S

Scalability

►Metrics
►Single System Image

Scalable Coherent Interface (SCI)

►SCI (Scalable Coherent Interface)

ScaLAPACK

Jack Dongarra, Piotr Luszczek
University of Tennessee, Knoxville, TN, USA

Definition

ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM [1] and/or MPI [2, 3]. It is a continuation of the LAPACK [4] project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers.

Discussion

Both LAPACK and ScaLAPACK libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems. The goals of both projects are efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so users can construct new routines from well-designed parts), and ease of use (by making the interface to LAPACK and ScaLAPACK look as similar as possible). Many of these goals, particularly portability, are aided by developing and promoting standards, especially for low-level communication and computation routines. These goals have been successfully attained, limiting most machine dependencies to two standard libraries called the BLAS, or Basic Linear Algebra Subprograms [5–8], and BLACS, or Basic Linear Algebra Communication Subprograms [9, 10]. LAPACK will run on any machine where the BLAS are available, and ScaLAPACK will run on any machine where both the BLAS and the BLACS are available.

The library is written in Fortran 77 (with the exception of a few symmetric eigenproblem auxiliary routines written in C to exploit IEEE arithmetic) in a Single Program Multiple Data (SPMD) style using explicit message passing for interprocessor communication. The name ScaLAPACK is an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK.

ScaLAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. ScaLAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS [11, 12], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [9, 10] for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, the majority of interprocessor communication occurs within the PBLAS. So the source code of the top software layer of ScaLAPACK looks similar to that of LAPACK.

ScaLAPACK contains *driver routines* for solving standard types of problems, *computational routines* to perform a distinct computational task, and *auxiliary routines* to perform a certain subtask or common

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,

low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software developers, so the Fortran source for these routines have been documented with the same level of detail used for the ScaLAPACK computational routines and driver routines.

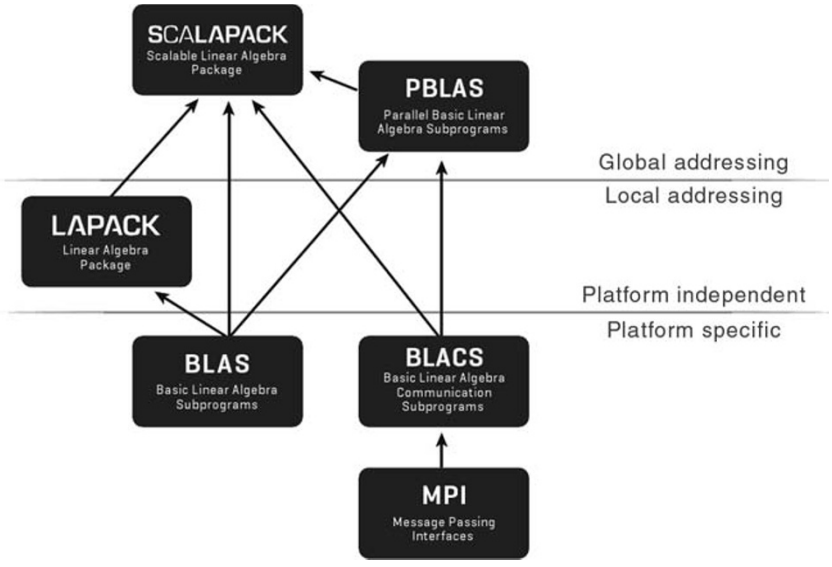
Dense and band matrices are provided for, but not general sparse matrices. Similar functionality is provided for real and complex matrices. However, not all the facilities of LAPACK are covered by ScaLAPACK yet.

ScaLAPACK is designed to give high efficiency on MIMD distributed-memory concurrent supercomputers, such as the older ones like Intel Paragon, IBM SP series, and the Cray T3 series, as well as the newer ones, such IBM Blue Gene series and Cray XT series of supercomputers. In addition, the software is designed so that it can be used with clusters of workstations through a networked environment and with a heterogeneous computing environment via PVM or MPI. Indeed, ScaLAPACK can run on any machine that supports either PVM or MPI.

The ScaLAPACK strategy for combining efficiency with portability is to construct the software so that as much as possible of the computation is performed by calls to the Parallel Basic Linear Algebra Subprograms (PBLAS). The PBLAS [11, 12] perform global computation by relying on the Basic Linear Algebra Subprograms (BLAS) [5–8] for local computation and the Basic Linear Algebra Communication Subprograms (BLACS) [9, 10] for communication.

The efficiency of ScaLAPACK software depends on the use of block-partitioned algorithms and on efficient implementations of the BLAS and the BLACS being provided by computer vendors (and others) for their machines. Thus, the BLAS and the BLACS form a low-level interface between ScaLAPACK software and different machine architectures. Above this level, all of the ScaLAPACK software is portable.

The BLAS, PBLAS, and the BLACS are not, strictly speaking, part of ScaLAPACK. C code for the PBLAS is included in the ScaLAPACK distribution. Since the performance of the package depends upon the BLAS and the BLACS being implemented efficiently, they have not been included with the ScaLAPACK distribution. A machine-specific implementation of the BLAS and the BLACS should be used. If a machine-optimized version of the BLAS is not available, a Fortran 77 reference implementation of the BLAS is available from Netlib [13]. This code constitutes the "model implementation" [14, 15]. The model implementation of the BLAS is not expected to perform as well as a specially tuned implementation on most high-performance computers – on



ScaLAPACK. Fig. 1 ScaLAPACK's software hierarchy

some machines it may give much worse performance – but it allows users to run ScaLAPACK codes on machines that do not offer any other implementation of the BLAS.

If a vendor-optimized version of the BLACS is not available for a specific architecture, efficiently ported versions of the BLACS are available on Netlib. Currently, the BLACS have been efficiently ported on machine-specific message-passing libraries such as the IBM (MPL) and Intel (NX) message-passing libraries, as well as more generic interfaces such as PVM and MPI. The BLACS overhead has been shown to be negligible [10]. Refer to the URL for the blacs directory on Netlib for more details: http://www.netlib.org/blacs/index.html

Figure 1 describes the ScaLAPACK software hierarchy. The components below the line, labeled *local addressing*, are called on a single processor, with arguments stored on single processors only. The components above the line, labeled *global addressing*, are synchronous parallel routines, whose arguments include matrices and vectors distributed across multiple processors.

LAPACK and ScaLAPACK are freely available software packages provided on the World Wide Web on Netlib. They can be, and have been, included in commercial packages. The authors ask only that proper credit be given to them which is very much like the modified BSD license.

Related Entries

►LAPACK
►Linear Algebra, Numerical

Bibliography

1. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994
2. MPI Forum, MPI: A message passing interface standard, International Journal of Supercomputer Applications and High Performance Computing, 8 (1994), pp 3–4. Special issue on MPI. Also available electronically, the URL is ftp://www.netlib.org/mpi/mpi-report.ps
3. Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra JJ (1996) MPI: The Complete Reference, MIT Press, Cambridge, MA
4. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel JW, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D, LAPACK Users' Guide, SIAM, 1992
5. Hanson R, Krogh F, Lawson CA (1973) A proposal for standard linear algebra subprograms, ACM SIGNUM Newsl, 8
6. Lawson CL, Hanson RJ, Kincaid D, Krogh FT (1979) Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans Math Soft 5:308–323
7. Dongarra JJ, Du Croz J, Hammarling Richard S, Hanson J (March 1988) An Extended Set of FORTRAN Basic Linear Algebra Subroutines, ACM Trans Math Soft 14(1):1–17
8. Dongarra JJ, Du Croz J, Duff IS, Hammarling S (March 1990) A Set of Level 3 Basic Linear Algebra Subprograms, ACM Trans Math Soft 16(1):1–17
9. Dongarra J, van de Geijn R (1991) Two dimensional basic linear algebra communication subprograms, Computer Science Dept. Technical Report CS-91–138, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #37)
10. Dongarra J, Whaley RC (1995) A user's guide to the BLACS v1.1, Computer Science Dept. Technical Report CS-95–281, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #94)
11. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley RC (May 1995) A proposal for a set of parallel basic linear algebra subprograms, Computer Science Dept. Technical Report CS-95–292, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #100)
12. Petitet A (1996) Algorithmic Redistribution Methods for Block Cyclic Decompositions, PhD thesis, University of Tennessee, Knoxville, TN
13. Dongarra JJ, Grosse E (July 1987) Distribution of Mathematical Software via Electronic Mail, Communications of the ACM 30(5): 403–407
14. Dongarra JJ, du Croz J, Duff IS, Hammarling S (1990) Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans Math Soft 16:18–28
15. Dongarra JJ, DU Croz J, Hammarling S, Hanson RJ (1998) Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subroutines, ACM Trans Math Soft 14:18–32

Scalasca

Felix Wolf
Aachen University, Aachen, Germany

Synonyms

The predecessor of Scalasca, from which Scalasca evolved, is known by the name of KOJAK.

Definition

Scalasca is an open-source software tool that supports the performance optimization of parallel programs by

measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes. Scalasca targets mainly scientific and engineering applications based on the programming interfaces MPI and OpenMP, including hybrid applications based on a combination of the two. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well suited for small- and medium-scale HPC platforms.

Discussion

Introduction

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers is expanding from generation to generation. As a consequence, supercomputing applications are required to harness much higher degrees of parallelism in order to satisfy their enormous demand for computing power. However, with today's leadership systems featuring more than a hundred thousand cores, writing efficient codes that exploit all the available parallelism becomes increasingly difficult. Performance optimization is therefore expected to become an even more essential software-process activity, critical for the success of many simulation projects. The situation is exacerbated by the fact that the growing number of cores imposes scalability demands not only on applications but also on the software tools needed for their development.

Making applications run efficiently on larger scales is often thwarted by excessive communication and synchronization overheads. Especially during simulations of irregular and dynamic domains, these overheads are often enlarged by wait states that appear in the wake of load or communication imbalance when processes fail to reach synchronization points simultaneously. Even small delays of single processes may spread wait states across the entire machine, and their accumulated duration can constitute a substantial fraction of the overall resource consumption. In particular, when trying to scale communication-intensive applications to large processor counts, such wait states can result in substantial performance degradation.

To address these challenges, Scalasca has been designed as a diagnostic tool to support application optimization on highly scalable systems. Although also covering single-node performance via hardware-counter measurements, Scalasca mainly targets communication and synchronization issues, whose understanding is critical for scaling applications to performance levels in the petaflops range. A distinctive feature of Scalasca is its ability to identify wait states that occur, for example, as a result of unevenly distributed workloads.

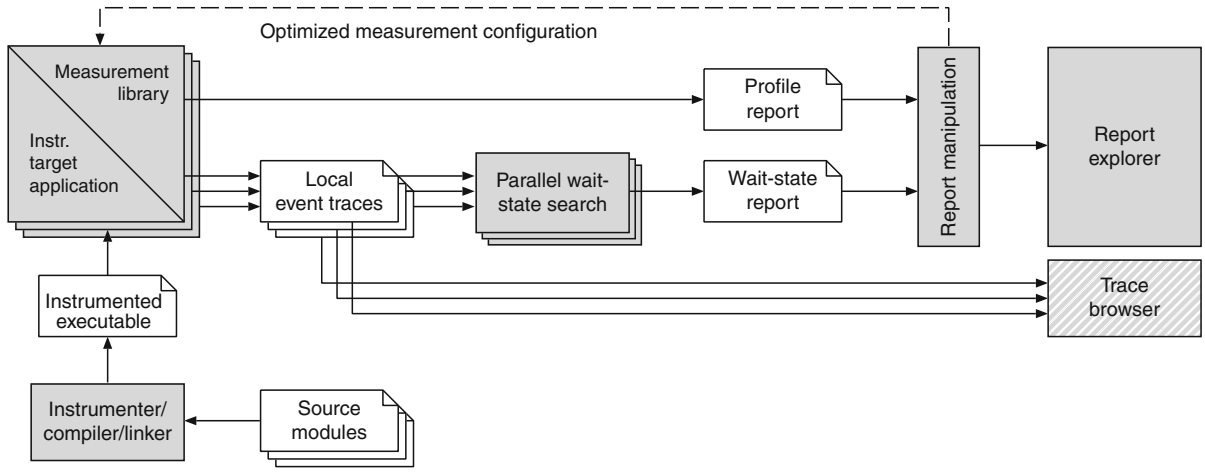
Functionality

To evaluate the behavior of parallel programs, Scalasca takes performance measurements at runtime to be analyzed *postmortem* (i.e., after program termination). The user of Scalasca can choose between two different analysis modes:

- Performance overview on the call-path level via profiling (called runtime summarization in Scalasca terminology)
- In-depth study of application behavior via event tracing

In profiling mode, Scalasca generates aggregate performance metrics for individual function call paths, which are useful for identifying the most resource-intensive parts of the program and assessing process-local performance via hardware-counter analysis. In tracing mode, Scalasca goes one step further and records individual performance-relevant events, allowing the automatic identification of call paths that exhibit wait states. This core feature is the reason why Scalasca is classified as an automatic tool. As an alternative, the resulting traces can be visualized in a traditional time-line browser such as ▸VAMPIR to study the detailed interactions among different processes or threads. While providing more behavioral detail, traces also consume significantly more storage space and therefore have to be generated with care.

Figure 1 shows the basic analysis workflow supported by Scalasca. Before any performance data can be collected, the target application must be instrumented, that is, probes must be inserted into the code which carry out the measurements. This can happen at different levels, including source code, object code, or library.



Scalasca. Fig. 1 Schematic overview of the performance data flow in Scalasca. *Gray rectangles* denote programs and *white rectangles* with the *upper right corner* turned down denote files. *Stacked symbols* denote multiple instances of programs or files, in most cases running or being processed in parallel. *Hatched boxes* represent optional third-party components

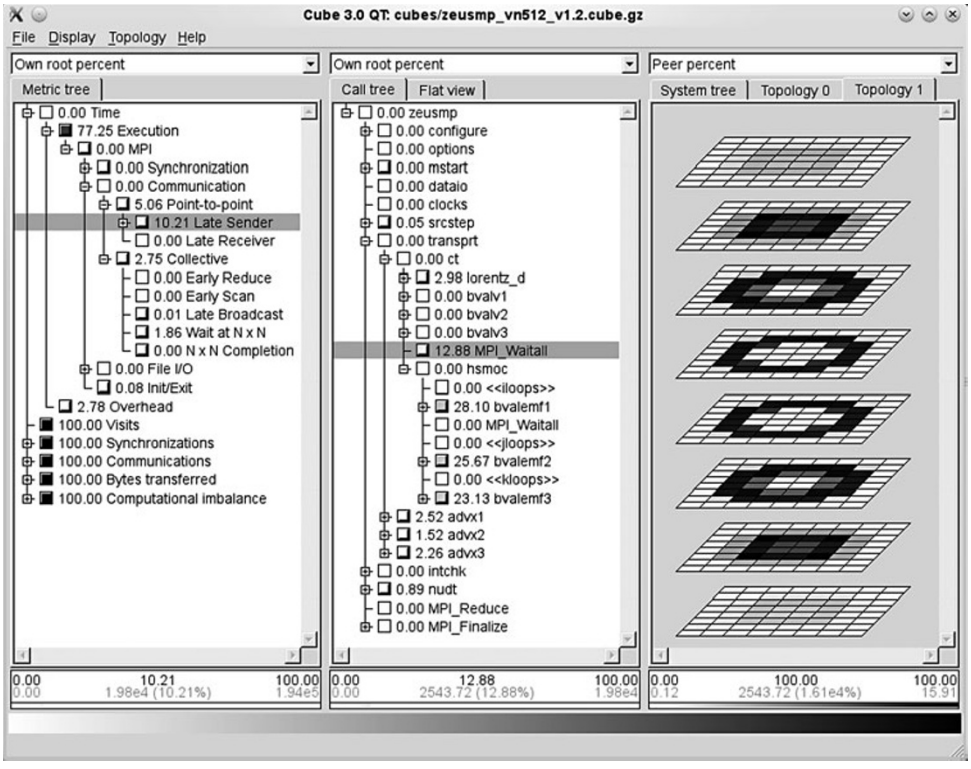
Before running the instrumented executable on the parallel machine, the user can choose between generating a profile or an event trace. When tracing is enabled, each process generates a trace file containing records for its process-local events. To prevent traces from becoming too large or inaccurate as a result of measurement intrusion, it is generally recommended to optimize the instrumentation based on a previously generated profile report. After program termination, Scalasca loads the trace files into main memory and analyzes them in parallel using as many cores as have been used for the target application itself. During the analysis, Scalasca searches for wait states, classifies detected instances by category, and quantifies their significance. The result is a wait-state report similar in structure to the profile report but enriched with higher-level communication and synchronization inefficiency metrics. Both profile and wait-state reports contain performance metrics for every combination of function call path and process/thread and can be interactively examined in the provided analysis report explorer (Fig. 2) along the dimensions of performance metric, call tree, and system. In addition, reports can be combined or manipulated to allow comparisons or aggregations, or to focus the analysis on specific extracts of a report. For example, the difference between two reports can be calculated to assess the effectiveness of an optimization or a new report can be generated after eliminating uninteresting phases (e.g., initialization).

Instrumentation

Preparation of a target application executable for measurement and analysis requires it to be *instrumented* to notify the measurement library, which is linked to the executable, of performance-relevant execution events whenever they occur at runtime. On all systems, a mix of manual and automatic instrumentation mechanisms is offered. Instrumentation configuration and processing of source files are achieved by prefixing selected compilation commands and the final link command with the Scalasca instrumenter, without requiring other changes to optimization levels or the build process, as in the following example for the file `foo.c`:

```
> scalasca -instrument mpicc -c foo.c
```

Scalasca follows a *direct instrumentation* approach. In contrast to interrupt-based sampling, which takes periodic measurements whenever a timer expires, Scalasca takes measurements when the control flow reaches certain points in the code. These points mark performance-relevant events, such as entering/leaving a function or sending/receiving a message. Although instrumentation points have to be chosen with care to minimize intrusion, direct instrumentation offers advantages for the global analysis of communication and synchronization operations. In addition to pure direct instrumentation, future versions of Scalasca will combine direct instrumentation with sampling in



Scalasca. Fig. 2 Interactive performance-report exploration for the Zeus MP/2 code [8] with Scalasca. The *left pane* lists different performance metrics arranged in a specialization hierarchy. The selected metric is a frequently occurring wait state called *Late Sender*, during which a process waits for a message that has not yet been sent (Fig. 3a). The number and the color of the icon to the *left* of the label indicate the percentage of execution time lost due to this wait state, in this case 10.21%. In the *middle pane*, the user can see which call paths are most affected. For example, 12.88% of the 10.21% is caused by the call path `zeusmp()` → `transprt()` → `ct()` → `MPI_Waitall()`. The *right pane* shows the distribution of waiting times for the selected combination of wait state and call path across the virtual process topology. The display indicates that most wait states occur at the outer rim of a spherical region in the center of the three-dimensional Cartesian topology

profiling mode to better control runtime dilation, while still supporting global communication analyses [15].

Measurement

Measurements are collected and analyzed under the control of a workflow manager that determines how the application should be run, and then configures measurement and analysis accordingly. When tracing is requested, it automatically configures and executes the parallel trace analyzer with the same number of processes as used for measurement. The following examples demonstrate how to request measurements from MPI application `bar` to be executed with 65,536 ranks, once in profiling and once in tracing mode (distinguished by the use of the "`-t`" option).

```
> scalasca -analyze mpiexec \
  -np 65536 bar <arglist>
> scalasca -analyze -t mpiexec \
  -np 65536 bar <arglist>
```

Call-Path Profiling

Scalasca can efficiently calculate many execution performance metrics by accumulating statistics during measurement, avoiding the cost of storing them with events for later analysis. For example, elapsed times and hardware-counter metrics for source regions

(e.g., routines or loops) can be immediately determined and the differences accumulated. Whereas trace storage requirements increase in proportion to the number of events (dependent on the measurement duration), summarized statistics for a call-path profile per thread have a fixed storage requirement (dependent on the number of threads and executed call paths).

In addition to call-path visit counts, execution times, and optional hardware counter metrics, Scalasca profiles include various MPI statistics, such as the numbers of synchronization, communication and file I/O operations along with the associated number of bytes transferred. Each metric is broken down into collective versus point-to-point/individual, sends/writes versus receives/reads, and so on. Call-path execution times separate MPI message-passing and OpenMP multithreading costs from purely local computation, and break them down further into initialization/finalization, synchronization, communication, file I/O and thread management overheads (as appropriate). For measurements using OpenMP, additional thread idle time and limited parallelism metrics are derived, assuming a dedicated core for each thread.

Scalasca provides accumulated metric values for every combination of call path and thread. A call path is defined as the list of code regions entered but not yet left on the way to the currently active one, typically starting from the main function, such as in the call chain `main()` → `foo()` → `bar()`. Which regions actually appear on a call path depends on which regions have been instrumented. When execution is complete, all locally executed call paths are combined into a global dynamic call tree for interactive exploration (as shown in the middle of Fig. 2, although the screen shot actually visualizes a wait-state report).

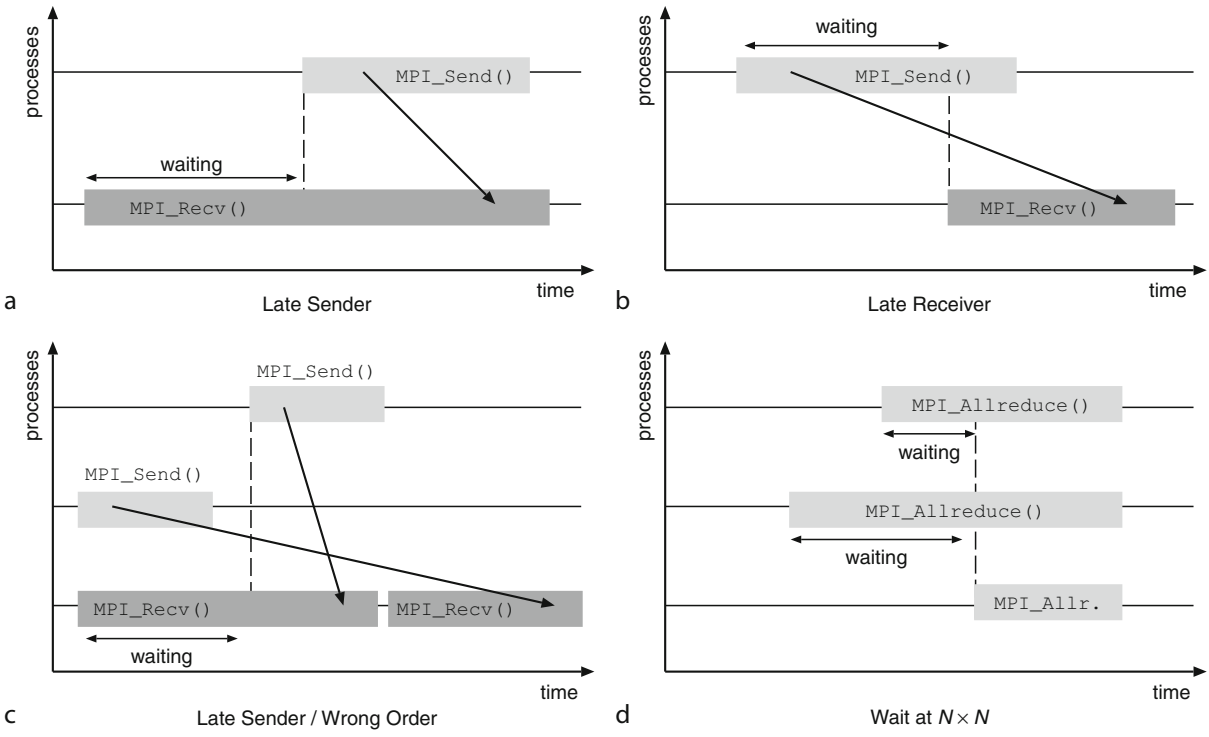
Wait-State Analysis

In message-passing applications, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. If a rendezvous protocol is used, this relationship also applies in the opposite direction. Collective synchronization is similar in that its completion requires each participating process to have reached a certain point. As a consequence, a significant fraction of the time spent in communication and synchronization routines can often be attributed to wait states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner. Scalasca provides a diagnostic method that allows the localization of wait states by automatically searching event traces for characteristic patterns.

Figure 3 shows several examples of wait states that can occur in message-passing programs. The first one is the *Late Sender* pattern (Fig. 3a), where a receiver is blocked while waiting for a message to arrive. That is, the receive operation is entered by the destination process before the corresponding send operation has been entered by the source process. The waiting time lost as a consequence is the time difference between entering the send and the receive operations. Conversely, the *Late Receiver* pattern (Fig. 3b) describes a sender that is likely to be blocked while waiting for the receiver when a rendezvous protocol is used. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default, or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation is blocked until the data is transferred to the receiver. The *Late Sender / Wrong Order* pattern (Fig. 3c) describes a receiver waiting for a message, although an earlier message is ready to be received by the same destination process (i.e., messages received in the wrong order). Finally, the *Wait at N×N* pattern (Fig. 3d) quantifies the waiting time due to the inherent synchronization in *n*-to-*n* operations, such as `MPI_Allreduce()`. A full list of the wait-state types supported by Scalasca including explanatory diagrams can be found online in the Scalasca documentation [11].

Parallel Wait-State Search

To accomplish the search in a scalable way, Scalasca exploits both distributed memory and parallel processing capabilities available on the target system. After the target application has terminated and the trace data have been flushed to disk, the trace analyzer is launched with one analysis process per (target) application process and loads the entire trace data into its distributed memory address space. Future versions of Scalasca may exploit persistent memory segments available on systems such as Blue Gene/P to pass the trace data to the analysis stage without involving any file I/O. While traversing the traces in parallel, the analyzer performs a replay of the application's original communication. During the replay, the analyzer



Scalasca. Fig. 3 Four examples of wait states (**a**–**d**) detected by Scalasca. The combination of MPI functions used in each of these examples represents just one possible case. For instance, the blocking receive operation in wait state (**a**) can be replaced by a non-blocking receive followed by a wait operation. In this case, the waiting time would occur during the wait operation

identifies wait states in communication and synchronization operations by measuring temporal differences between local and remote events after their time stamps have been exchanged using an operation of similar type. Detected wait-state instances are classified and quantified according to their significance for every call path and system resource involved. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, good scalability can be achieved even at previously intractable scales. Recent scalability improvements allowed Scalasca to complete trace analyses of runs with up to 294,912 cores on a 72-rack IBM Blue Gene/P system [23].

A modified form of the replay-based trace analysis scheme is also applied to detect wait states occurring in MPI-2 RMA operations. In this case, RMA communication is used to exchange the required information between processes. Finally, Scalasca also provides the ability to process traces from hybrid MPI/OpenMP and pure OpenMP applications. However, the parallel wait-state search does not yet recognize OpenMP-specific wait states, such as barrier waiting time or lock contention, previously supported by its predecessor.

Wait-State Search on Clusters without Global Clock

To allow accurate trace analyses on systems without globally synchronized clocks, Scalasca can synchronize inaccurate time stamps postmortem. Linear interpolation based on clock offset measurements during program initialization and finalization already accounts for differences in offset and drift, assuming that the drift of an individual processor is not time dependent. This step is mandatory on all systems without a global clock, such as Cray XT and most PC or compute blade clusters. However, inaccuracies and drifts

varying over time can still cause violations of the logical event order that are harmful to the accuracy of the analysis. For this reason, Scalasca compensates for such violations by shifting communication events in time as much as needed to restore the logical event order while trying to preserve the length of intervals between local events. This logical synchronization is currently optional and should be performed if the trace analysis reports (too many) violations of the logical event order.

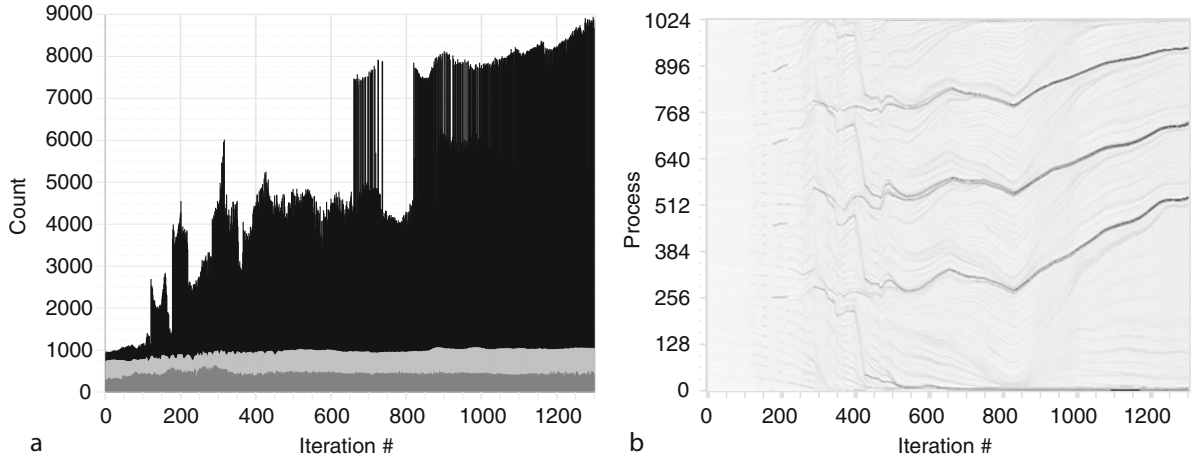
Future Directions

Future enhancements of Scalasca will aim at further improving both its functionality and its scalability. In addition to supporting the more advanced features of OpenMP such as nested parallelism and tasking as an immediate priority, Scalasca is expected to evolve toward emerging programming models and architectures including partitioned global address space (PGAS) languages and heterogeneous systems. Moreover, optimized data management and analysis workflows including in-memory trace analysis will allow Scalasca to master even larger processor configurations than it does today. A recent example in this direction is the substantial reduction of the trace-file creation overhead that was achieved by mapping large numbers of logical process-local trace files onto a small number of physical files [4], a feature that will become available in future releases of Scalasca.

In addition to keeping up with the rapid new developments in parallel hardware and software, research is also undertaken to expand the general understanding of parallel performance in simulation codes. The examples below summarize two ongoing projects aimed at increasing the expressive power of the analyses supported by Scalasca. The description reflects the status of March 2011.

Time-Series Call-Path Profiling

As scientific parallel applications simulate the temporal evolution of a system, their progress occurs via discrete points in time. Accordingly, the core of such an application is typically a loop that advances the simulated time step by step. However, the performance behavior may vary between individual iterations, for example, due to periodically reoccurring extra activities or when the state of the computation adjusts to new conditions in so-called adaptive codes. To study such time-dependent behavior, Scalasca can distinguish individual iterations in profiles and event traces. Figure 4 shows the distribution of point-to-point messages across the iteration-process space in the MPI-based PEPC [7] particle simulation code. Obviously, during later stages of the simulation the majority of messages are sent by only a small collection of processes with time-dependent



Scalasca. Fig. 4 Gradual development of a communication imbalance along 1,300 timesteps of PEPC on 1,024 processors. (**a**) Minimum (*bottom*), median (*middle*), and maximum (*top*) number of point-to-point messages sent or received by a process in an iteration. (**b**) Number of messages sent by each process in each iteration. The higher the number of messages the *darker the color* on the value map

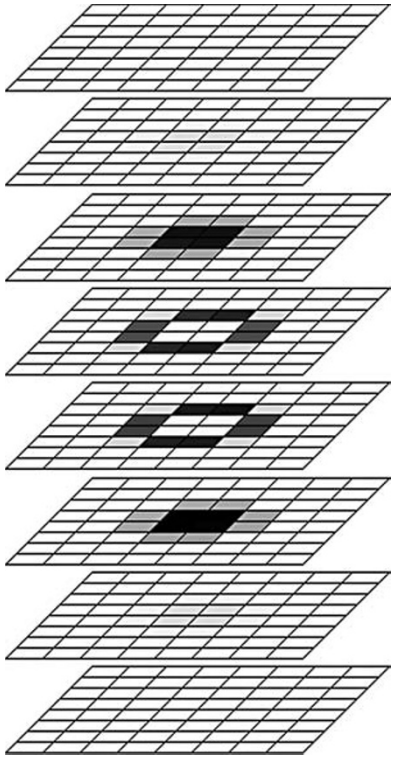
constituency, inducing wait states on other processes (not shown).

However, even generating call-path profiles (as opposed to traces) separately for thousands of iterations to identify the call paths responsible may exceed the available buffer space – especially when the call tree is large and more than one metric is collected. For this reason, a runtime approach for the semantic compression of a series of call-path profiles based on incrementally clustering single-iteration profiles was developed that scales in terms of the number of iterations without sacrificing important performance details [16]. This method, which will be integrated in future versions of Scalasca, offers low runtime overhead by using only a condensed version of the profile data when calculating distances and accounts for process-dependent variations by making all clustering decisions locally.

Identifying the Root Causes of Wait-State Formation

In general, the temporal or spatial distance between cause and symptom of a performance problem constitutes a major difficulty in deriving helpful conclusions from performance data. Merely knowing the locations of wait states in the program is often insufficient to understand the reason for their occurrence. Building on earlier work by Meira, Jr. et al. [12], the replay-based wait-state analysis was extended in such a way that it attributes the waiting times to their root causes [3], as exemplified in Fig. 5. Typically, these root causes are intervals during which a process performs some additional activity not performed by its peers, for example as a result of insufficiently balancing the load.

However, excess workload identified as the root cause of wait states usually cannot simply be removed. To achieve a better balance, optimization hypotheses drawn from such an analysis typically propose the redistribution of the excess load to other processes instead. Unfortunately, redistributing workloads in complex message-passing applications can have surprising side effects that may compromise the expected reduction of waiting times. Given that balancing the load statically or even introducing a dynamic load-balancing scheme constitute major code changes, such procedures should ideally be performed only if the prospective performance gain is likely to materialize. Other recent work [10] therefore concentrated on determining the savings we can realistically hope for when redistributing a given delay – before altering the application itself. Since the effects of such changes are hard to quantify analytically, they are simulated in a scalable manner via a parallel real-time replay of event traces after they have been modified to reflect the redistributed load.



Scalasca. Fig. 5 In the lorentz_d() subroutine of the Zeus MP/2 code, several processes primarily arranged on a small hollow sphere within the virtual process topology are responsible for wait states arranged on the enclosing hollow sphere shown earlier in Fig. 2. Since the inner region of the topology carries more load than the outer region, processes at the rim of the inner region delay those farther outside. The darkness of the color indicates the amount of waiting time induced by a process but materializing farther away. In this way, it becomes possible to study the root causes of wait-state formation

Related Entries

►Intel® Thread Profiler
►MPI (Message Passing Interface)
►OpenMP

- ►OpenMP Profiling with OmpP
- ►Performance Analysis Tools
- ►Periscope
- ►PMPI Tools
- ►Synchronization
- ►TAU
- ►Vampir

Bibliographic Notes and Further Reading

Scalasca is available for download including documentation under the New BSD license at www.scalasca.org.

Scalasca emerged from the KOJAK project, which was started in 1998 at the Jülich Supercomputing Centre in Germany to study the automatic evaluation of parallel performance data, and in particular, the automatic detection of wait states in event traces of parallel applications. The wait-state analysis first concentrated on MPI [20] and later on OpenMP and hybrid codes [21], motivating the definition of the POMP profiling interface [13] for OpenMP, which is still used today even beyond Scalasca by OpenMP-enabled profilers such as ompP (►OpenMP Profiling with OmpP) and ►TAU. A comprehensive description of the initial trace-analysis toolset resulting from this effort, which was publicly released for the first time in 2003 under the name KOJAK, is given in [19].

During the following years, KOJAK's wait-state search was optimized for speed, refined to exploit virtual process topologies, and extended to support MPI-2 RMA communication. In addition to the detection of wait states occurring during a single run, KOJAK also introduced a framework for comparing the analysis results of different runs [14], for example, to judge the effectiveness of optimization measures. An extensive snapshot of this more advanced version of KOJAK including further literature references is presented in [22]. However, KOJAK still analyzed the traces sequentially after the process-local trace data had been merged into a single global trace file, an undesired scalability limitation in view of the dramatically rising number of cores employed on modern parallel architectures.

In 2006, after the acquisition of a major grant from the Helmholtz Association of German Research Centres, the Scalasca project was started in Jülich as the successor to KOJAK with the objective of improving the scalability of the trace analysis by parallelizing the search for wait states. A detailed discussion of the parallel replay underlying the parallel search can be found in [6]. Variations of the scalable replay mechanism were applied to correct event time stamps taken on clusters without global clock [1], to simulate the effects of optimizations such as balancing the load of a function more evenly across the processes of a program [10], and to identify wait states in MPI-2 RMA communication in a scalable manner [9]. Moreover, the parallel trace analysis was also demonstrated to run on computational grids consisting of multiple geographically dispersed clusters that are used as a single coherent system [2]. Finally, a very recent replay-based method attributes the costs of wait states in terms of resource waste to their original cause [3].

Since the enormous data volume sometimes makes trace analysis challenging, runtime summarization capabilities were added to Scalasca both as a simple means to obtain a performance overview and as a basis to optimally configure the measurement for later trace generation. Scalasca integrates both measurement options in a unified tool architecture, whose details are described in [5]. Recently, a semantic compression algorithm was developed that will allow Scalasca to take time-series profiles in a space-efficient manner even if the target application performs large numbers of timesteps [16].

Major application studies with Scalasca include a survey of using it on leadership systems [24], a comprehensive analysis of how the performance of the SPEC MPI2007 benchmarks evolves as their execution progresses [17], and the investigation of a gradually developing communication imbalance in the PEPC particle simulation [18]. Finally, a recent study of the Sweep3D benchmark demonstrated performance measurements and analyses with up to 294,912 processes [23].

From 2003 until 2008, KOJAK and later Scalasca was jointly developed together with the Innovative Computing Laboratory at the University of Tennessee. During their lifetime, the two projects received funding from the Helmholtz Association of German Research Centres, the US Department of Energy, the US Department of Defense, the US National Science Foundation, the German Science Foundation, the German Federal Ministry of Education and Research, and the European Union. Today, Scalasca is a joint project between Jülich

and the German Research School for Simulation Sciences in nearby Aachen.

The following individuals have contributed to Scalasca and its predecessor: Erika Ábrahám, Daniel Becker, Nikhil Bhatia, David Böhme, Jack Dongarra, Dominic Eschweiler, Sebastian Flott, Wolfgang Frings, Karl Fürlinger, Christoph Geile, Markus Geimer, Marc-André Hermanns, Michael Knobloch, David Krings, Guido Kruschwitz, André Kühnal, Björn Kuhlmann, John Linford, Daniel Lorenz, Bernd Mohr, Shirley Moore, Ronal Muresano, Jan Mußler, Andreas Nett, Christian Rössel, Matthias Pfeifer, Peter Philippen, Farzona Pulatova, Divya Sankaranarayanan, Pavel Saviankou, Marc Schlütter, Christian Siebert, Fengguang Song, Alexandre Strube, Zoltán Szebenyi, Felix Voigtländer, Felix Wolf, and Brian Wylie.

Bibliography

1. Becker D, Rabenseifner R, Wolf F, Linford J (2009) Scalable timestamp synchronization for event traces of message-passing applications. Parallel Comput 35(12):595–607
2. Becker D, Wolf F, Frings W, Geimer M, Wylie BJN, Mohr B (2007) Automatic trace-based performance analysis of metacomputing applications. In: Proceedings of the international parallel and distributed processing symposium (IPDPS), Long Beach, CA, USA. IEEE Computer Society, Washington, DC
3. Böhme D, Geimer M, Wolf F, Arnold L (2010) Identifying the root causes of wait states in large-scale parallel applications. In: Proceedings of the 39th international conference on parallel processing (ICPP), San Diego, CA, USA. IEEE Computer Society, Washington, DC, pp 90–100
4. Frings W, Wolf F, Petkov V (2009) Scalable massively parallel I/O to task-local files. In: Proceedings of the ACM/IEEE conference on supercomputing (SC09), Portland, OR, USA, Nov 2009
5. Geimer M, Wolf F, Wylie BJN, Ábrahám E, Becker D, Mohr B (2010) The Scalasca performance toolset architecture. Concurr Comput Pract Exper 22(6):702–719
6. Geimer M, Wolf F, Wylie BJN, Mohr B (2009) A scalable tool architecture for diagnosing wait states in massively-parallel applications. Parallel Comput 35(7):375–388
7. Gibbon P, Frings W, Dominiczak S, Mohr B (2006) Performance analysis and visualization of the n-body tree code PEPC on massively parallel computers. In: Proceedings of the conference on parallel computing (ParCo), Málaga, Spain, Sept 2005 (NIC series), vol 33. John von Neumann-Institut für Computing, Jülich, pp 367–374
8. Hayes JC, Norman ML, Fiedler RA, Bordner JO, Li PS, Clark SE, ud-Doula A, Mac Low M-M (2006) Simulating radiating and magnetized flows in multiple dimensions with ZEUS-MP. Astrophys J Suppl 165(1):188–228
9. Hermanns M-A, Geimer M, Mohr B, Wolf F (2009) Scalable detection of MPI-2 remote memory access inefficiency patterns. In: Proceedings of the 16th European PVM/MPI users' group meeting (EuroPVM/MPI), Espoo, Finland. Lecture notes in computer science, vol 5759. Springer, Berlin, pp 31–41
10. Hermanns M-A, Geimer M, Wolf F, Wylie BJN (2009) Verifying causality between distant performance phenomena in large-scale MPI applications. In Proceedings of the 17th Euromicro international conference on parallel, distributed, and network-based processing (PDP), Weimar, Germany. IEEE Computer Society, Washington, DC, pp 78–84
11. Jülich Supercomputing Centre and German Research School for Simulation Sciences. Scalasca parallel performance analysis toolset documentation (performance properties). http://www.scalasca.org/download/documentation/
12. Meira W Jr, LeBlanc TJ, Poulos A (1996) Waiting time analysis and performance visualization in Carnival. In: Proceedings of the SIGMETRICS symposium on parallel and distributed tools (SPDT'96), Philadelphia, PA, USA. ACM
13. Mohr B, Malony A, Shende S, Wolf F (2002) Design and prototype of a performance tool interface for OpenMP. J Supercomput 23(1):105–128
14. Song F, Wolf F, Bhatia N, Dongarra J, Moore S (2004) An algebra for cross-experiment performance analysis. In: Proceedings of the international conference on parallel processing (ICPP), Montreal, Canada. IEEE Computer Society, Washington, DC, pp 63–72
15. Szebenyi Z, Gamblin T, Schulz M, de Supinski BR, Wolf F, Wylie BJN (2011) Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In: Proceedings of the international parallel and distributed processing symposium (IPDPS), Anchorage, AK, USA. IEEE Computer Society, Washington, DC
16. Szebenyi Z, Wolf F, Wylie BJN (2009) Space-efficient time-series call-path profiling of parallel applications. In: Proceedings of the ACM/IEEE conference on supercomputing (SC09), Portland, OR, USA, Nov 2009
17. Szebenyi Z, Wylie BJN, Wolf F (2008) SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proceedings of the 1st SPEC international performance evaluation workshop (SIPEW), Darmstadt, Germany. Lecture notes in computer science, vol 5119. Springer, Berlin, pp 99–123
18. Szebenyi Z, Wylie BJN, Wolf F (2009) Scalasca parallel performance analyses of PEPC. In: Proceedings of the workshop on productivity and performance (PROPER) in conjunction with Euro-Par, Las Palmas de Gran Canaria, Spain, August 2008. Lecture notes in computer science, vol 5415. Springer, Berlin, pp 305–314
19. Wolf F (2003) Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich. ISBN 3-00-010003-2
20. Wolf F, Mohr B (2001) Specifying performance properties of parallel applications using compound events. Parallel Distrib Comput Pract 4(3):301–317
21. Wolf F, Mohr B (2003) Automatic performance analysis of hybrid MPI/OpenMP applications. J Syst Archit 49(10–11):421–439
22. Wolf F, Mohr B, Dongarra J, Moore S (2007) Automatic analysis of inefficiency patterns in parallel applications. Concurr Comput Pract Exper 19(11):1481–1496

23. Wylie BJN, Geimer M, Mohr B, Böhme D, Szebenyi Z, Wolf F (2010) Large-scale performance analysis of Sweep3D with the Scalasca toolset. Parallel Process Lett 20(4):397–414
24. Wylie BJN, Geimer M, Wolf F (2008) Performance measurement and analysis of large-scale parallel applications on leadership computing systems. Sci Program 16(2–3):167–181

Scaled Speedup

►Gustafson's Law

Scan for Distributed Memory, Message-Passing Systems

Jesper Larsson Träff
University of Vienna, Vienna, Austria

Synonyms

All prefix sums; Prefix; Parallel prefix sums; Prefix reduction

Definition

Among a group of p consecutively numbered processing elements (nodes) each node has a data item x_i for $i = 0, \ldots p-1$. An associative, binary operator $\oplus$ over the data items is given. The nodes in parallel compute all p (or $p-1$) *prefix sums* over the input items: node i computes the *inclusive prefix sum* $\oplus_{j=0}^{i} x_j$ (or, for $i > 0$, the *exclusive prefix sum* $\oplus_{j=0}^{i-1} x_j$).

Discussion

For a general discussion, see the entry on "►Reduce and Scan."

For distributed memory, message-passing parallel systems, the scan operation plays an important role for load-balancing and data-redistribution operations. In this setting, each node possesses a local data item x_i, typically an n-element vector, and the given associative, binary operator $\oplus$ is typically an element-wise operation. The required prefix sums could be computed by arranging the input items in a linear sequence and "scanning" through this sequence in order, carrying along the corresponding inclusive (or exclusive) prefix sum. In a distributed memory setting, this intuition would be captured by arranging the nodes in a linear array, but much better algorithms usually exist. The inclusive scan operation is illustrated in Fig. 1.

The scan operation is included as a collective operation in many interfaces and libraries for distributed memory systems, for instance MPI [5], where both an inclusive (MPI_Scan) and an exclusive (MPI_Exscan) general scan operation is defined. These operations work on vectors of consecutive or non-consecutive elements and arbitrary (user-defined) associative, possibly also commutative operators.

Algorithms

On distributed memory parallel architectures parallel algorithms typically use tree-like patterns to compute the prefix sums in a logarithmic number of parallel communication rounds. Four such algorithms are described in the following. Algorithms also exist for hypercube and mesh- or torus-connected communication architectures [1, 4, 8].

The p nodes are consecutively numbered from 0 to $p-1$. Each node i has an input data item x_i. Nodes must communicate explicitly by send and receive operations. For convenience a fully connected model in which each node can communicate with all other nodes at the same cost is assumed. Only one communication operation per node can take place at a time. The data items x_i are assumed to all have the same size $n = |x_i|$. For scalar items, $n = 1$, while for vector items n is the number of elements. The communication cost of transferring a data item of size n is assumed to be $O(n)$ (linear).

Linear Array

The simplest scan algorithm organizes the nodes in a linear array. For $i > 0$ node i waits for a *partial sum*

Before			After		
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$\oplus_{j=0}^{0} x^{(j)}$	$\oplus_{j=0}^{1} x^{(j)}$	$\oplus_{j=0}^{2} x^{(j)}$

Scan for Distributed Memory, Message-Passing Systems. Fig. 1 The inclusive scan operation

$\oplus_{j=0}^{i-1} x_j$ from node $i-1$, adds its own item x_i to arrive at the partial sum $\oplus_{j=0}^{i} x_j$, and sends this to node $i+1$, unless $i = p-1$. This algorithm is strictly serial and takes $p-1$ communication rounds for the last node to finish its prefix computation, for a total cost of $O(pn)$. This is uninteresting in itself, but the algorithm can be pipelined (see below) to yield a time to complete of $O(p+n)$. When n is large compared to p this easily of implementable algorithm is often the fastest, due to its simplicity.

Binary Tree

The *binary tree* algorithm arranges the nodes in a balanced binary tree T with in-order numbering. This numbering has the property that the nodes in the subtree $T(i)$ rooted at node i have consecutive numbers in the interval $[s,\ldots,i,\ldots,e]$, where s and e denote the first (start) and last (end) node in the subtree $T(i)$, respectively. The algorithm consists of two phases. It suffices to describe the actions of a node in the tree with parent, right and left children. In the *up-phase*, node i first receives the *partial result* $\oplus_{j=s}^{i-1} x_j$ from its left child and adds its own item x_i to get the partial result $\oplus_{j=s}^{i} x_j$. This value is stored for the down-phase. Node i then receives the partial result $\oplus_{j=i+1}^{e} x_j$ from its right child and computes the partial result $\oplus_{j=s}^{e} x_j$. Node i sends this value upward in the tree without keeping it. In the *down-phase*, node i receives the partial result $\oplus_{j=0}^{s-1} x_j$ from its parent. This is first sent down to the left child and then added to the stored partial result $\oplus_{j=s}^{i} x_j$ to form the final result $\oplus_{j=0}^{i} x_j$ for node i. This final result is sent down to the right child.

With the obvious modifications, the general description covers also nodes that need not participate in all of these communications: Leaves have no children. Some nodes may only have a leftmost child. Nodes on the path between root and leftmost leaf do not receive data from their parent in the down-phase. Nodes on the path between rightmost child and root do not send data to their parent in the up-phase.

Since the depth of T is logarithmic in the number of nodes, and data of size n are sent up and down the tree, the time to complete the scan with the binary tree algorithm is $O(n \log p)$. This is unattractive for large n. If the nodes can only either send or receive in a communication operation but otherwise work simultaneously, the number of communication operations per node is at most 6, and the number of communication rounds is $6\lceil \log_2 p \rceil$.

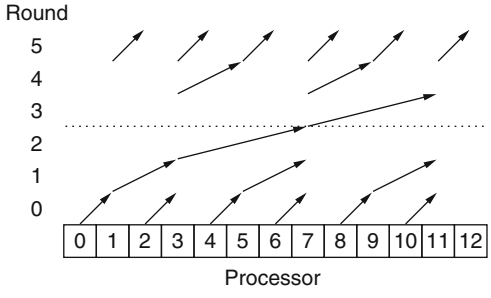
Binomial Tree

The *binomial tree* algorithm likewise consists of an *up-phase* and a *down-phase*, each of which takes k rounds where $k = \lfloor \log_2 p \rfloor$. In round j for $j = 0,\ldots,k-1$ of the up-phase each node i satisfying $i \wedge (2^{j+1}-1) \equiv 2^{j+1}-1$ (where $\wedge$ denotes "bitwise and") receives a partial result from node $i-2^j$ (provided $0 \leq i-2^j$). After sending to node i, node $i-2^j$ is inactive for the remainder of the up-phase. The receiving nodes add the partial results, and after round j have a partial result of the form $\oplus_{\ell=i-2^{j+1}+1}^{i} x_\ell$. The down-phase counts rounds downward from k to 1. Node i with $i \wedge (2^j-1) \equiv 2^j-1$ sends its partial result to node $i+2^{j-1}$ (provided $i+2^{j-1} < p$) which can now compute its final result $\oplus_{\ell=0}^{i+2^{j-1}} x_\ell$. The communication pattern is illustrated in Fig. 2.

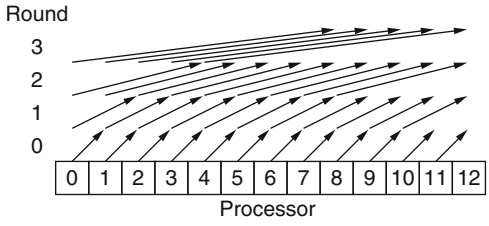
The number of communication rounds is $2\lceil \log p \rceil$, and the time to complete is $O(n \log p)$. Each node is either sending or receiving data in each round, with no possibility for overlapping of sending and receiving due to the computation of partial results.

Simultaneous Trees

The number of communication rounds can be improved by a factor of two by the *simultaneous binomial tree* algorithm. Starting from round $k = 0$, in round k, node i sends a computed, partial result $\oplus_{j=i-2^k+1}^{i} x_j$ to node $i+2^k$ (provided $i+2^k < p$) and receives a likewise computed, partial result from node $i-2^k$ (provided $i-2^k \geq 0$). Before the next round, the previous and received partial



Scan for Distributed Memory, Message-Passing Systems. Fig. 2 The communication pattern for the binomial tree algorithm for $p = 13$



Scan for Distributed Memory, Message-Passing Systems.
Fig. 3 The communication patterns for the simultaneous binomial tree algorithm for $p = 13$

results are added. It is easy to see that after round k, node i's partial result is $\oplus_{j=\max(0,i-2^{k+1}+1)}^{i} x_j$. Node i terminates when both $i - 2^k < 0$ (nothing to receive) and $i + 2^k \geq p$ (nothing to send). Provided that the nodes can simultaneously send and receive an item in the same communication operation this happens after $\lceil \log_2 p \rceil$ communication rounds for a total time of $O(n \log p)$. This is again not attractive for large n. This algorithm dates back at least to [2] and is illustrated in Fig. 3.

For large n the running time of $O(n \log p)$ is not attractive due to the logarithmic factor. For the binary tree algorithm pipelining can be employed, by which the running time can be improved to $O(n + \log p)$. Neither the binomial nor the simultaneous binomial tree algorithms admit pipelining. A pipelined implementation breaks each data item into a certain number of blocks N each of roughly equal size of n/N. As soon as a block has been processed and sent down the pipeline, the next block can be processed. The time of such an implementation is proportional to the depth of the pipeline plus the number of blocks times the time to process one block. From this an optimal block size can be determined, which gives the running time claimed for the binary tree algorithm. For very large vectors relative to the number of nodes, $n \gg p$, a linear pipeline seems to be the best practical choice due to very small constants. Pipelining can only be applied if the binary operator can work independently on the blocks. This is trivially the case if the items are vectors and the operator works element-wise on these.

Related Entries

►Collective Communication
►Message Passing Interface (MPI)
►Reduce and Scan

Bibliographic Notes and Further Reading

The parallel prefix problem was early recognized as a fundamental parallel processing primitive, and its history goes back at least as far as the early 1970s. The simultaneous binomial tree algorithm is from [2], but may have been discovered earlier as well. The binary tree algorithm easily admits pipelining but has the drawback that leaf nodes are only using either send or receive capabilities, and that simultaneous send-receive communication can be only partially exploited. In [6] it was shown how to overcome these limitations, yielding theoretically almost optimal scan (as well as reduce and broadcast) algorithms with (two) binary trees. Algorithms for distributed systems in the *LogP* performance model were presented in [7]. An algorithm for multiport message-passing systems was given in [3]. For results on meshes and hypercubes see, for instance [1, 4, 8].

Bibliography

1. Akl SG (1999) Parallel computation: models and methods. Prentice-Hall, Upper Saddle River
2. Daniel Hillis W, Steele GL Jr (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183
3. Lin Y-C, Yeh C-S (1999) Efficient parallel prefix algorithms on multiport message-passing systems. Inform Process Lett 71:91–95
4. Mayr EW, Plaxton CG (1993) Pipelined parallel prefix computations, and sorting on a pipelined hypercube. J Parallel Distrib Comput 17:374–380
5. MPI Forum (2009) MPI: A message-passing interface standard. Version 2.2. www.mpiforum.org. Accessed 4 Sept 2009
6. Sanders P, Speck J, Träff JL (2009) Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Comput 35:581–594
7. Santos EE (2002) Optimal and efficient algorithms for summing and prefix summing on parallel machines. J Parallel Distrib Comput 62(4):517–543
8. Ziavras SG, Mukherjee A (1996) Data broadcasting and reduction, prefix computation, and sorting on reduces hypercube parallel computer. Parallel Comput 22(4):595–606

Scan, Reduce and

►Reduce and Scan

Scatter

►Collective Communication

Scheduling

►Job Scheduling
►Task Graph Scheduling
►Scheduling Algorithms

Scheduling Algorithms

Patrice Quinton
ENS Cachan Bretagne, Bruz, France

Synonyms

Execution ordering

Definition

Scheduling algorithms aim at defining when operations of a program are to be executed. Such an ordering, called a *schedule*, has to make sure that the dependences between the operations are met. Scheduling for parallelism consists in looking for a schedule that allows a program to be efficiently executed on a parallel architecture. This efficiency may be evaluated in term of total execution time, utilization of the processors, power consumption, or any combination of this kind of criteria. Scheduling is often combined with *mapping*, which consists in assigning an operation to a resource of an architecture.

Discussion

Introduction

Scheduling is an essential design step to execute an algorithm on a parallel architecture. Usually, the algorithm is specified by means of a program from which dependences between statements or operations (also called *tasks*) can be isolated: an operation A depends on another operation B if the evaluation of B must precede that of A for the result of the algorithm to be correct.

Assuming the dependences of an algorithm are already known, scheduling has to find out a partial order, called a *schedule*, which says when each statement can be executed. A schedule may be resource-constrained, if the ordering must be such that some resource – for example, processor unit, memory, etc. – is available for the execution of the algorithm.

Scheduling algorithms are therefore classified according to the precision of the dependence analysis, the kind of resources needed, the optimization criteria considered, the specification of the algorithm, the granularity size of the tasks, and the kind of parallel architecture that is considered for execution.

Scheduling can also happen at run-time – *dynamic scheduling* – or at compile-time – *static scheduling*.

In the following, basic notions related to task graphs are first recalled, then scheduling algorithms for loops are presented. The presentation separates mono-dimensional and higher-dimensional loops. *Task Graph scheduling*, a topic that is covered in another essay of this site, is not treated here.

Task Graphs and Scheduling

Task graphs and their scheduling is the basis of many situations of parallel computations and has been the subject of a number of researches. Task graphs are a means of representing dependences between computations of a program: vertices V of the graph are elementary computations, called tasks, and oriented edges E represent dependences between computations. A task graph is therefore an acyclic, directed multigraph.

Assume that each task T has an estimated integer value $d(T)$ representing its execution duration. A schedule for a task graph is a function σ from the vertex set V of the graph to the set of positive integers, such that $\sigma(u) + d(u) \leq \sigma(v)$, where d represents the duration associated to the evaluation of task u, and v depends on u. Since it is acyclic, a task graph always admits a schedule.

Scheduling a task graph may be constrained by the number of processors p available to run the tasks. It is assumed that each task T, when ready to be evaluated, is allocated to a free processor, and occupies this processor

during its duration $d(T)$, after which the processor is released and becomes idle. When p is unlimited, we say that the schedule is *resource free*, otherwise, it is *resource constrained*.

The total computation time for executing a task graph on p processors, is also called its *makespan*. If the number of processors is illimited, that is, $p = +\infty$, it can be shown that finding out the makespan amounts to a simple traversal of the task graph.

When the number of processors is bounded, that is, $p < +\infty$, it can be shown that the problem becomes NP-complete. It is therefore required to rely on heuristics. In this case, a simple *greedy* heuristics is efficient: try to schedule at a given time as many tasks as possible on processors that are available at this time.

A *list schedule* is a schedule such that no processor is deliberately kept idle; at each time step t, if a processor P is available, and if some task T is *free*, that is to say, when all its predecessors have been executed, then T is scheduled to be executed on processor P. It can be shown that a list schedule allows a solution within 50% of the optimal makespan to be found.

List scheduling plays a very important rôle in scheduling theory and is therefore worth mentioning. A generic *list scheduling* algorithm looks as follows:

1. Initialization
   a. Assign to all free tasks a priority level (the choice of the priority function depends on the problem at hand).
   b. Set the time step t to 0.
2. While there remain tasks to execute:
   a. If the execution of a task terminates at time t, suppress this task from the predecessor list of all its successors. Add those tasks whose predecessor tasks has become empty to the free tasks, and assign them a priority.
   b. If there are q available processors, and r free tasks, execute the $\min(q, r)$ tasks of highest priority.
   c. Increase the time step t by one.

Scheduling One-Dimensional Loops

A particular field of interest for scheduling is how to execute efficiently a set of compuations that re-execute a large amount of time, also called *cyclic scheduling*. Such a problem is encountered when scheduling a one-dimensional loop. Consider for example, the following loop:

do $k = 0$, **step** 0, **to** $N - 1$

$\mathtt{A} : a(k) = c(k-1)$

$\mathtt{B} : b(k) = a(k-2) \times b(k-1)$

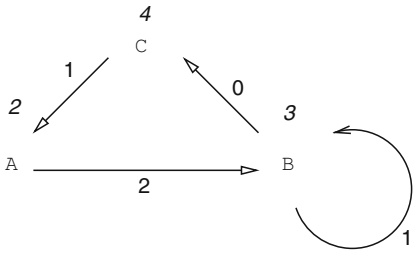
$\mathtt{C} : c(k) = b(k) + 1$

endo

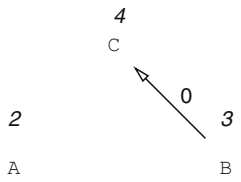
This loop could be modeled as a task graph where tasks would be $\mathtt{A}(k)$, $\mathtt{B}(k)$, and $\mathtt{C}(k)$, $k = 0, ..., N-1$. But if N is large, this graph becomes difficult to handle, hence the idea of finding out a generic way to schedule the operations by following the stucture of the loop.

Scheduling such a loop consists in finding out a time instant for each instruction to be executed, while respecting the dependences between these instructions. Call *operation* the execution of an instruction for a given value of index k, and denote $\mathtt{A}(k)$, $\mathtt{B}(k)$, and $\mathtt{C}(k)$ these operations. Notice that $\mathtt{A}(k)$ depends on the value $c(k-1)$, which is computed by operation $\mathtt{C}(k-1)$ during the previous iteration of the loop. Similarly, $\mathtt{B}(k)$ depends on $\mathtt{B}(k-1)$ and $\mathtt{A}(k-2)$; finally, $\mathtt{C}(k)$ depends on $\mathtt{B}(k)$, an operation of the same loop iteration. To model this situation, the notion of *reduced dependence graph*, which gather all information needed to schedule the loop, is introduced: its vertices are operation names (e.g., A, B, C etc.), and its edges represent dependences between operations. To each vertex v, associate an integer value $d(v)$ that represents its execution duration, and to each edge, associate a positive integer w that is 0 if the dependence lies within an iteration, and the difference between the iterations number otherwise. Fig. 1 represents the reduced dependence graph of the above loop.

With this representation, a schedule becomes a function σ from $V \times \mathbf{N}$ to $\mathbf{N}$, where $\mathbf{N}$ denotes the set of integers; $\sigma(v, k)$ gives the time at which instruction v of iteration k can be executed. Of course, a schedule must meet the dependence constraints, which can be expressed in the following way: for all edge $e = (u, v)$ of the dependence graph, it is needed that $\sigma(v, k + w(e)) \geq \sigma(u, k) + d(u)$.



Scheduling Algorithms. Fig. 1 Reduced dependence graph of the loop



Scheduling Algorithms. Fig. 2 Reduced dependence graph without nonzero edges. This graph is necessarily acyclic, and a simple traversal provides a schedule

For this kind of problem, the performance of a schedule is defined as the *average cycle time* λ defined as

$$\lambda = \lim_{N\to\infty} \frac{\max\{\sigma(v,k) + d(v) \mid v \in V, 0 \le k < N\}}{N}.$$

In order to define a *generic* schedule for such a loop, it is interesting to consider schedules of the form $\sigma(v,k) = \lambda k + c(v)$ where λ is a positive integer and $c(v)$ is a constant integer associated to instruction v. Constant λ is called the *initiation interval* of the loop, and it represents the number of cycles between the beginning of two successive iterations.

Finding out a schedule is quite easy: by restricting the reduced dependence graph to the intra-iteration dependences (i.e., dependences whose value w is 0), one obtains an acyclic graph that can be scheduled using a list scheduling algorithm. Let then λ be the makespan of such schedule, and let $ls(v)$ be the schedule assigned to vertex v by the list scheduling; then, a schedule of the form $\sigma(v,k) = \lambda k + ls(v)$ is a solution to the problem. Fig. 2 shows, for example, the task graph obtained when removing nonzero inter-iteration dependencies. One can see that the makespan of this graph is 7, and therefore, a schedule of the form $\sigma(v,k) = 7k + ls(v)$ is a solution. Figure 3 shows a possible execution, on two processors, with an initiation interval of 7.

However, one can do better in general, as shall be seen. Again, the *cyclic scheduling* problem can be stated with or without resources. Assuming, for the sake of simplicity, that instructions are carried out by a set of p identical processing units, the number of available processors constitutes a resource limit.

First, one can give a lower bound for the *initiation interval*. Let $\mathcal{C}$ be a cycle of the reduced dependence graph, and let $d(\mathcal{C})$ be its duration (the sum of the durations of its instructions) and $w(\mathcal{C})$ be its iteration weight, i.e., the sum of the iteration weights of its edges. Then, $\lambda \ge \frac{d(\mathcal{C})}{w(\mathcal{C})}$. The intuition behind this bound is that the duration of the tasks of the cycle have to be spread on the number of iterations spanned by this cycle, and as cycles have all the same duration λ, one cannot do better than this ratio. Applying this bound to this example, it can be seen that $\lambda \ge \frac{9}{3} = 3$ (cycle A → B → C → A) and $\lambda \ge 3$ (self dependence of B). Can the value $\lambda = 3$ be reached?

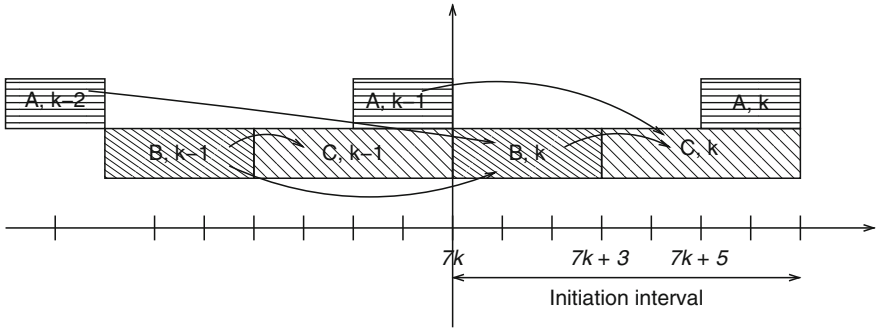
The answer depends again on the resource constraints. When an unlimited number of processors is available, one can show that the lower bound can be reached by solving a polynomial algorithm (essentially, Belman-Ford longest path algorithm). For this example, this would lead to the schedule $\sigma(\mathrm{B},k) = 3k$, $\sigma(\mathrm{C},k) = 3k+3$, and $\sigma(\mathrm{A},k) = 3k+4$, which meets the constraints and allows the execution of the loop on 4 processors (see Fig. 4).

Solving the problem for limited resources is NP-complete. Another lower bound is related to the number of processors available. If p processors are available, then $p\lambda \ge \sum_{v\in V} d(v)$. Intuitively, the total execution time available during one iteration on all processors ($p\lambda$) cannot be less than the total execution time needed to evaluate all tasks of an iteration.

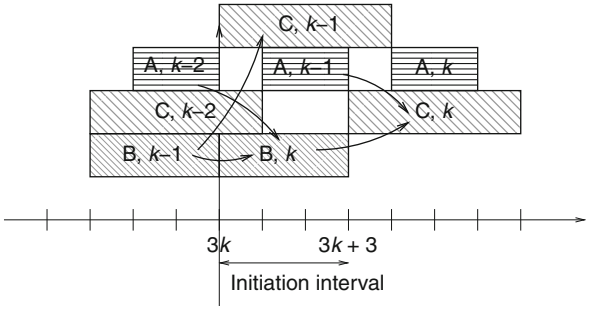
Several heuristics have been proposed in the literature to approach this bound: loop compaction, loop shifting, modulo scheduling, etc. All these methods rely upon list scheduling, as presented in the Sect. "Task graphs and Scheduling". Notice, finally, that there exits *guaranteed heuristics* for solving this problem. One such heuristics combines loop compaction, loop shifting and retiming to obtain such a result.

Scheduling Multidimensional Loops

Scheduling unidimensional loops is based on the idea of avoiding unrolling the loop and therefore, of keeping



Scheduling Algorithms. Fig. 3 Execution of the loop with an initiation interval of 7. The corresponding schedule is $\sigma(\mathtt{A},k) = 7k+5$, $\sigma(\mathtt{B},k) = 7k$, and $\sigma(\mathtt{C},k) = 7k+3$



Scheduling Algorithms. Fig. 4 Execution of the loop with an optimal initiation interval of 3. The corresponding schedule is $\sigma(\mathtt{A},k) = 3k+4$, $\sigma(\mathtt{B},k) = 3k$, and $\sigma(\mathtt{C},k) = 3k+3$

the model compact and tractable, independently on the loop bound N. To this end, the schedule is chosen to be a linear expression of the loop index. The same idea can be pushed further: instead of unrolling a multiple index loop, one tries to express the schedule of an operation by a linear (or affine) function of the loop indexes. This technique can be applied to imperative loops, but it interacts deeply with the detection of dependences in the loop, making the problem more difficult. Another way of looking at this problem is to consider algorithms expressed as recurrence equations, where parallelism is directly exposed.

Consider the following recurrences for the matrix multiplication algorithm $C = AB$ where A, B, and C are square matrices of size N:

$$1 \le i,j,k \le N \rightarrow C(i,j,k) = C(i,j,k-1) + a(i,k) \times b(k,j) \quad (1)$$

$$1 \le i,j \le N \rightarrow c(i,j) = C(i,j,N). \quad (2)$$

Each one of these equations is a single assignment, and it expresses a recurrence, whose indexes belong to the set defined by the left-hand inequality. The first recurrence (1) defines how C is computed, where k represents the iteration index; the second recurrence (2) says that value $c(i,j)$ is obtained as the result of the evaluation of $C(i,j,k)$ for $k = N$.

An easy way to schedule these computations for parallelism is to find out a schedule t that is an affine function of all the indexes of these calculations. Assume that C has a schedule of the form $t(i,j,k) = \lambda_1 i + \lambda_2 j + \lambda_3 k + \alpha$; also, assume that the evaluation of equations take at least one cycle, one must have, from Eq. 1:

$$t(i,j,k) > t(i,j,k-1), \ \forall i,j,k \ s.t. \ 1 \le i,j,k \le N$$

or once simplified, $\lambda_3 > 0$. This defines a set of possible values for t, with simplest form $t(i,j,k) = k$ (coefficient α can be chosen arbitrarily equal to 0.) Once a schedule is found, it is possible to allocate the computations of a recurrence equation on a parallel architecture. A simple way of doing it is to use a linear allocation function: for the matrix multiplication example, one could take i and j as the number of the processors, which would result in a parallel matrix multiplication executed on a square grid-connected array of size N. But many other allocations are possible.

This simple example contains all the ingredients of loop parallelization techniques, as will be seen now.

First, notice that this kind of recurrence is particular: it is said to be *uniform*, as left-hand side operations depend uniformly on right-hand side expressions through a dependence of the form $(0,0,1)$ (such a dependence is a simple translation of the left-hand side index). In general, one can consider *affine recurrences*

where any right-hand side index is an affine function of the left-hand side index.

Second, the set where the indexes must lie is an *integral convex polyhedron*, whose bounds may be parameterized by some symbolic values (here N for example).

This is not surprising: most of the loops that are of interest for parallelism share the property that array indexes are almost always affine, and that the index space of a loop can be modeled with a polyhedron.

Scheduling affine recurrences in general can be also done by extending the method. Assume that a recurrence has the form

$$\forall z \in P, V(z) = f(U(a(z)), \ldots)$$

where U and V are variables of a program, z is a n-dimensional integer vector, P is a polyhedron, and $a(z)$ is an affine mapping of z. The difference with the previous case is that the number of dependences may now be infinite: depending on the a function and the size of P, one may have a large, potentially infinite, number of dependences between instances of $V(z)$ and $U(a(z))$, for which the schedule t must meet $t(V(z)) > t(U(a(z)))$. However, properties of polyhedra may be used. A given polyhedron P may be represented equivalently by a set of inequalities, or by its set of generators, that is to say, its vertices, rays, and lines. Generators are in finite number, and it turns out that it is necessary and sufficient to check the property of t on vertices and rays, and this leads to a finite set of linear inequalities involving the coefficients of the t function. This is the basis of the so-called *vertex method* for scheduling recurrences. Another, equivalent method, called the *Farkas method*, involves directly the inequalities instead of the generator system of the polyhedron. In general, such methods lead to high complexity algorithms, but in practice, they are tractable: a typical simple algorithm leads to a few hundreds of linear constraints.

Without going into the details, these methods can be extended to cover interesting problems. First of all, it is not necessary to assume that all variables of the recurrences have the same schedule. One may instead suppose that variable V is scheduled with a function $t_V(i,j,\ldots) = \lambda_{V,1} i + \lambda_{V,2} j + \ldots + \alpha_V$ without making the problem more difficult. Secondly, one may also suppose that the schedule has several dimensions: if operation $V(z)$ is scheduled at time $\begin{pmatrix} t_{V,1}(z) \\ t_{V,2}(z) \end{pmatrix}$, then the time is understood as a lexicographical ordering. This allows modeling situations where more than one dimension of the index space are evaluated sequentially on a single processor, thus allowing architectures of various dimensions to be found.

Very few researches have yet explored scheduling loops under constraints, as was the case for cyclic scheduling.

A final remark: cyclic scheduling is a particular case of multidimensional scheduling, where the function has the form $\sigma(v,k) = \lambda k + \alpha_v$. Indeed, the α constant describes the relative time of execution of operation $v(k)$ during an iteration. In a similar way for higher-level recurrences, α_V plays the same rôle and allows one to refine the pipeline structure of one processor.

From Loops to Recurrences

The previous section (Sect. Scheduling Multi-Dimensional Loops) has shown how a set of recurrence equations can be scheduled using linear programming techniques. Here it is sketched how loops expressed in a some imperative language can be translated into single-assignment form that is equivalent to recurrence equations. Consider the following example:

```
for i := 0 to 2n do
    {S1}:  c[i] := 0.;
for i :=0 to n do
    for j := 0 to n do
        {S2}:  c[i+j] := c[i+j] +
        a[i]*b[j];
```

which computes the product of two polynomials of size $n+1$, where the coefficients of the input polynomials are in arrays `a` and `b` and the result in `c`. This program contains two loops, each one with one statement, labeled `S1` and `S2` respectively.

Each statement represents as many elementary *operations* as loop indexes surrounding it. Statement `S1` corresponds to $2n+1$ operations, denoted by $(\texttt{S1}, (i))$, and statement `S2` corresponds to operations denoted $(\texttt{S2}, (i,j))$.

The left-hand side element of statement `S2`, `c[i+j]`, writes several times in the same memory cell and its right-hand side makes use of the content of the same

memory cell. In order to rewrite these loops as recurrence equations, one need to carry out a *data flow* analysis to find out which statement made the last modification to `c[i+j]`. Since both statement modify `c`, it may be either `S2` itself, or `S1`.

Consider first the case when `S2` is the source of the operation that modifies `c[i+j]` for operation $(\texttt{S2},(i,j))$. Then this operation is $(\texttt{S2},(i',j'))$ where $i'+j' = i+j$, and indexes i' and j' must satisfy the limit constraints of the loops, that is, $0 \le i' \le n$ and $0 \le j' \le n$. There is another condition, also called *sequential predicate*: $(\texttt{S2},(i',j'))$ must be the operation that was the last to modify cell `c[i+j]` before $(\texttt{S2},(i,j))$ uses it. But (i',j') is smaller than (i,j) in lexicographic order, and thus, (i',j') is the lexical maximum of (i,j). All these conditions being affine inequalities, it turns out that finding out the source index can be done using *parameterized linear programming*, resulting in a so-called *quasi-affine selection tree* (quast) whose conditions on indices are also linear. One would thus find out that

$$(i',j') = \textit{if } i>1 \wedge j<n \text{ then } (i-1,j+1) \text{ else } \perp$$

where $\perp$ represents the situation when (i',j') was defined by a statement that precedes the loops.

Now if `S1` is the source of the operation, this operation is $(\texttt{S1},i')$ where $i' = i+j$, and index i' must lay inside the bounds, thus $0 \le i' \le 2n$. Moreover, this operation must be executed before, which is obviously always true. Thus, it can be seen that $(i') = (i+j)$. Combining both expressions, one finds that the source of (i,j) is given by

$$\textit{if } i>1 \wedge j<n \text{ then } (S2,i-1,j+1) \text{ else } (S1,i+j).$$

Once the source of each operation is found, rewriting the program as a system of recurrence equations is easy. For each statement `S`, a variable $S(i,j,\ldots)$ with as many indexes as surrounding loops is introduced, and the right hand side equation is rewritten by replacing expressions by their source translation. An extra equation can be provided to express the value of output variables of the program, here, `c` for example.

$$0 \le i \le 2n \rightarrow S1(i) = 0. \tag{3}$$

$$0 \le i \le n \wedge 0 \le j \le n \rightarrow S2(i,j) = \textit{if } i>1 \wedge j<n \tag{4}$$

$$\text{then } S1(i-1,j+1) + a(i) * b(j)$$

$$\text{else } S1(i+j) + a(i) * b(j) \tag{5}$$

$$0 \le i \le 2n \rightarrow c(i) = \textit{if } j=0 \text{ then } S2(i,j) \text{ else } S2(n,j) \tag{6}$$

The exact data flow analysis is a means of expressing implicit dependences that result from the encoding of the algorithm as loops. Although other methods exist in order to schedule loops without single assignement, this method is deeply related to scheduling techniques.

Scheduling Today and Future Directions

Loop scheduling is a central process of autoparallelization techniques, either implicit – when loops are rewritten after dependence analysis and then compiled, – or explicit, when the scheduling function is calculated and explicitly used during the parallelization process. Compilers incorporate more and more often sophisticated dependence and analysis tools to produce efficient code for target parallel architecture. Mono-dimensional loop scheduling is both used to massage loops for instruction-level parallelism, but also to design special-purpose hardware for embedded calculations. Finally, multi-dimensional scheduling is included in some autoparallelizers to produce code for parallel processors.

The need to produce efficient compilers is a limitation to the use of complex scheduling methods, as it has been shown that most practical problems are NP-complete. Progress in the knowledge of scheduling algorithms has made more and more likely for methods, previously rejected because of their complexity, to become common ingredients of parallelization tools. Moreover, the design of parallel programs tolerates a slightly longer production time than compiling everydays' program does, in particular because many applications of parallelism target embedded systems. Thus, the spreading of parallel computers – for example, GPU or multi-core – will certainly make these techniques useful.

Bibliographic Notes and Further Reading

Darte, Robert, and Vivien [2] wrote a reference book on scheduling for parallelism; it provides an in-depth coverage of the techniques presented here, in particular task scheduling and cyclic scheduling. It also provides a survey and comparison of various methods for dependence analysis. This entry borrows a large part of its material from this book.

El-Rewini, Lewis, and Ali give in [3] a survey on scheduling in general and task scheduling in particular. In [1], Benoit and Robert give complexity results for scheduling problems that are related to recent parallel platforms.

Lam [9] presents *software pipelining*, one of the most famous applications of cyclic scheduling to loops. Theoretical results on cyclic scheduling can be found in [7].

Karp, Miller, and Winograd [8] were among the first authors to consider scheduling for parallel execution, in a seminal paper on recurrence equations. Lamport [10] published one of the most famous papers on loop parallelization. Loop dependence analysis and loop transformations for parallelization are extensively presented by Zima and Chapman in [12]. Mauras, Quinton, Rajopadhye, and Saouter presented in [11] the vertex method to schedule affine recurrences, whereas Feautrier ([5] and [6]) describes the Farkas method for scheduling affine loop nests.

Data flow analysis is presented in great detail by Feautrier in [4].

Bibliography

1. Benoit A, Robert Y (Oct 2008) Complexity results for throughput and latency optimization of replicated and data-parallel workflows. Algorithmica 57(4):689–724
2. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhäuser, Boston
3. El-Rewini H, Lewis TG, Ali HH (1994) Task scheduling in parallel and distributed systems. Prentice Hall, Englewood Cliffs, New Jersey
4. Feautrier P (February 1991) Dataflow analysis of array and scalar references. Int J Parallel Program 20(1):23–53
5. Feautrier P (Oct 1992) Some efficient solutions to the affine scheduling problem, Part I, one dimensional time. Int J Parallel Program 21(5):313–347
6. Feautrier P (December 1992) Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. Int J Parallel Program 21(6):389–420
7. Hanen C, Munier A (1995) Cyclic Scheduling on parallel processors: An overview. In: Chrétienne P, Coffman EG Jr, Lenstra K, Lia Z (eds) Scheduling theory and its applications. John Wiley & Sons
8. Karp RM, Miller RE, Winograd S (July 1967) The organization of computations for uniform recurrence equations. J Assoc Comput Machin 14(3):563–590
9. Lam MS (1988) Software pipelining: an effective scheduling technique for VLIW Machines. In: SIGPLAN'88 conference on programming language, design and implementation, Atlanta, GA. ACM Press, pp 318–328
10. Lamport L (Feb 1974) The parallel execution of DO loops. Commun ACM 17(2):83–93
11. Mauras C, Quinton P, Rajopadhye S, Saouter Y (September 1990) Scheduling affine parameterized recurrences by means of variable dependent timing functions. In: Kung SY, Schwartzlander EE, Fortes JAB, Przytula KW (eds) Application Specific Array Processors, Princeton University, IEEE Computer Society Press, pp 100–110
12. Zima H, Chapman B (1989) Supercompilers for Parallel and Vector Computers. ACM Press, New York

SCI (Scalable Coherent Interface)

Hermann Hellwagner
Klagenfurt University, Klagenfurt, Austria

Synonyms

International standard ISO/IEC 13961:2000(E) and IEEE Std 1596, 1998 edition

Definition

Scalable Coherent Interface (SCI) is the specification (standardized by ISO/IEC and the IEEE) of a high-speed, flexible, scalable, point-to-point-based interconnect technology that was implemented in various ways to couple multiple processing nodes. SCI supports both the message-passing and shared-memory communication models, the latter in either the cache-coherent or non-coherent variants. SCI can be deployed as a system area network for compute clusters, as a memory interconnect for large-scale, cache-coherent, distributed-shared-memory multiprocessors, or as an I/O subsystem interconnect.

Discussion

Introduction

SCI originated in an effort of bus experts in the late 1980s to define a very high performance computer bus ("Superbus") that would support a significant degree of multiprocessing, i.e., number of processors. It was soon realized that backplane bus technology would not be able to meet this requirement, despite advanced concepts like split transactions and sophisticated implementations of the latest bus standards and products. The committee thus abandoned the bus-oriented view

and developed novel, distributed solutions to overcome the shared-resource and signaling problems of buses, while retaining the overall goal of defining an interconnect that offers the convenient services well known from centralized buses [3, 6].

The resulting specification of SCI, approved initially in 1992 [8] and finally in the late 1990s [9], thus describes hardware and protocols that provide processors with the shared-memory view of buses. SCI also specifies related transactions to read, write, and lock memory locations without software protocol involvement, as well as to transmit messages and interrupts. Hardware protocols to keep processor caches coherent are defined as an implementation option. The SCI interconnect, the memory system, and the associated protocols are fully distributed and scalable: an SCI network is based on point-to-point links only, implements a distributed shared memory (DSM) in hardware, and avoids serialization in almost any respect.

Goals of SCI's Development

Several ambitious goals guided the specification process of SCI [3, 5, 8].

High performance. The primary objective of SCI was to deliver high communication performance to parallel or distributed applications. This comprises:

- High sustained throughput;
- Low latency;
- Low CPU overhead for communication operations.

The performance goals set forth were in the range of Gbit/s link speeds and latencies in the low microseconds range in loosely-coupled systems, even less in tightly-coupled multiprocessors.

Scalability. SCI was devised to address scalability in many respects, among them [4]:

- Scalability of *performance* (aggregate bandwidth) as the number of nodes grows;
- Scalability of interconnect *distance*, from centimeters to tens of meters, depending on the media and physical layer implementation, but based on the same logical layer protocols;
- Scalability of the *memory system*, in particular of the *cache coherence protocols*, without a practical limit on the number of processors or memory modules that could be handled;
- *Technological scalability*, i.e., use of the same mechanisms in large-scale and small-scale as well as tightly-coupled and loosely-coupled systems, and the ability to readily make use of advances in technology, e.g., high-speed links;
- *Economic scalability*, i.e., use of the same mechanisms and components in low-end, high-volume and high-end, low-volume systems, opening the chance to leverage the economies of scale of mass production of SCI hardware;
- No short-term practical limits to the *addressing capability*, i.e., an addressing scheme for the DSM wide enough to support a large number of nodes and large node memories.

Coherent memory system. Caches are crucial for modern microprocessors to reduce average access time to data. This specifically holds for a DSM system with NUMA (non-uniform memory access) characteristics where remote accesses can be roughly an order of magnitude more expensive than local ones. To support a convenient programming model as known, e.g., from symmetric multiprocessors (SMPs), the caches should be kept coherent in hardware.

Interface characteristics. The SCI specification was intended to describe a standard interface to an interconnect to enable multiple devices from multiple vendors to be attached and to interoperate. In other words, SCI should serve as an "open distributed bus" connecting components like processors, memory modules, and intelligent I/O devices in a high-speed system.

Main Concepts of SCI

In the following, the main concepts and features of SCI will be summarized and the achievements, as represented by several implementations of SCI networks, will be assessed.

Point-to-point links. An SCI interconnect is defined to be built only from unidirectional, point-to-point links between participating nodes. These links can be used for concurrent data transfers, in contrast to the one-at-a-time communication characteristics of buses. The number of the links grows as nodes are added to the system, increasing the aggregate bandwidth of the network. The links can be made fast and their performance can scale with improvements in the underlying technology. They can be implemented in a bit-parallel manner

(for small distances) or in a bit-serial fashion (for larger distances), with the same logical layer protocols. Most implementations use parallel links over distances of a few centimeters or meters.

Sophisticated signaling technology. The data transfer rates and lengths of shared buses are inherently limited due to signal propagation delays and signaling problems on the transmission lines. The unidirectional, point-to-point SCI links avoid such signaling problems. Since there is only a single transmitter and a single receiver rather than multiple devices, the signaling speed can be increased significantly. High speeds are also fostered by low-voltage differential signals.

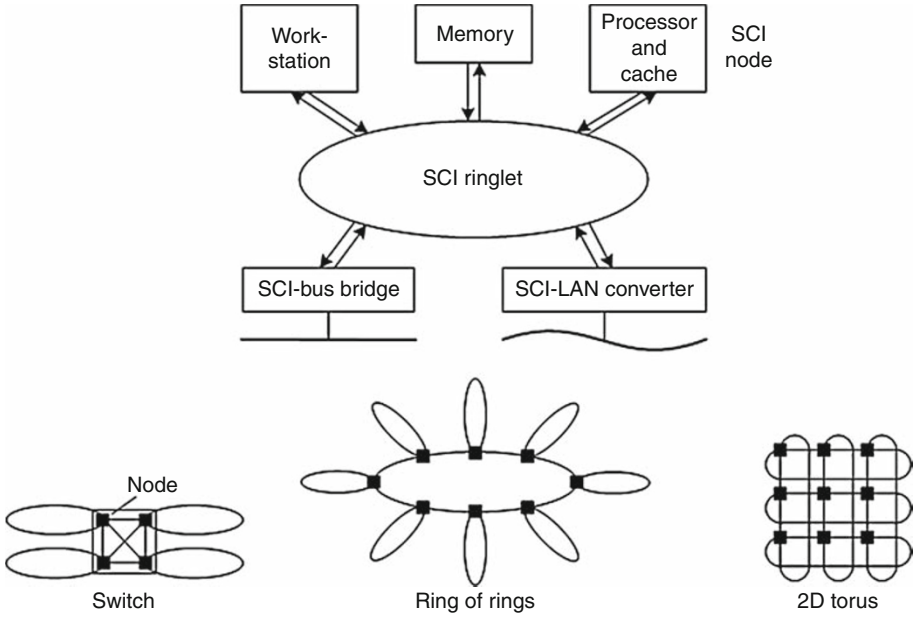
Furthermore, SCI strictly avoids back-propagating signals, even reverse flow control on the links, in favor of high signaling speeds and scalability. Flow control information becomes part of the normal data stream in the reverse direction, leading to the requirement that an SCI node must at least have one outgoing link and one incoming link.

Already in the mid 1990s, SCI link implementation speeds reached 500 Mbyte/s in system area networks (distances of a few meters, 16-bit parallel links) and 1 Gbyte/s in closely-coupled, cache-coherent shared-memory multiprocessors; transfer rates of 1 Gbyte/s have also been demonstrated over a distance of about 100 m, using parallel fiber-optic links; see [6].

Nodes. SCI was designed to connect up to 64 k nodes. A node can be a complete workstation or server machine, a processor and its associated cache only, a memory module, I/O controllers and devices, or bridges to other buses or interconnects, as illustrated exemplarily in Fig. 1. Each node is required to have a standard interface to attach to the SCI network, as described in [3, 5, 8]. In most SCI systems implemented so far, nodes are complete machines, often even multiprocessors.

Topology independence. In principle, SCI networks with complex topologies could be built; investigations into this area are described in several chapters of [6]. However, the standard anticipates simple topologies to be used. For small systems, for instance, the preferred topology is a small ring (a so-called *ringlet*); for larger systems, topologies like a single switch connecting multiple ringlets, rings of rings, or multidimensional tori are feasible; see Fig. 1. Most SCI systems implemented used single rings, a switch, multiple rings, or two-dimensional tori.

Fixed addressing scheme. SCI uses the 64-bit fixed addressing scheme defined by the Control and Status



SCI (Scalable Coherent Interface). Fig. 1 Simple SCI network topologies

Register (CSR) Architecture standard (IEEE Std 1212-1991) [7]. The 64-bit SCI address is divided into two fixed parts: the most significant 16 address bits specify the node ID (node address) so that an SCI network can comprise up to 64 k nodes; the remaining 48 bits are used for addressing within the nodes, in compliance with the CSR Architecture standard.

Hardware-based distributed shared memory (DSM). The SCI addressing scheme spans a global, 64-bit address space; in other words, a physically addressed, distributed shared memory system. The distribution of the memory is transparent to software and even to processors, i.e., the memory is logically shared. A memory access by a processor is mediated to the target memory module by the SCI hardware.

The major advantage of this feature is that inter-node communication can be effected by simple load and store operations by the processor, without invocation of a software protocol stack. The instructions accessing remote memory can be issued at user level; the operating system need not be involved in communication. This results in very low latencies for SCI communication, typically in the low microseconds range.

A major implementation challenge, however, is how to integrate the SCI network (and, thus, access to the system-wide SCI DSM) with the memory architecture of a standard single-processor workstation or a multiprocessor node. The common solutions, attaching SCI to the I/O bus or to the memory bus, will be outlined below.

Bus-like services. To complete the hardware DSM, SCI defines transactions to read, write, and lock memory locations, functionality well-known from computer buses. In addition, message passing and global time synchronization are supported, both as defined by the CSR Architecture; interrupts can be delivered remotely as well. Broadcast functionality is also defined.

Transactions can be tagged with four different priorities. In order to avoid starvation of low-priority nodes, fair protocols for bandwidth allocation and queue allocation have been developed. Bandwidth allocation is similar in effect to bus arbitration in that it assigns transfer bandwidth (if scarce) to nodes willing to send. Queue allocation apportions space in the input queues of heavily loaded, shared nodes, e.g., memory modules or switch ports, which are targeted by many nodes simultaneously. Since the services have to be implemented in a fully distributed fashion, the underlying protocols are rather complex.

Split transactions. Like multiprocessor buses, SCI strictly splits transactions into *request* and *response* phases. This is a vital feature to avoid scalability impediments; it makes signaling speed independent of the distance a transaction has to travel and avoids monopolizing network links. Transactions therefore have to be self-contained and are sent as packets, containing a transaction ID, addresses, commands, status, and data as needed. A consequence is that multiple transactions can be outstanding per node. Transactions can thus be pumped into the network at a high rate, using the interconnect in a pipeline fashion.

Optional cache coherence. SCI defines distributed cache coherence protocols, based on a distributed-directory approach, a multiple readers – single writer sharing regime, and write invalidation. The memory coherence model is purposely left open to the implementer. The standard provides optimizations for common situations such as pair-wise sharing that improve performance of frequent coherence operations.

The cache coherence protocols are designed to be implemented in hardware; however, they are highly sophisticated and complex. The complexity stems from a large number of states of coherent memory and cache blocks, correspondingly complex state transitions, and the advanced algorithms that ensure atomic modifications of the distributed-directory information (e.g., insertions, deletions, invalidations). The greatest complication arises from the integration of the SCI coherence protocols with the snooping protocols typically employed on the nodes' memory buses. An implementation is highly challenging and incurs some risks and potentially high costs. Not surprisingly, only a few companies have done implementations, as described below.

The cache coherence protocols are provided as options only. A compliant SCI implementation need not cover coherence; an SCI network even cannot participate in coherence actions when it is attached to the I/O bus as is the case in compute clusters. Yet, a common misconception was that cache coherence was required functionality at the core of SCI. This

misunderstanding has clearly hindered SCI's proliferation for non-coherent uses, e.g., as a system area network.

Reliability in hardware. In order to enable high-speed transmission, error detection is done in hardware, based on a 16-bit CRC code which protects each SCI packet. Transactions and hardware protocols are provided that allow a sender to detect failure due to packet corruption, and allow a receiver to notify the sender of its inability to accept packets (due to a full input queue) or to ask the sender to re-send the packet. Since this happens on a per-packet basis, SCI does not automatically guarantee in-order delivery of packets. This may have a considerable impact on software which would rely on a guaranteed packet sequence. An example is a message passing library delivering data into a remote buffer using a series of remote write transactions and finally updating the tail pointer of the buffer. Some SCI hardware provides functionality to enforce a certain memory access order, e.g., via memory barriers [6].

Various time-outs are provided to detect lost packets or transmission errors. Hardware retry mechanisms or software recovery protocols may be implemented based on transmission-error detection and isolation mechanisms; these are however not part of the standard. As a consequence, SCI implementations differed widely in the way errors are dealt with.

The protocols are designed to be robust, i.e., they should, e.g., survive the failure of a node with outstanding transactions. Among other mechanisms, error containment and logging procedures, ringlet maintenance functions and a packet time-out scheme are specified. Robustness is particularly important for the cache coherence protocols which are designed to behave correctly even if a node fails amidst the modification of a distributed-directory entry.

Layered specification. The SCI specification is structured into three layers.

At the *physical layer*, three initial physical link models are defined: a parallel electrical link operating at 1 Gbyte/s over short distances (meters); a serial electrical link that operates at 1 Gbit/s over intermediate distances (tens of meters); and a serial optical link that operates at 1 Gbit/s over long distances (kilometers).

Although the definitions include the electrical, mechanical, and thermal characteristics of SCI modules, connectors, and cables, the specifications were not adhered to in implementations. SCI systems typically used vendor-specific physical layer implementations, incompatible to others. It is fair to say, therefore, that SCI has not become the open, distributed interconnect system that the designers had envisaged to create.

The *logical layer* specifies transaction types and protocols, packet types and formats, packet encodings, the standard node interface structure, bandwidth and queue allocation protocols, error processing, addressing and initialization issues, and SCI-specific CSRs.

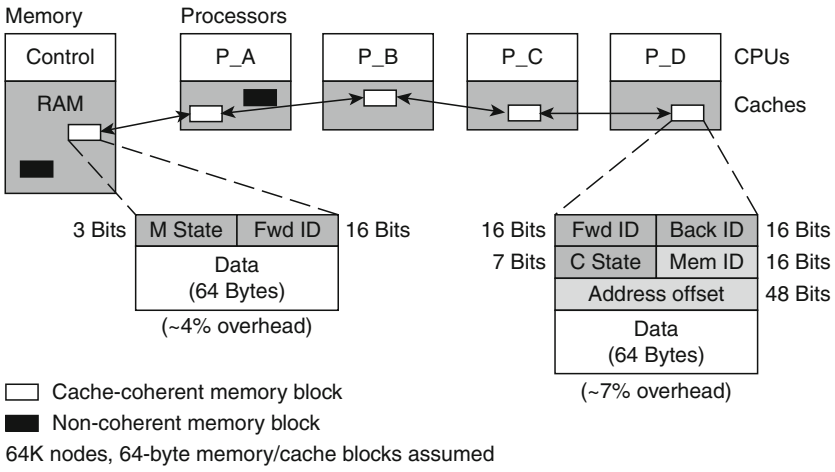
The *cache coherence layer* provides concepts and hardware protocols that allow processors to cache remote memory blocks while still maintaining coherence among multiple copies of the memory contents. Since an SCI network no longer has a central resource (i.e., a memory bus) that can be snooped by all attached processors to effect coherence actions, distributed-directory-based solutions to the cache coherence problem had to be devised.

At the core of the cache coherence protocols are the distributed sharing lists shown in Fig. 2. Each shared block of memory has an associated distributed, doubly-linked sharing list of processors that hold a copy of the block in their local caches. The memory controller and the participating processors cooperatively and concurrently create and update a block's sharing list, depending on the operations on the data.

C code. A remarkable feature of the SCI standard is that major portions are provided in terms of a "formal" specification, namely, C code. Text and figures are considered explanatory only, the definitive specification are the C listings. Exceptions are packet formats and the physical layer specifications. The major reasons for this approach are that C code is (largely) unambiguous and not easily misunderstood and that the specification becomes executable, as a simulation. In fact, much of the specification was validated by extensive simulations before release.

Implementations and Applications of SCI

SCI was originally conceived as a shared-memory interconnect, but SCI's flexibility and performance potential



SCI (Scalable Coherent Interface). Fig. 2 Sharing list and coherence tags of SCI cache coherence protocols

also for other applications was soon realized and leveraged by industry. In the following, a few of these "classical" applications of SCI are introduced and examples of commercial systems that exploited SCI technology are provided.

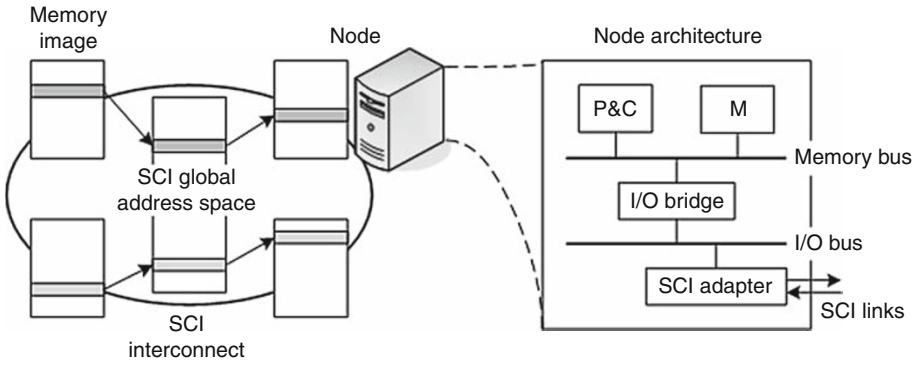
System Area Network for Clusters

In compute clusters, an SCI system area network can provide high-performance communication capabilities. In this application, the SCI interconnect is attached to the I/O bus of the nodes (e.g., PCI) by a peripheral adapter card, very similar to a LAN; see Fig. 3. In contrast to "standard" LAN technology and most other system area networks, though, the SCI cluster network, by virtue of the common SCI address space and associated transactions, provides hardware-based, *physical* distributed shared memory. Figure 4 shows a high-level view of the DSM. An SCI cluster thus is more tightly coupled than a LAN-based cluster, exhibiting the characteristics of a NUMA parallel machine.

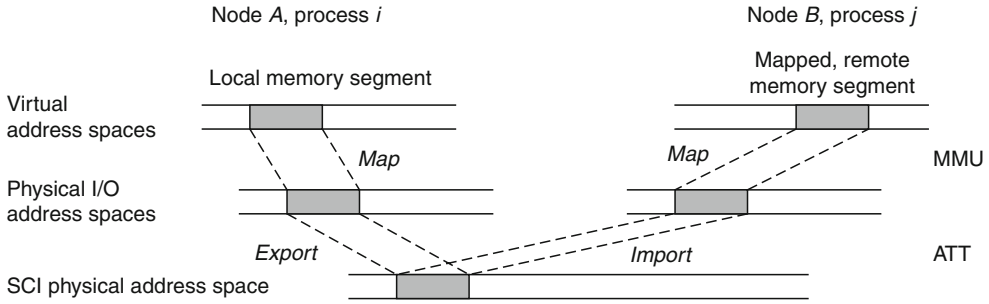
The SCI adapter cards, together with the SCI driver software, establish the DSM as depicted in Fig. 4. (This description pertains to the solutions taken by the pioneering vendor, Dolphin Interconnect Solutions, for their SBus-SCI and PCI-SCI adapter cards [2, 6].) A node that is willing to share memory with other nodes (e.g., *A*), creates shared memory segments in its physical memory and exports them to the SCI network (i.e., SCI address space). Other nodes (e.g., *B*) import these DSM segments into their I/O address space. Using on-board address translation tables (ATTs), the SCI adapters maintain the mappings between their local I/O addresses and the global SCI addresses. Processes on the nodes (e.g., *i* and *j*) may further map DSM segments into their virtual address spaces. The latter mappings are conventionally being maintained by the processors' MMUs.

Once the mappings have been set up, inter-node communication may be performed by the participating processes at *user level*, by simple CPU load and store operations into DSM segments mapped from remote memories. The SCI adapters translate I/O bus transactions that result from such memory accesses into SCI transactions, and vice versa, and perform them on behalf of the requesting processor. Thus, remote memory accesses are both transparent to the requesting processes and do not need intervention by the operating system. In other words, no protocol stack is involved in remote memory accesses, resulting in low communication latencies.

In the 1990s, the prevailing commercial implementations of such SCI cluster networks were the SBus-SCI and PCI-SCI adapter cards (and associated switches) offered by Dolphin Interconnect Solutions [2]. These cluster products were used by other companies to build key-turn cluster platforms; examples were Sun Microsystems, offering high-performance, high-availability server clusters, and Scali Computers and Siemens providing clusters for parallel computing based on MPI; see [6]. Similar SCI adapters are still offered at the time of writing and are being used in embedded systems, e.g., medical devices such as CT scanners [2].



SCI (Scalable Coherent Interface). Fig. 3 SCI cluster model



SCI (Scalable Coherent Interface). Fig. 4 Address spaces and address translations in SCI clusters

In addition, there were research implementations of adapter cards, among them one developed at CERN as a prototype to investigate SCI's feasibility and performance for demanding data acquisition systems in high-energy physics, i.e., in one of the LHC experiments; see [6].

The description of an SCI cluster interconnect given above does not address many of the low-level problems and functionality that the implementation has to cover. For instance, issues like the translation of 32-bit node addresses to 64-bit SCI addresses and vice versa, the choice of the shared segment size, error detection and handling, high-performance data transfers between node hardware and SCI adapter, and the design and implementation of low-level software (SCI drivers) as well as message-passing software (e.g., MPI) represent a spectrum of research and development problems. Several contributions in [6] describe these problems and corresponding solutions in some detail.

An important property of such SCI cluster interconnect adapters is worth pointing out here. Since an SCI cluster adapter attaches to the I/O bus of a node, it cannot directly observe, and participate in, the traffic on the memory bus of the node. This therefore precludes caching and coherence maintenance of memory regions mapped to the SCI address space. In other words, remote memory contents are basically treated as non-cacheable and are always accessed remotely. Standard SCI cluster interconnect hardware does not implement cache coherence capabilities therefore. This property raises a performance concern: remote accesses (round-trip operations such as reads) must be used judiciously since they are still an order of magnitude more expensive than local memory accesses (NUMA characteristics).

The basic approach to deal with the latter problem is to avoid remote operations that are inherently round-trip, i.e., reads, as far as possible. Rather, remote writes are used which are typically buffered by the SCI adapter and therefore, from the point of view of the processor issuing the write, experience latencies in the range of local accesses, several times faster than remote read operations. Again in [6], several chapters describe how considerations like this influence the design and

implementation of efficient message-passing libraries on top of SCI.

Finally, several contributions in [6] deal with how to overcome the limitations of non-coherent SCI cluster hardware, in particular for the implementation of shared-memory and shared-object programming models. Techniques from software DSM systems, e.g., replication and software coherence maintenance, are applied to provide a more convenient abstraction, e.g., a common virtual address space spanning all the nodes in the SCI cluster.

Memory Interconnect for Cache-Coherent Multiprocessors

The use of SCI as a cache-coherent memory interconnect allows nodes to be even more tightly coupled than in a non-coherent cluster. This application requires SCI to be attached to the memory bus of a node, as shown in Fig. 5. At this attachment point, SCI can participate in and "export," if necessary, the memory and cache coherence traffic on the bus and make the node's memory visible and accessible to other nodes. The nodes' memory address ranges (and the address mappings of processes) can be laid out to span a global (virtual) address space, giving processes transparent and coherent access to memory anywhere in the system. Typically, this approach is adopted to connect multiple bus-based commodity SMPs to form a large-scale, cache-coherent (CC) shared-memory system, often termed a CC-NUMA machine.

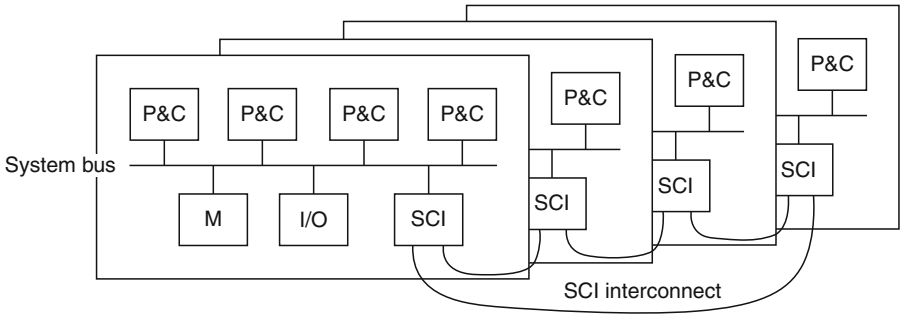
There were three notable examples of SCI-based CC-NUMA machines: the HP/Convex Exemplar series [1], the Sequent NUMA-Q multiprocessor [10], and the Data General AViiON scalable servers (see [6]). The latter two systems comprised bus-based SMP nodes with Intel processors, while the Exemplar used HP PA-RISC processors and a non-blocking crossbar switch within the nodes. The inter-node memory interconnects were proprietary implementations of the SCI standard, with specific adaptations and optimizations incorporated to ease implementation and integration with the node architecture and to foster overall performance.

The major challenge in building a CC-NUMA machine is to bridge the cache coherence mechanisms on the intra-node interconnect (e.g., the SMP bus running a snooping protocol) and the inter-node network (SCI). Since it is well documented, the Sequent NUMA-Q machine is used as a case study in [6] to illustrate the essential issues in building such a bridge and making the protocols interact correctly.

I/O Subsystem Interconnect

SCI can be used to connect one or more I/O subsystems to a computing system in novel ways. The shared SCI address space can include the I/O nodes which then are enabled to directly transfer data between the peripheral devices (in most cases, disks) and the compute nodes' memories using DMA; software needs not be involved in the actual transfer. Remote peripheral devices in a cluster, for instance, thus can become accessible like local devices, resulting in an I/O model similar to SMPs; remote interrupt capability can also be provided via SCI. High bandwidth and low latency, in addition to the direct remote memory access capability, make SCI an interesting candidate for an I/O network.

There were two commercial implementations of SCI-based I/O system networks. One was the GigaRing channel from SGI/Cray [12], the other one the



SCI (Scalable Coherent Interface). Fig. 5 SCI based CC-NUMA Multiproccessor Model

external I/O subsystem interconnect of the Siemens RM600 Enterprise Servers, based on Dolphin's cluster technology; see again [6].

Concluding Remarks

SCI addresses difficult interconnect problems and specifies innovative distributed structures and protocols for a scalable DSM architecture. The specification covers a wide spectrum of bus, network, and memory architecture problems, ranging from signaling considerations up to distributed directory-based cache coherence mechanisms.

In fact, this wide scope of SCI has raised criticism that the standard is actually "several standards in one" and difficult to understand and work with. This lack of a clear profile and the wide applicability of SCI have probably contributed to its relatively modest acceptance in industry. Furthermore, the SCI protocols are complex (albeit well devised) and not easily implemented in silicon. Thus, implementations are quite complex and therefore expensive.

In an attempt to reduce complexity and optimize speed, many of the implementers adopted the concepts and protocols which they regarded as appropriate for their application, and left out or changed other features according to their needs. This use of SCI led to a number of proprietary, incompatible implementations.

As a consequence, the goal of economic scalability has not been satisfactorily achieved in general. Further, SCI has clearly also missed the goal of evolving into an "open distributed bus" that multiple devices from different vendors could attach to and interoperate.

However, in terms of the technical objectives, predominantly high performance and scalability, SCI has well achieved its ambitious goals. The vendors that have adopted and implemented SCI (in various flavors), offered innovative high-throughput, low-latency interconnect products or full-scale cache-coherent shared-memory multiprocessing systems, as described above.

SCI had noticeable influence on further developments, e.g., interconnect standard projects like SyncLink, a high-speed memory interface, and Serial Express (SerialPlus), an extension to the SerialBus (IEEE 1394) interconnect. SCI also played a role in the debate on "future I/O systems" in the late 1990s, which then led to the InfiniBand™Architecture. The latest developments on SCI are explored in more detail in the final chapter of [6].

Related Entries

►Buses and Crossbars
►Cache Coherence
►Clusters
►Distributed-Memory Multiprocessor
►InfiniBand
►Myrinet
►NOW
►Nonuniform Memory Access (NUMA) Machines
►Quadrics
►Shared-Memory Multiprocessors

Bibliographic Notes and Further Reading

SCI was standardized and deployed in the 1990s. David B. Gustavson, one of the main designers, provides first-hand information on SCI's origins, development, and concepts [3, 4, 5]; even the standard document is stimulating material [8, 9]. Gustavson also maintained a Website that actively traced and documented the activities around SCI and related standards [11].

In the late 1990s, the interest in SCI largely disappeared. However, [6] was published as a comprehensive (and somehow concluding) summary of the technology, research and development projects, and achievements around SCI. For many projects and products, the corresponding chapters in [6] are the only documentation still available in the literature; in particular, information on products has disappeared from the Web meanwhile, but is still available to some extent in this book for reference.

This entry of the encyclopedia is a condensed version of the introductory chapter of [6].

Bibliography

1. Brewer T, Astfalk G (1997) The evolution of the HP/Convex Exemplar. In: Proceedings of COMPCON, San Jose
2. Dolphin Interconnect Solutions (2009) http://www.dolphinics.com. Accessed October 2009
3. Gustavson DB, (1992) The Scalable Coherent Interface and related standards projects. IEEE Micro 12(1):10–22

4. Gustavson DB (1994) The many dimensions of scalability. In: Proceedings of the COMPCON Spring, San Francisco
5. Gustavson DB Li Q (1996) The Scalable Coherent Interface (SCI). IEEE Comm Mag 348:52–63
6. Hellwagner H, Reinefeld A (eds) (1999) SCI: Scalable Coherent Interface. Architecture and software for high-performance compute clusters. LNCS vol 1734, Springer, Berlin
7. IEEE Std 1212-1991 (1991) IEEE standard Control and Status Register (CSR) Architecture for microcomputer buses
8. IEEE Std 1596-1992 (1993) IEEE standard for Scalable Coherent Interface (SCI)
9. International Standard ISO/IEC 13961:2000(E) and IEEE Std 1596, 1998 Edition (2000) Information technology - Scalable Coherent Interface (SCI). ISO/IEC and IEEE
10. Lovett T, Clapp R (1996) STiNG: a CC-NUMA computer system for the commercial marketplace. In: Proceedings of the 23rd Internationl Symposium on Computer Architecture (ISCA), May 1996, Philadelphia
11. SCIzzL: the local area memory port, local area multiprocessor, Scalable Coherent Interface and Serial Express users, developers, and manufacturers association. http://www.scizzl.com. Accessed October 2009
12. Scott S (1996) The GigaRing channel. IEEE Micro 16(1):27–34

Semantic Independence

Martin Fränzle[1], Christian Lengauer[2]
[1]Carl von Ossietzky Universität, Oldenburg, Germany
[2]University of Passau, Passau, Germany

Definition

Two program fragments are semantically independent if their computations are mutually irrelevant. A consequence for execution is that they can be run on separate processors in parallel, without any synchronization or data exchange. This entry describes a relational characterization of semantic independence for imperative programs.

Discussion

Introduction

The most crucial analysis for the discovery of parallelism in a sequential, imperative program is the dependence analysis. This analysis tells which program operations must definitely be executed in the order prescribed by the source code (see the entries on dependences and dependence analysis). The conclusion is that all others can be executed in either order or in parallel. The converse problem is that of discovering independence. Program fragments recognized as independent may be executed in either order or in parallel, all others must be executed in the order prescribed by the source code.

The search for dependences is at the basis of an automatic program parallelizer (see the entries on autoparallelization and on the polyhedron model). The determination of the independence of two program fragments is a software engineering technique that can help to map a specific application program to a specific parallel or distributed architecture or it can tell something specific about the information flow in the overall program.

While the search for dependences or independences has to be based on some form of analysis of the program text, the criteria underlying such an – in any Turing-complete programming language necessarily incomplete – analysis should be semantic. The reason is that the syntactic form of the source program will make certain (in)dependences easy to recognize and will occlude others. It is therefore much wiser to define independence in a model which does not entail such an arbitrary bias. An appropriate model for the discovery of independence between fragments of an imperative program is the relation between its input states and its output states.

Central Idea and Program Example

Program examples adhere to the following notational conventions:

- The value sets of variables are specified by means of a declaration of the form var $x, y, \ldots : \{0, \ldots, n\}$, for $n \geq 1$ (that is, they are finite).
- If the value set is $\{0,1\}$, Boolean operators are used with their usual meaning, where the number 0 is identified with false and 1 with true.
- Boolean operators bind more tightly than arithmetic ones.

The initial paper on semantic independence [2] contains a number of illustrative examples. The most complex one, Example 4, shall serve here as a guide:

$$\begin{aligned} &\text{var } x, y, z : \{0,1\}; \\ &\alpha_1 : \quad (x, y) := (x \wedge z, y \wedge \neg z) \\ &\alpha_2 : \quad (x, y) := (x \wedge \neg z, y \wedge z) \end{aligned}$$

The question is whether the effect of these two distinct multiple assigments can be achieved by two independent operations which could be executed mutually in parallel. At first glance, it does not seem possible, since x and y are subject to update and are both shared by both statements. However, this is due to the particular syntactic form of the program. One should not judge independence by looking at the program text but by looking at a semantic model.

One suitable model is a graph of program state transitions. The graph for the program above is depicted in Fig. 1. The nodes are program states. A program state consists of the three binary values of variables x, y, and z. State transitions are modelled by arrows between states. Solid arrows depict transitions of statement α_1, dashed arrows transitions of statement α_2.

The question to be answered in order to determine independence on any platform is: is there a transformation of the program's state space which leads to variables that can be accessed independently? It is not sufficient to require updates to be independent but allow reading to be shared, because the goal is to let semantically independent program fragments access disjoint pieces of memory. This obviates a cache coherence protocol in a multiprocessor system and improves the utilization of the cache space. Some special-purpose platforms, for example, systolic arrays, disallow shared reading altogether; see the entry on systolic arrays. Consequently, all accesses – writing and reading – must be independent.

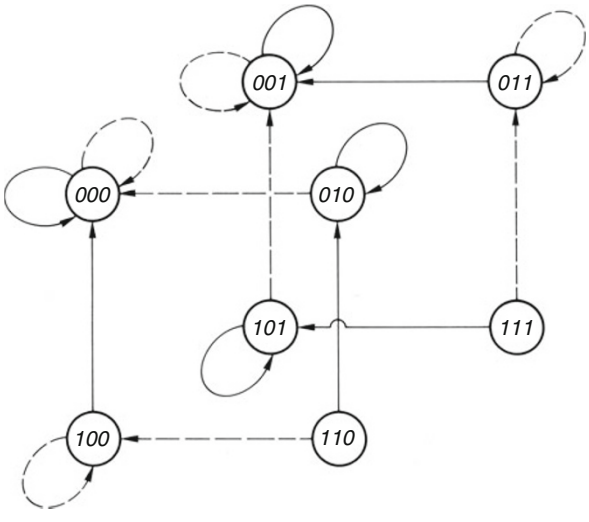
For the example, there is such a transformation of the variables x, y, and z to new variables V and W:

$$\begin{aligned}&\text{var } V : \{0,1\},\ W : \{0,1,2,3\};\\ &V = (x \wedge \neg z) \vee (y \wedge z)\\ &W = 2 * z + (x \wedge z) \vee (y \wedge \neg z)\end{aligned}$$

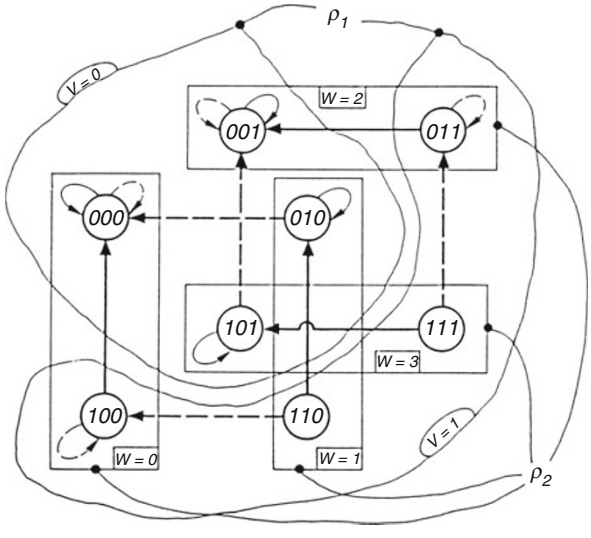
The transformation is depicted in Fig. 2. The figure shows the same state graph as Fig. 1 but imposes two partitionings on the state space. The set ρ_1 of the two amorphously encased partitions yields the two-valued variable V. As denoted in the figure, each partition models one distinct value of the variable. The set ρ_2 of the four rectangularly encased partitions yields the four-valued variable W. The partitionings must – and do – satisfy certain constraints to be made precise below.

The semantically equivalent program that results is:

$$\begin{aligned}&\text{var } V : \{0,1\},\ W : \{0,1,2,3\};\\ &\alpha_1' : \quad V := 0\\ &\alpha_2' : \quad W := 2 * (W \text{div } 2)\end{aligned}$$



Semantic Independence. Fig. 1 State relations of the program example



Semantic Independence. Fig. 2 State relations of the program example, partitioned

As is plainly evident, the new statements α_1' and α_2' access distinct variable spaces and are, thus, trivially independent. In Fig. 2, this property is reflected by the fact that α_1 (and, thus, α_1') transitions are exclusively inside partitions of partitioning ρ_2 and α_2 (and, thus, α_2') transitions exclusively inside partitions of partitioning ρ_1.

The notion of semantic independence illustrated above and formalized below specifies a transformation of the state space that induces two partitionings, each covering one update in the transformed state space, such that both updates can be performed in parallel, without any need for shared access to program variables: they are completely independent in the sense that no information flows between them.

If the parallel updates depend on a common context, that is, they constitute only a part of the overall program, the transformations between the original and the transformed state space must become part of the implementation. Then, the transformation from the original to the transformed space may require multiple read accesses to variables in the original state space and the backtransformation may require multiple read accesses to variables in the transformed state space. Depending on the abilities of the hardware, the code for these transformations may or may not be parallelizable (see the section on disjoint parallelism and its limitations).

Formal Definition

The first objective of this subsection is the independence of two imperative program fragments, as illustrated in the example. The generalization to more than two program fragments as well as to the interior of a single fragment follows subsequently.

Independence Between Two Statements

A transformation of the state space can be modelled as a function

$$\eta : X \to Y$$

from a state space X to a state space Y. In this subsection, consider two program fragments, R and S, which are modelled as relations on an arbitrary (finite, countable, or uncountable) state space X:

$$R \subseteq X \times X \quad \text{and} \quad S \subseteq X \times X.$$

η is supposed to expose binary independence. This leads to a partitioning of state space Y, that is, Y should be the Cartesian product of two nontrivial, that is, neither empty nor singleton state spaces A and B:

$$Y = A \times B.$$

Desired are two new program fragments $R' \subseteq Y \times Y$ and $S' \subseteq Y \times Y$, such that R' operates solely on A and S' operates solely on B. Given two relations

$$r \subseteq A \times A \quad \text{and} \quad s \subseteq B \times B$$

denote their product relation as follows:

$$\begin{aligned} r \otimes s = \{((a,b),(a',b')) \mid (a,a') \in r, (b,b') \in s\} \\ \subseteq (A \times B) \times (A \times B) \end{aligned}$$

Also, write I_M for the identity relation on set M, that is, the relation $\{(m,m) \mid m \in M\}$. Then, the above requirement that R' and S' operate solely on A and B, respectively, amounts to requiring that

$$R' = (r \otimes I_B) \quad \text{and} \quad S' = (I_A \otimes s).$$

for appropriate relations $r \subseteq A \times A$ and $s \subseteq B \times B$.

Furthermore, the composition of R' and S' should exhibit the same input–output behavior on the image $\eta(X) \subseteq Y$ of the transformation η as the composition of R and S on X. Actually, this requirement has been strengthened to a mutual correspondence in order to avoid useless parallelism in which either R' or S' does the full job of both R and S: R' must behave on its part of $\eta(X)$ like R on X and S' like S. With relational composition denoted as left-to-right juxtaposition, that is,

$$r\,s = \{(x,y) \mid \exists z : (x,z) \in r \wedge (z,y) \in s\},$$

the formal requirement is:

$$R\,\eta = \eta\,R' \quad \text{and} \quad S\,\eta = \eta\,S'.$$

In order to guarantee a correspondence between the resulting output state in Y and its equivalent in X, there is one more requirement: η's relational inverse $\eta^{-1} = \{(y,x) \mid y = \eta(x)\}$, when applied to $\eta(X)$, must not incur a loss of information of a poststate generated by program fragments R or S. This is expressed by the two equations:

$$R = \eta\,R'\,\eta^{-1} \quad \text{and} \quad S = \eta\,S'\,\eta^{-1}$$

Figure 3 depicts the previous two requirements as commuting diagrams, on the left for relation R and on the right for relation S. The dashed arrows represent the former and the dashed-dotted arrows the latter requirement. Together they imply:

$$R = R\,\eta\,\eta^{-1} \quad \text{and} \quad S = S\,\eta\,\eta^{-1}.$$

It is neither a consequence that η^{-1} is a function when confined to the images of R or S, nor that it is total on these images. However, if $\eta\,\eta^{-1}$ contains two states (x, y) and (x, y'), then y and y' are R-equivalent (and S-equivalent) in the sense that $(z, y) \in R$ iff $(z, y') \in R$, for each $z \in X$ (and likewise for S), that is, y and y' occur as poststates of exactly the same prestates. If R (or S) is a total function, which is the case for deterministic programs, then η^{-1} restricted to the image of R (or S, respectively) must be a total function.

Putting all requirements together, one arrives at the following definition of semantic independence between two imperative program fragments.

Definition 1 (Semantic independence [4]) *Two relations R and S on the state set X are called* semantically independent *if there are nontrivial (i.e., cardinality > 1) sets A and B, relations r on A and s on B, and a function $\eta : X \to A \times B$ such that*

$$\begin{aligned} &R\,\eta = \eta\,(r \otimes I_B), \quad R\,\eta\,\eta^{-1} = R, \quad \textit{and} \\ &S\,\eta = \eta\,(I_A \otimes s), \quad S\,\eta\,\eta^{-1} = S. \end{aligned} \tag{1}$$

Under the conditions of Eq. 1, R is simulated by r in the sense that

$$R = R\,\eta\,\eta^{-1} = \eta\,(r \otimes I_B)\,\eta^{-1}$$

and, likewise, S by s due to

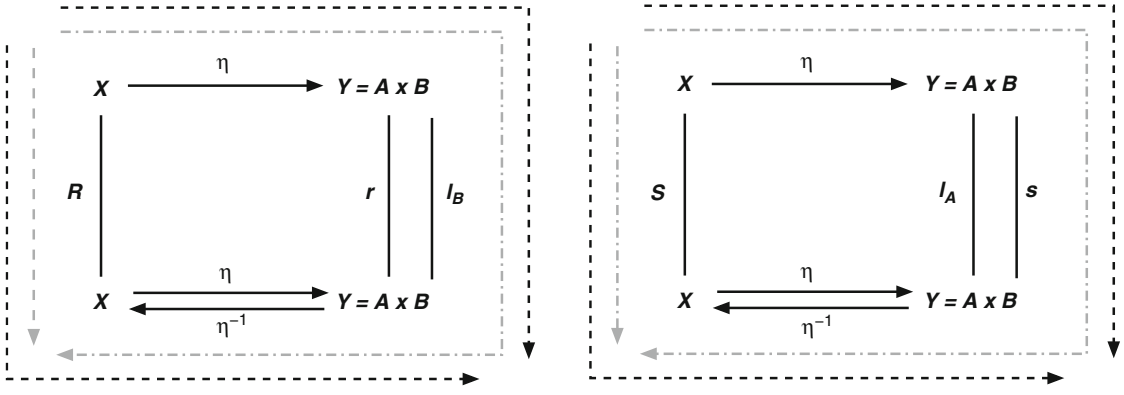
$$S = S\,\eta\,\eta^{-1} = \eta\,(I_A \otimes s)\,\eta^{-1}.$$

That is, after applying state transformation η, an effect equivalent to R can be obtained by executing r, with the result to be extracted by transforming the state space back via η^{-1}. The gain is that r only operates on the first component A of the transformed state space, not interfering with s which, in turn, simulates S on the second component B.

An immediate consequence of Definition 1 is that semantically independent relations commute with respect to relational composition, which models sequential composition in relational semantics. Take again R and S:

$$\begin{aligned} & RS \\ = \;& R\,S\,\eta\,\eta^{-1} \\ = \;& R\,\eta\,(I_A \otimes s)\,\eta^{-1} \\ = \;& \eta\,(r \otimes I_B)(I_A \otimes s)\,\eta^{-1} \\ = \;& \eta\,(I_A \otimes s)\,(r \otimes I_B)\,\eta^{-1} \\ = \;& S\,\eta\,(r \otimes I_B)\,\eta^{-1} \\ = \;& S\,R\,\eta\,\eta^{-1} \\ = \;& S\,R \end{aligned}$$

However, the converse is not true: semantic independence is more discriminative than commutativity. Intuitively, it has to be, since semantic independence



Semantic Independence. Fig. 3 Two requirements on semantic independence

must ensure truly parallel execution, without the risk of interference, while commutativity may hold for interfering program fragments. A case of two program fragments which commute but are not semantically independent was provided as Example 5 in the initial paper [2]:

$$\begin{array}{ll} \text{var } x : \{0,1,2,3\}; & \\ \alpha_1 : & x := (x+1) \mod 4 \\ \alpha_2 : & x := (x+2) \mod 4 \end{array}$$

To prove the dependence, consider the projection of a Cartesian product to its first component:

$$\downarrow_1 : A \times B \to A$$
$$\downarrow_1 ((a,b)) = a$$

Since α_1 corresponds to a transition relation that is a total and surjective function on $X = \{0,1,2,3\}$, $\eta\downarrow_1$ is necessarily a bijection between X and A whenever η satisfies Eq. 1. Consequently, relation S', which implements α_2 in the transformed state space, cannot leave the first component A of the transformed state space $A \times B$ unaffected, that is, it cannot be of the form $(I_A \otimes s)$, since α_2 is not the identity on X.

As pointed out before, the example in Fig. 1 is semantically independent in the sense of Definition 1. Taking $A = \{0,1\}$, that is, as the domain of variable V from the transformation on page 2, and $B = \{0,1,2,3\}$, that is, as the domain of variable W, and taking

$$\eta(x,y,z) = ((x \wedge \neg z) \vee (y \wedge z), 2 * z + (x \wedge z) \vee (y \wedge \neg z)),$$

the two relations

$$s = \{(a,0) \mid a \in A\} \quad \text{and} \quad r = \{(b, 2 * (b \; div \; 2)) \mid b \in B\}$$

satisfy Eq. 1. These two relations s and r are the relational models of the assignments α_1' and α_2' on page 4.

Mutual Independence Between More than Two Fragments

Semantic independence is easily extended to arbitrarily many program fragments. To avoid subscripts, consider only three program fragments given as relations R, S, T on the state set X, which illustrates the general case sufficiently. These relations are called semantically independent if they correspond to relations r, s, t on three nontrivial sets A, B, C, respectively, such that there is a function

$$\eta : X \to A \times B \times C$$

that can be used to represent R, S and T by the product relations

$$r \otimes I_B \otimes I_C, \quad I_A \otimes s \otimes I_C, \quad I_A \otimes I_B \otimes t$$

respectively. For R, this representation has the form

$$R\,\eta = \eta\,(r \otimes I_B \otimes I_C).$$

As before, the requirement is that $R\,\eta\,\eta^{-1} = R$. Relations S and T are represented analogously and are subject to corresponding requirements.

These conditions are considerably stronger than requiring the three pairwise independences of R with $S\,T$, of S with $R\,T$, and of T with $R\,S$. The individual checks of the latter may use up to three different transformations η_1 to η_3 in Eq. 1. This freedom of choice does not capture semantic independence adequately.

Independence Inside a Single Statement

Up to this point, an underlying assumption has been that a limit of the granularity of parallelization is given by the imperative program fragments considered. At the finest possible level of granularity, these are individual assigment statements. If some such assignment statement contains an expression whose evaluation one would like to parallelize, one must break it up into several assignments to intermediate variables and demonstrate their independence.

One can go one step further and ask for the potential parallelism within a single statement. The question then is whether there exists a transformation that permits a slicing of the statement into various independent ones in order to obtain a parallel implementation.

An appropriate condition for semantic independence within a single statement given as a relation $R \subseteq X \times X$ is that X can be embedded into a nontrivial Cartesian product $A \times B$ via a mapping $\eta : X \to A \times B$ such that

$$R\,\eta = \eta\,(r \otimes s), \quad R\,\eta\,\eta^{-1} = R \tag{2}$$

for appropriate relations $r \subseteq A \times A$ and $s \subseteq B \times B$. The generalization to more than two slices is straightforward.

Disjoint Parallelism and Its Limitations

One notable consequence of the presented notion of semantic independence is that, due to accessing disjoint pieces of memory, the concurrent execution of the transformed statements neither requires shared reading nor shared writing. These operations are eliminated

by the transformation or, if the context requires a state space representation in the original format X rather than the transform Y, localized in code preceding the fork of the concurrent statements or following their join. This is illustrated with our initial example, where the shared reading and writing of x and y as well as the shared reading of z present in the original statements

$$\begin{aligned} \alpha_1 &: \quad (x,y) := (x \wedge z, y \wedge \neg z) \\ \alpha_2 &: \quad (x,y) := (x \wedge \neg z, y \wedge z) \end{aligned}$$

disappear in the semantically equivalent form

$$\begin{aligned} \alpha_1' &: \quad V := 0 \\ \alpha_2' &: \quad W := 2 * (W \text{ div } 2) \end{aligned}$$

If the context requires the format of X, one will have to include the transformations η and η^{-1} in the implementation. For the example, this means that the code

$$\begin{aligned} &\text{var } V : \{0,1\},\ W : \{0,1,2,3\}; \\ &\beta_1 : \quad V := (x \wedge \neg z) \vee (y \wedge z) \\ &\beta_2 : \quad W := 2 * z + (x \wedge z) \vee (y \wedge \neg z) \end{aligned}$$

(or any alternative code issuing the same side effects on V and W) has to be inserted before the concurrent statements α_1' and α_2' and that the code

$$\begin{aligned} &\gamma_1 : \quad x := \textsf{if } (W \text{ div } 2) = 1 \textsf{ then } (W \bmod 2) \textsf{ else } V \\ &\gamma_2 : \quad y := \textsf{if } (W \text{ div } 2) = 0 \textsf{ then } (W \bmod 2) \textsf{ else } V \\ &\gamma_3 : \quad z := W \text{ div } 2 \\ &\text{endvar } V,\ W \end{aligned}$$

has to be appended behind their join. The keyword endvar indicates that, after the backtransformation, variables V and W are irrelevant to the remainder of the program. (The backtransformation γ_3 of z from the transformed to the original state space could be omitted, since the original statements do not have a side effect on z, that is, γ_3 turns out not to modify z).

Whether implementations such as β_1 and β_2 of the projections $\eta\downarrow_1$ and $\eta\downarrow_2$, which prepare the state spaces for the two parallel statements to be sparked, can themselves be executed in parallel depends on the ability to perform shared reading. Since these transformations have no side effects on the common state space X and only local ones on A or B, respectively, yet require shared reading on X, they can be executed in parallel if and only if the architecture supports shared reading on X – yet not necessarily on A or B, which need not be accessible to the respective statement executed in parallel. Conversely, a parallel execution of the reverse transformation does, in general, require shared reads on A and B, yet no shared reads or writes of the individual variables spanning X, as illustrated by the statements γ_1, γ_2, and γ_3 above.

Practical Relevance

In the parallelization of algorithms, the mapping η transforming the state space is often fixed, depending on the application, and, thus, left implicit. This leads to fixed parallelization schemes which can be applied uniformly, which makes the search for η and the calculation of the simulating relations r and s from the original state transformers R and S unnecessary. Instead, it suffices to justify the parallelization scheme once and for all. It is worth noting that, for many practical schemes, this can be done via the concept of semantic independence. Examples are the implementation of $*$-LISP on the Connection Machine [5] and Google's closely related MapReduce framework [3], which are based on a very simple transformation, or the use of residue number systems (RNS) in arithmetic hardware and signal processing [8], which exploits a much more elaborate transformation.

In the former two cases, transformation η maps a list of elements to the corresponding tuple of its elements, after which all elements are subjected to the same mapping, say f, in parallel. The implementation of η distributes the list elements to different processors. Then, f is applied independently in parallel to each list element and, finally, by the implementation of η^{-1}, the local results are collected in a new list.

In contrast, RNS, which has gained popularity in signal processing applications during the last two decades, transforms the state space rather intricately following the Chinese remainder theorem of modular arithmetic. In its simplest form, one considers assignments of the form $x := y \odot z$, where the variables are bounded-range, nonnegative integers and the operation $\odot$ can be addition, subtraction, or multiplication. RNS transforms the variables in such assignments to vectors of remainders with co-primal bases as follows. (The numbers of a co-primal base have no common positive factors other than 1.)

Let x, y, z be variables with ranges $\mathbb{N}_{<k}$, where k is a nonzero natural number and $\mathbb{N}_{<k}$ denotes the natural numbers up to $k-1$, and let $f_1, \ldots, f_n \ll k$ be co-primal natural numbers with $\prod_{i=1}^{n} f_i \geq k$. Then, each variable v of x, y, z is transformed to a vector $(v_1, \ldots, v_n)$ of variables such that v_i ranges over $\mathbb{N}_{<f_i}$ and $v_i = v \bmod f_i$. Consequently,

$$\eta(x, y, z) = (x \bmod f_1, y \bmod f_1, z \bmod f_1, \ldots, x \bmod f_n, y \bmod f_n, z \bmod f_n).$$

The independence of the slices of assignment $x := y \odot z$, one for each f_i, follows from the fact that

$$\begin{aligned} & (y \odot z) \bmod f_i \\ = & ((y \bmod f_i) \odot (z \bmod f_i)) \bmod f_i \\ = & (y_i \odot z_i) \bmod f_i \end{aligned}$$

for each of the three operations that can be substituted for $\odot$. This allows a parallel implementation in which the assignment $x := y \odot z$ is replaced by n semantically independent assignments

$$x_i := (y_i \odot z_i) \bmod f_i \quad \text{for} \quad i = 1, \ldots, n.$$

These are executed in parallel without any need for information exchange. A significant speedup results from the reduced bit width of the arithmetic operations involved, which shrinks from $\lceil \log_2 k \rceil$ to $\lceil \log_2 (\max_{i=1}^{n} f_i) \rceil$. If chains of assignments are performed, the speedup can easily amortize the computational effort involved in the transformations η and η^{-1}. Then this parallelization scheme remains attractive even when the data are supplied and expected in standard binary encoding, as is normally the case in signal processing.

Related Entries

►Connection Machine
►Dependences
►Dependence Analysis
►Parallelization, Automatic
►Polyhedron Model
►Systolic Arrays

Bibliographic Notes and Further Reading

Semantic independence is a generalization of the syntactic independence criterion proposed by Arthur J. Bernstein in 1966 [1]. This criterion states that independent program fragments are allowed to read shared variables at any time but must not update any shared variable.

Lengauer introduced a notion of independence based on Hoare logic in the early 1980s [6, 7]. It combines a semantic with a syntactic requirement. The semantic part of the independence relation is called full commutativity and permits an arbitrary interleaving of the program fragments at the statement level, the syntactic part is called non-interference and ensures the absence of shared updates within each statement.

The independence relation based on a program state graph, as proposed by Best and Lengauer in 1989 [2], is entirely semantic. Von Stengel [9] ported the model to universal algebra and Fränzle et al. [4] generalized it shortly thereafter to the notion presented here. Whether a general, automatic, and efficient method of finding η and constructing r and s in any practical programming language exists remains an open question.

Zhang [10] has ported the above notion of semantic independence to security. He adopts Definition 1 as the definition of a set of mutually secure transitions. Transformation η defines the local views of users and r and s are the locally visible effects of the globally secure transitions R and S.

The transformations of the state space, which make independence visible, reflect the same principle as the coordinate transformations of the iteration space in polyhedral loop parallelization. There are other correspondences between the two models, for example, the way in which one exposes parallelism in a single assignment statement. For a more detailed comparison, see the entry on the polyhedron model.

Bibliography

1. Bernstein AJ (1966) Analysis of programs for parallel processing. IEEE Trans Electr Comput EC-15(5):757–763
2. Best E, Lengauer C (1989) Semantic independence. Sci Comput Program 13(1):23–50
3. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM, 51(1):107–113, January 2008
4. Fränzle M, von Stengel B, Wittmüss A (1995) A generalized notion of semantic independence. Info Process Lett 53(1):5–9
5. Hillis WD (1986) The connection machine. MIT Press, Cambridge, MA

6. Lengauer C (1982) A methodology for programming with concurrency: the formalism. Sci Comput Program 2(1):19–52
7. Lengauer C, Hehner ECR (1982) A methodology for programming with concurrency: an informal presentation. Sci Comput Program 2(1):1–18
8. Omondi A, Premkumar B (2007) Residue number systems – theory and implementation, volume 2 of Advances in computer science and engineering. Imperial College Press, United Kingdom
9. von Stengel B (1991) An algebraic characterization of semantic independence. Info Process Lett 39(6):291–296
10. Zhang K (1997) A theory for system security. In: 10th IEEE computer security foundations workshop. IEEE Computer Society Press, June 1997, pp 148–155

Semaphores

►Synchronization

Sequential Consistency

►Memory Models

Server Farm

►Clusters
►Distributed-Memory Multiprocessor

Shared Interconnect

►Buses and Crossbars

Shared Virtual Memory

►Software Distributed Shared Memory

Shared-Medium Network

►Buses and Crossbars

Shared-Memory Multiprocessors

Luis H. Ceze
University of Washington, Seattle, WA, USA

Synonyms

Multiprocessors

Definition

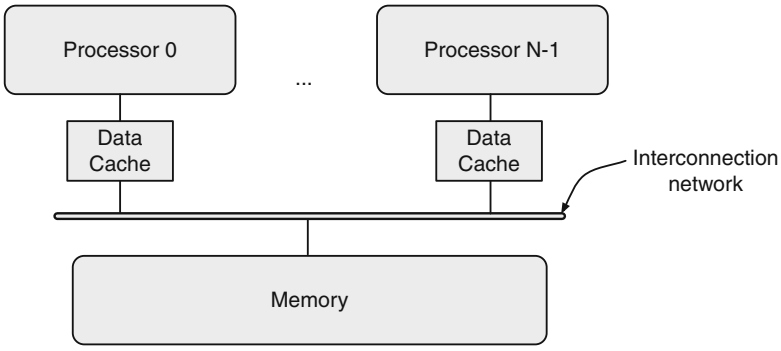
A shared-memory multiprocessor is a computer system composed of multiple independent processors that execute different instruction streams. Using Flynns's classification [1], an SMP is a multiple-instruction multiple-data (MIMD) architecture. The processors share a common memory address space and communicate with each other via memory. A typical shared-memory multiprocessor (Fig. 1) includes some number of processors with local caches, all interconnected with each other and with common memory via an interconnection (e.g., a bus).

Shared-memory multiprocessors can either be symmetric or asymmetric. Symmetric systems implies that all processors that compose the system are identical. Conversely, asymmetric systems have different types of processors sharing memory. Most multicore chips are single-chip symmetric shared-memory multiprocessors [2] (e.g., Intel Core 2 Duo).

Discussion

Communication between processors in shared-memory multiprocessors happens implicitly via read and write operations to common memory address. For example, a data producer processor writes to memory location A, and a consumer processor reads from the same memory location:

```
  P1        P2
------   -----
 ...       ...
A = 5     ...
...       ...
...      tmp = A
...       ...
```



Shared-Memory Multiprocessors. Fig. 1 Conceptual overview of a typical shared-memory multiprocessor

Processor P1 wrote to location A and later processor P2 read from A and therefore read the value "5" into local variable "tmp," written by P1. One way to think about the execution of a parallel program in a shared-memory multiprocessor is as some global interleaving of the memory operations of all threads. This global interleaving of instructions is nondeterministic, as there are no guarantees that if a given program is executed multiple times with the same input will lead to the same interleaving and therefore the same result.

Since communication happens nondeterministic, programs need to explicitly ensure that when communication happens, the data communicated is in a consistent state. This is done via synchronization, e.g., using mutual exclusion to prevent two threads from manipulating the same piece of data simultaneously.

One of the key components of a shared-memory multiprocessor is the cache coherence protocol. The cache coherence protocol ensures that all caches in each processor hold consistent data. The main job of the coherence protocol is to keep track of when cache lines are written to and propagating changes when that happens. Therefore, the granularity of data movement between processors is a cache line. There are many trade-offs in how cache coherence protocols are implemented. For example, whether updates should be propagated as they happen or if they should only be propagated when a remote processor tries to read the corresponding data. Another very important aspect of shared-memory multiprocessors is the memory consistency model, which determines when (what order) memory operations are made visible to processors in the system [3].

For the actual physical memory organization (and the interconnection network), it is common to have multiple partitions physically spread over the processors. This means that each partition is closer to some processors than others. Therefore, some regions of memory have faster access than others. Such organization is called non-uniform memory access, or NUMA. ccNUMA refers to NUMA multiprocessors with private caches per processor and cache coherence.

Given the general slower improvement in single-thread performance, processor manufacturers are now scaling raw compute performance in the forms of more cores (or processors) on a single chip, forming a symmetric shared-memory multiprocessor. Another recent trend points towards using specialized processors to further improver performance and energy efficiency, leading towards asymmetric shared-memory multiprocessor system.

Related Entries

►Cache Coherence
►Cedar Multiprocessor
►Locality of Reference and Parallel Processing
►Memory Models
►Race Conditions
►Race Detection Techniques

Bibliographic Notes and Further Reading

There are many notable shared-memory multiprocessor systems worth reading about, starting from mainframes

in the 60s (e.g., Burroughs B5500), many machines built in academia (e.g., UIUC CEDAR, Stanford Flash, Stanford DASH), and many systems manufactured and sold by Sequent and DEC (now defunct), as well as IBM, Sun, SGI, Fujitsu, HP, among several others.

Bibliography

1. Flynn M (1972) Some computer organizations and their effectiveness. IEEE Trans Comput C-21:948
2. Olukotun et al (1996) The case for a single-chip multiprocessor. In: Proceedings of the 7th international symposium architectural support for programming languages and operating systems (ASPLOS VII), Cambridge, MA, October 1996
3. Adve S, Gharachorloo K (1996) Shared memory consistency models: a tutorial. IEEE Comput Soc 29(12):66–76

SHMEM

►OpenSHMEM - Toward a Unified RMA Model

SIGMA-1

Kei Hiraki
The University of Tokyo, Tokyo, Japan

Synonyms

Dataflow supercomputer

Definition

SIGMA-1 is a large-scale computer based on fine-grain dataflow architecture designed to show feasibility of fine-grain dataflow computer to highly parallel computation over conventional von Neumann computers. SIGMA-1 project was stated on 1984 and the 128 processing element (PE) started working on 1988. SIGMA-1 was design and built at the Electrotechnical Laboratory (ETL for short), Ministry of International Trade and Industry, Japan. SIGMA-1 is still the largest scale dataflow computer so far built and it achieved more 100 Mflops as a maximum measured performance of the total system. As for the language for SIGMA-1, ETL developed Dataflow-C language as a subset of C programming language. Dataflow-C is a single assignment language that can compile C-like source programs to highly parallel executable dataflow machine codes (Fig. 1).

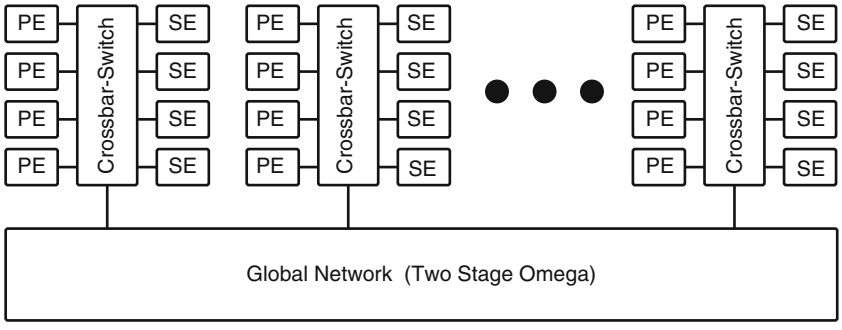
Architecture

Organization

The global organization of the SIGMA-1 is shown in Fig. 2. The SIGMA-1 consists of 128 processing elements, 128 structure elements, 32 local networks, a global network, 16 maintenance microprocessors, a service processor, and a host computer (Figs. 3 and 4). The processing elements and structure elements are divided into 32 groups, each of which consists of four processing elements, four structure elements, and a local network. All groups are connected via the global network. The global network consists of a two-stage omega network with a new adaptive load distribution mechanism. Its transfer rate is 2G bytes per second. A local network is a ten-by-ten crossbar packet switching network, eight ports of which are used for communication between processing elements and structure elements within a group and two to interface the global network. The transfer rate of a local network is 600M bytes per second. The whole system is synchronous and operates under a single clock (Fig. 5).

Figure 6 illustrates a processing element and a structure element. A processing element consists of an input buffer, a matching memory with a waiting matching function, an instruction memory, and a link register file. A processing element executes all the SIGMA-1 instructions except structure-handling instructions such as read, white, allocate, and deallocate used to access the structure memory in a structure element. A processing element works as a two-stage pipeline: the firing stage and the execution stage. In the firing stage, operand matching and instruction fetching are performed simultaneously. Input packets, stored in the input buffer, are sent to the matching memory. The matching memory enables the execution of instructions with two input operands. Successfully matched packets are sent to the execution stage with the instructions fetched from the instruction memory. If the match fails, the packet is stored in the matching memory and the instruction fetched is discarded. Chained hashing

SIGMA-1. Fig. 1 SIGMA-1 system

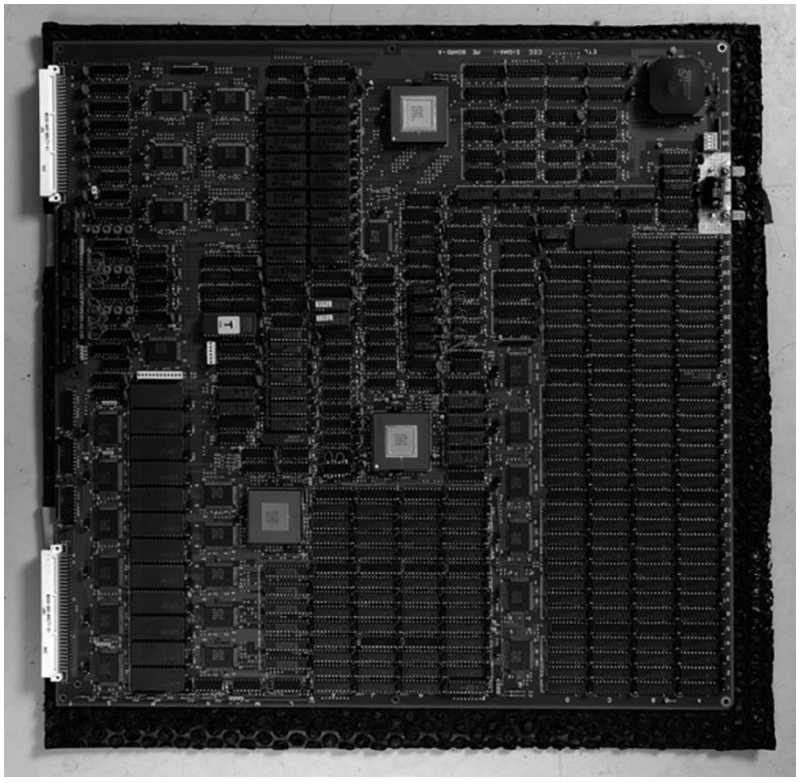


SIGMA-1. Fig. 2 Global architecture of the SIGMA-1

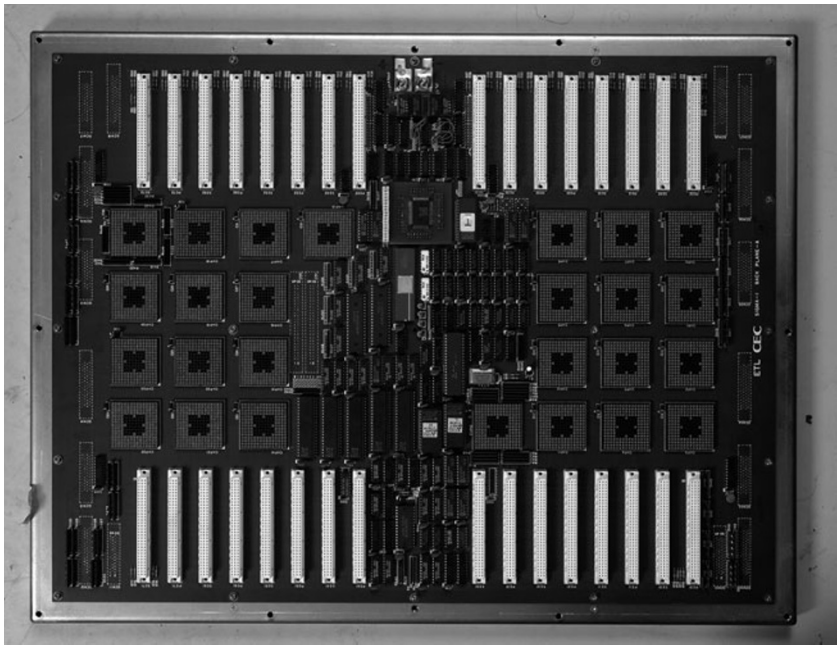
hardware is used to speed up the operand matching operation, where a first-in-first-out queue is attached to each key to enable multiple entries with the same key Table 1.

The link register is used to identify a parallel activity in a program and hold a base address of a procedure. In the execution stage, instructions are executed in parallel with packet identifier generation, where a packet identifier keeps the destination address (next instruction address) as well as a parallel activity identifier (a procedure identifier and an iteration counter). These operations are similar to those in a conventional computer. After combining the result value of the execution unit with the packet identifier, an output packet is produced and sent to the input buffer of the same or another processing element via the local network. If an array is treated as a set of scalar values, the matching memory provides automatic synchronization and space management. However, heavy packet traffic causes a serious bottleneck at the matching memory, since each element of the array must be handled separately. This is the reason additional structure elements

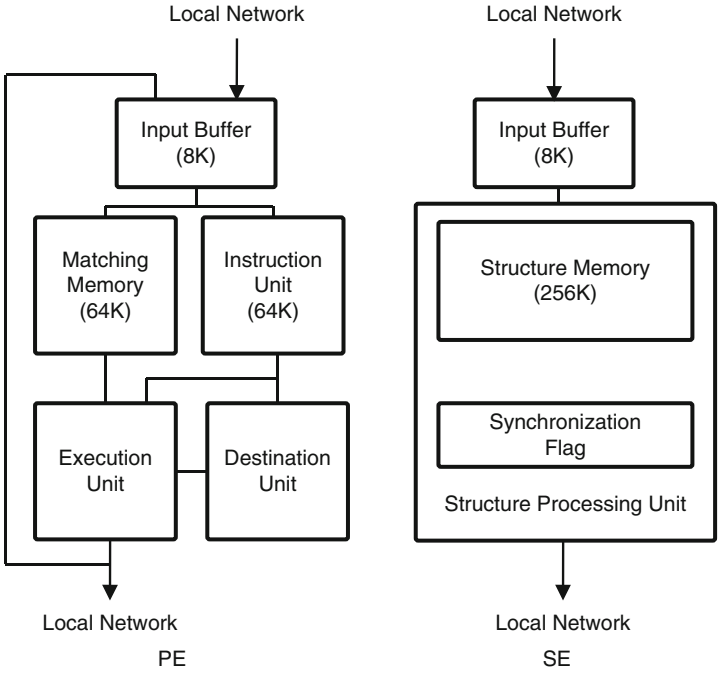
SIGMA-1. Fig. 3 SIGMA-1 Processing element (optional)



SIGMA-1. Fig. 4 SIGMA-1 Structure element (optional)



SIGMA-1. Fig. 5 SIGMA-1 Local network (optional)



SIGMA-1. Fig. 6 Processing element (PE) and SE of the SIGMA-1

are introduced in the SIGMA-1. A structure element consists of an input buffer, a structure memory, two flag logic functions, and a structure controller with a register file. The flag logic function is capable of test and setting all flags in an area arbitrarily specified in a single clock. The structure controller is responsible for waiting queue control and memory management. A structure element handles arrays, which are the most

SIGMA-1. Table 1 Hardware specification of the SIGMA-1

Technology:	CMOS Gate-array, SRAM, DRAM
Clock	10 MHz
Input buffer	8 K words(80bits/word)
Matching memory	64 K words(80bits/word)
Instruction memory	64 K words(40bits/word)
Structure memory	256 K words(40bits/word)
Total amount of memory	326 M Byte
Total no. of gates	19,013,447 gates
Total no. of ICs	91,029
Total no. of PE/SE	128 + 128

important data structure involved in numerical computations. Each structure cell is composed of a 40-bit data (data type and payload data) word and two flag bits used for read and write synchronization. Multiple read before write operations are realized by a waiting queue attached to each cell. The buddy system implemented in microprogramming is used to increase the speed of allocating and deallocating structures.

The architectural features of the SIGMA-1 are designed to achieve high speed by decreasing the parallel processing overheads in dataflow computers. For example, the short pipeline architecture enables quick response to small-sized programs with low parallelism. It was anticipated that the matching memory and the communication network would play a significant role in the system design from the viewpoint of synchronization and data transmission overhead. Besides the newly proposed mechanisms, high-speed and compact gate-array LSI elements have been developed for this purpose.

Testing, debugging, and maintenance are performed by a special purpose parallel architecture [6]. This maintenance architecture uses a set of conventional von Neumann microprocessors, since maintenance operations generally need history sensitivity utilizing side effects.

The architectural features of the SIGMA-1 are summarized as follows:

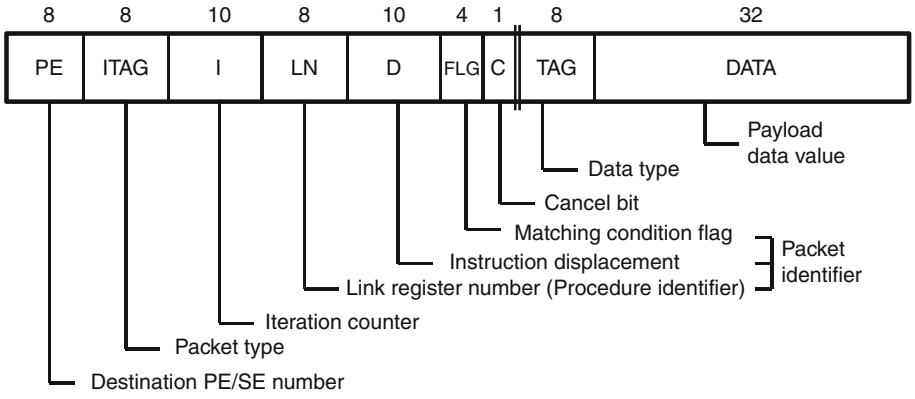
1. A short (two-stage) pipeline in a processing element, reducing the gap between the maximum and average performance
2. Chained hashing hardware for the matching memory, giving efficient and high-speed synchronization
3. An array-oriented structure element, minimizing structure handling overhead
4. A hierarchical network with a dynamic load distribution function, reducing performance degradation caused by load imbalance
5. A maintenance architecture for testing and debugging, providing high-speed input and output operations and performing precise performance measurements

Packet and Instruction Architecture

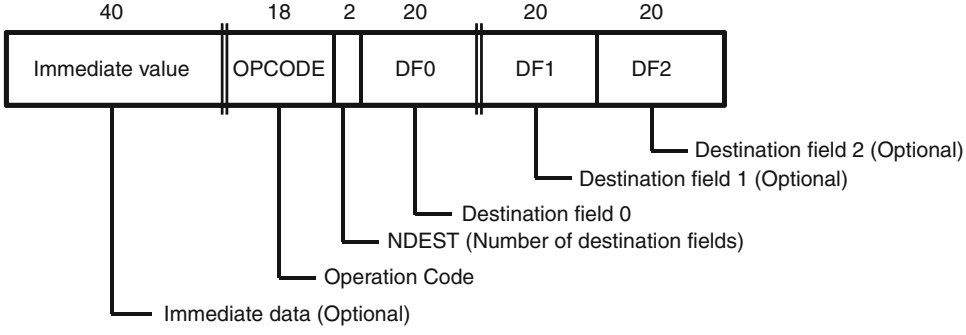
The SIGMA-1 adopts packet communication architecture. Processing elements and structure elements communicate using fixed length packets. Input and output to the host computer and maintenance system are also in packet form. Therefore, hardware initialization and hardware maintenance are carried out in packet form. As shown in Fig. 7, a packet contains a destination processing element number, a packet identifier and tagged data, and miscellaneous control information.

The 89-bit packet is divided into two 40-bit segments, a processing element number field (8 bits), and a cancel bit. A packet is transferred in two consecutive segments; the first segment contains a processing element or structure element number, a destination address in the processing element or structure element (32 bits), and packet type (8 bits). The second segment consists of data (32 bits), data type (8 bits), and one cancel bit. The cancel bit is used for canceling the preceding segment when a malfunction occurs. When the cancel bit is on, the whole packet becomes invalid.

A destination address consists of a procedure identifier (8 bits) for specifying a parallel activity, a relative instruction address (10 bits) within the activity, an iteration counter (10 bits), and some control information determining the firing rule in the matching memory (4 bits). The iteration counter is utilized to implement the loop construct efficiently. As in the ordinary model of dynamic dataflow, the concatenation of the procedure identifier and iteration counter is used to distinguish parallel activities in a program. The types of packets used are: result of instruction, procedure call and return, interrupt handling and system management, structure operation, and system initiation and maintenance for system resources.



SIGMA-1. Fig. 7 Packet format



SIGMA-1. Fig. 8 Instruction format

The minimum length of an instruction is 40 bits (one word) as illustrated in Fig. 8. The first 20 bits indicate the operation to be performed (18 bits) and the number of destination address fields (2 bits). The next 20 bits indicate the destination address of the result, which does not contain dynamically assigned information such as a procedure identifier and iteration counter. An immediate data operand can be located at the header part of each instruction. The maximum number of destination address fields is three, using two words. When an instruction contains an immediate data (constant) operand, another word is required.

The objective of the SIGMA-1 instruction set design is to execute efficiently application programs that cannot be efficiently executed on a vector-type supercomputer or a parallel von Neumann computer. Low efficiency in these program executions is caused by frequent procedure calls and returns, large amount of scalar arithmetic operations, and wide fluctuations of parallelism. Fine-grain parallelism has the advantage over coarse-grain parallelism to overcome this inefficiency. As a result, 200 instructions on a processing element and 97 instructions on a structure element have been implemented (Fig. 9).

Software

The DataFlow C (DFC) compiler is implemented using the language development tools of the UNIX operating system. The emphasis was not on efficiency but on ease of implementation. Efficiency and compactness are still to be considered. Basically, DFC is a subset of the C language with a single assignment rule. Therefore, assignment to a variable is generally allowed only once in each procedure. However, the loop construct, realized by the *for* statement, is an exception. A non-single assignment expression can be written as the third argument of the *for* statement as shown in the following summation program. The argument *n* of the main program is given by a trigger packet invoking the program.

SIGMA-1. Fig. 9 SIGMA-1 PEs and SEs (optional)

```
#define N 10
main(n)
int n;
{int i; float a[N], retval, sigma();
for(i=0;i<5;i=I + 1) a[i] = i;
retval=sigma(n, a); print(retval);}

float sigma(n, a)
int n; float a[];
{int i; float s;
for (i=0, s=0; i<n; s=s+a[i],i=i+1);
return(s);}
```

DFC may have multiple return values as an extension to the C language. DFC programs without multiple return values can be compiled by a C compiler. The advantage is that DFC programs can be verified by a C compiler on a host computer. The DFC compiler comprises approximately 7,000 steps in C. A nested structure of *for* or *if* statements is not implemented and is inhibited. The DFC compiler generates a macro-assembly language program. This corresponds to a dataflow graph with no restriction on the number of arcs.

System Performance

The whole system operates synchronously at a 10 MHz clock frequency. The execution time of an instruction is determined by the maximum execution time of the two pipeline stages. Single operand instructions are executed every two cycles and two operand instructions every three cycles. The firing stage normally takes three cycles. Since non-division arithmetic operations take at most two cycles, the execution stage completes most of the instruction execution in two cycles. This consideration implies that the maximum speed of a processing element is about 5 MIPS and 3 MFLOPS. For a structure element, each read and write instruction is carried out in two cycles. Since instructions for allocating and deallocating structures are implemented by a microprogrammed buddy system, the execution times estimated are between 10 and 340 cycles. The network transfers a packet every two cycles. Therefore, structure elements can utilize maximum performance of the network.

Performance is measured in terms of program execution time rather than raw machine cycles. The measurement study proved that there is no difference in computing ability of a single processing element between dataflow architecture and von Neumann architecture, in case of constructing them using the same device technology.

1. A speed degradation of 30% occurs when structure elements are adopted. This is due to the extra index and address calculation for structure handling.
2. The single processing element performance is almost constant over the vector length, because of the short pipeline architecture.
3. The speedup ratio for a single processing element to four processing element organization is 3.9 without structure elements and 3.5 with structure elements.

This performance evaluation shows that the average performance of the 128 processing element organization is more than 100 MFLOPS. For example, SIGMA-1 compute matrix multiplies at 117 Mflops.

Discussion

Table 2 shows development history of major MIMD parallel computers and dataflow computers.

SIGMA-1. Table 2 History of dataflow parallel processors

Year[a]	System name	#PE	Word length	Architecture
1976	DDM1	1 PE	4	Static dataflow
1978	TI DDP	4 PE	16	Static dataflow
1979	LAU	32 PE	32	Static dataflow
1981	Manchester dataflow machine	1 PE	24	Dynamic dataflow
1982	OKI DDDP	4 PE	16	Dynamic dataflow
1982	NTT Eddy	16 PE	16	Simulated dynamic dataflow
1984	NEC NEDIPS	8 PE	32	Static dataflow
1983	Q-p	1 PE	16	Static dataflow, Asyncronous
1984	**SIGMA-1 Prototype**	**1PE**	**32**	**Dynamic dataflow**
1988	**SIGMA-1**	**128 PE**	**32**	**Dynamic dataflow**
1988	MIT monsoon prototype	1 PE	64	Dynamic dataflow
1990	ETL EM4	80 PE	32	Dynamic/Hybrid dataflow
1991	MIT monsoon	8 PE	64	Dynamic dataflow
1996	ETL EM-X	80 PE	32	Dynamic/Hybrid dataflow

[a]Year denote the date start working in its full configuration

As shown in Table 2, SI-1 is the largest dataflow computer so far built.

Illustrations, diagrams, and code examples.

Bibliography

1. Hiraki K, Nishida K, Sekiguchi S, Shimada T (1986) Maintenance architecture and its LSI implementation of a dataflow computer with a large number of processors. Proceedings of International Conference on Parallel Processing, IEEE Computer Society, University Park, pp 584–591
2. Hiraki K, Sekiguchi S, Shimada T (1987) System architecture of a dataflow supercomputer. Proceedings of 1987 IEEE Region 10 Conference, IEEE computer Society, Los Alamitos, pp 1044–1049
3. Hiraki K, Shimada T, Nishida K (1984) A hardware design of the SIGMA-1—A data flow computer for scientific computations. Proceedings of 1984 International Conference on Parallel Processing, IEEE Computer Society, Los Alamitos, pp 524–531
4. Hiraki K, Shimada T, Sekiguchi S (1993) Empirical study of latency hiding on a fine-grain parallel processor. Proceedings of International Conference on Supercomputing, ACM, Tokyo, pp 220–229
5. Sekiguchi S, Shimada T, Hiraki K (1991) Sequential description and parallel execution language DFCII dataflow supercomputers. Proceedings of International Conference on Supercomputing, ACM, New York, pp 57–66
6. Shimada T, Hiraki K, Sekiguchi S, Nishida K (1986) Evaluation of a single processor of a prototype data flow computer SIGMA-1 for scientific computations. Proceedings of 12th Annual International Symposium on Computer Architecture, IEEE Computer Society, Chicago, pp 226–234

SIMD (Single Instruction, Multiple Data) Machines

►Cray Vector Computers
►Floating Point Systems FPS-120B and Derivatives
►Flynn's Taxonomy
►Fujitsu Vector Computers
►Illiac IV
►MasPar
►MPP
►NEC SX Series Vector Computers
►Vector Extensions, Instruction-Set Architecture (ISA)

SIMD Extensions

►Vector Extensions, Instruction-Set Architecture (ISA)

SIMD ISA

►Vector Extensions, Instruction-Set Architecture (ISA)

Single System Image

Rolf Riesen[1], Arthur B. Maccabe[2]
[1]IBM Research, Dublin, Ireland
[2]Oak Ridge National Laboratory, Oak Ridge, TN, USA

Synonyms

Distributed process management

Definition

A Single System Image (SSI) is an abstraction that provides the illusion that a multicomputer or cluster is a single machine. There are individual instances of the Operating Systems (OSs) running on each node of a multicomputer, processes working together are spread across multiple nodes, and files may reside on multiple disks. An SSI provides a unified view of this collection to users, programmers, and system administrators. This unification makes a system easier to use and more efficient to manage.

Discussion

Introduction

Multicomputers consist of nodes, each with its own memory, CPUs, and a network interface. In the case of clusters, each node is a stand-alone computer made of commodity, off-the-shelf parts. Instead of viewing this collection of computers as individual systems, it is easier and more economical if users, programmers, and system administrators can treat the collection as a single machine. For example, users want to submit a single job to the system, even if it consists of multiple processes working in parallel.

The task of an SSI is to provide the illusion of a single machine under the control of the users and system administrators. The SSI is a layer of abstraction that provides functions and utilities to use, program, and manage the system. An SSI consists of multiple pieces. Some of them are implemented as individual utilities, other pieces are found in system libraries, there are additional processes running to hold up the illusion, OSs are modified, and in some systems hardware enables some of the abstraction. No SSI provides a complete illusion. There are always ways to circumvent the abstraction, and sometimes it is necessary to interact with specific components of the system directly. Many SSI in daily use provide a limited abstraction that is enough to make use of the system practical, but do not add too much overhead or impede scalability.

An OS consists of a kernel, such as Linux, and a set of utilities and tools, such as a graphical user interface and tools to manipulate files. On a single computer, for example, a desktop workstation, the OS manages resources, such as CPUs, memory, disks, and other peripherals. It does that by providing a set of abstractions such as processes and files that allow users to interact with the system in a consistent manner independent of the specific hardware used in that system. For everyday tasks, it should not matter what particular CPU is installed, or whether files reside on a local or a network-attached disk drive. The goal of an SSI is to provide a similar experience to users of a parallel system.

That is not that difficult to achieve in a Symmetric Multi-Processor (SMP) where all memory is shared and all CPUs have, more or less, equally fast access to it. The OS for an SMP has additional features to manage the multiple CPUs, but the view it presents to a user is mostly the same that a user of a single-CPU system sees. For example, when launching an application, a user of an SMP does not need to specify which CPU should run it. The OS will select the least busy CPU and schedules the process to run there. There are additional functions that an SMP OS provides, for example, to synchronize processes, pin them to specific CPUs, and allow them to share memory. Nevertheless, the abstraction is still that of a single, whole machine.

In contrast, in a distributed memory system, such as a large-scale supercomputer or a cluster, each node runs its own copy of the OS. It is possible that not all nodes have the same capabilities; some may have access to a Graphic Processing Unit (GPU), the CPUs and memory sizes may be different, and some nodes may have connections to wide-area networks that are not available to all nodes in the system. The task of an SSI is to unify these individual components and present them to the user as if they were a single computer. This is difficult and sometimes not desired. Accessing data in the memory or disk of a remote node is much more time consuming than accessing data locally. It therefore makes sense for a user to specify that a process be run on the node where the data currently resides. It also makes

sense to give users the ability to specify that a process be run on a node that meets minimum memory requirements or has a certain type of CPU or GPU installed. On the other hand, for many tasks, or in a homogeneous system, the SSI should hide the complexity of the underlying system and present a view that lets users interact with it as if it were a large SMP.

An SSI is meant to make interaction with a parallel system easier and more efficient for three different groups of users: Application users who use the system to run applications such as a database or a simulation, programmers who create applications for parallel systems, and administrators who manage these systems. Each group has its own requirements for, and uses of, an SSI; and the view an SSI presents to each group is different. Of course, there is some overlap since there are tasks that need to be done by members of more than one group. The following sections discuss an SSI from these three different viewpoints. Functions that overlap are discussed in earlier viewpoints.

Application User's View

A user of a parallel system uses it for an application such as computing the path of an asteroid, the weekly payroll, or maintaining the checking accounts of a bank. This type of user does this by running an application which provides the necessary functionality. In some cases, like the banking example, the user never directly interacts with the system. Those users only see the application interface, for example, the banking application that presents information about customers and their account balances. Parallel systems used in science and engineering cater to users who each have their own applications to solve their particular scientific problems. These users do interact with the system and an SSI assists them in making these interactions smooth and reliable.

Large systems that are shared among many users usually dedicate several nodes as login nodes. Smaller clusters sometimes let users log into any of the nodes, though this is less efficient for running parallel applications. An SSI gathers these login nodes under the umbrella of a single address. Users remotely log into that address and the SSI chooses the least busy node as the actual login node.

Once logged in, users need the ability to launch serial or parallel jobs (applications) and control them. An SSI chooses the nodes to run the individual processes on and provides utilities to the user to obtain status information, stop, suspend, and resume jobs.

While an application is running, users need to interact with it. The SSI transparently directs user input to the application, independent of which node it is running on. Similarly, output from the application is funneled (if it comes from multiple processes) back to the user. This is the same as on any computing system. However, in a distributed system, the SSI needs to provide that functionality even if a process migrates to another node.

Input and output data for an application is often stored in files. Users must have the ability to see these files and manipulate them, for example, copy, move, and delete them. This requires a shared file system, since the application must be able to access the same files. Since the user, the application, and the files themselves may all reside on different nodes, a distributed file system is needed that presents a single root to all nodes in the system. That means the full path name of a file, that is, the hierarchy of directories (folders) above it and the file name itself, must be the same when viewed from any node.

Peripherals should be visible as if they were connected to the local node. The SSI must provide a view of that peripheral and transfer data to and from it, if the device is not local.

Programmer's View

Programmers of parallel applications have many of the same requirements as application users. They also need to launch applications, to test them for example, and need access to the file system. The same is true for writing and compiling applications.

For debugging, the SSI may need to route requests from the debugger to processes on another node to suspend them and to inspect their memory and registers. If this functionality is embedded in the debugger itself, then this is no longer a requirement for the SSI, but it does require services that are part of the OS.

Programmers do have additional requirements that an SSI provides. The programs they write need to access low-level system services provided by the OS. That is done through system calls that allow applications to trap into an OS kernel to obtain a specific service. Typical services provided are calls to open a file for reading and

writing, creating and starting a new process, and interact through Inter-Process Communication (IPC) with another process. In a distributed system, these functions become more complex.

A process creation request in a desktop system is an operation carried out by the local OS. In a distributed system managed by an SSI, process creation may mean interaction with the OS of another node. If the original process and the new process need to communicate, then data has to move between nodes instead of just being copied locally into another memory location. For an application that is not aware of the distributed nature of the system it is running on, the SSI must provide this functionality so that it is transparent to the application.

Many applications consist of multiple processes that work together by sharing memory locations. On a single system, that is an efficient way to move data from one process to another and lets multiple processes read the same data. If these processes, in a distributed system, do not have access to the same memory anymore, this becomes more difficult and expensive, than it was before. Some SSI provide the illusion of a shared memory space. This is enabled by specialized hardware or through Distributed Shared Memory (DSM) implemented in software.

Administrator's View

An administrator of a parallel system is responsible for managing user accounts, keeping system software up to date, replacing failed components, and in general have the system available for its intended use. Many of the administrators tasks are similar to the ones a user performs, for example, manipulating files, but with a higher privilege level.

In addition, an SSI should provide a single administrative domain. That means once the system administrator has logged in and has been authenticated by the system, the administrator should be able to perform all tasks for the entire system from that node. The SSI enables killing, suspending, and resuming any process, independent of its location. A system administrator further has the ability to trigger the migration of a process. This is sometimes necessary to vacate a portion of the system so it can be taken down for maintenance or repair. To do this, it must be possible to "down" network links, disks, CPUs, and whole nodes. Once down, they wont be allocated and used again until they have been fixed or are done with maintenance.

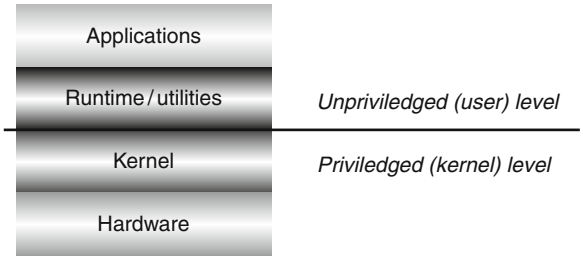
System administrators also need the ability to monitor nodes and links to troubleshoot problems or look for early signs of failures. An SSI can help with these tasks by providing utilities to search, filter, and rotate logs as an aggregate so an administrator does not need to login or access each node individually.

Often, each node has its own copy of system files, libraries, and the OS installed on a local disk. This increases performance and enhances scalability. An SSI provides functions to keep the versions of all these files synchronized and allows a system administrator to update the system software on all nodes with a single command.

Implementation of an SSI

The previous section looked at the functionality an SSI provides to users, programmers, and system administrators. This section provides a little bit more detail on how these abstractions are implemented and at what level of the system hierarchy. Figure 1 shows the layers of a system that contain SSI functions. User level is a CPU mode of operation where hardware prevents certain resource accesses. Applications and utilities run at this level. At the kernel level, all (local) resources are fully accessible. The OS kernel runs at this privilege level and manages resources on behalf of the applications running above it.

At the bottom of the hierarchy is the hardware. Some SSI features can only be implemented at that level. If the necessary hardware is missing, that feature will not be available or only very inefficiently in the form of a software layer. Shared memory on a system without hardware support for it is such an example.



Single System Image. Fig. 1 Privilege and abstraction layers in a typical system

One level above the hardware resides the OS kernel. Some SSI functionality, for example, process migration, needs to be implemented at that level. Above the OS is the runtime layer and system utilities. This layer is sometimes called middle-ware. A batch and job control system which determines when and were (on which nodes) a job should run can be found at this level. Of course, utilities at that layer need the support of the OS and ultimately the underlying hardware to do their job.

Above that are the applications that make use of the system. Some SSI functionality, for example, application-level checkpoint and restart can be implemented at that level, for example, in a library. Table 1 shows various SSI features and at what level they are implemented. Some can be implemented at more than one level. Only the levels that have been traditionally used, and where a feature is most efficient, are marked.

Shared Memory

Sharing memory that is distributed among several nodes requires hardware support to work well. Implementations in software of so-called Distributed Shared Memory (DSM) systems have been done, but they all suffer from poor performance and scalability.

Single System Image. Table 1 Features of an SSI and at what layer of the system hierarchy they are most commonly implemented

Feature	Implementation layer			
	HW	Kernel	Runtime	App.
Shared memory	•			
Up/down sys components	•	•	•	
Debugger support		•	•	
Process migration		•		
Stdin/stdout		•		
File system		•		
System calls		•		
Checkpoint/restart		•	•	•
Batch system			•	
Login load leveling			•	
Sys log management			•	
SW maintenance			•	

Management of System Components

In larger systems, it is desirable to mark components as being down so they wont be used anymore. Marking a node down, for example, should prevent the job scheduling system from allocating that node. In some systems, it is possible to mark links as being down and have the system route messages around that link. This is useful for system maintenance. Components that need repair wont be used until the system is brought down during the regular maintenance period, when all failed components can be repaired at the same time. Depending on the sophistication of this management feature, it requires hardware and kernel support. Simply telling the node allocator to avoid certain nodes can be done at user level in the runtime system.

Debugger Support

A debugger is a user-level program like any other application. However, in order to signal and control processes, and to inspect their memory and registers, the debugger needs support from the OS. Some SSI modify kernels such that standard Unix facilities, for example, signals and process control, are extended to any process in the system; independent of which node they are currently running on. In that case, the debugger does not need to change, although adding support to debug multiple processes at the same time makes it a more useful tool.

If the OS is not capable of providing these services, they can be implemented in the debugger with the help of runtime system features such as daemons and remote procedure calls (RPC).

Process Migration

Migrating a process to another node is useful for load balancing and as a precautionary measure before a node is marked down. Migrating a process requires that all data structures, such as the Process Control Block (PCB), which the OS maintains for this process, migrate in addition to the code and user-level data structures the process has created. A completely transparent migration also requires that file handles to open files and those used for IPC are migrated as well. That, in return, requires that some information remains in the local OS about where the process has migrated to. (Or that all connections to the migrating process are updated before the migration.) For example, another

process on the local node which was communicating with the migrated process via IPC will continue to do so. When data arrives for the migrated process, the OS needs to forward the data to the new location. If a process migrates multiple times, forwarding becomes more expensive and convoluted.

Stdin/Stdout

The default channels for input to and output from a process under UNIX are called stdin and stdout. If a user is logged into one node and starts a process on another, then the OS transfers data between these two nodes on behalf of the user. If an application consists of multiple processes, the OS ensures that output from all of them gets funneled through the stdout channel to the user. Data from the stdin channel usually does not get replicated and only the first process that reads it, will receive a copy. The OS must maintain these data channels even if processes migrate to other nodes, otherwise the illusion of a single system image would be broken.

File System

In a system managed by an SSI, all processes should see the same directory (folder) and file hierarchy. This feature, called shared root, is what is available on a desktop system where multiple processes can access files on the local disk. In a parallel system, files are often distributed among multiple disks to enhance performance and scalability. Distributing files like that complicates things, since metadata (information about files and directories) may not be located on the same device as the files themselves. Yet, metadata must be kept up-to-date to preserve file semantics. For example, two processes appending data to the end of a file must not overwrite each other's data. That requires that the file pointer always points to the true end of the file.

Some files which the OS frequently accesses should be local to the node. However, some files, such as/etc/passwd, should be the same on all nodes and a system administrator should have the ability to update only one of them and then propagate the changes to all copies. On the other hand, some (pseudo) files that describe local devices for example, need to be unique on each node. If a process migrates and had one of these files open, it must continue to interact with the device on the original node, even though a file with the same name exists on the new node as well. Yet for other files, temporary files for example, it may be desirable that they are only visible locally. File systems that can handle all these cases are complex and need to be tightly integrated with the OS.

System Calls

System calls are functions an application uses to interact with the OS. Calls for opening, closing, reading, and writing a file are typical. So are calls to create and control other processes. An SSI that wants to maintain these calls unchanged in a distributed system requires changes to the OS kernel. For example, process creation may not be strictly local anymore, if the system tries to balance load by creating new processes on less busy nodes.

Checkpoint/Restart

Long-running applications often do not finish, due to an interruption by a faulty component or a software problem. This is especially true for parallel applications, where it is more likely that at least one of the many nodes it runs on will fail. To combat this problem, an application occasionally writes intermediate data and its current state to disk. When a failure occurs, the application is restarted on a different node, or on the same node after it has been rebooted or repaired. The application then reads the checkpoint data written earlier and continues its computation.

This can be done by the application itself, in the runtime system, or in the OS kernel. The latter has the nice property that application writers do not need to worry about it, but it has the potential for waste, since the kernel cannot know what data the application will need to restart. Therefore, it has to checkpoint the complete application state and all of its data. Checkpoint/restart needs to be integrated with the SSI system, even if checkpointing is done at the application level.

Batch System

Many parallel systems use a batch scheduler to increase throughput of jobs through the system and to allow it to continue processing unattended, for example, at night or on weekends. Users submit scripts that describe the application they want to run, how much time they require, and how many nodes the application needs. The batch scheduler uses the job priority and knowledge about available nodes to select and run the jobs currently enqueued for processing.

A batch system is one of the most common SSI components in a cluster. It allows the user to treat the collection of machines as a single system and use a single command to submit jobs to it. A batch system provides additional commands to view status information about the currently enqueued and running jobs and to manage them. All these commands treat jobs as if they were a single process on the local system.

Login Load Leveling

Parallel systems usually have more than one node where users can log in. In a partitioned system, there is a service partition comprised of the nodes dedicated to user login and common tasks such as editing files and compiling applications. Parallel jobs are then run on nodes that have been assigned to the compute partition. Often, nodes in the service partition are of a different type; they may have more memory or local disks attached. Smaller clusters may not use partitioning, though that is less efficient. In either case, the SSI should present a single address to the outside world. When a user logs in at that address, the SSI picks one of the service nodes for that user to log in. The choice is usually made based on the current work load of each node in the service partition. As far as the user is concerned, there is only one login node.

System Log Management

A lot of information about the current state of the system is stored in log files that are unique to each node of a parallel system. System administrators consult these logs to diagnose problems and to detect and prevent faults before they occur (e.g., warnings about the rising temperature inside a CPU). An SSI should provide functions to aggregate these individual files and let system administrators search through, purge, or archive them with a single command.

Software Maintenance

The software versions of the system utilities, libraries, and applications installed on each node must be the same system-wide or at least be compatible with each other. As with system log management, an SSI should provide functions to system administrators that allow them to upgrade and check software packages on multiple nodes with a single command.

Imperfect SSI

The previous sections listed the features an SSI must provide and a glance at what implementing them entails. Many of these features are independent components and useful even when none of the other components are available. In fact, no SSI running on more than a couple hundred of nodes implements all features described thus far. The reason for that is scalability. Some features work well when used on eight nodes, but become unusably slow when extended much beyond that number.

Among the most costly features described earlier is process migration, especially if an SSI tries to achieve complete transparency. Once a process starts interacting with other processes on a node and obtains file handles to node-local resources, it becomes difficult to migrate that process. After migration, a forwarding mechanism must be in place to route information to the new location of the migrated process. If the original node retains information about the new location of the migrated process, a failure of that node will disrupt the flow of information. If no information is kept at the original location, then all processes that are communicating with the migrated process must be told about the new location. If migration is frequent, and interaction among processes is high, this becomes very costly. Furthermore, it does not always work, for example, if the migrated process was using a peripheral that is only available on the original node. Another high-cost SSI item is shared memory. Even when supported by hardware, the illusion of a single, large memory starts to fall apart as the number of nodes (CPUs) increases.

Not all is lost, though. Some systems deploy additional hardware to assist with some SSI features. For example, some systems have an additional network and disks dedicated to system software maintenance. Furthermore, complete transparency is not always required, and users are willing to use different commands and utilities when dealing with parallel processes, in return for better performance and scalability. For example, using a command such as `qsub` to submit a parallel job does not seem a high burden compared to the transparent method of just naming the application on the command line to start it. The UNIX command `ps` is used to obtain process status information. In a transparent SSI system, it would return that information even if the process is not local, or return information

about many processes, if they are part of a parallel application.

In contrast, users often only want to know whether their job is running and whether it is making progress. The batch system has a command that can answer at least the former question and a `ps` output for hundreds of processes scattered throughout the system would not provide any additional useful information.

Therefore, even a partial SSI can be of benefit. Application users benefit most from a batch scheduling system. This also assists system administrators who also need a way to efficiently manage the hardware and software of the system. Most other functionality, especially less scalable features, are not strictly necessary to make a parallel system useful. That may mean additional utilities that expose the distributed aspects of the system and modifications to applications to allow them to work in parallel and with greater efficiency.

Future Directions

Desktop systems with two or four CPU cores are common now. High-performance systems will be the first to receive CPUs with tens of cores, but those CPUs will soon be common. Desktop OSs will adapt to these systems and provide SSI features to manage and use the cores in a single system. Users will expect to see these features in parallel systems as well, which will drive the development of SSI features and integration into parallel OSs.

At the very high-end, exascale systems are on the horizon. Implementing SSI for these systems will be a challenge due to the size of these systems. Displaying the status of a hundred-thousand nodes in a readable fashion without endless scrollbars is only feasible with intelligent software that zooms in on interesting events and features of the system. Commercial tools for applications that span that many nodes do not exist yet. Simple tools have a better chance at scaling to these future systems, but managing and debugging such large applications and systems will remain a challenge.

Related Entries

►Checkpointing
►Clusters
►Fault Tolerance
►Operating System Strategies
►Scalability

Bibliographic Notes and Further Reading

Overviews of SSI are presented in [3], Chapter 11 of [13], and [8]. The descriptions of actual systems are out of date or the systems are not in use anymore. Nevertheless, the fundamental features and issues described are still valid.

Many papers discuss specific aspects of SSI. For example, [5] and [4] discuss partitioning and present job launch mechanisms. In [12], the authors discuss specific SSI challenges for petascale systems. SSI tools to make system administration easier are presented in [2, 7, 15].

Process migration is discussed in detail in [10] and specific implementations appear in MOSIX [1], BProc [6], and IBM's WPAR [9], among others. The survey in [16] is a little bit dated, but it gives a good description of so-called worms, which are processes specifically designed to migrate.

Implementations of SSI, most of them partial, abound even before the term SSI was coined. Examples include LOCUS [17], Plan 9 [14], MOSIX [1], and Kerrighed [11].

Bibliography

1. Barak A, La'adan O (1998) The MOSIX multicomputer operating system for high performance cluster computing. Future Gener Comput Syst 13(4–5):361–372
2. Brightwell R, Fisk LA, Greenberg DS, Hudson T, Levenhagen M, Maccabe AB, Riesen R (2000) Massively parallel computing using commodity components. Parallel Comput 26(2–3):243–266
3. Buyya R, Cortes T, Jin H (2001) Single system image. Int J High Perform Comput Appl 15(2):124–135
4. Frachtenberg E, Petrini F, Fernandez J, Pakin S, Coll S (2002) STORM: lightning-fast resource management. In: Supercomputing '02: proceedings of the 2002 ACM/IEEE conference on supercomputing, Baltimore. IEEE Computer Society, Los Alamitos, pp 1–26
5. Greenberg DS, Brightwell R, Fisk LA, Maccabe AB, Riesen R (1997) A system software architecture for high-end computing. In: SC'97: high performance networking and computing: proceedings of the 1997 ACM/IEEE SC97 conference, San Jose, Raleigh, 15–21 Nov 1997. ACM/IEEE Computer Society, New York
6. Hendriks E (2002) BPROC: the Beowulf distributed process space. In: ICS '02: proceedings of the 16th international conference on supercomputing, pp 129–136. ACM, New York
7. Hendriks EA, Minnich RG (2006) How to build a fast and reliable 1024 node cluster with only one disk. J Supercomput 36(2):171–181
8. Hwang K, Jin H, Chow E, Wang C-L, Xu Z (1999) Designing SSI clusters with hierarchical check pointing and single I/O space. IEEE Concurr 7(1):60–69

9. Kharat S, Mishra R, Das R, Vishwanathan S (2008) Migration of software partition in UNIX system. In: Compute '08: proceedings of the first Bangalore annual compute conference, Bangalore. ACM, New York, pp 1–4
10. Milojičić DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration. ACM Comput Surv 32(3):241–299
11. Morin C, Gallard P, Lottiaux R, Vallée G (2004) Towards an efficient single system image cluster operating system. Future Gener Comput Syst 20(4):505–521
12. Ong H, Vetter J, Studham RS, McCurdy C, Walker B, Cox A (2006) Kernel-level single system image for petascale computing. SIGOPS Oper Syst Rev 40(2):50–54
13. Pfister GF (1998) In search of clusters: the ongoing battle in lowly parallel computing, 2nd edn. Prentice-Hall, Upper Saddle River
14. Pike R, Presotto D, Thompson K, Trickey H, Winterbottom P (1993) The use of name spaces in Plan 9. SIGOPS Oper Syst Rev 27(2):72–76
15. Skjellum A, Dimitrov R, Angaluri SV, Lifka D, Coulouris G, Uthayopas P, Scott SL, Eskicioglu R (2001) Systems administration. Int J High Perform Comput Appl 15(2): 143–161
16. Smith JM (1988) A survey of process migration mechanisms. SIGOPS Oper Syst Rev 22(3):28–40
17. Walker B, Popek G, English R, Kline C, Thiel G (1983) The LOCUS distributed operating system. In: SOSP '83: proceedings of the ninth ACM symposium on operating systems principles, Bretton Woods. ACM, New York, pp 49–70

Singular-Value Decomposition (SVD)

►Eigenvalue and Singular-Value Problems

Sisal

John Feo
Pacific Northwest National Laboratory, Richland, WA, USA

Definition

Streams and Iterations in a Single Assignment Language (Sisal) was a general-purpose applicative language developed for shared-memory and vector supercomputer systems. It provided an hierarchical intermediate form, parallel runtime system, optimizing compiler, and programming environment. The language was strongly typed. It supported both array and stream data structures, and had both iterative and parallel loop constructs.

Discussion

Introduction

Streams and Iterations in a Single Assignment Language (Sisal) was a general-purpose applicative language defined by Lawrence Livermore National Laboratory, Colorado Sate University, University of Manchester and Digital Equipment Corporation in the early 1980s [1]. Lawrence Livermore and Colorado State developed the language over the next 2 decades. They maintained the language definition, compiler and runtime system, programming tools, and provided education and user support services. The language version used widely in the 1980s and 1990s was Sisal 1.2 [2]. Three Sisal user conferences were held in 1991, 1992, and 1993 [3, 4].

The Sisal Language Project had four major accomplishments: (1) IF1, an intermediate form used widely in research [5]; (2) OSC, an optimizing compiler that preallocated memory for most data structures and eliminated unnecessary coping and reference count operations [7]; (3) a highly efficient, threaded runtime system that supported most shared-memory, parallel computer systems [6], and (4) execution performance and scalability equivalent to imperative languages [1, 8].

Language Definition

Sisal's applicative semantics and keyword-centric syntax proved to be a simple, easy-to-use, easy-to-read parallel programming language that facilitated algorithm development and simplified compilation. Since the value of any expression depended only on the values of its subexpressions and not on the order of evaluation, Sisal programs defined the data dependencies among operations, but left the scheduling of operations, the communication of data values, and the synchronization of concurrent operations to the compiler and runtime system.

Sisal was strongly typed. Programmers specified the types of only function parameters; the compiler inferred all other types. All functions were mathematically sound. A function had access to only its arguments and there were no side effects. Each function invocation was independent and functions did not retain state

between invocations. Sisal programs were referentially transparent and single-assignment. Since a name was bound to one value and not a memory location, there was no aliasing. In general, a Sisal program defined a set of mathematical expressions where names stood for specific values, and computations progressed without state making the transformation from source code to dataflow graph trivial.

Sisal supported the standard scalar data types: boolean, character, integer, real, and double precision. It also included the aggregate types: array, stream, record, and union. All arrays were one-dimensional; multi-dimensional arrays were arrays of arrays. The type, size, and lower bound of an array was not defined explicitly, rather it was a consequence of execution. Arrays were created by gathering elements, "modifying" existing arrays, or catenations.

A stream was a sequence of values of uniform type. In Sisal, stream elements were accessible in order only; there was no random access. A stream could have only one producer, but many consumers. By definition, Sisal streams were nonstrict – each element was available as soon as it was produced. The Sisal runtime system supported the concurrent execution of the producer and consumer(s).

The two major language constructs were *for initial* and *for* expressions. The former expression substituted for sequential iteration in conventional languages, but retained single assignment semantics. The rebinding of loop names to values was implicit and occurred between iterations. Loop names prefixed with *old* referred to previous values. A returns clause defined the results and arity of the expression. Each result was either the final value of some loop name or a reduction of the values assigned to a loop name during loop execution. The following *for initial* expression returns the prefix sum of A

```
for initial
    i := 0;
    x := A[i]
while i < n repeat
    i := old i + 1;
    x := old x + A[i]
returns array of x
end for
```

Sisal supported seven intrinsic reductions: *array of*, *stream of*, *catenate*, *sum*, *product*, *least*, and *greatest*. The order of reduction was determinate.

The *for* expression provided a way to specify independent iterations. The semantics of the expression did not allow references to values defined in other iterations. The expression was controlled by a range expression that could be simple, a dot product of ranges, or a cross product of ranges. The next two expressions compute the dot and cross product of two arrays of size *n* and *m*

```
for i in 0 to n - 1 dot j
      in 0 to m - 1
   x := A[i] * B[j]
returns sum of x
end for
for i in 0 to n - 1 cross j
      in 0 to m - 1
   x := A[i] * B[j]
returns array of x
end for
```

Note that the expression does not indicate how the parallel iterations should be scheduled. The management of parallel threads was the responsibility of the compiler and runtime systems and was highly tuned for each architecture. This separation of responsibility greatly simplified Sisal programs, improved portability, and made the language easy to use.

Build-in-Place Analysis

The Sisal compile process comprised eight phases shown in Fig. 1. The two most important phases concerned build-in-place (IF2MEM) and update-in-place analysis (IF2UP). IF2 was a superset of IF1. It was not applicative as it included operations (AT-nodes) that directly referenced and manipulated memory.

Prior to Sisal, functional programming languages were considered very inefficient. Since the size and structure of aggregate data types are a consequence of execution, without analysis aggregates must be built one element at a time possibly requiring elements to be copied many times as the aggregate grows in size. Moreover, strict adherence to single-assignment semantics requires construction of a new array whenever a single element is modified.

Sisal. Fig. 1 OSC phases

Build-in-place analysis attacked the incremental construction problem [6]. The algorithm had two-passes. Pass 1 visited nodes in data flow order, and built, where possible, an expression to calculate an array's size at run time. The expression was a function of the semantics of the array constructor and the size expressions of its inputs. Determining the size of an array built during loop execution required an expression to calculate the number of loop iterations before the loop executes. Deriving this expression was not possible for all loops. Pass 1 concluded by pushing, in the order of encounter, the nodes for the size expression onto a stack.

Pass 2 removed nodes from the stack, inserted them into the data flow graph, and appropriately wired memory references among them by inserting edges transmitting pointers to memory. If a node was the parent of an already converted node, it could build its result directly into the memory allocated to its child, thus eliminating the intermediate array. If the node was not the parent of an already converted node, it built its result in a new memory location. Note that the ordering of nodes on the stack guaranteed processing of children before parents.

Update-in-Place Analysis

Update-in-place analysis attacked the aggregate update problem [7]. OSC introduced dependences in the data flow graph to schedule readers of an aggregate object before writers. It used reference counts to identify the last use of an object. If the last use was a write, then the write was executed in place. By re-ordering nodes and aggregating reference copy operations, OSC eliminated up to 98% of explicit reference count operations in larger programs without reducing parallelism. Consequently, the inefficiencies of reference counting largely disappeared and the benefits of updating aggregate objects were fully enjoyed by Sisal programs.

OSC considered an array as two separately reference-counted objects: a dope vector defining the array's logical extent and the physical space containing its constituents. As array elements are computed and stored, dope vectors cycle between dependent nodes to communicate the current status of the regions under construction. As a result, multiple dope vectors might reference the same or different regions of the physical space and the individual participants in the construction would produce new dope vectors to communicate the current status of the regions to their predecessors. Without update-in-place optimization, each stage in construction would copy a dope vector.

Runtime System

The runtime software supported the parallel execution of Sisal programs, provided general purpose dynamic storage allocation, implemented operations on major data structures, and interfaced with the operating system for input/output and command line processing [7].

The Sisal runtime made modest demands of the host operating system. At program initiation a command line option specified the number of operating system processes (workers) to instantiate for the duration of the program. The runtime system maintained two queues of threads: the For Pool and the Ready List. A worker was always in one of three modes of operation: executing a thread from the For Pool, executing a thread from the Ready Pool, or idle. The runtime system allocated stacks for threads on demand, but every effort was made to reuse previously allocated stacks and thus reduce allocation and deallocation overhead.

The For Pool maintained slices of *for* expressions. A worker would pick up a slice, execute it, and return to the pool for another slice. The worker that executed the last slice of an expression placed the parent thread on the Ready List, and returned to the For Pool to look for slices from other expressions. By persisting in executing slices of *for* expressions, a worker avoided deallocating and allocating its stack.

The Ready List maintained all other threads made executable by events, such as the main program or function calls. If a worker found no work on either Pool, it examined the Storage Deallocation List for deferred storage deallocation work.

Sisal relied on dynamic storage allocation for many data values such as arrays and streams, and for internal objects such as thread descriptors and execution stacks. It used a two-level allocation mechanism that was fast, efficient, and parallelizable. A standard boundary tag scheme was augmented with multiple entry points to a circular list of free blocks. Both the free list search and deallocation of blocks were parallelized, even though the former sometimes completely removed blocks from the list and the latter coalesced physically adjacent but logically distant blocks. To increase speed, an exact-fit caching mechanism was interposed before the boundary tag scheme. It used a working set of different sizes of recently freed blocks.

Performance

As noted above, most Sisal programs executed as fast as their imperative language counterparts. Consequently, Sisal attracted a wide user community that developed many different types of parallel applications, including benchmark codes, mathematical methods, scientific simulations, systems optimization, and bioinformatics.

Table 1 lists the execution times of four applications and Table 2 gives compilation statistics for each program. Columns 2–4 give the number of static arrays built, preallocated, and built-in-place; columns 5–8 list the number of copy, conditional copy and reference count operations after optimization, and the number of artificial dependency edges introduced by the compiler.

It is instructive to examine two of the applications. SIMPLE was a two-dimensional Lagrangian hydrodynamics code developed at Lawrence Livermore National Laboratory to simulate the behavior of a fluid in a sphere. The hydrodynamic and heat condition equations are solved by finite difference methods. A tabular ideal gas equation determines the relation between state variables. The implementation of SIMPLE in Sisal 1.2 was straightforward and highly parallel.

The compiler preallocated and built all arrays in place (261 of them), eliminated all absolute copy operations, marked 19 copy operations for run time check, and eliminated 2,005 out of 2,066 reference count operations. The 19 conditional copy operations were a result of row sharing. Since they always tested false at runtime, no copy operations actually occured. For 62 iterations of a 100 × 100 grid problem, the Sisal and Fortran versions of SIMPLE on one processor executed in 3,099.3 and 3,081.3 s, respectively. On ten processors, the Sisal code realized a speedup of 7.3. An equivalent of 1.5 processors

Sisal. Table 1 Execution times

Program	Fortran	Sisal (#p)	Sisal (#p)	Speedup
GaussJordan	54.0 s	54.5 s (1)	8.8 s (10)	6.2
RICARD	30.6 h	31.0 h	3.5 h (10)	9.0
SIMPLE	3081.3 s	3099.3 s(1)	422.0 (10)	7.3
Simulated annealing	476.6 s	956.2 s (1)	267.8 s (5)	3.6

Sisal. Table 2 OSC optimizations

Program	Arrays	PreAllocated	Built inplace	Copy	CCopy	Ref counts	Artificial edges
GaussJordan	7	7	7	0	0	1	9
RICARD	29	29	28	0	6	7	5
SIMPLE	261	261	261	0	19	61	347
Simulated annealing	46	46	42	0	4	41	168

was lost in allocating and deallocating two-dimensional arrays. The implementation of true multi-dimensional arrays was proposed in both Sisal 2.0 [9] and Sisal 90. [These language definitions were defined near the end of the project, but never implemented.]

Simulated annealing is a generic Monte Carlo optimization technique for solving many difficult combinatorial problems. In the case of the school timetable problem, the objective is to assign a set of tuples to a fixed set of time slots (periods) such that no critical resource is scheduled more than once in any period. Each tuple is a record of four fields: class, room, subject, and teacher. Classes, rooms, and teachers are critical resources; subjects are not. At each step of the procedure, a tuple is chosen at random and moved to another period. If the new schedule has equivalent or lower cost, the move is accepted. If the new schedule has higher cost, the move is accepted with a certain exponential probability. If the move is not accepted, the tuple is returned to its original period. The procedure can be paralleilized by simultaneously choosing one tuple from each nonempty period and applying the move criterion to each. The accepted moves are then carried out one at a time.

OSC preallocated memory for all the arrays, and built all but four of the arrays in place. It removed all absolute copy operations, marked four copy operations for run time check, and removed all but 41 reference count operations. The four arrays not built in pace resulted from adding a tuple to a period. Although the compiler did not mark the new periods for build-in-place, the periods were rarely copied. When an element was removed from the high-end of an array, the array's logical size shrunk by one, but its physical size remained constant. Thus, when an element was added to a period, there was often space to add the element without copying. Whenever copying occurred, extra space was allocated to accommodate future growth. This foresight saved over 15,000 copies in this application at the cost of a few hundred bytes of storage.

Bibliographic Notes and Further Reading

For a summary of the project and details on the compiler and runtime system see [1]. The language definition can be found in [2]. IF1 is described [5].

Build-in-place analysis and update-in-place analysis are described in more detail in [6] and [7], respectively.

In 1991 and then again in 1992 two new versions of the language, Sisal 2.0 and Sisal 90, were defined. These versions included: higher-order functions, user-defined reductions, parameterized data types, foreign language modules, array syntax, and rectangular arrays. While compilers for the version were never released there is some information available on the web.

Bibliography

1. Feo JT, Cann D, Oldehoeft R (1990) A report of the Sisal Language Project. J Parallel Distrib Comput 10(4):349–366
2. McGraw J et al (1985) Sisal: streams and iterations in a single-assignment language, reference manual version 1.2. Lawrence Livermore National Laboratory Manual M-146, Livermore, CA, September 1985
3. Feo J (ed) Proceedings of the 2nd Sisal User Conference. Lawrence Livermore National Laboratory, CONF-9210270, San Diego, CA, October 1992
4. Feo J (ed) Proceedings 3rd Sisal User Conference. Lawrence Livermore National Laboratory, CONF-9310206, San Diego, CA, October 1993
5. Skedzielewski S, Glauert J (1985) IF1 – an intermediate form for applicative languages. Lawrence Livermore National Laboratory Manual M-170, Livermore National Laboratory, Livermore, CA, September 1985
6. Ranelletti J (1996) Graph transformation algorithms for array memory optimization in applicative languages. PhD dissertation, University of California at Davis, CA, May 1996
7. Cann D (1998) Compilation techniques for high performance applicative computation. PhD dissertation, Colorado State University, CO, May 1998
8. Cann D (1992) Retire Fortran?: A debate rekindled. Commun ACM 35(8):81–89
9. Oldehoeft R et al (1991) The Sisal 2.0 reference manual. Lawrence Livermore National Laboratory, Technical Report UCRL-MA-109098, Livermore, CA, December 1991

Small-World Network Analysis and Partitioning (SNAP) Framework

►SNAP (Small-World Network Analysis and Partitioning) Framework

SNAP (Small-World Network Analysis and Partitioning) Framework

Kamesh Madduri
Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Synonyms

Graph analysis software; Small-world network analysis and partitioning (SNAP) framework

Definition

SNAP (Small-world Network Analysis and Partitioning) is a framework for exploratory analysis of large-scale complex networks. It provides a collection of optimized parallel implementations for common graph-theoretic problems.

Discussion

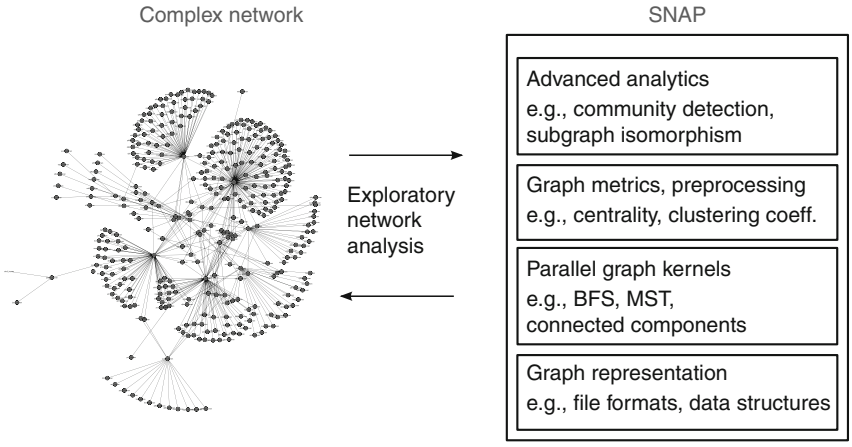
Introduction

Graphs are a fundamental abstraction for modeling and analyzing data, and are pervasive in real-world applications. Transportation networks (road and airline traffic), socio-economic interactions (friendship circles, organizational hierarchies, online collaboration networks), and biological systems (food webs, protein interaction networks) are a few examples of data that can be naturally represented as graphs. Understanding the dynamics and evolution of real-world network abstractions is an interdisciplinary research challenge with wide-ranging implications. Empirical studies on networks have led to a variety of models to characterize their topology and evolution. Quite surprisingly, it has been shown that graph abstractions arising from diverse systems such as the Internet, social interactions, and biological networks exhibit common structural features such as a low diameter, unbalanced degree distributions, self-similarity, and the presence of dense subgraphs [1, 11, 14]. Some of these topological features are captured by what is known as the *small-world* model or phenomenon.

The analysis of large graph abstractions, particularly small-world complex networks, raises interesting computational challenges. Graph algorithms are typically highly memory intensive, make heavy use of data structures such as lists, sets, queues, and hash tables, and exhibit a combination of data and task-level parallelism. On current workstations, it is infeasible to do exact in-core computations on graphs larger than 100 million vertex/edge entities (*large* in general refers to the typical problem size for which the graph and the associated data structures do not fit in 2–4 GB of main memory). In such cases, parallel computers can be utilized to obtain solutions for memory and compute-intensive graph problems quickly. Due to power constraints and diminishing returns from instruction-level parallelism, the computing industry is rapidly converging towards widespread use of multicore chips and accelerators. Unfortunately, several known parallel algorithms for graph-theoretic problems do not easily map onto clusters of multicore systems. The mismatch arises due to the fact that current systems lean towards efficient execution of regular computations with low memory footprints and working sets, and heavily penalize memory-intensive applications with irregular memory accesses; however, parallel graph algorithms in the past were mostly designed assuming an underlying, well-balanced compute-memory platform. The small-world characteristics of real networks, and the load-balancing constraints they impose during parallelization, represent an additional challenge to the design of scalable graph algorithms.

SNAP [2, 10] is an open-source computational framework for graph-theoretic analysis of large-scale complex networks. It is intended to be an optimized collection of computational kernels (or algorithmic building-blocks) that the end-user could readily use and compose to answer higher-level, ad-hoc graph analysis queries. The target platforms for SNAP are shared-memory multicore and symmetric multiprocessor systems. SNAP kernels are implemented in C and use OpenMP for parallelization. On distributed memory systems, SNAP can be used for intra-node parallelization, and the user has to manage inter-node communication, and also identify and implement parallelism at a coarser node-level granularity.

The parallel graph algorithms in SNAP are significantly faster than alternate implementations in other open-source graph software. This is due to a combination of the use of memory-efficient data structures, preprocessing kernels that are tuned for small-world



SNAP (Small-World Network Analysis and Partitioning) Framework. Fig. 1 A schematic of the SNAP graph analysis framework [8]

networks, as well as algorithms that are designed to specifically target cache-based multicore architectures. These issues are discussed in more detail in the following sections of this article.

As the project title suggests, the initial design goals of SNAP were to provide scalable parallel solutions for community structure detection [4], a problem variant of graph partitioning that is of great interest in social, and in general, small-world network analysis. Community structure detection is informally defined as identifying densely connected sets of vertices in the network, thereby revealing latent structure in a large-scale network. It is similar to the problem of graph partitioning in scientific computing, as is usually formulated as a graph clustering problem. SNAP includes several different parallel algorithms for solving this problem of community detection.

Graph Representation

The first issue that arises in the design of graph algorithms is the use of appropriate in-memory data structures for representing the graph. The data to be analyzed typically resides on disk in a database, or in multiple files. As the data is read from disk and loaded into main memory, a graph representation is simultaneously constructed. The minimal layout that would constitute an in-memory graph representation is a list of edge tuples with vertex identifiers indicating the source and destination vertices, and any attributes associated with the edges and vertices. However, this does not give one easy access to lists of edges originating from a specific vertex. Thus, the next commonly used representation is to sort the edge tuple list by the source vertex identifier, and store all the adjacencies of a particular vertex in a contiguous array. This is the primary *adjacency list* representation of graphs that is supported in SNAP. Edges can have multiple attributes associated with them, in which case they can either be stored along with the corresponding adjacency vertex identifier, or in separate auxiliary arrays. This representation is space-efficient, has a low computational overhead for degree and membership queries, and provides cache-friendly access to adjacency iterators.

In cases where one requires periodic structural updates to the graph, such as insertions and deletions of edges, SNAP uses alternate graph representations. An extension to the static representation is the use of dynamic, resizable adjacency arrays. Clearly, this would support fast insertions. Further, parallel insertions can be supported using non-blocking atomic increment operations on most of the current platforms. There are two potential parallel performance issues with this data structure. Edge deletions are expensive in this representation, as one needs to scan the entire adjacency list in the worst case to locate the required tuple. The scan is particularly expensive for high-degree vertices. Second, there may be load-balancing issues with parallel edge insertions (for instance, consider the case where there are a stream of insertions to the adjacency list of

a single vertex). These problems can be alleviated by batching the updates, or by randomly shuffling the updates before scheduling the insertions. If one uses sorted adjacency lists to address the deletion problem, then the cost of insertions goes up due to the overhead of maintaining the sorted order.

An alternative to arrays is the use of tree structures to support both quick insertions and deletions. Treaps are binary search trees with a priority (typically a random number) associated with each node. The priorities are maintained in heap order, and this data structure supports insertions, deletions, and searching in average-case logarithmic time. In addition, there are known efficient parallel algorithms for set operations on treaps such as union, intersection, and difference. Set operations are particularly useful to implement kernels such as graph traversal and induced subgraphs, and also for batch-processing updates.

To process both insertions and deletions efficiently, and also given the power-law degree distribution for small-world networks, SNAP supports a hybrid representation that uses dynamically resizable arrays to represent adjacencies of *low-degree* vertices, and treaps for *high-degree* vertices. The threshold to determine low and high-degree vertices can be varied based on the data set characteristics. By using dynamic arrays for low-degree vertices (which will be a majority of vertices in the graph), one can achieve good performance for insertions. Also, deletions are fast and cache-friendly for low-degree vertices, whereas they take logarithmic time for high-degree vertices represented using treaps. Madduri and Bader [9] discuss the parallel performance and space-time trade-offs involved with each of these representations in more detail.

Parallelization Strategies

There is a wide variety in the known approaches for exploiting parallelism in graph problems. Some of the easier ones to implement and analyze are computations involving iterations over vertex and edge lists, without much inter-iteration dependencies. For instance, queries such as determining the top-k high-degree vertices, or the maximum-weighted edge in the graph, can be easily parallelized. However, most parallel algorithms require the use of data structures such as priority queues and multisets. Further, one needs support for fast parallel operations on these structures, such as parallel insertions, membership queries, and batched deletions. Fine-grained, low overhead synchronization is an important requirement for several efficient parallel implementations. Further, the notion of *partitioning* a graph is closely related to several parallel graph algorithms. Given that inter-processor communication is expensive on distributed memory systems (compared to computation on a single node), several parallel graph approaches rely on a graph partitioning and vertex reordering preprocessing step to minimize communication in subsequent algorithm executions. Graph partitioning is relevant in the context of shared-memory SNAP algorithms as well, since it reduces parallel synchronization overhead and improves locality in memory accesses.

Consider the example of breadth-first graph traversal (BFS) as an illustration of more complex paradigms for parallelism in graph algorithms. Several SNAP graph kernels are designed to exploit fine-grained thread-level parallelism in graph traversal. There are two common parallel approaches to breadth-first search: *level-synchronous* graph traversal, where the adjacencies of vertices at each level in the graph are visited in parallel; and *path-limited searches*, where multiple searches from vertices that are *far apart* are concurrently executed, and the independent searches are aggregated. The level-synchronous approach is particularly suited for small-world networks due to their low graph diameter. Support for fine-grained efficient synchronization is critical in both these approaches. The SNAP implementation of BFS aggressively reduces locking and barrier constructs through algorithmic changes as well as architecture-specific optimizations. For instance, the SNAP BFS implementation uses a lock-free approach for tracking visited vertices and thread-local work queues, significantly reducing shared memory contention. While designing fine-grained algorithms for small-world networks, it is also important to take the unbalanced degree distributions into account. In a level-synchronized parallel BFS where vertices are statically assigned to multiple threads of execution, it is likely that there will be phases with severe work imbalance (due to the imbalance in vertex degrees). To avoid this, the SNAP implementation first estimates the processing work to be done at each vertex, and then assigns vertices accordingly to threads. Several other optimization techniques, and their relative performance

benefits, are discussed by Bader and Madduri in more detail [2].

Graph algorithms often involve performance tradeoffs associated with memory utilization and parallelization granularity. In cases where the input graph instance is small enough, one could create several replicates of the graph or the associated data structures for multiple threads to execute concurrently, and thus reduce synchronization overhead. SNAP utilizes this technique for the exact computation of betweenness centrality, which requires a graph traversal from each vertex. The fine-grained implementation parallelizing each graph traversal requires space linear in the number of vertices and edges, whereas the coarse-grained approach (the traversals are distributed among p threads) incurs a p-way multiplicative factor memory increase. Depending on the graph size, one could choose an appropriate number of replicates to reduce the synchronization overhead.

SNAP Kernels for Exploratory Network Analysis

Exploratory graph analysis often involves an iterative study of the structure and dynamics of a network, using a discriminating selection of topological metrics. SNAP supports fast exact and approximate computation of well-known social network analysis metrics, such as average vertex degree, clustering coefficient, average shortest path length, rich-club coefficient, and assortativity. Most of these metrics have a linear or sub-linear computational complexity, and are straightforward to implement. When used appropriately, they not only provide insight into the network structure, but also help speed up subsequent analysis algorithms. For instance, the average neighbor connectivity metric is a weighted average that gives the average neighbor degree of a degree-k vertex. It is an indicator of whether vertices of a given degree preferentially connect to high- or low-degree vertices. Assortativity coefficient is a related metric, which is an indicator of community structure in a network. Based on these metrics, it may be possible to say whether the input data is representative of a specific common graph class, such as bipartite graphs or networks with pronounced community structure. This helps one choose an appropriate community detection algorithm and also the right clustering measure to optimize for. Other preprocessing kernels that are beneficial for exposing parallelism in the problem include computation of connected and biconnected components of the graph. If a graph is composed of several large connected components, it can be decomposed and individual components can be analyzed concurrently. A combination of these preprocessing kernels before the execution of a compute-intensive routine often lead to a substantial performance benefit, either by reducing the computation by pruning vertices or edges, or by explosing more coarse-grained concurrency and locality in the problem.

Community Identification Algorithms in SNAP

Several routines in SNAP are devoted to solving the community identification problem in small-world networks. Graph partitioning and community identification are related problems, but with an important difference: the most commonly used objective function in partitioning is minimization of edge cut, while trying to *balance the number of vertices* in each partition. The number of partitions is typically an input parameter for a partitioning algorithm. Clustering, on the other hand, optimizes for an appropriate application-dependent measure, and the number of clusters is unknown beforehand. Multilevel and spectral partitioning algorithms (e.g., Chaco [6] and Metis [7]) have been shown to be very effective for partitioning graph abstractions derived from physical topologies, such as finite-element meshes arising in scientific computing. However, the edge cut when partitioning small-world networks using these tools is nearly two orders of magnitude higher than the corresponding edge cut value for a nearly-Euclidean topology, for graph instances that are comparable in the number of vertices and edges. Clearly, small-world networks lack the topological regularity found in scientific meshes and physical networks, and current graph partitioning algorithms cannot be adapted as-is for the problem of community identification.

The parallel community identification algorithms in SNAP optimize for a measure called modularity. Let $C = (C_1, \ldots, C_k)$ denote a partition of the set of vertices V such that $C_i \neq \phi$ and $C_i \cap C_j = \phi$. The cluster $G(C_i)$ is identified with the induced subgraph $G[C_i] := (C_i, E(C_i))$, where $E(C_i) := \{\langle u, v\rangle \in E : u, v \in C_i\}$. Then, $E(C) := \cup_{i=1}^{k} E(C_i)$ is the set of intra-cluster edges

and $\tilde{E}(C) := E - E(C)$ is the set of inter-cluster edges. Let $m(C_i)$ denote the number of inter-cluster edges in C_i. Then, the modularity measure $q(C)$ of a clustering C is defined as

$$q(C) = \sum_i \left[\frac{m(C_i)}{m} - \left(\frac{\sum_{v \in C_i} deg(v)}{2m} \right)^2 \right].$$

Intuitively, modularity captures the idea that a *good division* of a network into communities is one in which the inter-community edge count is less than what is expected by random chance. This measure does not try to minimize the edge cut in isolation, nor does it explicitly favor a balanced community partitioning. The general problem of modularity optimization has been shown to be $\mathcal{NP}$-complete, and so there are several known heuristics to maximize modularity. Most of the known techniques fall into one of the two broad categories: *divisive* and *agglomerative* clustering. In the agglomerative scheme, each vertex initially belongs to a singleton community, and communities whose amalgamation produces an increase in the modularity score are typically merged together. The divisive approach is a top-down scheme where the entire network is initially in one community, and the network is iteratively broken down into subcommunities. This hierarchical structure of community resolution can be represented using a tree (referred to as a dendrogram). The final list of the communities is given by the leaves of the dendrogram, and internal nodes correspond to splits (joins) in the divisive (agglomerative) approaches.

SNAP includes several parallel algorithms for community identification that use agglomerative and divisive clustering schemes [2]. One divisive clustering approach is based on the greedy betweenness-based algorithm proposed by Newman and Girvan [12]. In this approach, the community structure is resolved by iteratively filtering edges with high *edge betweenness centrality*, and tracking the modularity measure as edges are removed. The compute-intensive step in the algorithm is repeated computation of the edge betweenness scores in every iteration, and SNAP performs this computation in parallel. SNAP also supports several greedy agglomerative clustering approaches, and the main strategy to exploit parallelism in these schemes is to concurrently resolve communities that do not share intercommunity edges. Performance results on real-world networks indicate that the SNAP parallel community identification algorithms give substantial performance gains, without any loss in community structure (given by the modularity score) [2, 8].

Related Entries

►Chaco
►Graph Algorithms
►Graph Partitioning
►METIS and ParMETIS
►PaToH (Partitioning Tool for Hypergraphs)
►Social Networks

Bibliographic Notes and Further Reading

The community detection algorithms in SNAP are discussed in more detail in [3, 8]. Other popular libraries and frameworks for large-scale network analysis include Network Workbench [13], igraph [3], and the Parallel Boost Graph Library [5].

Bibliography

1. Amaral LAN, Scala A, Barthélémy M, Stanley HE (2000) Classes of small-world networks. Proc Natl Acad Sci USA 97(21): 11149–11152
2. Bader DA, Madduri K (April 2008) SNAP: Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), IEEE, Miami, FL
3. Csárdi G, Nepusz T (2006) The igraph software package for complex network research. InterJournal Complex Systems 1695. http://igraph.sf.net. Accessed May 2011
4. Fortunato S (Feb 2010) Community detection in graphs. Physics Reports 486(3–5):75–174
5. Gregor D, Lumsdaine A (July 2005) The Parallel BGL: a generic library for distributed graph computations. In: Proceedings of the Parallel/High-Performance Object-Oriented Scientific Computing (POOSC '05), IOS Press, Glasgow, UK
6. Hendrickson B, Leland R (Dec 1995) A multilevel algorithm for partitioning graphs. In: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC 1995), ACM/IEEE Computer Society, New York
7. Karypis G, Kumar V (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
8. Madduri K (July 2008) A high-performance framework for analyzing massive complex networks. Ph.D. thesis, Georgia Institute of Technology
9. Madduri K, Bader DA (May 2009) Compact graph representations and parallel connectivity algorithms for massive dynamic

network analysis. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), IEEE Computer Society, Rome, Italy
10. Madduri K, Bader DA, Riedy EJ (2011) SNAP: Small-world Network Analysis and Partitioning v0.4. http://snap-graph.sf.net. Accessed May 2011
11. Newman MEJ (2003) The structure and function of complex networks. SIAM Rev 45(2):167–256
12. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69:026113
13. NWB Team (2006) Network Workbench Tool. Indiana University, Northeastern University, and University of Michigan, http://nwb.slis.indiana.edu. Accessed May 2011
14. Watts DJ, Strogatz SH (1998) Collective dynamics of small world networks. Nature 393:440–442

SoC (System on Chip)

Tanguy Risset
INSA Lyon, Villeurbanne, France

Definition

A System on Chip (SoC) refers to a single-integrated circuit (chip) composed of all the components of an electronic system. A SoC is heterogeneous, in addition to classical digital components: processor, memory, bus, etc.; it may contain analog and radio components. The SoC market has been driven by embedded computing systems: mobile phones and handheld devices.

Discussion

Historical View

Gordon Moore predicted in 1965 the exponential growth of silicon integration and its consequences on the application of integrated circuits. Following this growth, known as *Moore's Law*, the number of transistor integrated on a single silicon chip has doubled every 18 months, leading to a constant growth in the semi conductor industry for over 30 years. This technological evolution implied constant changes in the design of digital circuits, with, for instance, the advent of gate level simulation and logic synthesis. Amongst these changes, the advent of System on Chip (SoC) represented a major technological shift.

The term SoC became increasingly widespread in the 1990s and is used to describe chips integrating on a single silicon die what was before spread on several circuits: processor, memory, bus, hardware device drivers, etc. SoC technology did not fundamentally change the functionality of the systems built, but drastically enlarged the *design space*: choosing the best way to assemble a complete system from beginning (system specification) to end (chip manufacturing) became a difficult task. In 1999, a book entitled *Surviving the SoC Revolution: A guide to plateform-based design* was published. It was written by a group of people from Cadence Design System, an important computer-aided design tool company: SoC was driving a revolution of the *design methodologies* for digital systems.

The production of SoC has been driven by the emerging market of embedded systems, more precisely *embedded computing systems*: cellular phones, PDA, digital camera, etc. Embedded computing systems require strong integration, computing power and energy saving, features that were provided by SoC integration. Moreover, most of these systems required radio communication features; hence, a SoC integrates digital components and analog components: the radio subsystem. The success of mobile devices has increased the economic pressure on SoCs, design cost and time-to-market have become major issues, leading to new design methodologies such as IP-based design and hardware/software codesign.

Another consequence of integration is the exponential growth of embedded software, shifting the complexity of SoC design from hardware to software. Emulation, simulation at various level of precision, and high level design techniques are used because nowadays SoC are incredibly complex: hundreds of millions of gates and millions of lines of code. Increasing digital computing power enables software radio, providing everything everywhere transparently. SoCs are now used to provide convergence between computing, telecommunication, and multimedia. Pervasive environments will also make extensive use of embedded "intelligence" with SoCs.

What Is a SoC?

A system on chip or SoC is an integrated circuit composed of many components including: one or several processors, on-chip memories, hardware accelerators, devices drivers, digital/analog converters, and analog components. It was initially named SoC because all the features of a complete "system" were integrated together

on the same chip. At that time, a system was dedicated to a single application: video processing or wireless communication for instance. Thanks to the increasing role of software, SoCs are no longer specific, in fact many of them are reused in several different telephony or multimedia devices.

Processors

The processor is the core of the SoC. Unlike desktop machine processors, a wide variety of processors is integrated in the SoC: general purpose processors, digital signal processors, micro-controllers, application specific processors, FPGA based soft cores, etc. The International Technology Road-Map for Semiconductors (ITRS) indicated that, in 2005, more than 70% of application specific integrated circuit (ASICS) contained a processor. Until 2010, most SoCs were composed of a single processor, responsible for the control of the whole system, associated with hardware accelerators and direct memory access components (DMA). In 2008, the majority of SoCs were built around a processor of one of the following form of processor architectures: ARM, MIPS, PowerPC, or x86. Multi-processor SoCs (MPSoC) are progressively appearing and making use of the available chip area to keep performance improvements. MPSoC appears to be a new major technological shift and MPSoC designers are facing problem traditionally associated with super-computing.

Busses

The bus, or more generally the interconnection medium, is also a major component of the SoC. Many SoCs contain several busses, including a fast system bus connecting the major SoC components (processor, memory, hardware accelerator) and a slower bus for other peripherals. For instance, the Advanced Microcontroller Bus Architecture (Amba) proposes the AHB specification (Advanced High-performance Bus), and the APB (Advanced Peripheral Bus), STBus (ST-Microelectronics), and CoreConnect (IBM) are other examples of SoC busses. The use of commercial bus protocol has been a major obstacle to SoC standardization: a bus comes with a dedicated communication protocol which might be incompatible with other busses. Networks on Chip (NoC) are progressively used with MPSoC, the major problem of NoCs today is their considerable power consumption.

Dedicated Components

A SoC will include standard control circuits such as UART for the serial port, USB controller, and control of specific devices: GPS, accelerometer, touch pad, etc. It might also include more compute intensive dedicated *intellectual properties* (IPS) usually targeted to signal or image processing: FFT, turbo decoder, H.263 coder, etc. Choosing between a dedicated component or a software implementation of the same functionality is done in early stages of the design, in the so-called *hardware/software partitioning*. A hardware IP will save power and execution time compared to a software implementation of the same functionality. Dedicated IPS are often mandatory to meet the performance requirements of the application (radio or multimedia), but they increase SoC price, complexity, and also reduce its flexibility.

Other Components

Analog and mixed signal components may be included to provide radio connexion: audio front-end, radio to baseband interfaces, analog to digital converters. Other application domains may require very specific components such as microelectromechanical systems (MEMS), optical computing, biochips, and nanomaterials.

Quality Criteria

The only objective indicator of the quality of a SoC is its economical success, which is obviously difficult to predict. Among the quality criteria, the most important are: power consumption, time to market, cost, and reusability.

Power consumption has been a major issue for decades. Integrating all components on the same chip reduces power consumption because inter chip wires have disappeared. On the other hand, increasing clock rate, program size, and number of computations has a negative impact on power consumption. Moreover, it is very difficult to statically predict the power consumption of a SoC as it heavily depends on low-level technological details. A precise electrical simulation of a SoC is extremely slow and cannot be used in practice for a complete system. Minimizing memory footprint and memory traffic, gating clocks for idle components, and dynamically adapting clock rate are techniques used to reduce power consumption in SoCs. In recent submicronic technologies, static power consumption is significant, leading to power consumption for idle devices.

It is usually said that, for a given product, the first commercially available SoC will capture half of the market. Hence, reducing the design time is a critical issue. Performing hardware and software design in parallel is a designer's dream: it becomes possible with virtual prototyping, that is, simulation of software on the virtual hardware. High-level design and refinement are also widely used: a good decision at a high level may save weeks of design.

The cost of a SoC is divided into two components: the nonrecurring engineering (NRE) cost which corresponds to the design cost and the unit manufacturing cost. Depending on the number of units manufactured and the implementation technology, a complex tradeoff between the two can be made, bearing in mind that some technological choices can shorten the time to market such as field programmable gate arrays (FPGA) for instance. NRE cost integrates, in addition to the design cost of the hardware and software part of the SoC, the cost of computer-aided design tools, computing resource for simulation/emulation, and development of tools for the SoC itself: compilers, linkers, debuggers, simulation models of the IPs, etc. To give a very rough idea, in 2008, the development cost of a complex SoC could reach several million dollars and could need more than one hundred man years. In this context, is seems obvious that the reuse of IPs, tools, and SoCs is a major issue because it shrinks both NRE costs and time to market.

An Example

Figure 1 shows the architecture of the Samsung S3CA400A01 SoC, designed to be associated with another chip containing processor and memory through the Cotulla interface. The Cotulla interface is associated with Xscale processor (strong ARM processor architecture manufactured by Intel). This chip was used for instance in association with the PXA250 chip of Intel in the PDA of Hewlett Packard iPaQ H5550. This SoC illustrates the different device driver components that can be found, one can also observe the hierarchy of busses mentioned before. It also shows that there is no *standard SoC*, this one has no processor inside. Thanks to the development of the open source software community, many SoC architectures have been detailed and Linux ports are available for many of them.

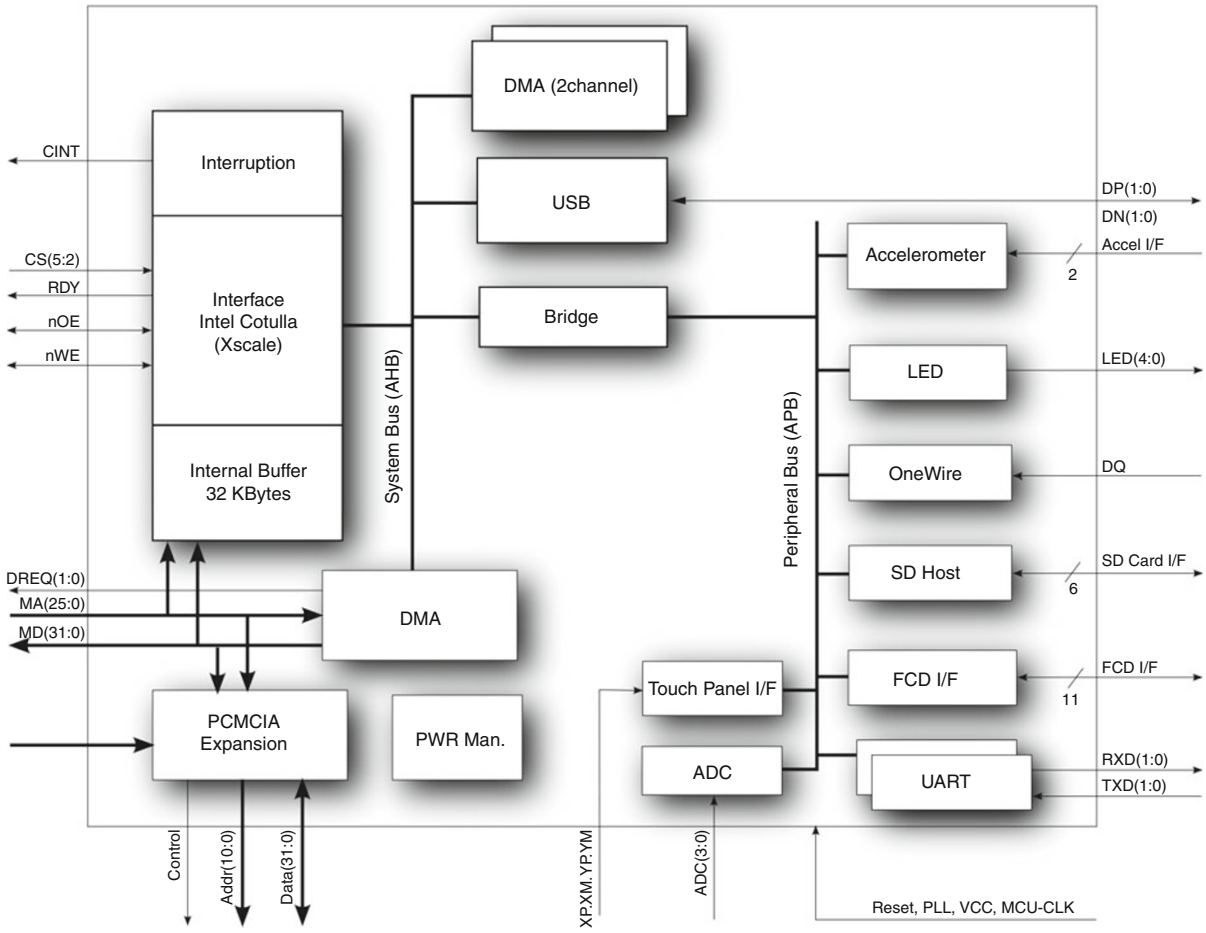
Why Is It a SoC "Revolution"?

Although its advent was predicted, SoC really caused a revolution in the semiconductor industry because it changed design methods and brought many software design problems in the microelectronics community.

Hardware design has been impacted by the arrival of SoCs. The complexity of chip mechanically increases with the number of gates on the chip. But the design and verification effort increases more than linearly with the number of gates. This is known as the *design gap*: How can the designer efficiently use the additional gates available without breaking design time and design reliability? A SoC requires different technologies to be integrated on the same silicon die (standard cells, memory, FPGA). The clock distribution tree is a major source of power dissipation leading to the advent of globally asynchronous, locally synchronous systems (GALS). Routing wires became a real problem partially solved by using additional metal layers.

Hardware design methods have drastically changed with the constant arrival of new Electronic Design Automation (EDA) tools. RTL design methods (RTL stands for *register transfer level*) have become the standard for hardware developers replacing transistor-level circuit design. The hardware description languages VHDL and Verilog are widely used for hardware specification before synthesis. However, higher-level hardware description language are needed, partly because SoC simulation time became prohibitive, requiring huge FPGA based emulators. In addition, the networked nature of embedded applications, and the non-determinism introduced by parallelism in MPSoC greatly complicates SoC simulation. As mentioned before, cost and time to market imposes pressure on engineers productivity, leading managers to new management methods. The hardware/software nature of a SoC requires a constant dialog between hardware and software engineers.

Embedded software has been introduced in circuit design, this is, in itself, a revolution. In 2006, the Siemens company employed more software developers than Microsoft did, and since the mid-2000 s, more than half of the SoC design efforts were spent in software development. Originally composed of a very basic infinite loop waiting for interrupts, embedded software has evolved by adding many device drivers, standard or real time operating system services: tasks or



SoC (System on Chip). Fig. 1 Architecture of the Samsung S3CA400A01 SoC

threads, resource management, shared memory, communication protocols, etc. Even if an important part of embedded code is written in C, there is a trend to use component based software design methodologies that reduces the conceptual difference between hardware and software objects. Verifying software reliability is very difficult, even if an important effort is dedicated to that during the design process, complete formal verification is impossible because of the complexity of the software. This explains the bugs that everybody has experienced on their mobile devices.

Both hardware and software design methods have been drastically modified, but the most important novelty concerns the whole SoC design methodology that tries to associate in a single framework hardware and software design.

SoC Design Methodology

There are many names associated with SoC design methodologies, all of them refer more or less to the same goal: designing hardware and software in the same framework such that the designer can quickly produce the most efficient implementation for a given system specification. The term frequently used today is *system level design*, the generic scheme of system level design is the following: (1) derive hardware components and software components from the specification, (2) map the software part on the hardware components, (3) prototype the resulting implementation, and (4) possibly provide changes at some stage of the design if the performance or cost is not satisfactory.

There is a global agreement on the fact that high-level specifications are very useful to reduce the

design time. Many design frameworks have been proposed for system level SoC design. The idea of refinement of the original specification down to hardware is present in many frameworks, high level synthesis (HLS) was once seen as the solution but appears to be a very difficult problem in general. In HLS, the hardware is completely derived from the initial specifications. During the 1990s, the arrival of Intellectual Properties (IP) introduced a clear distinction between circuit fabrication and circuit design, leading some companies to concentrate on IP design (ARM for instance). On the other hand, IP-based design and later *platform-based design* propose to start from fixed (or parameterizable) hardware library to reduce the design space exploration phase. Improvement in simulation techniques permits, with the use of systemC language, to have an approach between the two by using virtual prototypes: cycle true simulation of complete SoC before hardware implementation.

MPSoC and Future Trends

An open question remains at the present time: What will be the dominant model for MPSoCs that are predicted to include more than one hundred processors? There are two main trends: heterogeneous MPSoCs following actual heterogeneous SoC architecture or homogeneous SPMD-like multiprocessors on a single chip.

A heterogeneous MPSoC includes different types of processors with different instruction set architectures (ISA): general purpose processors, DSPs, application specific instruction set processors (ASIP). It also contains dedicated IPs that might not be considered as processors, DMAs, and memories. All these components are connected together through a network on chip.

Heterogeneous MPSoCs are a natural evolution of SoC, their main advantage is that they are more energy efficient than homogeneous MPSoCs. They have been driven by specific application domains such as software radio or embedded video processing. But, for the future, they suffer from many drawbacks. As they are usually tuned for a particular application, they are difficult to reuse in another context. Because of incompatibility between ISAs, tasks cannot migrate between processors inducing a much less flexible task mapping. Moreover, in the foreseen transistor technology, integrated circuits will have to be fault tolerant because of electronics defaults. Heterogeneous MPSoC are not fault tolerant by nature. Finally, their scalability is not obvious and code reuse between different heterogeneous platforms is impossible. However, because heterogeneous MPSoCs provide better performance with lower power consumption and lower cost, there are lots of commercial activities on this model.

Homogeneous MPSoC are more flexible, they can implement well known operating system services such as code portability between different architectures, dynamic task migration, virtual shared memory. Homogeneity can help in being fault tolerant, as any processor can be replaced by another. However, with more than a hundred processor on a chip in 2020, the MPSoC architecture cannot be completely homogeneous, it has to be hierarchical: clusters of processors tightly linked, these clusters being interconnected with a network on chip. Memory must be physically distributed with non-uniform access.

Ensuring cache coherency and providing efficient compilation and parallelization tools for these architectures are real challenges. In 2009, the ITRS road-map predicted for 2013 the advent of a concurrent software compiler that will "Enable compilation and software development in highly parallel processing SoC," this will be a critical step towards the advent of homogeneous MPSoC. However, many unknowns remain concerning the programming model to use, the ability of compilation tools to efficiently use the available resources, the reusability of the chip and the reusability of the embedded code between platforms, and the real power consumption of these homogeneous multiprocessors on chip.

A trade-off would be a homogeneous SoC with some level of heterogeneity: dedicated IPs for domain-specific treatments. New technological techniques, such as 3D VLSI packaging technology or more generally the arrival of so-called *system in package*, mixing various nanotechnologies on a single chip might also open a new road to embedded computing systems. But, whatever will be the successful MPSoC model, it will bring highly parallel computing on a chip and should lead to a revival of parallel computing engineering.

Bibliographic Notes and Further Reading

A number of books have been published on SoC, we have already talked about "surviving the SOC

revolution" [1]. Many interesting ideas were also present in the Polis project [4]. Daniel Gajski [9] worked on SoC from the beginning. Wayne Wolf published a general presentation [13]. More recently, Chris Rowen [11] proposed an interesting view of SoC design.

A good overview of the status of high-level synthesis can be found in Coussy and Moraviec book [2]. More practical details can be found on the Steve Furber book on ARM [8].

A number of useful information are present on the web, from the International Technology Roadmap for Semiconductors (ITRS [7]), from analyst or engineer [3] or open source software developer for embedded devices [5, 6]. A simple introduction to semiconductor industry can be found in Jim Turley's book [12].

A good introduction on MPSoC is presented in [10], but most of the interesting work on this subject are presented in the proceedings of the international conferences on the subject: Design Automation Conference (DAC), Design automation and Test in Europe (DATE), International Forum on Embedded MPSoC and Multicore (MPSOC), etc.

Bibliography

1. Chang H, Cooke L, Hunt M, Martin G, McNelly AJ, Todd L (1999) Surviving the SOC revolution: a guide to platform-based design. Kluwer Academic Publishers, Norwell, MA
2. Coussy P, Morawiec A (eds) (2008) High-level synthesis: from algorithm to digital circuit. Springer, Berlin, Germany
3. Embedded System Design (2009) http://www.embedded.com/
4. Balarin F et al (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Press, Dordrecht
5. Linux for device (2009) http://www.linuxfordevices.com/
6. Open Embedded: framework for embedded Linux (2009) http://wiki.openembedded.net
7. International Technology Roadmap for Semiconductors (2009) http://www.itrs.net/
8. Furber S (2000) ARM system-on-chip architecture. Addison Wesley, Boston, MA
9. Gajski DD, Abdi S, Gerstlauer A, Schirner G (2009) Embedded system design: modeling, synthesis and verification. Kluwer Academic Publishers, Dordrecht
10. Jerraya A, Wolf W (2004) Multiprocessor systems-on-chips (The Morgan Kaufmann Series in Systems on Silicon). Morgan Kaufmann, San Francisco, CA
11. Rowen C (2004) Engineering the complex SOC: fast, flexible design with configurable processors. Prentice-Hall Press, Upper Saddle River, NJ
12. Turley J (2002) The essential guide to semiconductors. Prentice Hall Press, Upper Saddle River, NJ
13. Wolf W (2002) Modern VLSI design: system-on-chip design. Prentice Hall Press, Upper Saddle River, NJ

Social Networks

Maleq Khan, V. S. Anil Kumar, Madha V. Marathe, Paula E. Stretz
Virginia Tech, Blacksburg, VA, USA

Introduction

Networks are pervasive in today's world. They provide appropriate representations for systems comprised of individual agents interacting locally. Examples of such systems are urban regional transportation systems, national electrical power markets and grids, the Internet, ad-hoc communication and computing systems, public health, etc. According to Wikipedia, *A social network is a social structure made of individuals (or organizations) called "nodes," which are tied (connected) by one or more specific types of interdependency, such as friendship, kinship, financial exchange, dislike, sexual relationships, or relationships of beliefs, knowledge or prestige.* Formally, a social network induced by a set V of agents is a graph $G = (V, E)$, with an edge $e = (u, v) \in E$ between individuals u and v, if they are interrelated or interdependent. Here, we use a more general definition of nodes and edges (that represent interdependency). The nodes represent living or virtual individuals. The edges can represent information flow, physical proximity, or any feature that induces a potential interaction. For example, the social contact networks, an edge signifies some form of physical co-location between the individuals – such a contact may either capture physical proximity, an explicit physical contact or a co-location in the virtual world, e.g., Facebook. Socio-technical networks are a generalization of social networks and consist of a large number of interacting physical, technological, and human/societal agents. The links in socio-technical networks can be physically real or a matter of convention such as those imposed by law or social norms, depending on the specific system being represented. This entry primarily deals with social networks.

Social networks have been studied for at least 100 years; see [9] for a detailed account of the history and development of social networks and the analytical tools that followed them. Scientists have used social networks to uncover interesting insights related to societies and social interactions. Social networks, their structural analysis, and dynamics on these networks (e.g., spread of diseases over social contact networks) are now a key part of social, economic, and behavioral sciences. Importantly, these concepts and tools are also becoming popular in other scientific disciplines such as public health epidemiology, ecology, and computer science due to their potential applications. For example, in computer science, interest in social networks has been spurred by web search and other online communication and information applications. Google and other search engines crucially use the structure of the web graph. Social networking sites, such as Facebook and Twitter, and blogging sites have grown at an amazing rate and play a crucial role in advertising and the spread of fads, fashion, public opinion, and politics. The growing importance of social networks in the scientific community can be gauged by a report from the National Academies [18] as well as a recent issue of Science [21]. The proliferation of Facebook, MySpace, Twitter, Blogsphere, and other online communities has made social networking a pervasive and essential part of our lingua.

An important and almost universal observation is that the network structure seems to have a significant impact on these systems and their associated dynamics. For instance, many infrastructure networks (such as the Internet) have been observed to be highly vulnerable to targeted attacks, but are much more robust to random attacks [2]. In contrast, social contact networks are known to be very robust to all types of attacks [7]. These robustness properties have significant implications for control and policies on such systems - for instance, protecting high degree nodes in Internet-like graphs has been found to be effective in stopping the spread of worms [19]. For many processes, e.g., the SIS model of diffusion (which models "endemic" diseases, such as Malaria and Internet malware), the dynamics have been found to be characterized quite effectively by the spectral properties of the underlying network [19]. Similarly, classical and new graph theoretic metrics, e.g., centrality (which, informally, determines the fraction of shortest paths passing through a node), have been found to be useful in identifying "important" nodes in networks and community structure. Therefore, computing the properties of these graphs, and simulating the dynamical processes defined on them are important research areas.

Most networks that arise in practice are very large and heterogeneous, and cannot be easily stored in memory - for instance, the social contact graph of a city could have millions of nodes and edges [13], while the web graph has over 11 billion nodes [7] and several billion edges. Additionally, these networks change at very short time scales. They are also naturally labeled structures, which adds to the memory requirements. Consequently, simple and well-understood problems such as finding shortest paths, which have almost linear time algorithms, become challenging to solve on such graphs. High-performance computing has been successfully used in a number of scientific applications involving large data sets with complex processes, and is, therefore, a natural and necessary approach to overcome such resource limitations.

Most of the successful applications of high-performance computing have been in physical sciences, such as molecular dynamics, radiation transport, n-body dynamics, and sequence analysis, and researchers routinely solve very large problems using massive distributed systems. Many techniques and general principles have been developed, which have made parallel computing a crucial tool in these applications. Social networks present fundamentally new challenges for parallel computing, since they have very different structure, and do not fit the paradigms that have been developed for other applications. As we discuss later, some of the key issues that arise in parallelization of social network problems include highly irregular structure, poor locality, variable granularity, and data-driven computations. Most social contact networks have a scale-free and "small world" structure (described formally below), which is very different from regular structures such as grids. As a result, social network problems often cannot be decomposed easily into smaller independent subproblems with low communication, making traditional parallel computing techniques very inefficient.

The goal of this article is to highlight the challenges for high-performance computing posed by the growing area of social networks. New paradigms are needed at all levels of hardware architecture, software methodologies, and algorithmic optimization.

Section "Background and Notation" presents some background into social networks, their history, and important problems. Section "Petascale Computing Challenges for Social Network Problems" identifies the main challenges for parallel computing. We discuss some recent developments in Section "Techniques" and conclude in section "Conclusions".

Background and Notation

Many of the results in sociology that have now become folklore have been based on insights from social network analysis. One of such classic results is the "six-degree of separation" experiment, in which Stanley Milgram [22] asked 296 randomly chosen people to forward a letter to a stock broker in Boston, by sending it to one of their acquaintances, who is most likely to know the target. Milgram found that the median length of chains that successfully reached the target was about 6, suggesting that the global social contact network was very highly connected. Attempts to explain this phenomenon have led to the "small-world" graph models, which show that a small fraction of long-range contacts to individuals spatially located far away is adequate to bring the diameter of the network down. Another classic result is Granovetter's concept of the "strength of weak ties" [13], which has become one of the most influential papers in sociology. He examined the role of information that individuals obtain as a result of their position in the social contact network, and found that information from acquaintances (weak ties), and not close friends, was the most useful, during major lifestyle changes, e.g., finding jobs. This finding has led to the notions of "homophily" (i.e., nodes with similar attributes form contacts) and "triadic closure" (i.e., nodes with a common neighbor are more likely to form a link), and the idea that weak ties form bridges that span different parts of the network.

Concepts such as these have been refined and formalized using graph theoretic measures; see [5, 19] for a formal discussion. Some of the key graph measures are: (1) *(Weighted) Degree distribution*: The degree of a node is the number of contacts it has, and the degree distribution is the frequency distribution of degrees. This measure is commonly used to characterize and distinguish various families of graphs. In particular, it has been found that many real networks have power law or lognormal degree distributions, instead of Poisson, which has been used to develop models to explain the construction and evolution of these networks. (2) The *Clustering coefficient*, $C(v)$, of a node v is the probability that two randomly picked neighbors of node v are connected, and models the notion of triadic closure, and its extensions. (3) *Centrality*: Let $f(s,t)$ denote the number of shortest paths from s to t, and let $f_v(s,t)$ denote the number of shortest paths from s to t that pass through node v. The centrality of node v is defined as $bc(v) = \sum_{s,t} f_v(s,t)/f(s,t)$, and captures its "importance". (4) *Robustness to node and edge deletions*, which is quantified in terms of the giant component size, as a result of node and edge deletions, and has been found to be a useful measure to compare graphs. (5) *Page Rank and Hubs* have been used to identify "important" web pages for search queries, but have been extended to other applications. The Page Rank of a node is, informally, related to the stationary probability of a node in a random walk model of a web surfer. (6) A *community* or a *cluster* is a "loosely connected" set of nodes with similar attributes, which are different from nodes in other communities. There are many different methods for identifying communities, including modularity and spectral structure [19, 22].

In applications such as epidemiology and viral marketing, the network is usually associated with a diffusion process, e.g., the spread of disease or fads. There are many models of diffusion, and some of the most widely used classes of models are *Stochastic cascade models* [11, 19]. In these models, we are given a probability $p(e)$ for each edge $e \in E$ in a graph $G = (V, E)$. The process starts at a node s initially. If node v becomes active at time t, it activates each neighbor w with probability $p(v,w)$, independently (and also independent of the history); no node can be reactivated in this process. In the case of viral marketing, active nodes are those that adopt a certain product, while in the SIR model of epidemics, the process corresponds to the spread of a disease, and the active nodes are the ones that get infected. Another class of models that has been studied extensively involves *Threshold functions*; one of the earliest uses of threshold models was by Granovetter and Schelling [10], who used it for modeling segregation. The *Linear Threshold Model* [14] is at the core of many of these models. In this model, each node v has a threshold Θ_v (which may be chosen randomly), and each edge (v,w) has a weight $b_{v,w}$. A node v becomes active at time t, if $\sum_{w \in N(v) \cap A_t} b_{v,w} \geq \Theta_v$,

where $N(v)$ denotes the set of neighbors of v and A_t denotes the set of active nodes at time t (in this model, an active node remains active throughout). Such diffusion processes are instances of more general models of dynamical systems, called *Sequential Dynamical Systems (SDSs)* [7, 19], which generalize other models, such as cellular automata, Hopfield networks, and communicating finite state machines. An SDS S is defined as a tuple (G, F, π), where: (a) $G = (V, E)$ is the underlying graph on a set V of nodes, (b) $F = (f_v)$ denotes a set of local functions for each node $v \in V$, on some fixed domain. Each node v computes its state by applying the local function f_v on the states of its neighbors. (c) π denotes a permutation (or a word) on V, and specifies the order in which the node states are to be updated by applying the local functions. One update of the SDS involves applying the local functions in the order specified by π. It is easy to see that the above diffusion models can be captured by suitable choice of the local functions f_v and the order π. It is easy to extend the basic definition of SDS to accommodate local functions that are stochastic, the edges that are dynamic, and the graph that is represented hierarchically (to capture organizational structure).

Fundamental questions in social networks include (1) understanding the structure of the networks and the associated dynamics, especially how the dynamics is affected by the network properties; (2) techniques to control the dynamics; (3) identifying the most critical and vulnerable nodes crucial for the dynamics; and (4) coevolution between the network and dynamics – this issue brings in behavioral changes by individuals as a result of the diffusion process. The above mentioned problems require large-scale simulations, and parallel computing is a natural approach for designing them.

Petascale Computing Challenges for Social Network Problems

Lumsdaine et al. [15] justifiably assert that traditional parallel computing techniques (e.g., those developed in the context of applications such as molecular dynamics and sequencing) are not well suited for large-scale social network problems because of the following reasons: (1) Graph algorithms chiefly involve data driven computations, in which the computations are guided by the graph structure. Therefore, the structure and sequence of computations are not known in advance and are hard to predict, making parallelization difficult. (2) Social networks are typically very irregular and strongly connected, which makes partitioning into "independent" sub-problems difficult [15]. As observed by many researchers, these networks usually have a small-world structure with high clustering, and such graphs have low diameter and large separators (unlike regular graphs such as grids, which commonly arise in other applications). (3) Locality is one of the key properties that helps in parallelization; however, computations on social networks tend to give low locality because of the irregular structure, leading to computations and data access patterns with global properties. (4) As noted in [15, 16], graph computations often involve exploration of the graph structure, which are highly memory intensive and there is very little computation to hide the latency to memory accesses. Thus, the performance of memory subsystem, rather than the memory clock frequency, dominates the performance of graph algorithms.

Traditional parallel architectures have been developed for applications that do not have these constraints, and therefore, are not completely suited for social network algorithms. The most common paradigm of distributed computing, the distributed memory architectures with message-passing interface (MPI), leads to high message passing, because of the lack of locality and high data access to computation ratio. The shared memory model is much more suited for the kinds of data-driven computations that arise in graph algorithms, and multithreaded approaches have been found to be more effective in exploiting the parallelism that arises. Some of the main hardware and software challenges that arise are as follows: (1) In many graph problems, e.g., shortest paths, parallelism can be found at a fairly fine level of granularity, though in other problems, e.g., computing centrality, there is coarse grained granularity, where each path computation is an independent task. (2) While multithreading is crucial for graph algorithms, the unstructured nature of these graphs implies that there are significant memory contention and cache-coherence problems. (3) It is difficult to achieve load balancing on graph computations, where the level of parallelism varies with time, e.g., as in breadth-first search.

Techniques

Parallel algorithms for social network problems is an active area of research and we now discuss some of the key techniques that have been found to be effective for some problems that arise in analyzing social networks. These techniques are still application specific, and developing general paradigms is still an active research area.

Massive Multithreading Techniques

In [12], Hendrickson and Berry discussed how massively multithreaded architectures, such as the Cray MTA-2 and its successor the XMT, can be used to boost the performance of parallel graph algorithms. Instead of trying to reduce latency for single-memory access, the MTA-2 tolerates it by ensuring that the processor has other work to do while waiting for a memory request to be satisfied by supporting many concurrent threads in a single processor and switching between them in a single clock cycle. When a thread issues memory request, the processor immediately switches to another ready-to-execute thread. MTA-2 supports fast and dynamic thread creation and destruction allowing the program to dynamically determine the number of threads based on the data to be processed. Support of virtualization of threads allows adaptive parallelism and dynamic load balancing. MTA-2 also supports word-level locking of the data items, which decreases access contention and minimizes impact on the execution of other threads.

Drawbacks of massively multithreaded machines include higher price and much slower clock rate than mainstream systems, and the difficulty in porting to other architectures. Hendrickson and Berry [12], using the massively multithreaded architectures, extended a small subset of the Boost Graph Library (BGL) into the Multithreaded Graph Library (MTGL). This library, which is their ongoing current work, retains the BGL's look and feel, yet encapsulates the use of nonstandard features of massively multithreaded architectures. Madduri et al. [16] also discussed architectural features of the massively multithreaded Cray XMT system and present implementation details and optimization of betweenness centrality computations. They further showed how the parallel algorithm for betweenness centrality can be modified so that locking is not necessary to avoid concurrent access to a memory unit, which they call lock-free parallel algorithm.

Distributed Streaming Algorithms

Data streaming is a useful approach to deal with memory constraints, in which processors keep a "sketch" of the data, while making one or more passes through the entire data (or stream), which is assumed to be too big to store. Streaming is usually sequential, and Google's MapReduce [6] and Apache's Hadoop [1] provide a generic programming framework for distributed streaming. MapReduce/Hadoop provide a transparent implementation which takes care of issues like data distribution, synchronization, load balancing, and processor failures. It can, thus, greatly simplify computation over large-scale data sets, and has been proven to be a useful abstraction for solving many problems, especially those related to web applications.

We briefly describe the MapReduce framework here. In the map step, the master node takes a task, partitions it into smaller *independent subtasks* and passes them down to the worker nodes. The worker nodes may recursively do the same, or process the subtask and pass it back to the master node. The master node then collects the solutions to each subtask that it had assigned and processes these solutions to obtain the final solution for the original task. More formally, the input to the MapReduce system is represented as a set of (key, value) pairs, which can be defined in a completely general manner. There are two functions: *Map* and *Reduce*, which need to be specified by the user. A Map function processes a key/value pair (k_1, v_1) and produces a list of zero or more intermediate key/value pairs (k_2, v_2). Once the Map function completes processing of the input key/value pairs, the system groups the intermediate pairs by each key k_2 and makes a list of all values associated with k_2. These lists are then provided as input to the Reduce function. The Reduce function is then applied independently to each key/value-list pair to obtain a list of final values. The whole sequence can be repeated as needed.

Programs written in this functional style are automatically parallelized and executed on a large cluster of machines, insulating users from all the run-time issues. The MapReduce framework is quite powerful, and can be used to efficiently compute any symmetric

order-invariant function [8], a large subclass of problems that can be solved by streaming algorithms. This class includes many stream statistics that arise in web applications. Kang et al. [13] use this approach to estimate the diameter of a massive instance of the web graph with over 10 billion edges. The (key, value) pairs in their algorithm capture adjacencies of nodes and estimates of the number of nodes within a d-neighborhood of a node; the algorithm is run in multiple stages in order to keep the keys and associated values simple and small, and also uses efficient sketches for representing set unions. The uniqueness of a MapReduce framework lies in its ability to hide the underlying computing architecture from the end-user. Thus, the user can compute using a MapReduce framework on parallel clusters as well as loosely coupled machines in a cloud or over volunteer computing resources comprised of independent, heterogeneous machines.

Dynamical Processes on Social Networks

So far, much of the discussion has focused on computational considerations related to social network structure. But many more interesting questions arise when one studies dynamical processes on these networks; in fact, it is fair to say that social networks exist to serve the function of one or more dynamical processes on them. While researchers have worked extensively on structural properties of social networks, the literature on the study of dynamical processes on social networks is relatively sparse. When it does exist, it is usually in the context of very small networks or stylized and regular networks. Unfortunately, many of these results do not scale or apply to large and realistic social networks. In [3], the authors discuss new algorithms and their implementations for modeling certain classes of dynamical processes on large social networks. In general, the problem is *hard* computationally; it becomes even harder when the social network structure, the dynamical processes and individual node behavior coevolve. For certain class of dynamical processes that capture a number of interesting social, economic, and behavioral theories of collective behavior, it is possible to develop fast parallel algorithms. Intuitively, such dynamical processes can be expressed as SDSs with a certain symmetry condition imposed on local transition functions.

Conclusions

As our society is becoming more connected, there is an increasing need to develop innovative computational tools to synthesize, analyze, and reason about large and progressively realistic social networks. Applications of social networks for analyzing real-world problems will require the study of multi-network, multi-theory systems – systems composed of multiple networks among agents whose behavior is governed by multiple behaviors. Advances in computing and information are also giving rise to new classes of social networks. Two examples of these networks are: (1) botnet networks in which individual nodes are software agents (bots) and (2) wireless-social networks in which social networks between individuals are being supported by wireless devices that allow them to communicate and interact anytime and anywhere. As the society becomes more connected, these applications require support for real-time individualized decision making over progressively larger evolving social networks. This same technology will form the basis of new modeling and data processing environments. These environments will allow us to leverage the next generation computing and communication resources, including cloud computing, massively parallel peta-scale machines and pervasive wireless sensor networks. The dynamic models will generate new synthetic data sets that cannot be created in any other way (e.g., direct measurement). This will enable social scientists to investigate entirely new research questions about functioning societal infrastructures and the individuals interacting with these systems. Together, these advances also allow scientists, policy makers, educators, planners, and emergency responders unprecedented opportunities for multi-perspective reasoning.

Acknowledgments

This work has been partially supported by NSF Grant CNS-0626964, SES-0729441, CNS-0831633, and OCI-0904844, and DTRA Grant HDTRA1-0901-0017 and HDTRA1-07-C-0113.

Bibliography

1. Apache hadoop. Code and documentation are available at http://developer.yahoo.com/hadoop/
2. Barabasi A, Albert R (1999) Emergence of scaling in random networks. Science 286:509–512

3. Barrett C, Bisset K, Marathe A, Marathe M (2009) An integrated modeling environment to study the co-evolution of networks, individual behavior and epidemics. AI Magazine, 2009
4. Barrett C, Eubuank S, Marathe M (2005) Modeling and simulation of large biological, information and socio-technical systems: an interaction based approach interactive computation. In: Goldin D, Smolka S, Wegner P (2005) The new paradigm, Springer, Berlin, Heidelberg, New York
5. Baur M, Brandes U, Lerner J, Wagner D (2009), Group-level analysis and visualization of social networks. In: Lerner J, Wagner D, Zweig K (eds) Algorithmics of large and complex networks, vol 5515, LNCS. Springer, Berlin, Heidelberg, New York, pp 330–358
6. Dean J, Ghemawat S (2004) Mapreduce: Simplified data processing on large clusters. In: Proceedings of the sixth symposium on operating system design and implementation (OSDI), San Francisco, December 2004
7. Eubank S, Guclu H, Anil Kumar VS, Marathe MV, Srinivasan A, Toroczkai Z, Wang N (2004) Modelling disease outbreaks in realistic urban social networks. Nature 429(6998):180–184
8. Feldman J, Muthukrishnan S, Sidiropoulos A, Stein C, Svitkina Z (2008) On distributing symmetric streaming computations. In: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA), San Francisco, pp 710–719
9. Freeman L (2004) The development of social network analysis: a study in the sociology of science. Empirical Press, Vancouver
10. Granovetter M (1978) Threshold models of collective behavior. Am J Sociològy 83(6):1420–1443
11. Grimmett G (1999) Percolation. Springer, New York
12. Hendrickson B, Berry J (2008) Graph analysis with high-performance computing. Comput Sci Eng 10:14–19
13. Kang U, Tsourakakis CE, Appel AP, Faloutsos C, Leskovec J (2008) Hadi: fast diameter estimation and mining in massive graphs with hadoop. Technical Report CMU-ML-08-117, Carnegie Mellon University
14. Kempe D, Kleinberg JM, Tardos E (2003) Maximizing the spread of influence through a social network. In: SIGKDD '03, Washington
15. Lumsdaine A, Gregor D, Hendrickson B, Berry J (2007) Challenges in parallel graph processing. Parallel Process Lett 17:5–20
16. Madduri K, Ediger D, Jiang K, Bader DA, Chavarra-Miranda DG (2009) A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: Proceedings of the 3rd Workshop on Multithreaded Architectures and Applications (MTAAP), Miami, May 2009
17. Mortveit HS, Reidys CM (2000) Discrete sequential dynamical systems. Discrete Math 226:281–295
18. National Research Council of the National Academies (2005) Network Science. The National Academies Press, Washington, DC
19. Newman M (2003) The structure and function of complex networks. SIAM Rev 45:167–256
20. Newman MEJ (2004) Detecting community structure in networks. European Phy J B, 38:321–330
21. Special issue on complex systems and networks, Science, 24 July 2009, 325 (5939):357–504
22. Travers J, Milgram S (1969) An experimental study of the small world problem. Sociometry 32(4):425–443

Software Autotuning

►Autotuning

Software Distributed Shared Memory

Sandhya Dwarkadas
University of Rochester, Rochester, NY, USA

Synonyms

Implementations of shared memory in software; Shared virtual memory; Virtual shared memory

Definition

Software distributed shared memory (SDSM) refers to the implementation of shared memory in software on systems that do not provide hardware support for data coherence and consistency across nodes (and the memory therein).

Discussion

Introduction

Executing an application in parallel requires the coordination of computation and the communication of data to respect dependences among tasks. Historically, multiprocessor machines provide either a shared memory or a message passing model for data communication in hardware (with models such as partitioned global address spaces that fall in between the two extremes). In the message passing model, communication and coordination across processes is achieved via explicit messages. In the shared memory model, communication and coordination is achieved by directly reading and writing memory that is accessible by multiple processes and mapped into their address space. Either model can be emulated in software if the hardware does not

directly implement it. This entry focuses on implementations of shared memory in software on machines that do not support shared memory in hardware across all nodes.

In the shared memory model, data communication is implicitly performed when shared data is accessed. Programmers (or compilers) must still ensure, however, that shared data users synchronize (coordinate) with each other in order to respect program dependencies. The message passing model typically requires explicit communication management by the programmer or compiler – software must specify *what* data to communicate, with *whom* to communicate the data, and *when* the communication must be performed (typically on both the sender and the receiver). Coordination is often implicit in the data communication. While parallel programming has proven inherently more difficult than sequential programming, shared memory is considered conceptually easier as a parallel programming model than message passing. In contrast to message passing, the shared memory model hides the need for explicit data communication from the user, thereby avoiding the need to answer the "*what, when, and whom*" questions.

In terms of implementation, typical shared memory-based hardware requires support for coherence – ensuring that any modifications made by a processor propagate to all copies of the data — and consistency – ensuring a well-defined ordering in terms of when and in what order modifications are visible to other processors. Such support and the need for access to the physically shared memory impacts the scalability of the design. In contrast, message-based hardware is typically easier to build and to scale to large numbers of processors.

Software distributed shared memory (SDSM) attempts to combine the conceptual appeal of shared memory with the scalability of distributed message-based communication by allowing shared memory programs to run on message-based machines. SDSM systems target the knee of the price-performance curve by using commodity components as the nodes in a network of machines. SDSM provides processes with the illusion of shared address spaces on machines that do not physically share memory. Kai Li's [27, 28] pioneering work in the mid-1980s was the first to consider implementing SDSM on a network of workstations. Since then, there has been a tremendous amount of work exploring the state space of possible implementations and programming models, targeted at reducing the software overheads required and hiding the larger communication overheads of message-based hardware.

Implementation Issues

In order to implement coherence, accesses to shared data must be detected and controlled. Implementing coherence in software can typically be accomplished either using virtual memory hardware [8, 23, 28], using instrumentation [34], or using language-level hooks [3, 18]. The former two approaches assume that memory is a linear untyped array of bytes. The latter approach takes advantage of object- and type-specific information to optimize the protocol. The trade-offs are discussed below.

Implementations Using Virtual Memory

The virtual memory mechanisms available on general-purpose processors may be used to implement coherence [8, 23, 28]. The virtual memory framework combines protection with address translation at a granularity of a page, which is typically on the order of a few (e.g., 4) KBytes on today's machines. Hardware protection modes can be exercised to ensure that read or write accesses to pages that are shared are trapped if protection is violated. Software handlers installed by the runtime can then determine the necessary actions required to maintain coherence. The advantage of the virtual memory approach is that there is no overhead for data that is private or already cached. The disadvantage is the large sharing granularity, which can result in data being falsely (i.e., different processors reading and writing to independent locations that are colocated on the same page) shared, and the large cost of handling a miss.

Implementations Using Instrumentation

An alternative approach is to instrument the program in order to monitor every load and store operation to shared data. The advantage of this approach is that coherence can be maintained at any desired granularity [35], resulting in lower miss penalties. The disadvantage is that overhead is incurred regardless of whether

data is actually shared, resulting in higher hit latency. Optimizations that identify and instrument accesses to only shared data and that hide the instrumentation behind existing program computation [34] help reduce this overhead. Alternatively, hardware support, such as ECC (error-correcting code) bits in memory [35], can be leveraged to eliminate the software instrumentation by using the ECC bits to raise an exception when the data is not valid. However, complications do arise if these exceptions are not precise.

Language-Level Implementations

Language-level programming systems (e.g., [3, 18]) provide the opportunity to use objects and types to improve the efficiency of recognizing accesses to shared data. Coherence is usually maintained at the granularity of an object, providing application developers with control over how data is managed. The DOSA system [18] shows how a handle-based implementation (an implementation in which all object references are redirected through a handle) enables efficient fine- and coarse-grain data sharing. Handle-based implementations work transparently when used in conjunction with safe languages (ones in which no pointer arithmetic is allowed). The handles make it easy to relocate objects in memory, making it possible to use virtual memory techniques for access and modification detection and control. They also interact well with the garbage collection algorithms used in these type-safe languages, allowing good performance for these systems. The redirection allows the implementation to avoid false sharing for fine-grained access patterns.

Single- Versus Multiple-Writer Protocols

Traditional hardware-based cache coherence usually maintains the coherence invariant of allowing only a single writer for each coherence granularity unit of data. This approach has the advantage of simplicity, works well under the intuitive assumption that two processes will not intentionally write to the same location concurrently without some form of coordination, and makes it easy to ensure write serialization. However, coherence granularities are often larger than a single word. Even with hardware coherence, it is possible for a cache line to bounce ("ping-pong") between caches because processes modify logically disjoint regions of the same cache line. This problem is further exacerbated by the larger coherence granularities of software coherence, particularly when using a virtual memory implementation.

In order to combat the resulting performance penalties caused by such false sharing, multiple-writer protocols have been proposed [8]. In these protocols, the single writer state invariant of traditional hardware coherence no longer holds. At any given time, multiple processes may be writing to the same coherence unit. Write serialization is ensured by making sure that modifications (determined by keeping track of exactly what locations in the coherence unit were written) are propagated to all valid copies or that these copies are invalidated. Modified addresses can be determined as in the previous section, either by using instrumentation or virtual memory techniques. In the latter case, a *twin* or copy of the virtual memory page is made prior to modification, which is subsequently used to compare to the modified page in order to create a *diff* or an encoding of only the data that has changed on the page. Multiple-writer protocols work particularly well when combined with a more relaxed memory consistency model that allows coherence actions to be delayed to program synchronization points (elaborated on in the next section).

Memory Models

In order to help hide long communication latencies, SDSM systems typically use some form of relaxed consistency model, which enable aggregation of coherence traffic. In release consistency [16], ordinary accesses to shared data are distinguished from synchronization. Synchronization is further split into acquires and releases. An acquire roughly corresponds to a request for access to data, such as a lock operation. A release roughly corresponds to making such data available, such as an unlock operation. SDSM systems leverage the release consistency model to delay coherence actions on writes until the point of a release [8]. This enables both aggregation of coherence messages and a reduction in the false sharing penalty. The former is beneficial because of the high per message costs on typical network-based platforms. The latter is a result of data "ping-ponging" between sharers due to the fact that logically differentiated data resides in the same coherence

unit. By delaying coherence actions to the release point, the number of such "ping-pongs" is reduced.

Release consistency mandates, however, that before the release is visible to *any* other process, all ordinary data accesses made prior to the release are visible at *all* other processes. This results in extra messages between processes at the time of a release. The TreadMarks system [2] proposed the use of a lazy release consistency model [22] by making the observation that for data race free [1] programs, such ordinary data accesses need only be visible at the time of an acquire. Communication can thus be further aggregated and limited to between an acquirer and a releaser.

While lazy release consistency reduces the number of coherence messages used to the minimum possible, it comes at a cost in terms of implementation complexity. Since data accesses are not made visible everywhere at the time of a release, the runtime system must keep track of the ordering of such coherence events and transmit this information when a process eventually performs an acquire. TreadMarks accomplishes this via the use of vector timestamps. The execution of each process is divided into intervals delineated by synchronization operations (either an acquire or a release). Each interval is assigned a monotonically increasing number. Intervals of different processes are partially ordered: Intervals on a single process are totally ordered by program order and intervals on different processes are ordered by acquire–release causality. This partial order is what is captured by assigning a vector timestamp to each interval. Write notices for all data written in an interval are associated with its vector timestamp. This information is piggybacked on release messages to ensure coherence. Each process must then ensure that write notices from all intervals that precede its current interval are applied prior to execution of the current interval.

On system area networks such as Infiniband [21], where the latency and CPU occupancy of communication is an order of magnitude lower than on traditional networks, it can sometimes be beneficial to overlap communication with computation. Cashmere [25] leverages this observation in a moderately lazy release consistent implementation on the Memory Channel [16] network. Write notices are pushed to all processes at the time of a release. They are only applied at the time of the next acquire, thereby avoiding the need to interrupt the remote process. Another advantage of this implementation is that it removes the need to maintain vector timestamps along with the associated metadata management complexity.

An alternative consistency model, *entry consistency*, was defined by Bershad and Zekauskas [5]. All shared data are required to be explicitly associated with some synchronization (lock) variable. On a lock acquisition, only the shared data associated with the lock is guaranteed to be made coherent. This model has some implementation and efficiency advantages. However, the cost is a change to the programming model, requiring explicit association of shared data with synchronization and additional synchronization beyond what is usually required for data race free programs. Scope consistency [20] provides a model similar to entry consistency, but helps eliminate the burden of explicit binding by implicitly associating data with the locks under which they are modified.

Data and Metadata Location

In order to get an up-to-date copy of data, the runtime system must determine the location of the latest version/s. Most implementations fall into two categories – ones in which the information is distributed across all nodes at the time of synchronization, and ones in which each coherence and metadata unit has a *home* that keeps track of where the latest version resides, similar to a hardware directory-based protocol. The former implies that every process knows where the latest version is or has the latest version of the data propagated directly to them. The latter implies a level of indirection (through the home) in order to locate the latest version of the data or owner of the metadata. Variations of the latter protocol include making sure that the home node is kept up to date so that any process requesting a copy can retrieve it directly from the home, and allowing migration of the home to (one of) the most active user/s.

Leveraging Hardware Coherence

As multiprocessor desktop machines become more common, and with the advent of multicore chips, a cluster of multicore multiprocessor machines is a fairly common computing platform. In implementing shared memory on these platforms, the challenge is to take advantage of hardware-based sharing whenever possible and ensure that software overhead is incurred *only* when actively sharing data across nodes in a cluster.

SoftFLASH [12] is a kernel-level implementation of a two-level coherence protocol on a cluster of symmetric multiprocessors (SMPs). SoftFLASH implements coherence using virtual memory techniques, which implies that coherence is implemented by changing read/write permissions in the process's page table. These permissions are normally cached in the translation lookaside buffer (TLB) in each of the processors of a single node. Since the TLB is not typically hardware coherent, in order to ensure that all threads/processes on a single node have appropriate levels of permission to access shared data, the TLBs in each processor within a node must be examined and flushed if necessary. This process, called TLB shootdown, is usually accomplished with costly inter-processor interrupts.

Cashmere-2L [36] avoids the need for these expensive TLB shootdowns through a combination of a relaxed memory model and the use of a novel two-way diffing technique. Leveraging the observation that in a data race free model, accesses by different processes between two synchronization points will be to different memory locations, Cashmere-2L avoids the need to interrupt other processes during a software coherence operation. Updates to a page are applied through a reverse diffing process that identifies and updates only the bytes that have changed, allowing concurrent processes to continue accessing and modifying unrelated data. Invalidations are applied individually by each process and once again leverage the use of diffing in order to avoid potential conflicts with concurrent writers on the node.

Shasta [33] uses instrumentation to implement a finer granularity coherence protocol across SMP nodes. The lack of atomicity of coherence state checks with respect to the actual load or store (the two actions consist of multiple instructions) results in protocol race conditions that require extra overhead when data is shared by processes within an SMP node. A naive solution would involve sufficient synchronization to avoid the race. Shasta avoids this costly synchronization through the selective use of explicit messages. Since protocol metadata is visible to all processes, per-process state tables are used to determine processes that are actively sharing data. Messages are sent only to these processes and overhead incurred only with active sharing.

Leveraging Additional Hardware Support

Systems such as Ivy, TreadMarks, and Munin are implemented under the assumption that memory on remote machines is not directly accessible and must be read/accessed by requesting service from a handler on the remote machine. This is typically accomplished by sending a message over the network, which can be detected and received at the responding end either by periodically polling or by using interrupts. On traditional local area networks, interrupts are typically expensive because of the need for operating system intervention. However, the advantage is that overhead is incurred only when communication is required. Polling (periodically checking the status of the network interface), on the other hand, is relatively cheap (on the order of a few tens of processor cycles), but because the runtime system has no knowledge of when communication will occur, the frequency of checks can result in a significant overhead that is incurred regardless of whether or not data is actively shared. TreadMarks attempted to minimize the number of messages using a lazy protocol on the assumption that the per message costs (both network and processor occupancy as well as the cost to interrupt a remote processor in order to elicit a response) are at least two orders of magnitude higher than memory read and write latencies.

High-performance network technologies, such as those that conform to the Infiniband [21] standard (as well as research prototypes such as Princeton's Shrimp [7] and earlier commercial offerings such as DEC's Memory Channel, Myrinet, and Quadrics QsNet), provide low latency and high bandwidth communication. These networks achieve low latency by virtualizing the network interface. Issues of protection are separated from actual data communication. The former involves the operating system and is performed once at setup time. The latter is performed at user level, thereby achieving lower latency. The majority of these networks allow the possibility of direct access (reads and/or writes) to remote memory, changing the equation in terms of the trade-off in the number of messages used and the eagerness and laziness of the protocol. Protocols such as Cashmere and HLRC [6] leverage such direct access to perform protocol actions eagerly without the need to interrupt the remote processor.

Sharing in the Wide Area

As "computing in the cloud," i.e., taking advantage of geographically distributed compute resources, becomes more ubiquitous, the ability to seamlessly share information across the wide area will improve programmability. Several projects [3, 10, 14, 26, 31, 37] have examined techniques to allow data sharing in the wide area. Most enforce a strongly object-oriented programming model. Specifically, InterWeave [9] supports both language and machine heterogeneity, using an intermediate format to provide persistent data storage and to communicate seamlessly across heterogeneous machines. InterWeave leverages existing hardware and network characteristics (including support for coherence in hardware) whenever possible. The memory model is further relaxed to incorporate application-specific tolerances for delays in consistency. Writes are, however, serialized using a centralized approach (per object) for coordination.

Compiler and Language-Level Support

Early work in incorporating compiler support with SDSMs [11] examined techniques by which data communication could be aggregated or eliminated entirely by understanding the specific access patterns of the application. In particular, the shortcomings of a generalized runtime system for shared memory relative to a program written for message passing is that in order to minimize the volume of data communicated, data communication is often separated from synchronization and data is fetched on demand at the granularity of the coherence unit. Compile-time analysis can identify data that will be accessed and inform the runtime system so that the data can be explicitly prefetched. Similarly, by using appropriate directives to the runtime to identify when entire coherence units are being written without being read, coherence communication can be eliminated entirely. Subsequently, several efforts have examined the use of SDSM as a back end for programming models such as OpenMP [17, 29]. They discuss the trade-offs between using a threaded (with a default of sharing the entire address space) versus a process model (with a default of private address spaces with shared data being specifically identified). Their work shows that naive implementations can perform poorly, and that identification of shared data is important to the feasibility and scalability of using programming environments such as OpenMP on SMP clusters.

Future Directions

Despite intense research and significant advances in the development of SDSM systems in the 1980s and 1990s, they remain in limited use due to their trade of scalability for the platform transparency they achieve. As multicore platforms become more ubiquitous, and as the number of cores increases, the scalability of pure hardware-based coherence is also in question, and SDSM systems may see a bigger role. Several researchers continue to explore the possibility of combining hardware and software coherence, in terms of hardware assists for a software-based coherence implementation [13] and in terms of alternating between automatic hardware coherence and software-managed incoherence based on application or compiler knowledge [24]. There is also renewed interest in the use of SDSM techniques in order to support heterogeneous platforms composed of a combination of general-purpose CPUs and accelerators [4, 32].

Related Entries

►Cache Coherence
►Distributed-Memory Multiprocessor
►Linda
►Memory Models
►Network of Workstations
►POSIX Threads (Pthreads)
►Processes, Tasks, and Threads
►Shared-Memory Multiprocessors
►SPMD Computational Model
►Synchronization

Bibliographic Notes and Further Reading

Protic et al. [30] published a compendium of works in the area of distributed shared memory circa 1994. Iftode and Singh [19] wrote an excellent survey article that encompasses both network interface advances and the incorporation of multiprocessor nodes.

Acknowledgment

This work was supported in part by NSF grants CCF-1016902, CCF-0702505, CNS-0834451, and CNS-0509270. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the granting agencies.

Bibliography

1. Adve S, Hill M (1990) Weak ordering: a new definition. In: Proceedings of the 17th annual international symposium on computer architecture, May 1990. ACM, New York, pp 2–14
2. Amza C, Cox A, Dwarkadas S, Keleher P, Lu H, Rajamony R, Zwaenepoel W (1996) Tread-marks: shared memory computing on networks of workstations. IEEE Comput 29(2):18–28
3. Bal H, Kaashoek M, Tanenbaum A (1992) Orca: a language for parallel programming of distributed systems. IEEE Trans Softw Eng 18(3):190–205
4. Becchi M, Cadambi S, Chakradhar S (2010) Enabling legacy applications on heterogeneous platforms. Poster paper, 2nd USENIX workshop on hot topics in parallelism (HOTPAR), Berkley, June 2010
5. Bershad B, Zekauskas M (1991) Midway: shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Sept 1991
6. Bilas A, Jiang D, Singh JP (2001) Accelerating shared virtual memory via general-purpose network interface support. ACM Trans Comput Syst 19:1–35
7. Blumrich M, Li K, Alpert R, Dubnicki C, Felten E, Sandberg J (1994) Virtual memory mapped network interface for the SHRIMP multicomputer. In: Proceedings of the 21st annual international symposium on computer architecture. ACM, New York, pp 142–153
8. Carter J, Bennett J, Zwaenepoel W (1991) Implementation and performance of Munin. In: Proceedings of the 13th ACM symposium on operating systems principles, ACM Press, New York, pp 152–164
9. Chen D, Tang C, Chen X, Dwarkadas S, Scott ML (2001) Beyond S-DSM: shared state for distributed systems. Technical report 744, University of Rochester, Mar 2001
10. Chen D, Tang C, Chen X, Dwarkadas S, Scott ML (2002) Multi-level shared state for distributed systems. In: International conference on parallel processing, Aug 2002, Vancouver
11. Dwarkadas S, Cox A, Zwaenepoel W (1996) An integrated compile-time/run-time software distributed shared memory system. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, pp 186–197, Oct 1996
12. Erlichson A, Nuckolls N, Chesson G, Hennessy J (1996) SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, Oct 1996. ACM Press, New York, pp 210–220
13. Fensch C, Cintra M (2008) An OS-based alternative to full hardware coherence on tiled CMPs. In: Proceedings of the fourteenth international symposium on high-performance computer architecture symposium, February 2008, Phoenix
14. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. Int J Supercomputer Appl 11(2):115–128
15. Gharachorloo K, Lenoski D, Laudon J, Gibbons P, Gupta A, Hennessy J (1990) Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of the 17th annual international symposium on computer architecture, May 1990. ACM, New York, pp 15–26
16. Gillett R (1996) Memory channel: an optimized cluster interconnect. IEEE Micro 16(2):12–18
17. Hu Y, Lu H, Cox AL, Zwaenepoel W (1999) OpenMP for networks of SMPs. In: Proceedings of the 13th international parallel processing symposium (IPPS/SPDP), Apr 1999. IEEE, New York, pp 302–310
18. Hu YC, Yu W, Cox AL, Wallach D, Zwaenepoel W (2003) Runtime support for distributed sharing in sage languages. ACM Trans Comput Syst 21(1):1–35
19. Iftode L, Singh JP (1999) Shared virtual memory: progress and challenges. Proceedings of the IEEE 87(3):498–507
20. Iftode L, Singh JP, Li K (1996) Scope consistency: a bridge between release consistency and entry consistency. In: ACM symposium on parallelism in algorithms and architectures, June 1996. ACM Press, New York, pp 277–287
21. Association (2010) InfiniBand. http://www.infinibandta.org
22. Keleher P, Cox AL, Zwaenepoel W (1992) Lazy release consistency for software distributed shared memory. In: Proceedings of the 19th annual international symposium on computer architecture, May 1992. ACM Press, New York, pp 13–21
23. Keleher P, Dwarkadas S, Cox A, Zwaenepoel W (1994) Tread-marks: distributed shared memory on standard workstations and operating systems. In: Proceedings of the 1994 winter Usenix conference, Jan 1994. USENIX Association, Berkeley, pp 115–131
24. Kelm JH, Johnson DR, Tuohy W, Lumetta SS, Patel SJ (2010) Cohesion: a hybrid memory model for accelerators. In: Proceedings of the international symposium on computer architecture (ISCA), June 2010, St Malo
25. Kontothanassis L, Hunt G, Stets R, Hardavellas N, Cierniak M, Parthasarathy S, Meira W, Dwarkadas S, Scott M (1997) VM-based shared memory on low-latency, remote-memory-access networks. In: 24th international symposium on computer architecture, June 1997. ACM Press, New York, pp 157–169
26. Lewis M, Grimshaw A (1996) The core legion object model. In: Proceedings of the 5th high performance distributed computing conference, Aug 1996, Syracuse
27. Li K (1986) Shared virtual memory on loosely coupled multiprocessors. Ph.D. thesis, Yale University
28. Li K, Hudak P (1989) Memory coherence in shared virtual memory systems. ACM Trans Comput Syst 7(4):321–359
29. Min S-J, Basumallik A, Eigenmann R (2003) Optimizing openmp programs on software distributed shared memory systems. Int J Parallel Prog 31:225–249
30. Protic J, Tomasevic M, Milutinovic V (1998) Distributed shared memory: concepts and systems. IEEE Computer Society Press, Piscataway, p 365

31. Rogerson D (1997) Inside COM. Microsoft Press, Redmond
32. Saha B, Zhou X, Chen H, Gao Y, Yan S, Rajagopalan M, Fang J, Zhang P, Ronen R, Mendelson (2009) A Programming Model for a Heterogeneous x86 Platform. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, June 2009
33. Scales D, Gharachorloo K, Aggarwal A (1998) Fine-grain software distributed shared memory on smp clusters. In: Proceedings of the fourth international symposium on high-performance computer architecture symposium, Feb 1998. ACM, New York, pp 125–136
34. Scales D, Gharachorloo K, Thekkath C (1996) Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, Oct 1996, pp 174–185
35. Schoinas I, Falsafi B, Lebeck AR, Reinhardt SK, Larus JR, Wood DA (1994) Fine-grain access control for distributed shared memory. In: Proceedings of the 6th symposium on architectural support for programming languages and operating systems, Oct 1994. ACM, New York, pp 297–306
36. Stets R, Dwarkadas S, Hardavellas N, Hunt G, Kontothanassis L, Parthasarathy S, Scott M (1997) Cashmere-2L: software coherent shared memory on a clustered remote-write network. In: Proceedings of the 16th ACM symposium on operating systems principles, Oct 1997. ACM, New York, pp 170–183
37. van Steen M, Homburg P, Tanenbaum AS (1999) Globe: a wide-area distributed system. IEEE Concurr 7(1):70–78

Sorting

Laxmikant V. Kalé[1], Edgar Solomonik[2]
[1]University of Illinois at Urbana-Champaign, Urbana, IL, USA
[2]University of California at Berkeley, Berkeley, CA, USA

Definition

Parallel sorting is a process that given n keys distributed over p processors (numbered 0 through $p-1$), migrates the keys so that all keys on processor k, for $k \in [0, p-2]$, are sorted locally and are smaller than or equal to all keys on processor $k+1$.

Discussion

Introduction

Parallel sorting algorithms have been studied in a variety of contexts. Early studies in parallel sorting addressed the theoretical problem of sorting n keys distributed over n processors, using fixed interconnection networks. Modern parallel sorting algorithms focus on the scenario where n is much larger than the number of processors, p. However, even contemporary algorithms need to satisfy an array of use-cases and architectures, so various parallel sorting techniques have been analyzed for GPU-based sorting, shared memory sorting, distributed memory sorting, and external memory sorting. On most if not all of these architectures, parallel sorting is typically dominated by communication, in particular, the movement of data values associated with the keys.

Parallel sorting has a wide breadth of practical applications. Some scientific applications in the field of high-performance computing (e.g., ChaNGa) perform sorting each iteration, placing high demand on good scalability and adaptivity of parallel sorting algorithms. Parallel sorting is also utilized in the commercial field for processing of numeric as well as nonnumeric data (i.e., parallel database queries). Moreover, Integer Sort is one of the NAS parallel benchmarks.

Parallel Sorting Algorithms

There are a few important parallel sorting algorithms that dominate multiple use-cases and serve as building blocks for more specialized sorting algorithms. The algorithms will be described for the distributed memory paradigm, though most are also applicable to other architectures and models.

Parallel Quicksort

Parallelization of Quicksort can be done in a variety of ways. A simplified, recursive version is described below. A more thorough analysis can be found in [1] or [2].

1. A processor broadcasts a pivot element to all p processors.
2. Each processor then splits its keys into two sections (smaller keys and larger keys) according to the pivot.
3. Two prefix sums calculate the total number of smaller keys and larger keys on the first k processors, for $k \in [0, p-1]$. If, $small_k$ and $large_k$ are, respectively, the total numbers of smaller and larger keys on processors 0 through k, then, after this operation, processor k knows $small_{k-1}$ and $large_{k-1}$ as well as $small_k$ and $large_k$.
4. Processor $p-1$ knows the total sums, $small_{p-1}$ and $large_{p-1}$. It can therefore divide the set of processors

in proportion with $small_{p-1}$ and $large_{p-1}$, thus determining two sets of processors (P_s and P_l) that should be given the smaller and larger keys. This processor should also broadcast the average number of keys these two sets of processors should receive ($small_{avg}$ and $large_{avg}$).

5. Each processor k can now decide where its keys need to be sent. For example, the smaller keys should go to processor $\left\lfloor \frac{small_{k-1}}{small_{avg}} \right\rfloor$ through processor $\left\lfloor \frac{small_k}{small_{avg}} \right\rfloor$.
6. After the communication, it is sufficient for the Parallel Quicksort procedure to recurse on the two processor sets (P_s and P_l) until all data values are on the correct processors and can be sorted locally.

This algorithm generally achieves good load balance by determining the correct portions of processors that should receive smaller and larger keys. However, the necessity of moving half the data for every iteration is costly on large distributed systems. In the average case, Parallel Quicksort necessitates $\Theta(n \log p)$ data movement. Moreover, both the quality of load balance achieved for small processor sets and the total number of recursive levels are dependent on pivot selection. Efficient implementations of Parallel Quicksort typically use more complex pivoting techniques and multiple pivots [2]. Nevertheless, Parallel Quicksort is easy to implement and can achieve good performance on some shared memory and smaller distributed systems. Additionally, the number of messages sent by each processor in step 5 is constant – typically no more than four. Therefore, Parallel Quicksort has a relatively small message latency cost of $\Theta(p \log p)$ messages.

Bitonic Sort

Introduced in 1968 by Batcher [3], Bitonic Sort is one of the oldest parallel sorting algorithms. This algorithm is based on the sorting of *bitonic sequences*. A *bitonic sequence* is a sequence S or any cyclic shift of S, such that $S = S_1 S_2$ where S_1 is monotonically nondecreasing and S_2 is monotonically nonincreasing. Further, any unsorted sequence can be treated as a series of bitonic subsequences of length two. A *bitonic merge* turns a bitonic subsequence into a fully sorted subsequence. Bitonic Sort works by application of a series of bitonic merges until the entire sequence is sorted. Applying bitonic merges on a series of bitonic subsequences effectively doubles the length of each sorted subsequence and cuts the number of bitonic subsequences in half. Therefore, for an unsorted sequence of length k, where k is a power of two, Bitonic Sort requires $\log k$ merges.

The main insight of Bitonic Sort is in the bitonic merge operation. A bitonic merge recursively slices a bitonic sequence into two bitonic sequences, with all elements in one sequence larger than all elements in the other. When the bitonic sequence is sliced into pieces of unit length, the entire input is sorted. Given a bitonic sequence of length s, where s is a power of two, every slice operation compares and swaps each element k, where $k \in [0, s/2)$ with element $k + s/2$. These swaps result in two bitonic sequences of length $s/2$, with the elements in one being larger than the elements of the other. Thus, if s is a power of two, a bitonic merge requires $\log s$ slices, which amounts to $\log s$ comparison operations on every element.

In the case of $n = p$ on a hypercube network, a bitonic merge requires $\Theta(\log n)$ swaps, one swap in each hypercube dimension. The algorithm requires $\log n$ merges on such a network, for a composed running time of $\Theta(\log^2 n)$. Adaptive Bitonic Sorting [4] avoids the redundant comparisons in Bitonic Sort and achieves a runtime complexity of $\Theta(\log n)$. Further, Bitonic Sort is also asymptotically optimal on mesh connected networks [5].

As proposed by Blelloch [6], Bitonic Sort can be extended for the case of $n \geq p$, by introducing virtual hypercube dimensions within each processor. Alternatively, a more efficient sequential sorting algorithm can be used for the sequential sorting work. Though historically significant and elegant in nature, Bitonic Sort is not widely used on modern supercomputers, since the algorithm requires data to be migrated through the network $\Theta(\log^2 p)$ times or, in the case of Adaptive Bitonic Sorting, $\Theta(\log p)$ times. Nevertheless, Bitonic Sort has been used in a wide variety of applications, ranging from network router hardware to sorting on GPUs [7–9]. A more thorough analysis and justification of correctness of this algorithm can be found in the Bitonic Sort article.

Parallel Radix Sort

Radix Sort is a counting-based sorting algorithm that relies on the bitwise representation of keys. An r-bit radix looks at r bits of each key at a time and permutes the keys to move to one of the 2^r buckets, where the r-bit value of each key corresponds to its destination bucket. If each key has b bits, by looking at the r least-significant bits first, Radix Sort can sort the entire

dataset in $\lceil \frac{b}{r} \rceil$ permutations. Therefore, this algorithm has a complexity of $\Theta\left(\frac{b}{r}n\right)$. Notably, this complexity is linear with respect to n, a feature that cannot be matched by comparison-based sorting algorithms, which must do at least $\Theta(n \log n)$ comparison operations. However, Radix Sort is inherently cache-inefficient since each key may need to go to any one of the 2^r buckets for each iteration, independent of which bucket it resided in the previous iteration [10]. A further limitation on Radix Sort is its reliance on the bitwise representation of keys, which is satisfied for integers and requires a simple transformation for floats, but is not necessarily possible or simple for strings and other data types.

Parallel Radix Sort is implemented by assigning a subset of the buckets to each processor [6]. Thus, each of $\lceil \frac{b}{r} \rceil$ permutations would result in a step of all-to-all communication. Load balancing is also achieved relatively easily by computing a histogram of the number of keys headed for each bucket at the beginning of each step. These histograms can be computed using local data on each processor first, then summed up using a reduction. Given a summed-up histogram, a single processor can then adaptively decide the set of buckets to assign to each processor and broadcast this information to all other processors. Each processor then sends all of its keys to the processor that owns the appropriate bucket for each key, using all-to-all communication.

A good way to improve the efficiency of local histogram computation in Radix Sort is to use an auxiliary set of low-bit counters [11]. Given a large r, it is unlikely that an array of 2^r 32-bit counters can fit into the L1 cache. However, by using an array of 2^r 8-bit counters and incrementing the 32-bit counter array only when the 8-bit counters are about to overflow, cache performance can be significantly improved.

The simplicity of Radix Sort as well as its relatively good all-around performance and scalability have made it a popular parallel sorting algorithm in a variety of contexts. However, Radix Sort still suffers from cache-efficiency problems and requires multiple steps of all-to-all communication, which can be an extremely costly operation on a large enough system.

Sample Sort

Sample Sorting is a *splitter-based* method which performs data partitioning by collecting a sample of the entire dataset [12]. *Splitter-based* parallel sorting algorithms determine the destinations for each key by determining a range bounded by two *splitters* for each processor. These *splitters* are simply values meant to subdivide the entire key range into p approximately equal chunks, so that each processor can be assigned a roughly even-sized chunk. After the splitters have been determined and broadcasted to all processors, a single all-to-all communication step suffices in giving each processor the correct data.

Parallel Sorting by Regular Sampling is a Sample Sorting algorithm introduced by Shi and Schaeffer [13]. This algorithm determines the correct splitters by collecting a sample of data from each processor. Sorting by Regular Sampling uses a regular sample of size $p-1$ and generally operates as follows:

1. Sort local data on each processor.
2. Collect sample of size $p-1$ on each processor with the kth element of each sample as element $\frac{n}{p} \times \frac{k+1}{p}$ of the local data.
3. Merge the p samples to form a combined sorted sample of size $p \times (p-1)$.
4. Define $p-1$ splitters with the kth splitter as element $p \times \left(k + \frac{1}{2}\right)$ of the sorted sample.
5. Broadcast splitters to all processors.
6. On each processor, subdivide the local keys into p chunks (numbered 0 through $p-1$) according to the splitters. Send each chunk to equivalently numbered processor.
7. Merge the incoming data on each processor.

Collecting a regular sample of size $p-1$ from each processor has been proven to guarantee no more than $\frac{2n}{p}$ elements on any processor [14] and shown to achieve almost perfect load balance for most practical distributions. The algorithm has also been shown to be asymptotically optimal as long as $n \geq p^3$.

Regular Sample Sort is easy to implement, insensitive to key distribution, and optimal in terms of the data movement required (there is a single all-to-all step, so each key gets moved only once). The algorithm has been shown to perform very well for $n \gg p$ and has been the parallel sorting algorithm of choice for many modern applications. One important issue with the traditional Sorting by Regular Sampling technique is the requirement of a combined sample of size $\Theta(p^2)$. For a small-enough p this is not a major cost; however, for high-performance computing applications running on thousands of processors, this cost begins to overwhelm

the running time of the sorting algorithm and the $n \geq p^3$ assumption crumbles.

Sorting by Random Sampling [6] is a parallel sorting technique that can potentially alleviate some of the drawbacks of Parallel Sorting by Regular Sampling. Instead of selecting an evenly distributed sample of size $p - 1$ from each processor, random samples of size s are collected from the initial local datasets to form a combined sample of size $s \times p$. The s parameter has to be carefully chosen, but sometimes sufficient load balance can be achieved for $s < p$. Additionally, Sorting by Random Sampling allows for better potential overlap between computation and communication since the sample can be collected before the local sorting is done.

Another interesting variation of Sample Sort was introduced by Helman et al. [15]. Instead of collecting a sample, this sorting procedure first permutes the elements randomly with a randomized data transpose, then simply selects the splitters on one processor. To execute the transpose, each processor randomly assigns each of its local keys to one of the p buckets, then sends the jth bucket to the jth processor. With high probability, this permutation guarantees that any processor's elements will be representative of the entire key set. Thus, this algorithm avoids the cost of collecting and analyzing a large sample on a single processor. One processor still needs to select and broadcast splitters but this is a relatively cheap operation. The main disadvantage of this technique is the extra all-to-all communication round, which is very expensive on a large system. Additionally, as p scales to n/p, the load balance achieved by the algorithm deteriorates.

Histogram Sort

Histogram Sort [16] is another splitter-based method for parallel sorting. Like Sample Sort, Histogram Sort determines a set of $p - 1$ splitters to divide the keys into p evenly sized sections. However, it achieves this task by taking an iterative guessing approach rather than simply collecting one big sample. Each set of splitter-guesses, called the *probe*, is matched up to the data then adjusted, until *satisfactory values* for all splitters have been determined. A *satisfactory value* for the kth splitter needs to divide the data so that approximately $\frac{k+1}{p} \times n$ keys are smaller than the splitter value. Typically, a threshold range is established for each splitter so that the splitter-guesses can converge quicker. The kth splitter must divide the data within the range of keys, $\left(\frac{nk}{p} - \frac{nT}{p}, \frac{nk}{p} + \frac{nT}{p}\right)$, where T is the given threshold. A basic implementation of Histogram Sort operates as follows:

1. Sort local data on each processor.
2. Define a probe of $p - 1$ splitter-guesses distributed evenly over the key data range.
3. Broadcast the probe to all processors.
4. Produce local histograms by determining how much of each processor's local data fits between each key range defined by the splitter-guesses.
5. Sum up the histograms from each processor using a reduction to form a complete histogram.
6. Analyze the complete histogram on a single processor, determining any splitter values satisfied by a splitter-guess, and bounding any unsatisfied splitter values by the closest splitter-guesses.
7. If any splitters have not been satisfied, produce a new probe and go back to step 3.
8. Broadcast splitters to all processors.
9. On each processor, subdivide the local keys into p chunks (numbered 0 through $p - 1$) according to the splitters. Send each chunk to equivalently numbered processor.
10. Merge the incoming data on each processor.

This iterative technique can refine the splitter-guesses to an arbitrarily narrow threshold range. Quick convergence can be guaranteed by defining each new probe to contain a guess in the middle of the bounded range for each unsatisfied splitter.

Like any splitter-based sort, Histogram Sort is optimal in terms of the data movement required. However, unlike all previously described sorting algorithms, the running time of Histogram Sort depends on the distribution of the data through the data range. The iterative approach employed by this sorting procedure guarantees desired level of load balance which Sample Sort and Radix Sort cannot. Additionally, the probing technique is flexible and does not require that local data be sorted immediately [17]. This advantage allows an excellent opportunity for the exploitation of communication and computation overlap. However, Histogram Sort is generally more difficult to implement than common alternatives such as Radix Sort or Sample Sort.

Architectures and Theoretical Models

Parallelism is exhibited by a variety of computer architectures. Shared memory multiprocessors, distributed systems, supercomputers, sorting networks, and GPUs are all fundamentally different parallel computing constructions. As such, parallel sorting algorithms need to be designed and tuned for each of these architectures specifically.

Sorting Networks and Early Theoretical Models

Traditional parallel sorting targeted the problem of sorting n numbers on n processors using a fixed interconnection network. Bitonic Sort [3] was an early success as it provided a $\Theta(\log^2 n)$-depth sorting network. The bitonic sorting network also yielded a practical algorithm for sorting n keys in parallel using n processors in $\Theta(\log^2 n)$ time on network topologies such as the hypercube and shuffle-exchange. In 1983, Ajtai et al. [18], introduced an $\Theta(\log n)$-depth sorting network capable of sorting n keys in $\Theta(\log n)$ time using $\Theta(n \log n)$ comparators. However, this construction was shown to lead to less-efficient networks than Bitonic Sort for reasonable values of n. Leighton [19] introduced the Column Sort algorithm. He showed that, based on Column Sort, for any sorting network with $\Theta(n \log n)$ comparators and $\Theta(\log n)$ depth, one can construct a corresponding constant-degree network of n processors that can sort in $\Theta(\log n)$ time.

Much work has targeted the complexity of parallel sorting on the less-restrictive PRAM model where each processor can access the memory of all other processors in constant time. In 1986, Cole [20] introduced an efficient and practical parallel sorting algorithm with versions for the CREW (concurrent read only) and EREW (no concurrent access) PRAM models. This algorithm was based on a simple tree-based Mergesort, but was elegantly pipelined to achieve a $\Theta(\log n)$ complexity for sorting n keys using n processors. Cole's merge sort used a $\Theta(\log n)$ time merging algorithm, which naturally leads to a $\Theta(\log^2 n)$ sorting algorithm. However, his sorting algorithm used results from lower levels of the merge tree to partially precompute the merging done in higher levels of the merge tree. Thus, the sorting algorithm was designed so that at every node of the merge tree only a constant amount of work needed to be done, yielding a $\Theta(\log n)$ overall sorting complexity.

Sorting networks have also been extensively studied for the VLSI model of computation. The VLSI model focuses on area-time complexity, that is, the area of the chip on which the network is constructed and the running time. A good analysis of lower bounds for the complexity of such VLSI sorters can be found in [21].

These theoretical methods have been studied intensively in literature and have yielded many elegant parallel sorting algorithms. However, the theoretical machine models they were designed for are no longer representative or directly useful for current parallel computer architectures. Nevertheless, these studies provide valuable groundwork for modern and future parallel sorting algorithms. Moreover, sorting networks may prove to be useful for emerging architectures such as GPUs and chip multicores.

GPU-Based Sorting

Early GPU-based sorting algorithms utilized a limited graphics API which, among other restrictions, did not allow scatter operations and made Bitonic Sort the dominant choice. GPUTeraSort [7] is an early efficient hybrid algorithm that uses Radix Sort and Bitonic Sort. GPUTeraSort was designed for GPU-based external sorting, but is also general to in-memory GPU-based sorting. A weakness of the GPUTeraSort algorithm is its $\Theta(n \log^2 n)$ running time, typical of parallel sorting algorithms based on Bitonic Sort. GPU-ABiSort [8] improved over GPUTeraSort by using Adaptive Bitonic Sorting [4], lowering the theoretical complexity to $\Theta(n \log n)$ and often demonstrating a lower practical running time.

Newer GPUs, assisted by the CUDA software environment, allow for efficient scan primitives and a much broader set of parallel sorting algorithms. Efficient versions of Radix Sort and a parallel Mergesort are presented by Satish et al. [9]. Newer GPU-based sorting algorithms also exploit instruction-level parallelism by performing steps such as merging using custom vector operations. An array of various other GPU-based sorting algorithms, which are not detailed here, can be found in literature.

Current results on modern GPUs suggest that Radix Sort typically performs best, particularly when the key size is small [9]. Radix Sort is well fit for GPU execution since keys can be processed independently and synchronization is almost purely in the form of prefix sums,

which can be executed with high efficiency on GPUs. Radix Sort also requires few or no low-level branches, unlike comparison-based sorting algorithms. Finally, the cache inefficiency of Radix Sort has been less costly on GPUs since most GPUs have no cache. However, newer GPUs, such as the NVIDIA GPUs of compute capability 2.0, already have small caches and are rapidly evolving. It is hard to predict whether Radix Sort or a different sorting algorithm will prove most efficient on emerging GPU and accelerator architectures.

Shared Memory Sorting

Parallel merging techniques have commonly been used to produce simple shared memory parallel sorting algorithms. Francis and Mathieson [22] present a k-processor merge that allows for all processors to participate in a parallel merge tree, with two processors participating in each merge during the first merging stage and all processors participating in the final merge at the head of the tree. Good performance is achieved by subdividing each of the two arrays of size a and b being merged into k sections, where k is the number of processors participating in the merge. The subdivisions are selected so that the ith processor merges the ith section of each of the two arrays and produces elements $\frac{i}{k}(a+b)$ through $\frac{i+1}{k}(a+b)$ of the merged array.

Merge-based algorithms, such as the one detailed above, as well as parallel versions of Quicksort and Mergesort, are predominant on contemporary shared memory multiprocessor architectures. However, with the advent of increased parallelism in chip architectures, techniques such as sampling and histogramming may become more viable.

Distributed Memory Sorting

Parallel Sorting algorithms for distributed memory architectures are typically used in the high-performance computing field, and often require good scaling on modern supercomputers. In the 1990s Radix Sort and Bitonic Sort were widely used. However, as architectures evolved, these sorting techniques proved insufficient. Modern machines have thousands of cores, so, to achieve good scaling interprocessor communication needs to be minimized. Therefore, splitter-based algorithms such as Sample Sort and Histogram Sort are now more commonly used for distributed memory sorting. Splitter-based parallel sorting algorithms have minimal communication since they only move data once.

Future Directions

Despite the extraordinarily large amount of literature on parallel sorting, the demand for optimized parallel sorting algorithms continues to motivate novel algorithms and more research on the topic. Moreover, the continuously changing and, more recently, diverging nature of parallel architectures has made parallel sorting an evolving problem.

High-performance computer architectures are rapidly growing in size, creating a premium on parallel sorting algorithms that have minimal communication and good overlap between computation and communication. Since parallel sorting necessitates communication between all processors, the creation and optimization of topology-aware all-to-all personalized communication strategies is extremely valuable for many splitter-based parallel sorting algorithms. As previously mentioned, algorithms similar to Sample Sort and Histogram Sort are the most viable candidates for this field due to the minimal nature of the communication they need to perform.

Shared memory sorting algorithms are beginning to face a challenge that most modern sequential algorithms have to endure. The demand for good cache efficiency is now a key constraint for all algorithms due to the increasing relative cost of memory accesses. Parallel sorting is being studied under the cache-oblivious model [23] in an attempt to reevaluate previously established algorithms and introduce better ones.

Currently, parallel sorting using accelerators, such as GPUs, is probably the most active of all parallel sorting research areas due to the rapid changes and advances happening in the accelerator architecture field. Little can be said about which algorithms will dominate this field in the future, due to the influx of novel GPU-based sorting algorithms and newer accelerators in recent years.

The wide use of sorting in computer science along with the popularization of parallel architectures and parallel programming necessitates the implementation of parallel sorting libraries. Such libraries can be difficult to standardize, however, especially

for the efficiency-sensitive field of high-performance computing. Nevertheless, these libraries are quickly emerging, especially under the shared memory computing paradigm.

Related Entries

►Algorithm Engineering
►All-to-All
►Bitonic Sort
►Bitonic Sorting, Adaptive
►Collective Communication
►Data Mining
►NAS Parallel Benchmarks
►PRAM (Parallel Random Access Machines)

Bibliographic Notes and Further Reading

The literature on parallel sorting is very large and this entry is forced to cite only a select few. The sources cited are a mixture of the largest impact publications and the most modern publications. A few of the sources also give useful information on multiple parallel sorting algorithms. Blelloch et al. [6] provide an in-depth experimental and theoretical comparative analysis of a few of the most important distributed memory parallel sorting algorithms. Satish et al. [9] provide good analysis of a few of the most modern GPU-sorting algorithms. Vitter [24] presents a good analysis of external memory parallel sorting.

Bibliography

1. Kumar V, Grama A, Gupta A, Karypis G (1994) Introduction to parallel computing: design and analysis of algorithms. Benjamin-Cummings, Redwood City, CA
2. Sanders P, Hansch T (1997) Efficient massively parallel quicksort. In: IRREGULAR '97: Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel, pp 13–24. Springer-Verlag, London, UK
3. Batcher K (1968) Sorting networks and their application. Proc. SICC, AFIPS 32:307–314
4. Bilardi G, Nicolau A (1986) Adaptive bitonic sorting: an optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY
5. Thompson CD, Kung HT (1977) Sorting on a mesh-connected parallel computer. Commun ACM 20(4):263–271
6. Blelloch G et al. (1991) A comparison of sorting algorithms for the Connection Machine CM-2. In: Proceedings of the Symposium on Parallel Algorithms and Architectures, July 1991
7. Govindaraju N, Gray J, Kumar R, Manocha D (2006) Gputerasort: high performance graphics co-processor sorting for large database management. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp 325–336. ACM, New York
8. Greb A, Zachmann G (2006) Gpu-abisort: optimal parallel sorting on stream architectures. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, 20th International, pp 45–54, April 2006
9. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore gpus. Parallel and Distributed Processing Symposium, International, Rome, pp 1–10
10. LaMarca A, Ladner RE (1997) The influence of caches on the performance of sorting. In: SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, pp 370–379. Society for Industrial and Applied Mathematics, Philadelphia, PA
11. Thearling K, Smith S (1992) An improved supercomputer sorting benchmark. In: Proceedings of the Supercomputing, November 1992.
12. Huang JS, Chow YC (1983) Parallel sorting and data partitioning by sampling. In: Proceedings of the Seventh International Computer Software and Applications Conference, November 1983
13. Shi H, Schaeffer J (1992) Parallel sorting by regular sampling. J Parallel Distrib Comput 14:361–372
14. Li X, Lu P, Schaeffer J, Shillington J, Wong PS, Shi H (1993) On the versatility of parallel sorting by regular sampling. Parallel Comput 19(10):1079–1103
15. Helman DR, Bader DA, JáJá J (1998) A randomized parallel sorting algorithm with an experimental study. J Parallel Distrib Comput 52(1):1–23
16. Kale LV, Krishnan S (1993) A comparison based parallel sorting algorithm. In: Proceedings of the 22nd International Conference on Parallel Processing, pp 196–200, St. Charles, IL, August 1993
17. Solomonik E, Kale LV (2010) Highly scalable parallel sorting. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Urbana, IL, April 2010
18. Ajtai M, Komlós J, Szemerédi E (1983) Sorting in c log n parallel steps. Combinatorica 3(1):1–19
19. Leighton T (1984) Tight bounds on the complexity of parallel sorting. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp 71–80. ACM, New York
20. Cole R (1986) Parallel merge sort. In: SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science, pp 511–516. IEEE Computer Society, Washington, DC
21. Bilardi G, Preparata FP (1986) Area-time lower-bound techniques with applications to sorting. Algorithmica 1(1):65–91
22. Francis RS, Mathieson ID (1988) A benchmark parallel sort for shared memory multiprocessors. Comput IEEE Trans 37(12):1619–1626

23. Blelloch GE, Gibbons PB, Simhadri HV (2009) Brief announcement: low depth cache-oblivious sorting. In: SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, pp 121–123. ACM, New York
24. Vitter JS (2001) External memory algorithms and data structures: dealing with massive data. ACM Comput Surv 33(2):209–271

Space-Filling Curves

Michael Bader[1], Hans-Joachim Bungartz[2], Miriam Mehl[2]
[1]Universität Stuttgart, Stuttgart, Germany
[2]Technische Universität München, Garching, Germany

Synonyms

FASS (space-**f**illing, self-**a**voiding, **s**imple, and self-**s**imilar)-curves

Definition

A space-filling curve is a continuous and surjective mapping from a 1D parameter interval, say $[0,1]$, onto a higher-dimensional domain, say the unit square in 2D or the unit cube in 3D. Although this, at first glance, seems to be of a purely mathematical interest, space-filling curves and their recursive construction process have obtained a broad impact on scientific computing in general and on the parallelization of numerical algorithms for spatially discretized problems in particular.

Discussion

Introduction

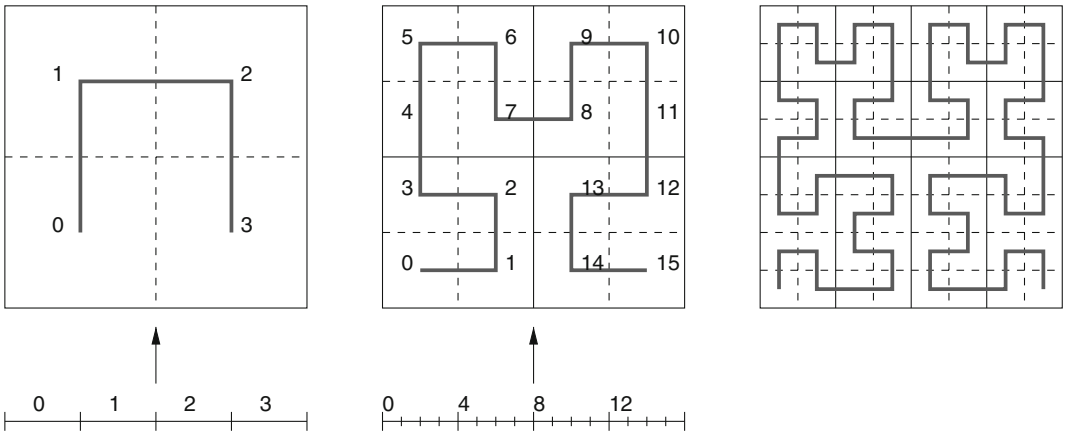
Space-filling curves (SFC) were presented at the end of the nineteenth century – first by Peano (1890) and Hilbert (1891), and later by Moore, Lebesgue, Sierpinski, and others. The idea that some curves (i.e., something actually one-dimensional) may completely cover an area or a volume sounds somewhat strange and formerly caused mathematicians to call them "topological monsters." The construction of all SFC follows basically the same principle: start with a *generator* indicating an order of traversal through the similar first-level substructures of the initial domain (the unit square, unit cube, etc.), and produce the next iterates by successively subdividing the domain in the same way as well as placing and connecting shrinked, rotated, or reflected versions of the generator in the next-level subdomains. This has to happen in an appropriate way, ensuring the two properties *neighborhood* (neighboring subintervals are mapped to neighboring subdomains) and *inclusion* (subintervals of an interval are mapped to subdomains of the interval's image). If all is done properly, it can be proven that the limit of this recursive process in fact defines a curve that completely fills the target domain and, hence, results in an SFC.

While the SFC itself as the asymptotical result of a continuous limit process is more a playground of mathematics, the iterative or recursive construction or, to be precise, the underlying mapping can be used for sequentializing higher-dimensional domains and data. Roughly speaking, these higher-dimensional data (elements of a finite element mesh, particles in a molecular dynamics simulation, stars in an astrophysics simulation, pixels in an image, voxels in a geometric model, or even entries in a data base, e.g.) now appear as pearls on a thread. Thus, via the locality properties of the SFC mapping, clusters of data can be easily identified (interacting finite elements, neighboring stars, similar data base entries, etc.). This helps for the efficient processing of tasks such as answering data base queries, defining hardware-aware traversal strategies through adaptively refined meshes or heterogeneous particle sets, and, of course, subdividing the data across cores or processors in the sense of (static or dynamic) load distribution in parallel computing. The main idea for the latter is that the linear (1D) arrangement of the data via the mapping basically reduces the load distribution problem to the sorting of indices.

Construction

Space-filling curves are typically constructed via an iterative or recursive process. The source interval and the target domain are recursively substructured into smaller intervals and domains. From each level of recursion to the next, a mapping between subintervals and subdomains is constructed, where the child intervals of a subinterval are typically mapped to the children of the image of the parent interval. The SFC is then defined as the image of the limit of these mappings.

Figure 1 illustrates this recursive construction process for the 2D Hilbert curve. From each level to the next, the subintervals are split into four congruent



Space-Filling Curves. Fig. 1 The first three iterations of the Hilbert curve

subintervals. Likewise, the square subdomains are split into four subsquares. In the nth iteration, an interval $[i \cdot 4^{-n}, (i+1) \cdot 4^{-n}]$ is mapped to the ith subsquare, as indicated in the figure. The curves in Fig. 1 connect the subsquares of the nth level according to their source intervals and are called *iterations* of the Hilbert curve. For the limit $n \to \infty$, the iterations shall, in an intuitive sense, converge to the Hilbert curve.

More formal: for any given parameter $t \in [0,1]$, there exists a sequence of nested intervals $[i_n \cdot 4^{-n}, (i_n+1) \cdot 4^{-n}]$ that all contain t. The corresponding subsquares converge to a point $h(t) \in [0,1]^2$. The image of the mapping h defined by that construction is called the Hilbert curve. h shall be called the Hilbert mapping.

Computation of Mappings

Figure 1 shows that the nth Hilbert iteration consists of the connection of four $(n-1)$th Hilbert iterations, which are scaled, rotated, and translated appropriately. For $n \to \infty$, this turns into a fix-point argument: the Hilbert curve consists of the connection of four suitably scaled, rotated, and translated Hilbert curves. The respective transformations shall be given by operations H_q, where $q \in \{0,1,2,3\}$ determines the relative position of the transformed Hilbert curve. This leads to the following recursive equation:

$$h(0_4.q_1q_2q_3q_4\ldots) = H_{q_1} \circ h(0_4.q_2q_3q_4\ldots). \quad (1)$$

If the parameter t is given as a quarternary fraction, i.e., $t = 0_4.q_1q_2q_3\cdots = \sum_n q_n \frac{1}{4^n}$, then the interval numbers i_n and the relative position of subintervals within their parent can be obtained from the quarternary digits q_n.

For finite quarternary fractions, successive application of Eq. 1 leads to the following formula to compute h:

$$h(0_4.q_1q_2\ldots q_n) = H_{q_1} \circ H_{q_2} \circ \ldots \circ H_{q_n} \circ h(0). \quad (2)$$

For the Hilbert curve, the operators H_q are defined as:

$$H_0 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}y \\ \frac{1}{2}x \end{pmatrix}$$

$$H_1 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}x \\ \frac{1}{2}y + \frac{1}{2} \end{pmatrix}$$

$$H_2 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}x + \frac{1}{2} \\ \frac{1}{2}y + \frac{1}{2} \end{pmatrix}$$

$$H_3 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} -\frac{1}{2}y + 1 \\ -\frac{1}{2}x + \frac{1}{2} \end{pmatrix}$$

Equations 1 and 2 may be easily turned into an algorithm to compute the image point $h(t)$ from any given parameter t.

Inverting this process leads to algorithms that find a parameter t that is mapped to a given point $p = h(t)$. However, note that the Hilbert mapping h is not bijective; hence, an inverse mapping h^{-1} does not exist. Still, it is possible to construct mappings $\tilde{h}^{-1}$ that return a uniquely defined parameter $t = \tilde{h}^{-1}(p)$ with $p = h(t)$. Note that for practical applications, only the discrete orders induced by h are of interest. These orders are usually bijective – for example, the relation between

subsquares and subintervals during the construction of the Hilbert mapping is a bijective one. Bijectivity is only lost with $n \to \infty$.

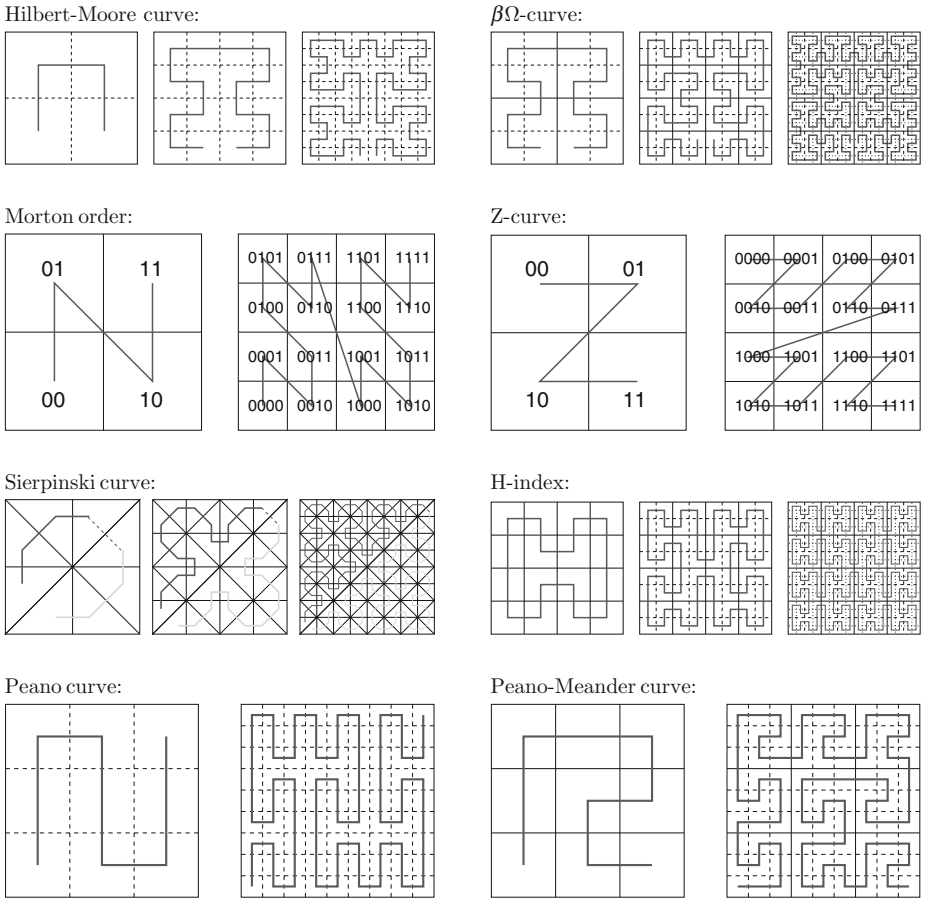
Examples of Space-Filling Curves

Figure 2 illustrates the construction of different SFC. All of them are constructed in a similar way as the Hilbert curve and can be computed via an analogous approach. The Hilbert-Moore curve combines four regular, scaled-down Hilbert curves to a closed curve. The $\beta\Omega$-curve is a Hilbert-like curve that uses nonuniform refinement patterns throughout the iterations. Similar to the Hilbert-Moore curve, it is a closed curve. Morton order and the Z-curve result from a bit-interleaving code for 2D grids. They lead to discontinuous mappings from the unit interval to the unit square. However, the Lebesgue curve uses the same construction, but maps the Cantor set to the unit square in order to obtain continuity. The Sierpinski curve is a curve that is generated via recursive substructuring of triangles. The combination of two triangle-filling Sierpinski curves leads to a curve that fills the unit square. The H-index follows a construction compatible to that for the Sierpinski curve, but generates a discrete order on Cartesian grids. Infinite refinement of the H-index, then, leads to the Sierpinski curve. The Peano curve, finally, as well as its variant, the Peano-Meander curve, are square-filling curves that are based on a recursive 3×3-refinement of the unit square.

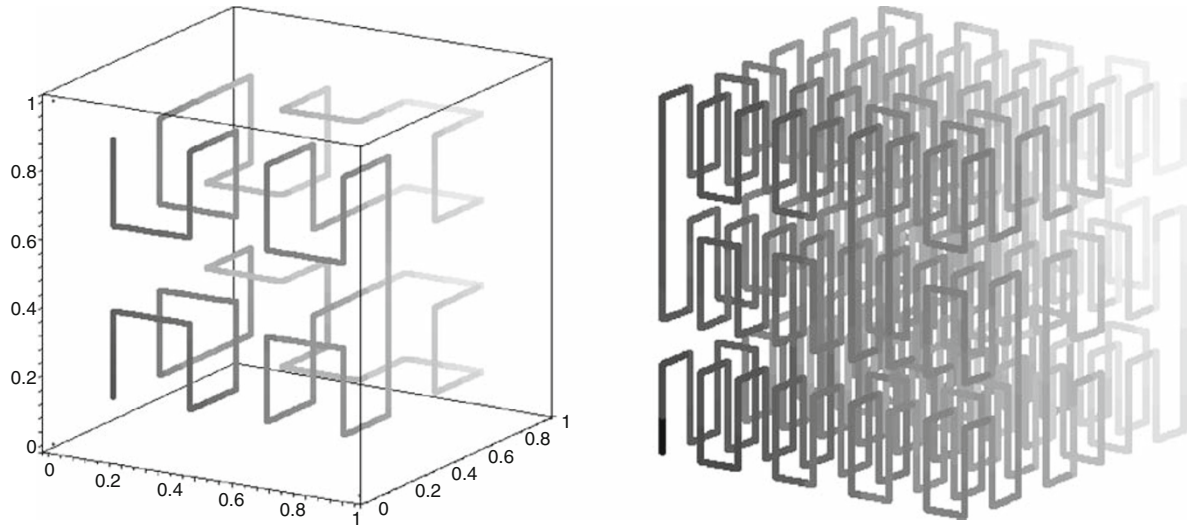
Figure 3 provides snapshots of the construction processes of 3D Hilbert and Peano curves.

Locality Properties of Space-Filling Curves

The recursive construction of SFC leads to locality properties that can be exploited for efficient load distribution and load balancing. The Hilbert curve, for example,



Space-Filling Curves. Fig. 2 Several examples for the construction of space-filling curves on the unit square



Space-Filling Curves. Fig. 3 Second iterations of three-dimensional Hilbert and Peano curves

and curves that follow a similar construction, can be shown to be Hölder continuous, i.e., for two parameters t_0 and t_1, the distance of the images $h(t_0)$ and $h(t_1)$ is bounded by

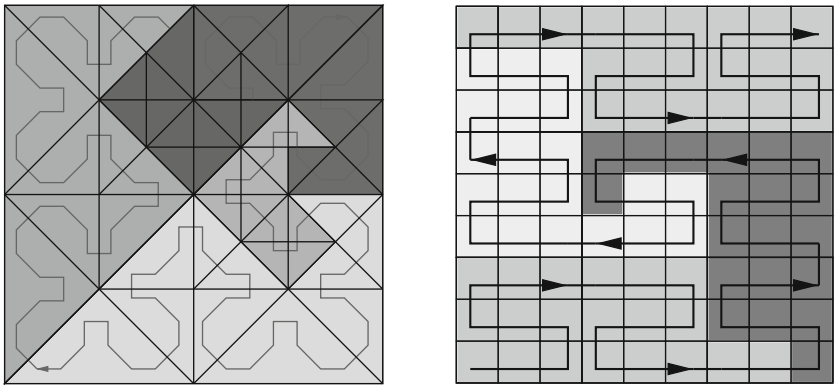
$$\|h(t_0) - h(t_1)\|_2 \leq C\,|t_0 - t_1|^{1/d}, \qquad (3)$$

where d denotes the dimension. As $|t_0 - t_1|$ is also equal to the area covered by the space-filling curve segment defined by the parameter interval $[t_0, t_1]$ (i.e., h is parameterized by area or volume), Eq. 3 gives a relation between the area covered by a curve segment and the distance between the end points of the curve. It is thus a measure for the compactness of partitions defined by curve segments. For d-dimensional objects such as spheres or cubes, the volume typically grows with the dth power of the extent (diameter, e.g.,) of the object. Hence, an exponent of d^{-1} is asympotically optimal, and the constant C characterizes the compactness of an SFC.

High Performance Computing and Load Balancing with Space-Filling Curves

The ability of recursively substructuring and linearizing high-dimensional domains is mainly responsible for the revival of SFC – or their arrival in computational applications – starting in the 1960s of the last century. As some milestones, the Z-order (Morton-order) curve has been proposed for file sequencing in geodetic data bases in 1966 by G.M. Morton [15]; *quadtrees* (nothing else than 2D Lebesgue SFC) were introduced in image processing [9, 19]; *octrees*, their 3D counterparts, marked the crucial idea for breakthroughs in the complexity of particle simulations, the *Barnes Hut* [3] as well as the *Fast Multipole* algorithms [11]. Octrees were also successfully used in CAD [10, 12], as well as for organizational tasks such as grid generation [20, 21] or steering volume-oriented simulations [16], and the portfolio of SFC of practical relevance is still widening.

The locality properties of SFC yield a high locality of data access and, therewith, a high efficiency of cache-usage for various kinds of applications. This will be a crucial aspect for future computing architectures, in particular multi-core architectures, where the memory bottleneck will become even more severe than already for today's high-performance computers. For example, the Peano curve is used for highly efficient block-structured matrix–matrix products [2], and it can serve as a general paradigm for PDE frameworks, where all the steps of geometry representation, adaptive grid generation, grid traversal, data structure design, and parallelization follow an SFC-based strategy [8, 13]. While all the SFC discussed so far refer to structured Cartesian grids or subdomain structures, the Sierpinski curve is defined on a triangular master domain and has been recently used to benefit from SFC characteristics also in the context of managing, traversing, and distributing triangular finite element meshes [1].



Space-Filling Curves. Fig. 4 Partitioning of a triangulation according to the Sierpinski curve and a Cartesian grid according to the Peano curve

Hilbert or Hilbert-Peano curves were probably the first SFC to see a broad application for load distribution and load balancing [6, 17]. Meanwhile, SFC-based strategies have become a well-established tool frequently outperforming alternatives such as graph partitioning. The classical SFC-based load-balancing approach is to queue up grid elements like pearls on a thread and to cut this queue into pieces with equal workload [4, 6, 7, 17, 18, 22, 24]. Fig. 4 shows examples for the partitioning of a triangulation according to the Sierpinski curve and a Cartesian grid according to the Peano curve. This reduces the load balancing problem to a problem of sorting data according to their positions on a space-filling curve. The locality properties of SFC yield connected partitions with quasi-minimal surfaces that is quasi-minimal communication costs. However, the constants involved in the quasi-optimality statement can be rather large [24].

More recent approaches combine the space-filling curve ordering with a tree-based domain decomposition [5, 14, 23]. This approach has the advantage that already the domain decomposition itself as well as dynamical rebalancing can easily be done fully parallel and that it fits in a natural way with highly efficient multilevel numerical methods such as multigrid solvers.

Summarizing, it is obvious that although load balancing is the most attractive application of SFC in the context of parallel computing, the scope of SFC has become much wider with their parallelization characteristics often just being one side effect.

Related Entries

►Domain Decomposition
►Hierarchical data format

Bibliographic Notes and Further Reading

1. Sagan H (1994) Space-filling curves, Springer, New York.
2. Bader M Space-filling curves – an introduction with applications in scientific computing, Texts in Computational Science and Engineering, Springer, submitted.

Bibliography

1. Bader M, Schraufstetter S, Vigh CA, Behrens J (2008) Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. Int J Comput Sci Eng 4(1):12–21
2. Bader M, Zenger Ch (2006) Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra Appl 417(2–3):301–313
3. Barnes J, Hut P (1986) A hierarchical O(n log n) force-calculation algorithm. Nature 324:446–449
4. Brázdová V, Bowler DR (2008) Automatic data distribution and load balancing with space-filling curves: implementation in conquest. J Phy Condens Matt 20
5. Brenk M, Bungartz H-J, Mehl M, Muntean IL, Neckel T, Weinzierl T (2008) Numerical simulation of particle transport in a drift ratchet. SIAM J Sci Comput 30(6):2777–2798
6. Griebel M, Zumbusch GW (1999) Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. Parallel Comput 25:827–843

7. Günther F, Krahnke A, Langlotz M, Mehl M, Pögl M, Zenger Ch (2004) On the parallelization of a cache-optimal iterative solver for PDES based on hierarchical data structures and space-filling curves. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users Group Meeting Budapest, Hungary, September 19–22, 2004. Proceedings, vol 3241 of Lecture Notes in Computer Science. Springer, Heidelberg
8. Günther F, Mehl M, Pögl M, Zenger C (2006) A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. SIAM J Sci Comput 28(5):1634–1650
9. Hunter GM, Steiglitz K (1979) Operations on images using quad trees. IEEE Trans Pattern Analy Machine Intell PAMI-1(2): 145–154
10. Jackings C, Tanimoto SL (1980) Octrees and their use in representing three-dimensional objects. Comp Graph Image Process 14(31):249–270
11. Rokhlin V, Greengard L (1987) A fast algorithms for particle simulations. J Comput Phys 73:325–348
12. Meagher D (1980) Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3d objects by computer. Technical Report, IPL-TR-80-111
13. Mehl M, Weinzierl T, Zenger C (2006) A cache-oblivious self-adaptive full multigrid method. Numer Linear Algebr 13(2–3):275–291
14. Mitchell WF (2007) A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. J Parallel Distrib Comput 67(4):417–429
15. Morton GM (1966) A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, IBM Ltd., Ottawa, Ontario
16. Mundani R-P, Bungartz H-J, Niggl A, Rank E (2006) Embedding, organisation, and control of simulation processes in an octree-based cscw framework. In: Proceedings of the 11th International Conference on Computing in Civil and Building Engineering, Montreal, pp 3208–3215
17. Patra A, Oden JT (1995) Problem decomposition for adaptive hp finite element methods. Comput Syst Eng 6(2):97–109
18. Roberts S, Klyanasundaram S, Cardew-Hall M, Clarke W (1998) A key based parallel adaptive refinement technique for finite element methods. In: Proceedings of the Computational Techniques and Applications: CTAC '97, Singapore, pp 577–584
19. Samet H (1980) Region representation: quadtrees from binary arrays. Comput Graph Image Process 13(1):88–93
20. Saxena M, Finnigan1 PM, Graichen CM, Hathaway AF, Parthasarathy VN (1995) Octree-based automatic mesh generation for non-manifold domains. Eng Comput 11(1): 1–14
21. Schroeder WJ, Shephard MS (1988) A combined octree/delaunay method for fully automatic 3-d mesh generation. Int J Numer Methods Eng 26(1):37–55
22. Sundar H, Sampath RS, Biros G (2008) Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. SIAM J Sci Comput 30(5):2675–2708
23. Weinzierl T (2009) A framework for parallel PDE solvers on multiscale adaptive Cartesian grids. Verlag Dr. Hut
24. Zumbusch GW (2001) On the quality of space-filling curve induced partitions. Z Angew Math Mech 81:25–28

SPAI (SParse Approximate Inverse)

Thomas Huckle, Matous Sedlacek
Technische Universität München, Garching, Germany

Synonyms

Sparse approximate inverse matrix

Definition

For a given sparse matrix A a sparse matrix $M \approx A^{-1}$ is computed by minimizing $\|AM - I\|_F$ in the Frobenius norm over all matrices with a certain sparsity pattern. In the SPAI algorithm the pattern of M is updated dynamically to improve the approximation until a certain stopping criterion is reached.

Discussion

Introduction

For applying an iterative solution method like the conjugate gradient method (CG), GMRES, BiCGStab, QMR, or similar algorithms, to a system of linear equations $Ax = b$ with sparse matrix A, it is often crucial to include an efficient preconditioner. Here, the original problem $Ax = b$ is replaced by the preconditioned system $MAx = Mb$ or $Ax = A(My) = b$. In a parallel environment a preconditioner should satisfy the following conditions:

- M can be computed efficiently in parallel.
- Mc can be computed efficiently in parallel for any given vector c.
- The iterative solver applied on $AMx = b$ or $MAx = Mb$ converges much faster than for $Ax = b$ (e.g., it holds $cond(MA) \ll cond(A)$).

The first two conditions can be easily satisfied by using a sparse matrix M as approximation to A^{-1}. Note, that the inverse of a sparse A is nearly dense, but in many

cases the entries of A^{-1} are rapidly decaying, so most of the entries are very small [11].

Benson and Frederickson [4] were the first to propose a sparse approximate inverse preconditioner in a static way by computing

$$\min_{M} \|AM - I\|_F \tag{1}$$

for a prescribed a priori chosen sparsity pattern for M. The computation of M can be split into n independent subproblems $\min_{M_k} \|AM_k - e_k\|_2$, $k = 1, ..., n$ with M_k the columns of M and e_k the k-th column of the identity matrix I. In view of the sparsity of these Least Squares (LS) problems, each subproblem is related to a small matrix $\hat{A}_k := A(I_k, J_k)$ with index set J_k which is given by the allowed pattern for M_k and the so-called shadow I_k of J_k, that is, the indices of nonzero rows in $A(:, J_k)$. These n small LS problems can be solved independently, for example, based on QR decompositions of the matrices $\hat{A}_k$ by using the Householder method or the modified Gram-Schmidt algorithm.

The SPAI Algorithm

The SPAI algorithm is an additional feature in this Frobenius norm minimization that introduces different strategies for choosing new profitable indices in M_k that lead to an improved approximation. Assume that, by solving (1) for a given index set J, an optimal solution $M_k(J_k)$ has been already determined resulting in the sparse vector M_k with residual r_k. Dynamically there will be defined new entries in M_k. Therefore, (1) has to be solved for this enlarged index set $\tilde{J}_k$ such that a reduction in the norm of the new residual $\tilde{r}_k = A(\tilde{I}_k, \tilde{J}_k)M_k(\tilde{J}_k) - e_k(\tilde{I}_k)$ is achieved.

Following Cosgrove, Griewank, Díaz [10], and Grote, Huckle [13], one possible new index $j \in J_{new}$ out of a given set of possible new indices J_{new} is tested to improve M_k. Therefore, the reduced 1D problem

$$\min_{\lambda_j} \|A(M_k + \lambda_j e_j) - e_k\| = \min_{\lambda_j} \|\lambda_j A_j + r_k\| \tag{2}$$

has to be considered. The solution of this problem is given by

$$\lambda_j = -\frac{r_k^T A e_j}{\|Ae_j\|^2}$$

which leads to an improved squared residual norm

$$\rho_j^2 = \|r_k\|^2 - \frac{(r_k^T A e_j)^2}{\|Ae_j\|^2} .$$

Obviously, for improving M_k one has to consider only indices j in rows of A that are related to the nonzero entries in the old residual r_k; otherwise they do not lead to a reduction in the residual norm. Thus, the column indices j have to be determined that satisfy $r_k^T A e_j \neq 0$ with the old residual r_k. Let the index set of nonzero entries in r_k be denoted by L. Furthermore, let $\tilde{J}_i$ denote the set of new indices that are related to the nonzero elements in the i-th row of A, and let $J_{new} = \cup_{i \in L} \tilde{J}_i$ denote the set of all possible new indices that can lead to a reduction of the residual norm. Then, one or more indices J_c are chosen as a subset of J_{new} that corresponds to a large reduction in r_k. For this enlarged index set $J_k \cup J_c$ the QR decomposition of the related LS submatrix has to be updated and solved for the new column M_k.

Inside SPAI there are different parameters that steer the computation of the preconditioner M:

- How many entries are added in one step
- How many steps of adding new entries are allowed
- Start pattern
- Maximum allowed pattern
- What residual $\|r_k\|$ should be reached
- How to solve the LS problems

Modifications of SPAI

A different and more expensive way to determine a new profitable index j with $\tilde{J}_k := J_k \cup \{j\}$ considers the more accurate problem

$$\min_{M_k(\tilde{J}_k)} \|A(:, \tilde{J}_k)M_k(\tilde{J}_k) - e_k\|$$

introduced by Gould and Scott [12]. For $\tilde{J}_k$ the optimal reduction of the residual is determined for the full minimization problem instead of the 1D minimization in SPAI.

Chow [9] showed ways to prescribe an efficient static pattern a priori and developed the software package PARASAILS.

Holland, Shaw, and Wathen [17] have generalized this ansatz allowing a sparse target matrix on the right side in the form $\min_M \|AM - B\|_F$. This approach is

useful in connection with some kind of two-level preconditioning: First compute a standard sparse preconditioner B for A and then improve this preconditioner by an additional Frobenius norm minimization with target B. From the algorithmic point of view the minimization with target matrix B instead of I introduces no additional difficulties. Only the pattern of M should be chosen more carefully with respect to A and B.

Zhang [23] introduced an iterative form of SPAI where in each step a thin M is derived starting with $\min_{M_1} ||AM_1 - I||_F$. In the second step the sparse matrix AM_1 is used and $\min_{M_2} ||(AM_1)M_2 - I||_F$ is solved, and so on. The advantage is, that because of the very sparse patterns in M_i the Least Squares problems are very cheap.

Chan and Tang [8] applied SPAI not to the original matrix but first used a Wavelet transform W and computed the sparse approximate inverse preconditioner for WAW^T that is assumed to be more diagonal dominant.

Yeremin, Kolotilina, Nikishin, and Kaporin [19, 20] introduced factorized sparse approximate inverses of the form $A^{-1} \approx LU$. Huckle generalized the factorized preconditioners adding new entries dynamically like in SPAI [14].

Grote and Barnard [2] developed a software package for SPAI and also introduced a block version of SPAI.

Huckle and Kallischko [15] generalized SPAI and the target approach. They combined SPAI with the probing method [7] in the form

$$\min_M \left(||AM - I||_F^2 + \rho^2 ||e^T AM - e^T||^2 \right)$$

for probing vectors e on which the preconditioner should be especially improved. Furthermore, they developed a software package for MSPAI.

Properties and Applications

Advantages of SPAI:

- Good parallel scalability.
- SPAI allows modifications like factorized approximation or including probing conditions to improve the preconditioner relative to certain subspaces, for example, as smoother in Multigrid or for regularization [16].
- It is especially efficient for preconditioning dense problems (see Benzi [1] et al.).

Disadvantages of SPAI:

- SPAI is sequentially more expensive, especially for denser patterns of M.
- Sometimes it shows poor approximation of A^{-1} and slow convergence as preconditioner.

Related Entries

►Preconditioners for Sparse Iterative Methods

Bibliographic Notes and Further Reading

Books

1. Axelsson O (1996) Iterative solution methods. Cambridge University Press, Cambridge
2. Saad Y (2003) Iterative methods for sparse linear systems. SIAM Philadelpha, PA
3. Bruaset AM (1995) A survey of preconditioned iterative methods. Longman Scientific & Technical, Harlow, Essex
4. Chen K (2005) Matrix preconditioning techniques and applications. Cambridge University Press, Cambridge

Software

1. Chow E. Parasails, https://computation.llnl.gov/casc/parasails/parasails.html
2. Barnard S, Bröker O, Grote M, Hagemann M. SPAI and Block SPAI, http://www.computational.unibas.ch/software/spai
3. Huckle T, Kallischko A, Sedlacek M. MSPAI, http://www5.in.tum.de/wiki/index.php/MSPAI

Bibliography

1. Alleon G, Benzi M, Giraud L (1997) Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. Numer Algorith 16(1):1–15
2. Barnard S, Grote M (1999) A block version of the SPAI preconditioner. Proceedings of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, TX
3. Barnard ST, Clay RL (1997) A portable MPI implemention of the SPAI preconditioner in ISIS++. In: Heath M, et al (eds) Proceedings of the eighth SIAM conference on parallel processing for scientific computing, Philadelphia, PA

4. Benson MW, Frederickson PO (1982) Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. Utilitas Math 22:127–140
5. Bröker O, Grote M, Mayer C, Reusken A (2001) Robust parallel smoothing for multigrid via sparse approximate inverses. SIAM J Scient Comput 23(4):1396–1417
6. Bröker O, Grote M (2002) Sparse approximate inverse smoothers for geometric and algebraic multigrid. Appl Num Math 41(1): 61–80
7. Chan TFC, Mathew TP (1992) The interface probing technique in domain decomposition. SIAM J Matrix Anal Appl 13(1): 212–238
8. Chan TF, Tang WP, Wan WL (1997) Wavelet sparse approximate inverse preconditioners. BIT 37(3):644–660
9. Chow E (2000) A priori sparsity patterns for parallel sparse approximate inverse preconditioners. SIAM J Sci Comput 21(5):1804–1822
10. Cosgrove JDF, D´iaz JC, Griewank A (1992) Approximate inverse preconditionings for sparse linear systems. Int J Comput Math 44:91–110
11. Demko S, Moss WF, Smith PW (1984) Decay rates of inverses of band matrices. Math Comp 43:491–499
12. Gould NIM, Scott JA (1995) On approximate-inverse preconditioners. Technical Report RAL-TR-95-026, Rutherford Appleton Laboratory, Oxfordshire, England
13. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. SIAM J Sci Comput 18(3):838–853
14. Huckle T (2003) Factorized sparse approximate inverses for preconditioning. J Supercomput 25:109–117
15. Huckle T, Kallischko A (2007) Frobenius norm minimization and probing for preconditioning. Int J Comp Math 84(8):1225–1248
16. Huckle T, Sedlacek M (2010) Smoothing and regularization with modified sparse approximate inverses. Journal of Electrical and Computer Engineering – Special Issue on Iterative Signal Processing in Communications, Appearing (2010)
17. Holland RM, Shaw GJ, Wathen AJ (2005) Sparse approximate inverses and target matrices. SIAM J Sci Comput 26(3):1000–1011
18. Kaporin IE (1994) New convergence results and preconditioning strategies for the conjugate gradient method. Numer Linear Algebra Appl 1:179–210
19. Kolotilina LY, Yeremin AY (1993) Factorized sparse approximate inverse preconditionings I: Theory. SIAM J Matrix Anal Appl 14(1):45–58
20. Kolotilina LY, Yeremin AY (1995) Factorized sparse approximate inverse preconditionings II: Solution of 3D FE systems on massively parallel computers. Inter J High Speed Comput 7(2): 191–215
21. Tang W-P (1999) Toward an effective sparse approximate inverse preconditioner. SIAM J Matrix Anal Appl 20(4):970–986
22. Tang WP, Wan WL (2000) Sparse approximate inverse smoother for multigrid. SIAM J Matrix Anal Appl 21(4):1236–1252
23. Zhang J (2002) A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. Appl Math Comput 130(1):63–85

Spanning Tree, Minimum Weight

David A. Bader[1], Guojing Cong[2]
[1]Georgia Institute of Technology, Atlanta, GA, USA
[2]IBM, Yorktown Heights, NY, USA

Definition

Given an undirected connected graph G with n vertices and m edges, the minimum-weight spanning tree (MST) problem consists in finding a spanning tree with the minimum sum of edge weights. A single graph can have multiple MSTs. If the graph is not connected, then it has a minimum spanning forest (MSF) that is a union of minimum spanning trees for its connected components. MST is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, and distributed networks, recent problems in biology and medicine such as cancer detection, medical imaging, and proteomics.

With regard to any MST of graph G, two properties hold: *Cycle property*: the heaviest edge (edge with the maximum weight) in any cycle of G does not appear in the MST. *Cut property*: if the weight of an edge e of any cut C of G is smaller than the weights of other edges of C, then this edge belongs to all MSTs of the graph.

When all edges of G are of unique weights, the MST is unique. When the edge weights are not unique, they can be made unique by numbering the edges and break ties using the edge number.

Sequential Algorithms

Three classical sequential algorithms, Prim, Kruskal, and Borůvka, are known for MST. Each algorithm grows a forest in stages, adding at each stage one or more tree edges, whose membership in the MST is guaranteed by the cut property. They differ in how the tree edges are chosen and the order that they are added.

Prim starts with one vertex and takes a greedy approach in growing the tree. In each step, it always maintains a connected tree by choosing the edge of the smallest weight that connects the current tree to a vertex that is outside the tree.

Kruskal starts with isolated vertices. As the algorithm progresses, multiple trees may appear, and eventually they merge into one.

Borůvka selects for each vertex the incident edge of the smallest weight as a tree edge. It then compacts the graph by contracting each connected component into a super-vertex. Note that finding the tree edges can be done in parallel for each vertices. Borůvka's algorithm lends itself naturally to parallelization.

These algorithms can easily be made to run in $O(m \log n)$ time. Better complexities can be achieved with Prim's algorithm if Fibonacci heap is used instead of binary heap.

Graham and Hell [14] gave a good introduction for the history of MST algorithms up to 1985. More complex algorithms with better asymptotic run times have since been proposed. For example, Gabow et al. designed an algorithm that runs in almost linear time, i.e., $O(m \log \beta(m,n))$, where $\beta(m,n) = \min\{i | \log^{(i)} n \leq m/n\}$. Karger, Klein, and Tarjan presented a randomized linear-time algorithm to find minimum spanning trees. This algorithm uses the random sampling technique together with a linear time verification algorithm (e.g., King's verification algorithm). Pettie and Ramachandran presented an optimal MST algorithm that runs in $O(T^*(m,n))$, where T^* is the minimum number of edge-weight comparisons needed to determine the solution.

Moret and Shapiro [20] presented a comprehensive experimental study on the performance MST algorithms. Prim's algorithm with binary heap is found in general to be a fast solution. Katriel et al. [19] have developed a pipelined algorithm that uses the cycle property and provide an experimental evaluation on the special-purpose NEC SX-5 vector computer.

Cache-oblivious MST is presented by Arge et al. [2]. Their algorithm is based on a cache-oblivious priority queue that supports *insertion*, *deletion*, and *deletemin* operations in $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ amortized memory transfers, where M and B are the memory and block transfer sizes. The cache-oblivious implementation of MST runs in $O(sort(m) \log\log(n/c) + c)$ memory transfers ($c = m/B$). The MST algorithm combines two phases: Borůvka and Prim.

The memory access behavior of some MST algorithms are characterized in [11].

Parallel Algorithms

Fast theoretical parallel MST algorithms also exist in the literature. Pettie and Ramachandran [21] designed a randomized, time-work optimal MST algorithm for the EREW PRAM, and using EREW to QSM and QSM to BSP emulations from [13], mapped the performance onto QSM and BSP models. Cole et. al. [7, 8] and Poon and Ramachandran [22] earlier had randomized linear-work algorithms on CRCW and EREW PRAM. Chong et al. [5] gave a deterministic EREW PRAM algorithm that runs in logarithmic time with a linear number of processors. On the BSP model, Adler et al. [1] presented a communication-optimal MST algorithm.

Most parallel and some fast sequential MST algorithms employ the Borůvka iteration. Three steps characterize a Borůvka iteration: *find-min*, *connect-components*, and *compact-graph*.

1. *find-min*: for each vertex v label the incident edge with the smallest weight to be in the MST.
2. *connect-components*: identify connected components of the induced graph with edges found in Step 1.
3. *compact-graph*: compact each connected component into a single supervertex, remove self-loops and multiple edges; and relabel the vertices for consistency.

The Borůvka algorithm iterates until no new tree edges can be found. After each Borůvka iteration, the number of vertices in the graph is reduced at least by half. Some algorithms invoke several rounds of the Borůvka iterations to reduce the input size (e.g., [1]) and/or to increase the edge density (e.g., [18]).

Implementation of Parallel Borůvka

Many of the fast theoretic MST algorithms are considered impractical for input of realistic size because they are too complicated and have large constant factors hidden in the asymptotic complexity. Complex MST algorithms are hard to implement and usually do not achieve good parallel speedups on current architectures. Most existing implementations are based on the Borůvka algorithm.

For a Borůvka iteration, *Find-min* and *connect-components* are simple and straightforward to implement. The *compact-graph* step performs bookkeeping

that is often left as a trivial exercise to the reader. JáJá [16] describes a compact-graph algorithm for dense inputs. For sparse graphs, though, the compact-graph step often is the most expensive step in the Borůvka iteration. Implementations and data structures for parallel Borůvka on shared-memory machines are described in [3].

Edge List Representation

This implementation of Borůvka's algorithm (designated **Bor-EL**) uses the edge list representation of graphs, with each edge (u, v) appearing twice in the list for both directions (u, v) and (v, u). An elegant implementation of the compact-graph step sorts the edge list (using an efficient parallel sample sort [15]) with the supervertex of the first endpoint as the primary key, the supervertex of the second endpoint as the secondary key, and the edge weight as the tertiary key. When sorting completes, all of the self-loops and multiple edges between two supervertices appear in consecutive locations and can be merged efficiently using parallel prefix-sums.

Adjacency List Representation

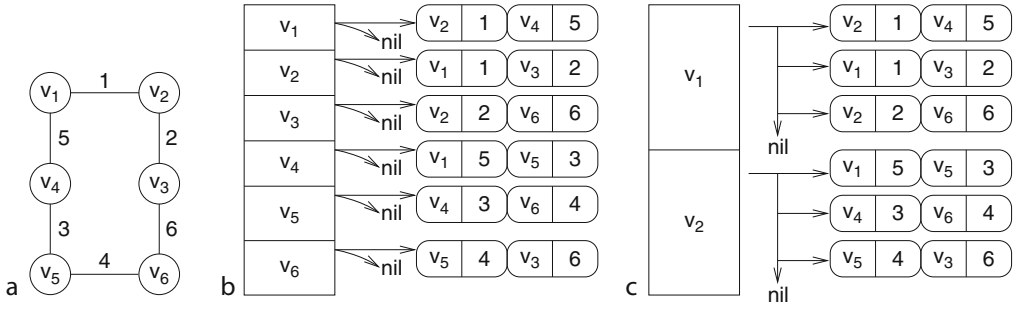
With the adjacency list representation, each entry of an index array of vertices points to a list of its incident edges. The compact-graph step first sorts the vertex array according to the supervertex label, then concurrently sorts each vertex's adjacency list using the supervertex of the other endpoint of the edge as the key. After sorting, the set of vertices with the same supervertex label are contiguous in the array, and can be merged efficient. This approach is designated as **Bor-AL**.

Both **Bor-EL** and **Bor-AL** achieve the same goal that self-loops and multiple edges are moved to consecutive locations to be merged. **Bor-EL** uses one call to sample sort, while **Bor-AL** calls a smaller parallel sort and then a number of concurrent sequential sorts.

Flexible Adjacency List Representation

The flexible adjacency list augments the traditional adjacency list representation by allowing each vertex to hold multiple adjacency lists instead of just a single one; in fact, it is a linked list of adjacency lists. During initialization, each vertex points to only one adjacency list. After the connect-components step, each vertex appends its adjacency list to its supervertex's adjacency list by sorting together the vertices that are labeled with the same supervertex. The compact-graph step is simplified, allowing each supervertex to have self-loops and multiple edges inside its adjacency list. Thus, the compact-graph step now uses a smaller parallel sort plus several pointer operations instead of costly sortings and memory copies, while the find-min step gets the added responsibility of filtering out the self-loops and multiple edges. This approach is designated as **Bor-FAL**.

Figure 1 illustrates the use of the flexible adjacency list for a 6-vertex input graph. After one Borůvka iteration, vertices 1, 2, and 3 form one supervertex, and vertices 4, 5, and 6 form a second supervertex. Vertex labels 1 and 4 represent the supervertices and receive the adjacency lists of vertices 2 and 3, and vertices 5 and 6, respectively. Vertices 1 and 4 are relabeled as 1 and 2. Note that most of the original data structure is kept intact. Instead of relabeling vertices in the adjacency list, a separate lookup table is maintained that holds the



Spanning Tree, Minimum Weight. Fig. 1 Example of flexible adjacency list representation. (**a**) Input graph. (**b**) Initialized flexible adjacency list. (**c**) Flexible adjacency list after one iteration

supervertex label for each vertex. The find-min step uses this table to filter out self-loops and multiple edges.

Analysis of Implementations

Helman and JáJá's SMP complexity model [15] provides a reasonable framework for the realistic analysis that favors cache-friendly algorithms by penalizing noncontiguous memory accesses. Under this model, there are two parts to an algorithm's complexity: M_E, the memory access complexity and, T_C, the computation complexity. The M_E term is the number of noncontiguous memory accesses, and the T_C term is the running time. The M_E term recognizes the effect that memory accesses have over an algorithm's performance. Parameters of the model includes the problem size n and the number of processors p.

For a sparse graph G with n vertices and m edges, as the algorithm iterates, the number of vertices decreases by at least half in each iteration, so there are at most $\log n$ iterations for all of the Borůvka variants.

Hence, the complexity of **Bor-EL** is given as, where c and z are constants related to cache size and sampling ratio [15].

$$T(n,p) = \langle M_E\,;\ T_C\rangle = \left\langle \left(\frac{8m+n+n\log n}{p} + \frac{4mc\log(2m/p)}{p\log z} \right)\log n\,;\ \mathrm{O}\left(\frac{m}{p}\log m\log n\right)\right\rangle.$$

As in each iteration these **Bor-AL** and **Bor-EL** compute similar results in different ways, it suffices to compare the complexity of the first iteration. For **Bor-AL**, the complexity of the first iteration is

$$T(n,p) = \langle M_E;\ T_C\rangle = \left\langle \left(\frac{8n+5m+n\log n}{p} + \frac{2nc\log(n/p)+2mc\log(m/n)}{p\log z} \right);\ \mathrm{O}\left(\frac{n}{p}\log m + \frac{m}{p}\log(m/n)\right)\right\rangle.$$

While for **Bor-EL**, the complexity of the first iteration is

$$T(n,p) = \langle M_E;\ T_C\rangle = \left\langle \left(\frac{8m+n+n\log n}{p} + \frac{4mc\log(2m/p)}{p\log z} \right);\ \mathrm{O}\left(\frac{m}{p}\log m\right)\right\rangle.$$

Bor-AL is a faster algorithm than **Bor-EL**, as expected, since the input for **Bor-AL** is "bucketed" into adjacency lists, versus **Bor-EL** that is an unordered list of edges, and sorting each bucket first in **Bor-AL** saves unnecessary comparisons between edges that have no vertices in common. The complexity of **Bor-EL** can be considered to be an upper bound of **Bor-AL**.

In **Bor-FAL**, n reduces at least by half while m stays the same. Compact-graph first sorts the n vertices, then assigns $\mathrm{O}(n)$ pointers to append each vertex's adjacency list to its supervertex's. For each processor, sorting takes $\mathrm{O}\left(\frac{n}{p}\log n\right)$ time, and assigning pointers takes $\mathrm{O}(n/p)$ time assuming each processor gets to assign roughly the same amount of pointers. Updating the lookup table costs each processor $\mathrm{O}(n/p)$ time. With **Bor-FAL**, to find the smallest weight edge for the supervertices, all the m edges will be checked, with each processor covering $\mathrm{O}(m/p)$ edges. The aggregate running time is $T_C(n,p)_{fm} = \mathrm{O}(m\log n/p)$, and the memory access complexity is $M_E(n,p)_{fm} = m/p$. For the finding connected component step, each processor takes $T_{cc} = \mathrm{O}\left(n\log\frac{n}{p}\right)$time, and $M_E(n,p)_{cc} \le 2n\log n$. The complexity for the whole Borůvka's algorithm is

$$T(n,p) = T(n,p)_{fm} + T(n,p)_{cc} + T(n,p)_{cg} \le \left\langle \frac{8n+2n\log n+m\log n}{p} + \frac{4cn\log(n/p)}{p\log z}\,;\ \mathrm{O}\left(\frac{m+n}{p}\log n\right)\right\rangle$$

A Hybrid Parallel MST Algorithm

An MST algorithm has been proposed in [3] that marries Prim's algorithm (known as an efficient sequential algorithm for MST) with that of the naturally parallel Borůvka approach. In this algorithm, essentially each processor simultaneously runs Prim's algorithm from different starting vertices. A tree is said to be *growing* when there exists a lightweight edge that connects the tree to a vertex not yet in another tree, and *mature* otherwise. When all of the vertices have been incorporated into mature subtrees, the algorithm contracts each subtree into a supervertex and call the approach recursively until only one supervertex remains. When the problem size is small enough, one processor solves the remaining problem using the best sequential MST algorithm.

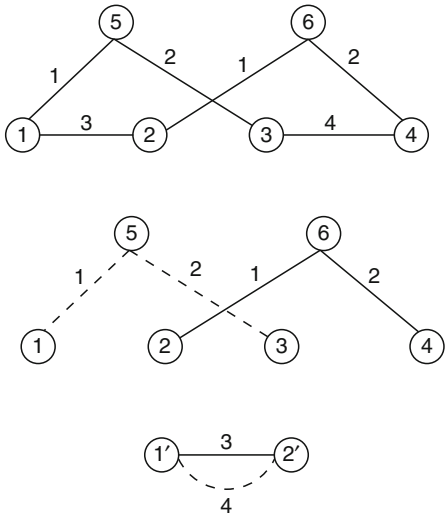
This parallel MST algorithm possesses an interesting feature: when run on one processor the algorithm behaves as Prim's, and on n processors becomes Borůvka's, and runs as a hybrid combination for $1 < p < n$, where p is the number of processors. Each of p processors in the algorithm finds for its starting vertex the smallest-weight edge, contracts that edge, and then finds the smallest-weight edge again for the contracted supervertex. It does not find all the smallest-weight edges for all vertices, synchronize, and then compact as in the parallel Borůvka's algorithm. The algorithm adapts for any number p of processors in a practical way for SMPs, where p is often much less than n, rather than in parallel implementations of Borůvka's approach that appear as PRAM emulations with p coarse-grained processors that emulate n virtual processors.

Implementation with Fine-Grained Locks

Most parallel graph algorithms are designed without locks. Indeed it is hard to measure contention for these algorithms. Yet proper use of locks can simplify implementation and improve performance. Cong and Bader [10] presented an implementation of Borůvka's algorithm (**Bor-spinlock**) that uses locks and avoids modifying the input data structure. In **Bor-spinlock**, the *compact-graph* step is completely eliminated.

The main idea is that instead of compacting connected components, for each vertex there is now an associated label *supervertex* showing to which supervertex it belongs. In each iteration, all the vertices are partitioned as evenly as possible among the processors. Processor p finds the adjacent edge with smallest weight for a supervertex v'. As the graph is not compacted, the adjacent edges for v' are scattered among the adjacent edges of all vertices that share the same supervertex v', and different processors may work on these edges simultaneously. Now the problem is that these processors need to synchronize properly in order to find the edge with the minimum weight. Figure 2 illustrates the specific problem for the MST case.

On the top in Fig. 2 is an input graph with six vertices. Suppose there are two processors P_1 and P_2. Vertices 1, 2, and 3 are partitioned on to processor P_1, and vertices 4, 5, and 6 are partitioned on to processor P_2. It takes two iterations for Borůvka's algorithm to find the MST. In the first iteration, the *find-min* step of **Bor-spinlock** labels $< 1,5 >$, $< 5,3 >$, $< 2,6 >$, and $< 6,4 >$, to be in the MST. *Connected-components* finds vertices 1, 3, and 5 in one component, and vertices 2, 4, and 6 in another component. The MST edges and components are shown in the middle of Fig. 2. Vertices connected by dashed lines are in one component, and vertices connected by solid lines are in the other component. At this time, vertices 1, 3, and 5 belong to supervertex $1'$, and vertices 2, 4, and 6 belong to supervertex $2'$. In the second iteration, processor P_1 again inspects vertices 1, 2, and 3, and processor P_2 inspects vertices 4, 5, and 6. Previous MST edges $< 1,5 >$, $< 5,3 >$, $< 2,6 >$, and $< 6,4 >$ are found to be edges inside supervertices and are ignored. On the bottom of Fig. 2 are the two supervertices with two edges between them. Edges $< 1,2 >$ and $< 3,4 >$ are found by P_1 to be the edges between supervertices $1'$ and $2'$, edge $< 3,4 >$ is found by P_2 to be the edge between the two supervertices. For supervertex $2'$, P_1 tries to label $< 1,2 >$ as the MST edge, while P_2 tries to label $< 3,4 >$. This is a race condition between the two processors, and locks are used in **Bor-spinlock** to ensure correctness.



Spanning Tree, Minimum Weight. Fig. 2 Example of the race condition between two processors when Borůvka's algorithm is used to solve the MST problem

In addition to locks and barriers, recent development in transactional memory provides a new mechanism for synchronization among processors. Kang and Bader implemented minimum spanning forest algorithms with transactional memory [17]. Their

implementation achieved good scalability on some current architectures.

Implementation on Distributed-Memory Machines

Partitioned global address space (PGAS) languages such as UPC and X10 [4, 23] have been proposed recently that present a shared-memory abstraction to the programmer for distributed-memory machines. They allow the programmer to control the data layout and work assignment for the processors. Mapping shared-memory graph algorithms onto distributed-memory machines is straightforward with PGAS languages.

Figure 3 shows both the SMP implementation and UPC implementation of parallel Borůvka. The two implementations are also almost identical. The differences are shown in underscore. Performance wise, straightforward PGAS implementation for irregular graph algorithms does not usually achieve high performance due to the aggregate startup cost of many small messages. Cong, Almasi, and Saraswat presented their study in optimizing the UPC implementation of graph algorithm in [9]. They apply communication coalescing together with other techniques for improving the performance. The idea is to merge the small messages to/from a processor into a single, large message. As all operations in each step of a typical PRAM algorithm are parallel, reads and writes can be scheduled in an order such that communication coalescing is possible. After communication coalescing, these data can be accessed in one communication round where one processor sends at most one message to another processor.

Experimental Results

Chung and Condon [6] implement parallel Borůvka's algorithm on the TMC CM-5. On a 16-processor machine, for geometric, structured graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieves a relative parallel speedup of about 4, on 16-processors, over the sequential Borůvka's algorithm, which was already 2–3 times slower than their sequential Kruskal

```
grafted = 0;
upc_forall(l=0;l<m; l++; i)
{
    i = El[l].v1; w = El[l].w;
    j = El[l].v2;
    i = D[i]; j = D[j];
    if(i!=j ){
      upc_lock(lock_array[i]);
      if(Min[i] > w) {
        Min[i] = w;
        Min_ind[i] = j;
        grafted = 1;
      }
      upc_unlock(lock_array[i]);
    }
}
upc_barrier;
grafted = all_reduce_i(grafted, UPC_MAX);
if(grafted ==0) break;

upc_forall(i=0;i<n;i++;i)
     if(Min_ind[i]!=-1)
         D[i]=Min_ind[i];
upc_barrier;

upc_forall(i=0; i<n; i++; i)
  while(D[i]!=D[D[i]]) D[i]=D[D[i]];
```

```
grafted = 0;
pardo(l,0,m,1)
{
      i = El[l].v1; w = El[l].w
      j = El]l].v2
      i = D[i]; j = D[j];
      if(i!=j ){
        pthread_lock(lock_array[i]);
        if(Min[i]>w) {
          Min[i] = w;
          Min_ind[i]=j;
          grafted=1;
        }
        pthread_unlock(lock_array[i]);
    }
}
node_barrier();
grafted = node_Reduce_i(grafted, MAX, TH);
if(grafted==0) break;

pardo(i,0,n,1)
  if(Min_ind[i]!=-1)
      D[i]=Min_ind[i];
node_Barrier();

pardo(i,0,n,1)
 while(D[i]!=D[D[i]]) D[i]=D[D[i]];
```

Spanning Tree, Minimum Weight. Fig. 3 UPC implementation and SMP implementation of MST: the main loop bodies

algorithm. Dehne and Götz [12] studied practical parallel algorithms for MST using the BSP model. They implement a dense Borůvka parallel algorithm, on a 16-processor Parsytec CC-48, that works well for sufficiently dense input graphs. Using a fixed-sized input graph with 1,000 vertices and 400,000 edges, their code achieves a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for the more challenging sparse graphs.

Bader and Cong presented their studies of parallel MST on symmetric multiprocessors (SMPs) in [3, 10]. Their implementation achieved for the first time good parallel speedups over a wide range of inputs on SMPs. Their Experimental results show that for **Bor-EL** and **Bor-AL** the compact-graph step dominates the running time. **Bor-EL** takes much more time than **Bor-AL**, and only gets worse when the graphs get denser. In contrast the execution time of compact-graph step of **Bor-FAL** is greatly reduced: in the experimental section with a random graph of 1M vertices and 10M edges, it is over 50 times faster than **Bor-EL**, and over 7 times faster than **Bor-AL**. Actually the execution time of the compact-graph step of **Bor-FAL** is almost the same for the three input graphs because it only depends on the number of vertices. As predicted, the execution time of the find-min step of **Bor-FAL** increases. And the connect-components step only takes a small fraction of the execution time for all approaches.

Cong, Almasi and Saraswat presented a UPC implementation of distributed MST in [9]. For input graphs with billions of edges, the distributed implementation achieved significant speedups over the SMP implementation and the best sequential implementation.

Bibliography

1. Adler M, Dittrich W, Juurlink B, Kutyłowski M, Rieping I (1998) Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In: SPAA '98: proceedings of the tenth annual ACM symposium on parallel algorithms and architectures, Puerto Vallarta, Mexico. ACM, New York, pp 27–36
2. Arge L, Bender MA, Demaine ED, Holland-Minkley B, Munro JI (2002) Cache-oblivious priority queue and graph algorithm applications. In: Proceedings of the 34th annual ACM symposium on theory of computing, Montreal, Canada. ACM, New York, pp 268–276
3. Bader DA, Cong G (2006) Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J Parallel Distrib Comput 66:1366–1378
4. Charles P, Donawa C, Ebcioglu K, Grothoff C, Kielstra A, Van Praun C, Saraswat V, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 2005 ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA), San Diego, CA, pp 519–538
5. Chong KW, Han Y, Lam TW (2001) Concurrent threads and optimal parallel minimum spanning tree algorithm. J ACM 48: 297–323
6. Chung S, Condon A (1996) Parallel implementation of Borůvka's minimum spanning tree algorithm. In: Proceedings of the 10th international parallel processing symposium (IPPS'96), Honolulu, Hawaii, pp 302–315
7. Cole R, Klein PN, Tarjan RE (1996) Finding minimum spanning forests in logarithmic time and linear work using random sampling. In: Proceedings of the 8th annual symposium parallel algorithms and architectures (SPAA-96), Newport, RI. ACM, New York, pp 243–250
8. Cole R, Klein PN, Tarjan RE (1994) A linear-work parallel algorithm for finding minimum spanning trees. In: Proceedings of the 6th annual ACM symposium on parallel algorithms and architectures, Cape May, NJ, ACM, New York, pp 11–15
9. Cong G, Almasi G, Saraswat V (2010) Fast PGAS implementation of distributed graph algorithms. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC '10), IEEE Computer Society, Washington, DC, pp 1–11
10. Cong G, Bader DA (2004) Lock-free parallel algorithms: an experimental study. In: Proceeding of the 33rd international conference on high-performance computing (HiPC 2004), Banglore, India
11. Cong G, Sbaraglia S (2006) A study of the locality behavior of minimum spanning tree algorithms. In: The 13th international conference on high performance computing (HiPC 2006), Bangalore, India. IEEE Computer Society, pp 583–594
12. Dehne F, Götz S (1998) Practical parallel algorithms for minimum spanning trees. In: Proceedings of the seventeenth symposium on reliable distributed systems, West Lafayette, IN. IEEE Computer Society, pp 366–371
13. Gibbons PB, Matias Y, Ramachandran V (1997) Can shared-memory model serve as a bridging model for parallel computation? In: Proceedings 9th annual symposium parallel algorithms and architectures (SPAA-97), Newport, RI, ACM, New York pp 72–83
14. Graham RL, Hell P (1985) On the history of the minimum spanning tree problem. IEEE Ann History Comput 7(1):43–57
15. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99), Baltimore, MD, Lecture notes in computer science, vol 1619. Springer-Verlag, Heidelberg, pp 37–56
16. JáJá J (1992) An Introduction to parallel algorithms. Addison-Wesley, New York
17. Kang S, Bader DA (2009) An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In: Proceedings of the 14th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), Raleigh, NC

18. Karger DR, Klein PN, Tarjan RE (1995) A randomized linear-time algorithm to find minimum spanning trees. J ACM 42(2):321–328
19. Katriel I, Sanders P, Träff JL (2003) A practical minimum spanning tree algorithm using the cycle property. In: 11th Annual European symposium on algorithms (ESA 2003), Budapest, Hungary, Lecture notes in computer science, vol 2832. Springer-Verlag, Heidelberg, pp 679–690
20. Moret BME, Shapiro HD (1994) An empirical assessment of algorithms for constructing a minimal spanning tree. In: DIMACS monographs in discrete mathematics and theoretical computer science: computational support for discrete mathematics vol 15, American Mathematical Society, Providence, RI, pp 99–117
21. Pettie S, Ramachandran V (2002) A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. SIAM J Comput 31(6):1879–1895
22. Poon CK, Ramachandran V (1997) A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In: Proceedings of the 8th international symposium algorithms and computation (ISAAC'97), Lecture notes in computer science, vol 1350. Springer-Verlag, Heidelberg, pp 212–222
23. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and Language Specification. CCS-TR-99-157. IDA/CCS, Bowie, Maryland

Sparse Approximate Inverse Matrix

►SPAI (SParse approximate inverse)

Sparse Direct Methods

Anshul Gupta
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Gaussian elimination; Linear equations solvers; Sparse gaussian elimination

Definition

Direct methods for solving linear systems of the form $Ax = b$ are based on computing $A = LU$, where L and U are lower and upper triangular, respectively. Computing the triangular factors of the coefficient matrix A is also known as *LU decomposition*. Following the factorization, the original system is trivially solved by solving the triangular systems $Ly = b$ and $Ux = y$. If A is symmetric, then a factorization of the form $A = LL^T$ or $A = LDL^T$ is computed via *Cholesky factorization*, where L is a lower triangular matrix (unit lower triangular in the case of $A = LDL^T$ factorization) and D is a diagonal matrix. One set of common formulations of LU decomposition and Cholesky factorization for dense matrices are shown in Figs. 1 and 2, respectively. Note that other mathematically equivalent formulations are possible by rearranging the loops in these algorithms. These algorithms must be adapted for sparse matrices, in which a large fraction of entries are zero. For example, if $A[j, i]$ in the division step is zero, then this operation need not be performed. Similarly, the update steps can be avoided if either $A[j, i]$ or $A[i, k]$ ($A[k, i]$ if A is symmetric) is zero.

When A is sparse, the triangular factors L and U typically have nonzero entries in many more locations than A does. This phenomenon is known as *fill-in*, and results in a superlinear growth in the memory and time requirements of a direct method to solve a sparse system with respect to the size of the system. Despite a high memory requirement, direct methods are often used in many real applications due to their generality and robustness. In applications requiring solutions with respect to several right-hand side vectors and the same coefficient matrix, direct methods are often the solvers of choice because the one-time cost of factorization can be amortized over several inexpensive triangular solves.

Discussion

The direct solution of a sparse linear system typically involves four phases. The two computational phases, *factorization* and *triangular solutions* have already been mentioned. The number of nonzeros in the factors and sometimes their numerical properties are functions of the initial permutation of the rows and columns of the coefficient matrix. In many parallel formulations of sparse factorization, this permutation can also have an effect on load balance. The first step in the direct solution of a sparse linear system, therefore, is to apply heuristics to compute a desirable permutation the matrix. This step is known as *ordering*. A sparse matrix can be viewed as the adjacency matrix of a graph. Ordering heuristics typically use the graph view of the matrix

```
1.  begin LU_Decomp (A, n)
2.    for i = 1, n
3.      for j = i + 1, n
4.        A[j, i] = A[j, i]/A[i, i]; /* division step, computes column i of L */
5.      end for
6.      for k = i + 1, n
7.        for j = i + 1, n
8.          A[j, k] = A[j, k] − A[j, i]×A[i, k]; /* update step*/
9.        end for
10.     end for
11.   end for
12. end LU_Decomp
```

Sparse Direct Methods. Fig. 1 A simple column-based algorithm for LU decomposition of an $n \times n$ dense matrix A. The algorithm overwrites A by L and U such that $A = LU$, where L is unit lower triangular and U is upper triangular. The diagonal entries after factorization belong to U; the unit diagonal of L is not explicitly stored

```
1.  begin Cholesky (A, n)
2.    for i = 1, n
3.      A[i, i] = √A[i, i];
4.      for j = i + 1, n
5.        A[j, i] = A[j, i]/A[i, i]; /* division step, computes column i of L */
6.      end for
7.      for k = i + 1, n
8.        for j = k, n
9.          A[j, k] = A[j, k] − A[j, i] × A[k, i]; /* update step*/
10.       end for
11.     end for
12.   end for
13. end Cholesky
```

Sparse Direct Methods. Fig. 2 A simple column-based algorithm for Cholesky factorization of an $n \times n$ dense symmetric positive definite matrix A. The lower triangular part of A is overwritten by L, such that $A = LL^T$

and label the vertices in a particular order that is equivalent to computing a permutation of the coefficient matrix with desirable properties. In the second phase, known as *symbolic factorization*, the nonzero pattern of the factors is computed. Knowing the nonzero pattern of the factors before actually computing them is useful for several reasons. The memory requirements of numerical factorization can be predicted during symbolic factorization. With the number and locations of nonzeros known before hand, a significant amount of indirect addressing can be avoided during numerical factorization, thus boosting performance. In a parallel implementation, symbolic factorization helps in the distribution of data and computation among processing units. The ordering and symbolic factorization phases are also referred to as preprocessing or analysis steps.

Of the four phases, numerical factorization typically consumes the most memory and time. Many applications involve factoring several matrices with different numerical values but the same sparsity structure. In such cases, some or all of the results of the ordering and symbolic factorization steps can be reused. This is also advantageous for parallel sparse solvers because parallel ordering and symbolic factorization are typically less scalable. Amortization of the cost of these steps over several factorization steps helps maintain the overall scalability of the solver close to that of numerical factorization. The parallelization of the triangular solves is highly dependent on the parallelization of the numerical factorization phase. The parallel formulation of numerical factorization dictates how the factors are distributed among parallel tasks. The subsequent triangular solution steps must use a parallelization scheme

that works on this data distribution, particularly in a distributed-memory parallel environment. Given its prominent role in the parallel direct solution of sparse linear system, the numerical factorization phase is the primary focus of this entry.

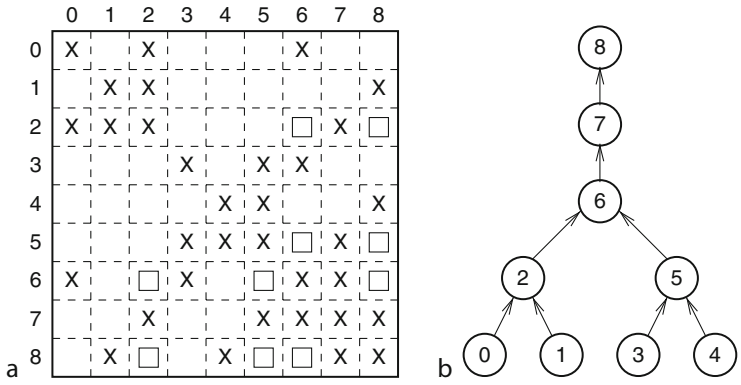
The algorithms used for preprocessing and factoring a sparse coefficient matrix depend on the properties of the matrix, such as symmetry, diagonal dominance, positive definiteness, etc. However, there are common elements in most sparse factorization algorithms. Two of these, namely, *task graphs* and *supernodes*, are key to the discussion of parallel sparse matrix factorization of all types for both practical and pedagogical reasons.

Task Graph Model of Sparse Factorization

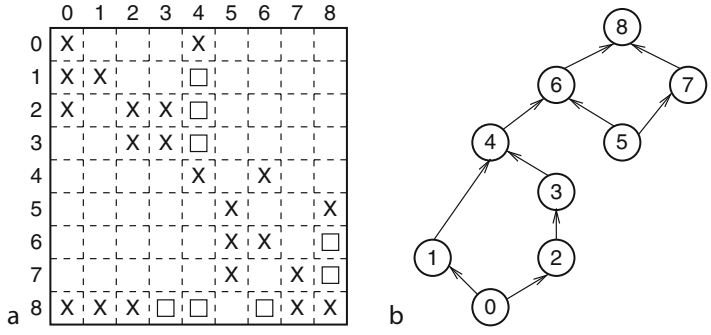
A parallel computation is usually the most efficient when running at the maximum possible level of granularity that ensures a good load balance among all the processors. Dense matrix factorization is computationally rich and requires $O(n^3)$ operations for factoring an $n \times n$ matrix. Sparse factorization involves a much smaller overall number of operations per row or column of the matrix than its dense counterpart. The sparsity results in additional challenges, as well as additional opportunities to extract parallelism. The challenges are centered around finding ways of orchestrating the unstructured computations in a load-balanced fashion and of containing the overheads of interaction between parallel tasks in the face of a relatively small number or operations per row or column of the matrix. The added opportunity for parallelism results from the fact that, unlike the dense algorithms of Figs. 1 and 2, the columns of the factors in the sparse case do not need to be computed one after the other. Note that in the algorithms shown in Figs. 1 and 2, row and column i are updated by rows and columns $1 \ldots i-1$. In the sparse case, column i is updated by a column $j < i$ only if $U[j,i] \neq 0$, and a row i is updated by a row $j < i$ only if $L[i,j] \neq 0$. Therefore, as the sparse factorization begins, the division step can proceed in parallel for all columns i for which $A[i,j] = 0$ and $A[j,i] = 0$ for all $j < i$. Similarly, at any stage in the factorization process, there could be large pool of columns that are ready of the division step. Any unfactored column i would belong to this pool iff all columns $j < i$ with a nonzero entry in row i of L and all rows $j < i$ with a nonzero entry in column i of U have been factored.

A task dependency graph is an excellent tool for capturing parallelism and the various dependencies in sparse matrix factorization. It is a directed acyclic graph (DAG) whose vertices denote tasks and the edges specify the dependencies among the tasks. A task is associated with each row and column (column only in the symmetric case) of the sparse matrix to be factored. The vertex i of the task graph denotes the task responsible for computing column i of L and row i of U. A task is ready for execution if and only if all tasks with incoming edges to it have completed. Task graphs are often explicitly constructed during symbolic factorization to guide the numerical factorization phase. This permits the numerical factorization to avoid expensive searches in order to determine which tasks are ready for execution at any given stage of the parallel factorization process. The task graphs corresponding to matrices with a symmetric structure are trees and are known as *elimination trees* in the sparse matrix literature.

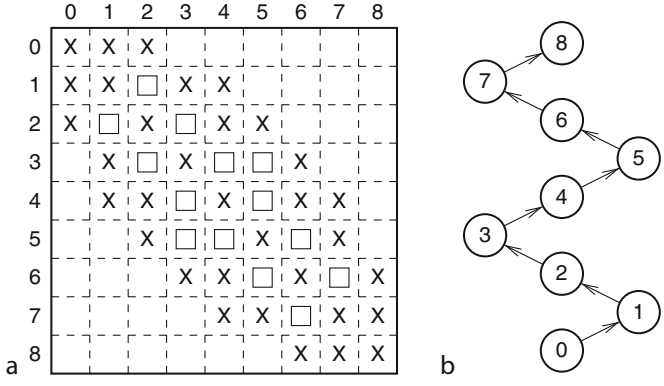
Figure 3 shows the elimination tree for a structurally symmetric sparse matrix and Fig. 4 shows the task DAG for a structurally unsymmetric matrix. Once a task graph is constructed, then parallel factorization (and even parallel triangular solution) can be viewed as the problem of scheduling the tasks onto parallel processes or threads. Static scheduling is generally preferred in a distributed-memory environment and dynamic scheduling in a shared-memory environment. The shape of the task graph is a function of the initial permutation of rows and columns of the sparse matrix, and is therefore determined by the outcome of the ordering phase. Figure 5 shows the elimination tree corresponding to the same matrix as in Fig. 3a, but with a different initial permutation. The structure of the task DAG usually affects how effectively it can be scheduled for parallel factorization. For example, it may be intuitively recognizable to readers that the elimination tree in Fig. 3 is more amenable to parallel scheduling than the tree corresponding to a different permutation of the same matrix in Fig. 5. Figure 6 illustrates that the matrices in Figs. 3 and 5 have the same underlying graph. The only difference is in the labeling of the vertices of the graph, which results in a different permutation of the rows and columns of the matrix, different amount



Sparse Direct Methods. Fig. 3 A structurally symmetric sparse matrix and its elimination tree. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in



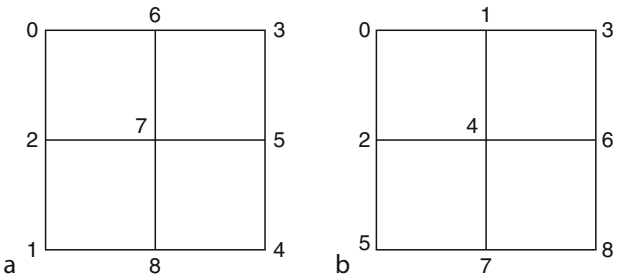
Sparse Direct Methods. Fig. 4 An unsymmetric sparse matrix and the corresponding task DAG. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in



Sparse Direct Methods. Fig. 5 A permutation of the sparse matrix of Fig. 3a and its elimination tree

of fill-in, and different shapes of task graphs. In general, long and skinny task graphs result in limited parallelism and a long critical path. Short and broad task graphs have a high degree of parallelism and shorter critical paths.

Since the shape, and hence the amenability to efficient parallel scheduling of the task graph is sensitive to ordering, heuristics that result in balanced and broad task graphs are preferred for parallel factorization. The best known ordering heuristic in this class is called



Sparse Direct Methods. Fig. 6 An illustration of the duality between graph vertex labeling and row/column permutation of a structurally symmetric sparse matrix. Grid (**a**), with vertices labeled based on nested dissection, is the adjacency graph of the matrix in Fig. 3a and grid (**b**) is the adjacency graph of the matrix in Fig. 5a

nested dissection. Nested dissection is based on recursively computing balanced bisections of a graph by finding small vertex separators. The vertices in the two disconnected partitions of the graph are labeled before the vertices of the separator. The same heuristic is applied recursively for labeling the vertices of each partition. The ordering in Fig. 3 is actually based on nested dissection. Note that the vertex set 6, 7, 8 forms a separator, dividing the graph into two disconnected components, 0, 1, 2 and 3, 4, 5. Within the two components, vertices 2 and 5 are the separators, and hence have the highest label in their respective partitions.

Supernodes

In sparse matrix terminology, a set of consecutive rows or columns that have the same nonzero structure is loosely referred to as a supernode. The notion of supernodes is crucial to efficient implementation of sparse factorization for a large class of sparse matrices arising in real applications.

Coefficient matrices in many applications have natural supernodes. In graph terms, there are sets of vertices with identical adjacency structures. Graphs like these can be compressed by having one supervertex represent the whole set that has the same adjacency structure. When most vertices of a graph belong to supernodes and the supernodes are of roughly the same size (in terms of the number of vertices in them) with an average size of, say, m, then it can be shown that the compressed graph has $O(m)$ fewer vertices and $O(m^2)$ fewer edges than in the original graph. It can also be shown that an ordering of original graph can be derived from an ordering of the compressed graph, while preserving the properties of the ordering, by simply labeling the vertices of the original graph consecutively in the order of the supernodes of the compressed graph. Thus, the space and the time requirements of ordering can be dramatically reduced. This is particularly useful for parallel sparse solvers because parallel ordering heuristics often yield orderings of lower quality than their serial counterparts. For matrices with highly compressible graphs, it is possible to compute the ordering in serial with only a small impact on the overall scalability of the entire solver because ordering is performed on a graph with $O(m^2)$ fewer edges.

While the natural supernodes in the coefficient matrix, if any, can be useful during ordering, it is the presence of supernodes in the factors that have the biggest impact on the performance of the factorization and triangular solution steps. Although there can be multiple ways of defining supernodes in matrices with an unsymmetric structure, the most useful form involves groups of indices with identical nonzero pattern in the corresponding columns of L and rows of U. Even if there are no supernodes in the original matrix, supernodes in the factors are almost inevitable for matrices in most real applications. This is due to fill-in. Examples of supernodes in factors include indices 6–8 in Fig. 3a, indices 2–3 and 7–8 in Fig. 4a, and indices 4–5 and 6–8 in Fig. 5. Some practitioners prefer to artificially increase the size (i.e., the number of member rows and columns) of supernodes by padding the rows and columns that have only slightly different nonzero patterns, so that they can be merged into the same supernode. The supernodes in the factors are typically detected and recorded as they emerge during symbolic

factorization. In the remainder of this chapter, the term supernode refers to a supernode in the factors.

It can be seen from the algorithms in Figs. 1 and 2 that there are two primary computations in a column-based factorization: the division step and the update step. A supernode-based sparse factorization too has the same two basic computation steps, except that these are now matrix operations on row/column blocks corresponding to the various supernodes.

Supernodes impart efficiency to numerical factorization and triangular solves because they permit floating point operations to be performed on dense submatrices instead of individual nonzeros, thus improving memory hierarchy utilization. Since rows and columns in supernodes share the nonzero structure, indirect addressing is minimized because the structure needs to be stored only once for these rows and columns. Supernodes help to increase the granularity of tasks, which is useful for improving computation to overhead ratio in a parallel implementation. The task graph model of sparse matrix factorization was introduced earlier with a task k defined as the factorization of row and column k of the matrix. With supernodes, a task can be defined as the factorization of all rows and columns associated with a supernode. Actual task graphs in practical implementations of parallel sparse solvers are almost always supernodal task graphs.

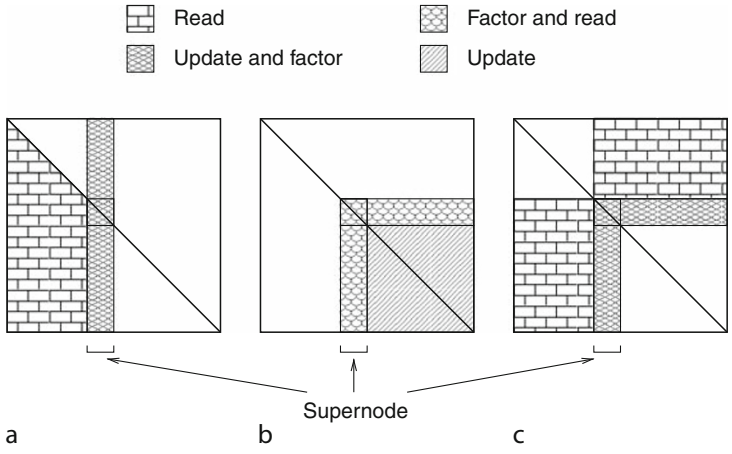
Note that some applications, such as power grid analysis, in which the basis of the linear system is not a finite-element or finite-difference discretization of a physical domain, can give rise to sparse matrices that incur very little fill-in during factorization. The factors of these matrices may have very small supernodes.

An Effective Parallelization Strategy

The task graphs for sparse matrix factorization have some typical properties that make scheduling somewhat different from traditional DAG scheduling. Note that the task graphs corresponding to irreducible matrices have a distinct root; that is, one node that has no outgoing edges. This corresponds to the last (rightmost) supernode in the matrix. The number of member rows and columns in supernodes typically increases away from the leaves and toward the root of the task graph. The reason is that a supernode accumulates fill-in from all its predecessors in the task graph. As a result, the portions of the factors that correspond to task graph nodes with a large number of predecessors tend to get denser. Due to their larger supernodes, the tasks that are relatively close to the root tend to have more work associated with them. On the other hand, the width of the task graph shrinks close to the root. In other words, a typical task graph for sparse matrix factorization tends to have a large number of small independent tasks closer to the leaves, but a small number of large tasks closer to the root. An ideal parallelization strategy that would match the characteristics of the problem is as follows. Starting out, the relatively plentiful independent tasks at or near the leaves would be scheduled to parallel threads or processes. As tasks complete, other tasks become available and would be scheduled similarly. This could continue until there are enough independent tasks to keep all the threads or processes busy. When the number of available parallel tasks becomes smaller than the number of available threads or processes, then the only way to keep the latter busy would be to utilize more than one of them per task. The number of threads or processes working on individual tasks would increase as the number of parallel tasks decreases. Eventually, all threads or processes would work on the root task. The computation corresponding to the root task is equivalent to factoring a dense matrix of the size of the root supernode.

Sparse Factorization Formulations Based on Task Roles

So far in this entry, the tasks have been defined somewhat ambiguously. There are multiple ways of defining the tasks precisely, which can result in different parallel implementations of sparse matrix factorization. Clearly, a task is associated with a supernode and is responsible for computing that supernode of the factors; that is, performing the computation equivalent to the division steps in the algorithms in Figs. 1 and 2. However, a task does not own all the data that is required to compute the final values of its supernode's rows and columns. The data for performing the update steps on a supernode may be contributed by many other supernodes. Based on the tasks' responsibilities, sparse LU factorization has traditionally been classified into three categories, namely, *left-looking*, *right-looking*, and *Crout*. These variations are illustrated in Fig. 7. The traditional left-looking variant uses nonconforming supernodes made up of columns of both L and U, which are not very



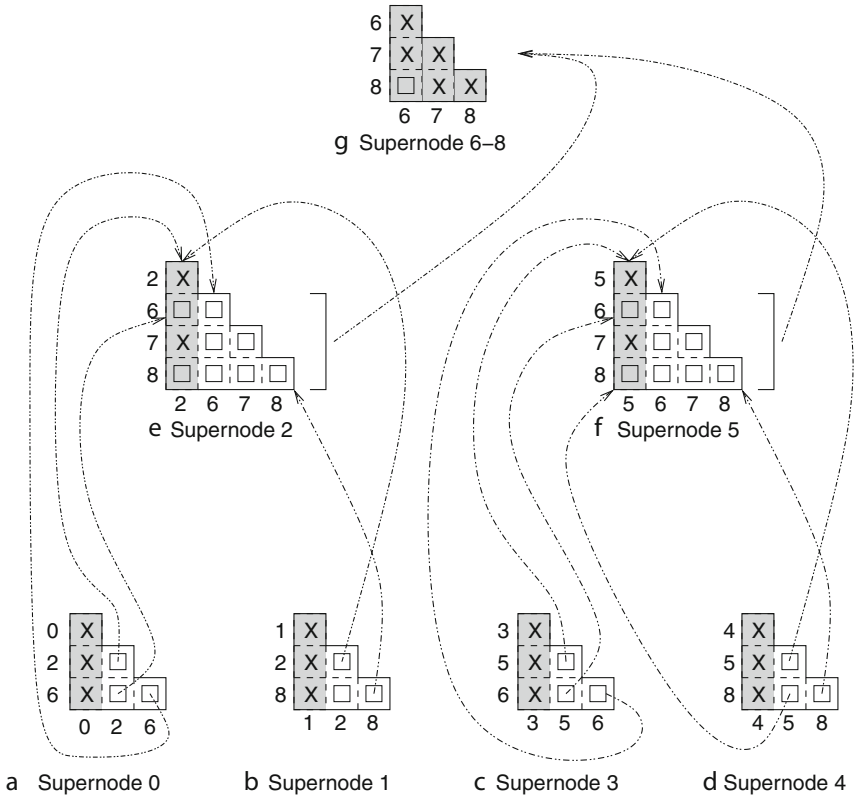
Sparse Direct Methods. Fig. 7 The left-looking (**a**), right-looking (**b**), and Crout (**c**) variations of sparse LU factorization. Different patterns indicate the parts of the matrix that are read, updated, and factored by the task corresponding to a supernode. *Blank portions* of the matrix are not accessed by this task

common in practice. In this variant, a task is responsible for gathering all the data required for its own columns from other tasks and for updating and factoring its columns. The left-looking formulation is rarely used in modern high-performance sparse direct solvers. In the right-looking variation of sparse LU, a task factors the supernode that it owns and performs all the updates that use the data from this supernode. In the Crout variation, a task is responsible for updating and factoring the supernode that it owns. Only the right-looking and Crout variants have symmetric counterparts.

A fourth variation, known as the *multifrontal method*, incorporates elements of both right-looking and Crout formulations. In the multifrontal method, the task that owns a supernode computes its own contribution to updating the remainder of the matrix (like the right-looking formulation), but does not actually apply the updates. Each task is responsible for collecting all relevant precomputed updates and applying them to its supernode (like the Crout formulation) before factoring the supernode. The supernode data and its update contribution in the multifrontal method is organized into small dense matrices called *frontal matrices*. Integer arrays maintain a mapping of the local contiguous indices of the frontal matrices to the global indices of the sparse factor matrices. Figure 8 illustrates the complete supernodal multifrontal Cholesky factorization of the symmetric matrix shown in Fig. 3a. Note that, since rows and columns with indices 6–8 form a supernode, there would be only one task (Fig. 8g) corresponding to these in the supernodal task graph (elimination tree).

When a task is ready for execution, it first constructs its frontal matrix by accumulating contributions from the frontal matrices of its children and from the coefficient matrix. It then factors its supernode, which is the portion of the frontal matrix that is shaded in Fig. 8. After factorization, the unshaded portion (this submatrix of a frontal matrix is called the *update matrix*) is updated based on the update step of the algorithm in Fig. 2. The update matrix is then used by the parent task to construct its frontal matrix.

Note that Fig. 8 illustrates a symmetric multifrontal factorization; hence, the frontal and update matrices are triangular. For general LU decomposition, these matrices would be square or rectangular. In symmetric multifrontal factorization, a child's update matrix in the elimination tree contributes only to its parent's frontal matrix. The task graph for general matrices is usually not a tree, but a DAG, as shown in Fig. 4. Apart from the shape of the frontal and update matrices, unsymmetric pattern multifrontal method differs from its symmetric counterpart in two other ways. First, an update matrix can contribute to more than one frontal matrices. Secondly, the frontal matrices receiving data from an update matrix can belong to the contributing supernode's ancestors (not necessarily parents) in the task graph.



Sparse Direct Methods. Fig. 8 Frontal matrices and data movement among them in the supernodal multifrontal Cholesky factorization of the sparse matrix shown in Fig. 3a

The multifrontal method is often the formulation of choice for highly parallel implementations of sparse matrix factorization. This is because of its natural data locality (most of the work of the factorization is performed in the well-contained dense frontal matrices) and the ease of synchronization that it permits. In general, each supernode is updated by multiple other supernodes and it can potentially update many other supernodes during the course of factorization. If implemented naively, all these updates may require excessive locking and synchronization in a shared-memory environment or generate excessive message traffic in a distributed environment. In the multifrontal method, the updates are accumulated and channeled along the paths from the leaves of the task graph to its the root. This gives a manageable structure to the potentially haphazard interaction among the tasks.

Recall that the typical supernodal sparse factorization task graph is such that the size of tasks generally increases and the number of parallel tasks generally diminishes on the way to the root from the leaves. The multifrontal method is well suited for both task parallelism (close to the leaves) and data parallelism (close to the root). Larger tasks working on large frontal matrices close to the root can readily employ multiple threads or processes to perform parallel dense matrix operations, which not only have well-understood data-parallel algorithms, but also a well-developed software base.

Pivoting in Parallel Sparse LDL^T and LU Factorization

The discussion in this entry so far has focussed on the scenario in which the rows and columns of the matrix are permuted during the ordering phase and this permutation stays static during numerical factorization. While this assumption is valid for a large class of practical problems, there are applications that would generate matrices that could encounter a zero or a very small entry on the diagonal during the factorization process.

This will cause the division step of the LU decomposition algorithm to fail or to result in numerical instability. For nonsingular matrices, this problem can be solved by interchanging rows and columns of the matrix by a process known as *partial pivoting*. When a small or zero entry is encountered at $A[i,i]$ before the division step, then row i is interchanged with another row j ($i < j \leq n$) such that $A[j,i]$ (which would occupy the location $A[i,i]$ after the interchange) is sufficiently greater in magnitude compared to other entries $A[k,i]$ ($i < k \leq n$, $k \neq j$). Similarly, instead of row i, column i could be exchanged with a suitable column j ($i < j \leq n$). In symmetric LDL^T factorization, both row and column i are interchanged simultaneously with a suitable row–column pair to maintain symmetry.

Until recently, it was believed that due to unpredictable changes in the structure of the factors due to partial pivoting, a priori ordering and symbolic factorization could not be performed, and these steps needed to be combined with numerical factorization. Keeping the analysis and numerical factorization steps separate has substantial performance and parallelization benefits, which would be lost if these steps are combined. Fortunately, modern parallel sparse solvers are able to perform partial pivoting and maintain numerical stability without mixing the analysis and numerical steps. The multifrontal method permits effective implementation of partial pivoting in parallel and keeps its effects as localized as possible.

Before computing a fill-reducing ordering, the rows or columns of the coefficient matrix are permuted such that the absolute value of the product of the magnitude of the diagonal entries is maximized. Special graph matching algorithms are used to compute this permutation. This step ensures that the diagonal entries of the matrix have relatively large magnitudes at the beginning of factorization. It has been observed that once the matrix has been permuted this way, in most cases, very few interchanges are required during the factorization process to keep it numerically stable. As a result, factorization can be performed using the static task graph and the static structures of the supernodes of L and U predicted by symbolic factorization. When an interchange is necessary, the resulting changes in the data structures are registered. Since such interchanges are rare, the resulting disruption and the overhead is usually well contained.

The first line of defense against numerical instability is to perform partial pivoting within a frontal matrix. Exchanging rows or columns within a supernode is local, and if all rows and columns of a supernode can be successfully factored by simply altering their order, then nothing outside the supernode is affected. Sometimes, a supernode cannot be factored completely by local interchanges. This can happen when all candidate rows or columns for interchange have indices greater than that of the last row–column pair of the supernode. In this case, a technique known as *delayed pivoting* is employed. The unfactored rows and columns are simply removed from the current supernode and passed onto the parent (or parents) in the task graph. Merged with the parent supernode, these rows and columns have additional candidate rows and columns available for interchange, which increases the chances of their successful factorization. The process of upward migration of unsuccessful pivots continues until they are resolved, which is guaranteed to happen at the root supernode for a nonsingular matrix.

In the multifrontal framework, delayed pivoting simply involves adding extra rows and columns to the frontal matrices of the parents of supernode with failed pivots. The process is straightforward for the supernodes whose tasks are mapped onto individual threads or processes. For the tasks that require data-parallel involvement of multiple threads or processes, the extra rows and columns can be partitioned using the same strategy that is used to partition the original frontal matrix.

Parallel Solution of Triangular Systems

As mentioned earlier, solving the original system after factoring the coefficient matrix involves solving a sparse lower triangular and a sparse upper triangular system. The task graph constructed for factorization can be used for the triangular solves too. For matrices with an unsymmetric pattern, a subset of edges of the task DAG may be redundant in each of the solve phases, but these redundant edges can be easily marked during symbolic factorization. Just like factorization, the computation for the lower triangular solve phase starts at the leaves of the task graph and proceeds toward the root. In the upper triangular solve phase, computation starts at the root and fans out toward the leaves (in other

words, the direction of the edges in the task graph is effectively reversed).

Related Entries

►Dense Linear System Solvers
►Multifrontal Method
►Reordering

Bibliographic Notes and Further Reading

Books by George and Liu [6] and Duff et al. [5] are excellent sources for a background on sparse direct methods. A comprehensive survey by Demmel et al. [4] sums up the developments in parallel sparse direct solvers until the early 1990s. Some remarkable progress was made in the development of parallel algorithms and software for sparse direct methods during a decade starting in the early 1990s. Gupta et al. [9] developed the framework for highly scalable parallel formulations of symmetric sparse factorization based on the multifrontal method (see tutorial by Liu [12] for details), and recently demonstrated scalable performance of an industrial strength implementation of their algorithms on thousands of cores [10]. Demmel et al. [3] developed one of the first scalable algorithms and software for solving unsymmetric sparse systems without partial pivoting. Amestoy et al. [1, 2] developed parallel algorithms and software that incorporated partial pivoting for solving unsymmetric systems with (either natural or forced) symmetric pattern. Hadfield [11] and Gupta [7] laid the theoretical foundation for a general unsymmetric pattern parallel multifrontal algorithm with partial pivoting, with the latter following up with a practical implementation [8].

Bibliography

1. Amestoy PR, Duff IS, Koster J, L'Excellent JY (2001) A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J Matrix Anal Appl 23(1):15–41
2. Amestoy PR, Duff IS, L'Excellent JY (2000) Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput Methods Appl Mech Eng 184:501–520
3. Demmel JW, Gilbert JR, Li XS (1999) An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. SIAM J Matrix Anal Appl 20(4):915–952
4. Demmel JW, Heath MT, van der Vorst HA (1993) Parallel numerical linear algebra. Acta Numerica 2:111–197
5. Duff IS, Erisman AM, Reid JK (1990) Direct methods for sparse matrices. Oxford University Press, Oxford, UK
6. George A, Liu JW-H (1981) Computer solution of large sparse positive definite systems. Prentice-Hall, NJ
7. Gupta A (2002) Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. SIAM J Matrix Anal Appl 24(2):529–552
8. Gupta A (2007) A shared- and distributed-memory parallel general sparse direct solver. Appl Algebra Eng Commun Comput 18(3):263–277
9. Gupta A, Karypis G, Kumar V (1997) Highly scalable parallel algorithms for sparse matrix factorization. IEEE Trans Parallel Distrib Syst 8(5):502–520
10. Gupta A, Koric S, George T (2009) Sparse matrix factorization on massively parallel computers. In: SC09 Proceedings, ACM, Portland, OR, USA
11. Hadfield SM (1992) On the LU factorization of sequences of identically structured sparse matrices within a distributed memory environment. PhD thesis, University of Florida, Gainsville, FL
12. Liu JW-H (1992) The multifrontal method for sparse matrix solution: theory and practice. SIAM Rev 34(1):82–109

Sparse Gaussian Elimination

►Sparse Direct Methods
►SuperLU

Sparse Iterative Methods, Preconditioners for

►Preconditioners for Sparse Iterative Methods

SPEC Benchmarks

Matthias Müller[1], Brian Whitney[2]
Robert Henschel[3], Kalyan Kumaran[4]
[1]Technische Universität Dresden, Dresden, Germany
[2]Oracle Corporation, Hillsboro, OR, USA
[3]Indiana University, Bloomington, IN, USA
[4]Argonne National Laboratory, Argonne, IL, USA

Synonyms

SPEC HPC96; SPEC HPC2002; SPEC MPI2007; SPEC OMP2001

Definition

Application-based Benchmarks measure the performance of computer systems by running a set of applications with a well defined configuration and workload.

Discussion

Introduction

The Standard Performance Evaluation Corporation (SPEC [Product and service names mentioned herein may be the trademarks of their respective owners]) is an organization for creating industry-standard benchmarks to measure various aspects of modern computer system performance. SPEC was founded in 1988. In January 1994, the High-Performance Group of the Standard Performance Evaluation Corporation (SPEC HPG) was founded with the mission to establish, maintain, and endorse a suite of benchmarks representative of real-world, high-performance computing applications. Several efforts joined forces to form SPEC HPG and to initiate a new benchmarking venture that is supported broadly. Founding partners included the member organizations of SPEC, former members of the Perfect Benchmark effort, and representatives of area-specific benchmarking activities. Other benchmarking organizations have joined the SPEC HPG committee since its formation.

SPEC HPG has developed various benchmark suites and its run rules over the last few years. The purpose of those benchmarks and their run rules is to further the cause of fair and objective benchmarking of high-performance computing systems. Results obtained with the benchmark suites are to be reviewed to see whether the individual run rules have been followed. Once they are accepted, the results are published on the SPEC web site (http://www.spec.org). All results, including a comprehensive description of the hardware they were produced on, are freely available. SPEC believes that the user community benefits from an objective series of tests which serve as a common reference.

The development of the benchmark suites includes obtaining candidate benchmark codes, putting these codes into the SPEC harness, testing and improving the codes' portability across as many operating systems, compilers, interconnects, runtime libraries as possible, and testing the codes for correctness and scalability.

The codes are put into the SPEC harness. The SPEC harness is a set of tools that allow users of the benchmark suite to easily run the suite, and obtain validated and publishable results. The users then only need to submit the results obtained to SPEC for review and publication on the SPEC web site.

The goals of the run rules of the benchmark suites are to ensure that published results are meaningful, comparable to other results, and reproducible. A result must contain enough information to allow another user to reproduce the result. The performance tuning methods employed when attaining a result should be more than just "prototype" or "experimental" or "research" methods; there must be a certain level of maturity and general applicability in the performance methods employed, e.g., the used compiler optimization techniques should be beneficial for other applications as well and the compiler should be generally available and supported.

Two set of metrics can be measured with the benchmark suite: "Peak" and "Base" metrics. "Peak" metrics may also be referred to as "aggressive compilation," e.g., they may be produced by building each benchmark in the suite with a set of optimizations individually selected for that benchmark, and running them with environment settings individually selected for that benchmark. The optimizations selected must adhere to the set of general benchmark optimization rules. Base optimizations must adhere to a stricter set of rules than the peak optimizations. For example, the "Base" metrics must be produced by building all the benchmarks in the suite with a common set of optimizations and running them with environment settings common to all the benchmarks in the suite.

SPEC HPC96

The efforts of SPEC HPG began in 1994 when a group from industry and academia came together to try and provide a benchmark suite based upon the principles that had started with SPEC. Two of the more popular benchmarks suites at the time were the NAS Parallel Benchmarks and the PERFECT Club Benchmarks. The group built upon the direction these benchmarks provided to produce their first benchmark, SPEC HPC96.

The benchmark SPEC HPC96 came out originally with two components, SPECseis96 and SPECchem96, with a third component SPECclimate available later.

Each of these benchmarks provided a set of rules, code, and validation which allowed benchmarking across a wide variety of hardware, including parallel platforms.

SPECseis96 is a benchmark application that was originally developed at Atlantic Richfield Corporation (ARCO). This benchmark was designed to test computation that was of interest to the oil and gas industry, in particular, time and depth migrations which are used to locate gas and oil deposits.

SPECchem96 is a benchmark based upon the application GAMESS (General Atomic and Molecular Electronic Structure System). This computational chemistry code was used in the pharmaceutical and chemical industries for drug design and bonding analysis.

SPECclimate is a benchmark based upon the application MM5, the PSU/NCAR limited-area, hydrostatic or non-hydrostatic, sigma-coordinate model designed to simulate or predict mesoscale and regional-scale atmospheric circulation. MM5 was developed by the Pennsylvania State University (Penn State) and the University Corporation for Atmospheric Research (UCAR).

SPEC HPC96 was retired in February 2003, a little after the introduction of SPEC HPC2002 and SPEC OMP2001. The results remain accessible on the SPEC web site, for reference purposes.

SPEC HPC2002

The benchmark SPEC HPC2002 was a follow-on to SPEC HPC96. The update involved using newer versions of some of the software, as well as additional parallelism models. The use of MM5 was replaced with the application WRF.

The benchmark was suitable for shared and distributed memory machines or clusters of shared memory nodes. SPEC HPC applications have been collected from among the largest, most realistic computational applications that are available for distribution by SPEC. In contrast to SPEC OMP, they were not restricted to any particular programming model or system architecture. Both shared-memory and message passing methods are supported. All codes of the current SPEC HPC2002 suite were available in an MPI and an OpenMP programming model and they included two data set sizes.

The benchmark consisted of three scientific applications:

SPECenv (WRF) is based on the WRF weather model, a state-of-the-art, non-hydrostatic mesoscale weather model, see http://www.wrf-model.org. The code consists of 25,000 lines of C and 145,000 lines of F90.

SPECseis was developed by ARCO beginning in 1995 to gain an accurate measure of performance of computing systems as it relates to the seismic processing industry for procurement of new computing resources. The code is written in F77 and C and has approximately 25,000 lines.

SPECchem used to simulate molecules ab initio, at the quantum level, and optimize atomic positions. It is a research interest under the name of GAMESS at the Gordon Research Group of Iowa State University and is of interest to the pharmaceutical industry. It consists of 120,000 lines of F77 and C.

The SPEC HPC2002 suite was retired in June 2007. The results remain accessible on the SPEC web site, for reference purposes.

SPEC OMP2001

SPEC's benchmark suite that measures performance using applications based on the OpenMP standard for shared-memory parallel processing. Two levels of workload (OMPM2001 and OMPL2001) characterize the performance of medium and large sized systems. Benchmarks running under SPEC OMPM2001 use up to 1.6 GB of memory, whereas the applications of SPEC OMPL2001 require about 6.4 GB in a 16-thread run.

The SPEC OMPM2001 benchmark suite consists of 11 large application programs, which represent the type of software used in scientific technical computing. The applications include modeling and simulation programs from the fields of chemistry, mechanical engineering, climate modeling, and physics. Of the 11 application programs, 8 are written in Fortran and 3 (AMMP, ART, and EQUAKE) are written in C. The benchmarks require a virtual address space of about 1.5 GB in a 1-processor execution. The rationales for this size were to provide data sets fitting in a 32-bit address space.

SPEC OMPL2001 consists of 9 application programs, of which 7 are written in Fortran and 2 (ART and

EQUAKE) are written in C. The benchmarks require a virtual address space of about 6.4 GB in a 16-processor run. The rationale for this size were to provide data sets significantly larger than those of the SPEC OMPM benchmarks, with a requirement for a 64-bit address space.

The following is a short description of the application programs of OMP2001:

APPLU Solves five coupled non-linear PDEs on a 3-dimensional logically structured grid, using the Symmetric Successive Over-Relaxation implicit time-marching scheme.

APSI Lake environmental model, which predicts the concentration of pollutants. It solves the model for the mesoscale and synoptic variations of potential temperature, wind components, and for the mesoscale vertical velocity, pressure, and distribution of pollutants.

MGRID Simple multigrid solver, which computes a 3-dimensional potential field.

SWIM Weather prediction model, which solves the shallow water equations using a finite difference method.

FMA3D Crash simulation program. It simulates the inelastic, transient dynamic response of 3-dimensional solids and structures subjected to impulsively or suddenly applied loads. It uses an explicit finite element method.

ART (Adaptive Resonance Theory) neural network, which is used to recognize objects in a thermal image. The objects in the benchmark are a helicopter and an airplane.

GAFORT Computes the global maximum fitness using a genetic algorithm. It starts with an initial population and then generates children who go through crossover, jump mutation, and creep mutation with certain probabilities.

EQUAKE Is an earthquake-modeling program. It simulates the propagation of elastic seismic waves in large, heterogeneous valleys in order to recover the time history of the ground motion everywhere in the valley due to a specific seismic event. It uses a finite element method on an unstructured mesh.

WUPWISE (Wuppertal Wilson Fermion Solver) is a program in the field of lattice gauge theory. Lattice gauge theory is a discretization of quantum chromodynamics. Quark propagators are computed within a chromodynamic background field. The inhomogeneous lattice-Dirac equation is solved.

GALGEL This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids. This program is only part of OMPM2001.

AMMP (Another Molecular Modeling Program) is a molecular mechanics, dynamics, and modeling program. The benchmark performs a molecular dynamics simulation of a protein-inhibitor complex, which is embedded in water. This program is only part of OMPM2001.

SPEC MPI2007

SPEC MPI2007 is SPEC's benchmark suite for evaluating MPI-parallel, floating point, compute-intensive performance across a wide range of cluster and SMP hardware. MPI2007 continues the SPEC tradition of giving users the most objective and representative benchmark suite for measuring and comparing high-performance computer systems.

SPEC MPI2007 focuses on performance of compute intensive applications using the Message-Passing Interface (MPI), which means these benchmarks emphasize the performance of the type of computer processor (CPU), the number of computer processors, the MPI Library, the communication interconnect,the memory architecture,the compilers, and the shared file system.

It is important to remember the contribution of all these components. SPEC MPI performance intentionally depends on more than just the processor. MPI2007 is not intended to stress other computer components such as the operating system, graphics, or the I/O system. Table 1 contains the list of codes, together with information on the benchmark set size, programming language, and application area.

104.milc,142.dmilc stands for MIMD Lattice Computation and is a quantum chromodynamics (QCD) code for lattice gauge theory with dynamical quarks. Lattice gauge theory involves the study of some of the fundamental constituents of matter, namely

SPEC Benchmarks. Table 1 List of applications in SPEC MPI2007

Benchmark	Suite	Language	Application domain
104.milc	medium	C	Physics: Quantum Chromodynamics (QCD)
107.leslie3d	medium	Fortran	Computational Fluid Dynamics (CFD)
113.GemsFDTD	medium	Fortran	Computational Electromagnetics (CEM)
115.fds4	medium	C/Fortran	Computational Fluid Dynamics (CFD)
121.pop2	medium, large	C/Fortran	Ocean Modeling
122.tachyon	medium, large	C	Graphics: Parallel Ray Tracing
125.RAxML	large	C	DNA Matching
126.lammps	medium, large	C++	Molecular Dynamics Simulation
127.wrf2	medium	C/Fortran	Weather Prediction
128.GAPgeofem	medium, large	C/Fortran	Heat Transfer using Finite Element Methods (FEM)
129.tera_tf	medium, large	Fortran	3D Eulerian Hydrodynamics
130.socorro	medium	C/Fortran	Molecular Dynamics using Density-Functional Theory (DFT)
132.zeusmp2	medium, large	C/Fortran	Physics: Computational Fluid Dynamics (CFD)
137.lu	medium, large	Fortran	Computational Fluid Dynamics (CFD)
142.dmilc	large	C	Physics: Quantum Chromodynamics (QCD)
143.dleslie	large	Fortran	Computational Fluid Dynamics (CFD)
145.lGemsFDTD	large	Fortran	Computational Electromagnetics (CEM)
147.l2wrf2	large	C/Fortran	Weather Prediction

quarks and gluons. In this area of quantum field theory traditional perturbative expansions are not useful and introducing a discrete lattice of space-time points is the method of choice.

107.leslie3d,143.dleslie The main purpose of this code is to model chemically reacting (i.e., burning) turbulent flows. Various different physical models are available in this algorithm. For MPI2007, the program has been set up a to solve a test problem which represents a subset of such flows, namely the temporal mixing layer. This type of flow occurs in the mixing regions of all combustors that employ fuel injection (which is nearly all combustors). Also, this sort of mixing layer is a benchmark problem used to understand physics of turbulent mixing.

LESlie3d uses a strongly conservative, finite-volume algorithm with the MacCormack Predictor-Corrector time integration scheme. The accuracy is fourth-order spatially and second-order temporally.

113.GemsFDTD,145.lGemsFDTD GemsFDTD solves the Maxwell equations in 3D in the time domain using the finite-difference time-domain (FDTD) method. The radar cross section (RCS) of a perfectly conducting (PEC) object is computed. GemsFDTD is a subset of the code GemsTD developed in the General ElectroMagnetic Solvers (GEMS) project.

The core of the FDTD method are second-order accurate central-difference approximations of the Faraday's and Ampere's laws. These central-differences are employed on a staggered Cartesian grid resulting in an explicit finite-difference method. The FDTD method is also referred to as the Yee scheme. It is the standard time-domain method within computational electrodynamics (CEM).

An incident plane wave is generated using so-called Huygens' surfaces. This means that the computational domain is split into a total field part and a scattered field part, where the scattered field part surrounds the total field part. A time-domain near-to-far-field transformation computes the RCS according to Martin and Pettersson. Fast Fourier transforms (FFT) are employed in the post-processing.

145.lGemsFDTD contains extensive performance improvements as compared with 113.GemsFDTD.

115.fds4 is a computational fluid dynamics (CFD) model of fire-driven fluid flow. The software solves numerically a form of the Navier-Stokes equations appropriate for low-speed, thermally driven flow with an emphasis on smoke and heat transport from fires. It uses the block64 test case as dataset. This dataset is similar to ones used to simulate fires in the World Trade Center. The author's agency did the investigation of the collapse.

121.pop2 The Parallel Ocean Program (POP) is a descendant of the Bryan-Cox-Semtner class of ocean models first developed by Kirk Bryan and Michael Cox at the NOAA Geophysical Fluid Dynamics Laboratory in Princeton, NJ, in the late 1960s. POP had its origins in a version of the model developed by Semtner and Chervin. POP is the ocean component of the Community Climate System Model. Time integration of the model is split into two parts. The three-dimensional vertically varying (baroclinic) tendencies are integrated explicitly using a leapfrog scheme. The very fast vertically uniform (barotropic) modes are integrated using an implicit free surface formulation in which a preconditioned conjugate gradient solver is used to solve for the two-dimensional surface pressure.

122.tachyon is a ray tracing program. It implements all of the basic geometric primitives such as triangles, planes, spheres, cylinders, etc. Tachyon is nearly embarrassingly parallel. As a result, MPI usage tends to be much lower as compared to other types of MPI applications. The scene to be rendered is partitioned into a fixed number of pieces, which are distributed out by the master process to each processor participating in the computation. Each processor then renders its piece of the scene in parallel, independent of the other processors. Once a processor completes the rendering of its particular piece of the scene, it waits until the other processors have rendered their pieces of the scene, and then transmits its piece back to the master process. The process is repeated until all pieces of the scene have been rendered.

126.lammps is a classical molecular dynamics simulation code designed to run efficiently on parallel computers. It was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. LAMMPS divides 3D space into 3D sub-volumes, e.g., a $P = A \times B \times C$ grid of processors, where P is the total number of processors. It tries to make the sub-volumes as cubic as possible, since the volume of data exchanged is proportional to the surface of the sub-volume.

127.wrf2,147.l2wrf is a weather forecasting code based on the Weather Research and Forecasting (WRF) Model, which is a next-generation mesocale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. The code is written in Fortran 90. WRF features multiple dynamical cores, a 3-dimensional variational (3DVAR) data assimilation system, and a software architecture allowing for computational parallelism and system extensibility. Multi-level parallelism support includes distributed memory (MPI), shared memory (OpenMP), and hybrid shared/distributed modes of execution. In the SPEC MPI2007 version of WRF, all OpenMP directives have been switched off. WRF version 2.0.2 is used in the 127.wrf2 benchmark version. WRF version 2.1.2 is used in the benchmark version, 147.l2wrf2.

128.GAPgeofem is a GeoFEM-based parallel finite-element code for transient thermal conduction with gap radiation and very heterogeneous material property. GeoFEM is an acronym for Geophysical Finite Element Methods, and it is a software used on the Japanese Earth Simulator system for the modeling of solid earth phenomena such as mantle-core convection, plate tectonics, and seismic wave propagation and their coupled phenomena. A backward Euler implicit time-marching scheme has been adopted. Linear equations are solved by parallel CG (conjugate gradient) iterative solvers with point Jacobi preconditioning.

129.tera_tf is a three dimensional Eulerian hydrodynamics application using a 2nd order Godunov-type scheme and a 3rd order remapping. It uses mostly point-to-point messages, and some reductions use non-blocking messages. The global domain is a cube, with N cells in each direction, which amounts to a total number of N^3 cells. To set up the problem, one needs to define the number of cells in each direction, and the number of blocks in each direction. Each block corresponds to an MPI task.

130.socorro is a modular, object oriented code for performing self-consistent electronic-structure calculations utilizing the Kohn-Sham formulation of density-functional theory. Calculations are performed using a plane wave basis and either norm-conserving pseudopotentials or projector augmented wave functions. Several exchange-correlation functionals are available for use including the local-density approximation (Perdew-Zunger or Perdew-Wang parameterizations of the Ceperley-Alder QMC correlation results) and the generalized-gradient approximation (PW91, PBE, and BLYP). Both Fourier-space and real-space projectors have been implemented, and a variety of methods are available for relaxing atom positions, optimizing cell parameters, and performing molecular dynamics calculations.

132.zeusmp2 is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, SDSC, University of Illinois at Urbana-Champaign, UC San Diego) for the simulation of astrophysical phenomena. The program solves the equations of ideal (non-resistive), non-relativistic, hydrodynamics and magnetohydrodynamics, including externally applied gravitational fields and self-gravity. The gas can be adiabatic or isothermal, and the thermal pressure is isotropic. Boundary conditions may be specified as reflecting, periodic, inflow, or outflow.

132.zeusmp2 is based on ZEUS-MP Version 2, which is a Fortran 90 rewrite of ZEUS-MP under development at UCSD (by John Hayes). It includes new physics models such as flux-limited radiation diffusion (FLD), and multispecies fluid advection is added to the physics set. ZEUS-MP divides the computational space into 3D tiles that are distributed across processors. The physical problem solved in SPEC MPI2007 is a 3D blastwave simulated with the presence of a uniform magnetic field along the x-direction. A Cartesian grid is used and the boundaries are "outflow."

137.lu has a rich ancestry in benchmarking. Its immediate predecessor is the LU benchmark in NPB3.2-MPI, part of the NAS Parallel Benchmark suite. It is sometimes referred to as APPLU (a version of which was 173.applu in CPU2000) or NAS-LU. The NAS-LU code is a simplified compressible Navier-Stokes equation solver. It does not perform an LU factorization, but instead implements a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system.

The code computes the solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit pseudo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. This scheme is functionally equivalent to a nonlinear block SSOR iterative scheme with lexicographic ordering. Spatial discretization of the differential operators are based on a second-order accurate finite volume scheme. It insists on the strict lexicographic ordering during the solution of the regular sparse lower and upper triangular matrices. As a result, the degree of exploitable parallelism during this phase is limited to $O(N^2)$ as opposed to $O(N^3)$ in other phases and its spatial distribution is non-homogenous. This fact also creates challenges during the loop re-ordering to enhance the cache locality.

Related Entries

►Benchmarks
►HPC Challenge Benchmark
►NAS Parallel Benchmarks
►Perfect Benchmarks

Bibliographic Notes and Further Reading

There are numerous efforts to create benchmarks for different purposes. One of the early application benchmarks are the so-called "Perfect Club Benchmarks"[6], an effort that among others initiated the SPEC High Performance Group (HPG) activities[9]. The goal of SPEC HPG is to create application benchmarks to measure the performance of High Performance Computers. There are also other efforts that share this goal. Often such a collection is assembled during a procurement process. However, it normally is not used outside this specific process, nor does such a collection claim to be representative for a wider community. Two of the collections that are in wider use are the NAS Parallel Benchmarks [5] and the HPC Challenge benchmark [8]. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel

supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. HPCC consists of synthetic kernels measuring different aspects of a parallel system, like CPU, memory subsystem, and interconnect.

Two other benchmarks that consist of applications are the ASCI-benchmarks used in the procurement process of various ASCI-machines. ASCI-Purple [1] consists of nine benchmarks and three stress tests. Its last update was in 2003. The ASCI-Sequoia benchmark [2] is the successor, it consists of 17 applications and synthetic benchmarks, but only seven codes are MPI parallel. A similar collection is the DEISA benchmark suite [7]. It contains 14 codes, but for licensing reasons, three of the benchmarks must be obtained directly from the code authors and placed in the appropriate location within the benchmarking framework.

There are also a number of publications with more details about the SPEC HPG benchmarks. Details of SPEC HPC2002 are available in [10].Characteristics of the SPEC benchmark suite OMP2001 are described by Saito et al. [14] and Müller et. al [12]. Aslot et al. [3] have presented the benchmark suite. Aslot et al.[4] and Iwashita et al.[11] have described performance characteristics of the benchmark suite. The SPEC High Performance Group published a more detailed description of MPI2007 in [13]. Some performance characteristics are depicted in [15].

Bibliography

1. ASCI-Purple home page (2003) https://asc.llnl.gov/computing_resources/purple/archive/benchmarks
2. ASCI-Sequoia home page. https://asc.llnl.gov/sequoia/benchmarks.
3. Aslot V, Domeika M, Eigenmann R, Gaertner G, Jones WB, Parady B (2001) SPEComp: a new benchmark suite for measuring parallel computer performance. In: Eigenmann R, Voss MJ (eds) WOMPAT'01: workshop on openmp applications and tools. LNCS, vol 2104. Springer, Heidelberg, pp 1–10
4. Aslot V, Eigenmann R (2001) Performance characteristics of the SPEC OMP2001 benchmarks. In: 3rd European workshop on OpenMP, EWOMP'01, Barcelona, Spain, September 2001
5. Bailey D, Harris T, Saphir W, van der Wijngaart R, Woo A, Yarrow M (1995) The NAS parallel benchmarks 2.0. Technical report NAS-95-020, NASA Ames Research Center, Moffett Field, CA. http://www.nas.nasa.gov/Software/NPB
6. Berry M, Chen D, Koss P, Kuck D, Lo S, Pang Y, Pointer, Rolo R, Sameh A, Clementi E, Chin S, Schneider D, Fox G, Messina P, Walker D, Hsiung C, Schwarzmeier J, Lue K, Orszag S, Seidl F, Johnson O, Goodrum R, Martin J (1989) The perfect club benchmarks: effective performance evaluation of supercomputers. Int J High Perform Comp Appl 3(3):5–40
7. DEISA benchmark suite home page. http://www.deisa.eu/science/benchmarking
8. Dongarra J, Luszczek P (2005) Introduction to the hpcchallenge benchmark suite. ICL technical report ICL-UT-05-01, ICL 2005
9. Eigenmann R, Hassanzadeh S (1996) Benchmarking with real industrial applications: the SPEC high-performance group. IEEE Comp Sci & Eng 3(1):18–23
10. Eigenmann R, Gaertner G, Jones W (2002) SPEC HPC2002: the next high-performance computer benchmark. In: Lecture notes in computer science, vol 2327. Springer, Heidelberg, pp 7–10
11. Iwashita H, Yamanaka E, Sueyasu N, van Waveren M, Miura K (2001) The SPEC OMP 2001 benchmark on the Fujitsu PRIMEPOWER system. In: 3rd European workshop on OpenMP, EWOMP'01, Barcelona, Spain, September 2001
12. Müller MS, Kalyanasundaram K, Gaertner G, Jones W, Eigenmann R, Lieberman R, van Waveren M, Whitney B (2004) SPEC HPG benchmarks for high performance systems. Int J High Perform Comp Netw 1(4):162–170
13. Müuller MS, van Waveren M, Lieberman R, Whitney B, Saito H, Kumaran K, Baron J, Brantley WC, Parrott C, Elken T, Feng H, Ponder C (2010) SPEC MPI2007 – an application benchmark suite for parallel systems using MPI. Concurrency Computat: Pract Exper 22(2):191–205
14. Saito H, Gaertner G, Jones W, Eigenmann R, Iwashita H, Lieberman R, van Waveren M, Whitney B (2002) Large system performance of SPEC OMP2001 benchmarks. In: Zima HP, Joe K, Sata M, Seo Y, Shimasaki M (eds) High performance computing, 4th international symposium, ISHPC 2002. Lecture notes in computer science, vol 2327. Springer, Heidelberg, pp 370–379
15. Szebenyi Z, Wylie BJN, Wolf F (2008) SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proceedings of the 1st SPEC international performance evaluation workshop (SIPEW). LNCS, vol 5119. Springer, Heidelberg, pp 99–123

SPEC HPC2002

►SPEC Benchmarks

SPEC HPC96

►SPEC Benchmarks

SPEC MPI2007

►SPEC Benchmarks

SPEC OMP2001

►SPEC Benchmarks

Special-Purpose Machines

►Anton, a Special-Purpose Molecular Simulation Machine
►GRAPE
►JANUS FPGA-Based Machine
►QCD apeNEXT Machines
►QCDSP and QCDOC Computers

Speculation

►Speculative Parallelization of Loops
►Speculation, Thread-Level
►Transactional Memory

Speculation, Thread-Level

Josep Torrellas
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Speculative multithreading (SM); Speculative parallelization; Speculative run-time parallelization; Speculative threading; Speculative thread-level parallelization; Thread-level data speculation (TLDS); Thread level speculation (TLS) parallelization; TLS

Definition

Thread-Level Speculation (TLS) refers to an environment where execution threads operate speculatively, performing potentially unsafe operations, and temporarily buffering the state they generate in a buffer or cache. At a certain point, the operations of a thread are declared to be correct or incorrect. If they are correct, the thread commits, merging the state it generated with the correct state of the program; if they are incorrect, the thread is squashed and typically restarted from its beginning. The term TLS is most often associated to a scenario where the purpose is to execute a sequential application in parallel. In this case, the compiler or the hardware breaks down the application into speculative threads that execute in parallel. However, strictly speaking, TLS can be applied to any environment where threads are executed speculatively and can be squashed and restarted.

Discussion

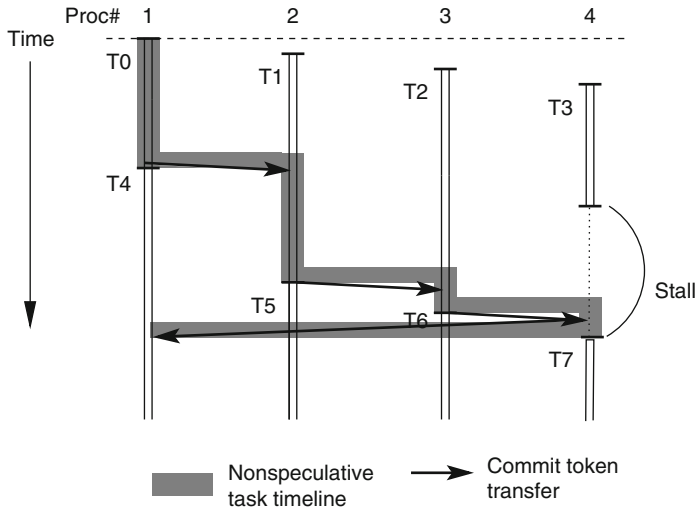
Basic Concepts in Thread-Level Speculation

In its most common use, Thread-Level Speculation (TLS) consists of extracting units of work (i.e., tasks) from a sequential application and executing them on different threads in parallel, hoping not to violate sequential semantics. The control flow in the sequential code imposes a relative ordering between the tasks, which is expressed in terms of predecessor and successor tasks. The sequential code also induces a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is *Speculative* when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. Otherwise, the task is nonspeculative. The memory accesses issued by speculative tasks are called speculative memory accesses.

When a nonspeculative task finishes execution, it is ready to *Commit*. The role of commit is to inform the rest of the system that the data generated by the task is now part of the safe, nonspeculative program state. Among other operations, committing always involves passing the *Commit Token* to the immediate successor task. This is because maintaining correct sequential semantics in the parallel execution requires that tasks commit in order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires nonspeculative status and all its predecessors have committed.

Figure 1 shows an example of several tasks running on four processors. In this example, when task T3 executing on processor 4 finishes the execution, it cannot commit until its predecessor tasks T0, T1, and T2 also finish and commit. In the meantime, depending on



Speculation, Thread-Level. Fig. 1 A set of tasks executing on four processors. The figure shows the nonspeculative task timeline and the transfer of the commit token

the hardware support, processor 4 may have to stall or may be able to start executing speculative task T7. The example also shows how the nonspeculative task status changes as tasks finish and commit, and the passing of the commit token.

Memory accesses issued by a speculative task must be handled carefully. Stores generate *Speculative Versions* of data that cannot simply be merged with the nonspeculative state of the program. The reason is that they may be incorrect. Consequently, these versions are stored in a *Speculative Buffer* local to the processor running the task – e.g., the first-level cache. Only when the task becomes nonspeculative are its versions safe.

Loads issued by a speculative task try to find the requested datum in the local speculative buffer. If they miss, they fetch the correct version from the memory subsystem, i.e., the closest predecessor version from the speculative buffers of other tasks. If no such version exists, they fetch the datum from memory.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware or software support that tracks, for each individual task, the data that the task wrote and the data that the task read without first writing it. A data-dependence violation is flagged when a task modifies a datum that has been read earlier by a successor task. At this point, the consumer task is *squashed* and all the data versions that it has produced are discarded. Then, the task is re-executed.

Figure 2 shows an example of a data-dependence violation. In the example, each iteration of a loop is a task. Each iteration issues two accesses to an array, through an un-analyzable subscripted subscript. At run-time, iteration J writes A[5] after its successor iteration J+2 reads A[5]. This is a Read After Write (RAW) dependence that gets violated due to the parallel execution. Consequently, iteration J+2 is squashed and restarted. Ordinarily, all the successor tasks of iteration J+2 are also squashed at this time because they may have consumed versions generated by the squashed task. While it is possible to selectively squash only tasks that used incorrect data, it would involve extra complexity. Finally, as iteration J+2 re-executes, it will re-read A[5]. However, at this time, the value read will be the version generated by iteration J.

Note that WAR and WAW dependence violations do not need to induce task squashes. The successor task has prematurely written the datum, but the datum remains buffered in its speculative buffer. A subsequent read from a predecessor task (in a WAR violation) will get a correct version, while a subsequent write from a predecessor task (in a WAW violation) will generate a version that will be merged with main memory before the one from the successor task.

Speculation, Thread-Level. Fig. 2 Example of a data-dependence violation

However, many proposed TLS schemes, to reduce hardware complexity, induce squashes in a variety of situations. For instance, if the system has no support to keep different versions of the same datum in different speculative buffers in the machine, cross-task WAR and WAW dependence violations induce squashes. Moreover, if the system only tracks accesses on a per-line basis, it cannot disambiguate accesses to different words in the same memory line. In this case, false sharing of a cache line by two different processors can appear as a data-dependence violation and also trigger a squash.

Finally, while TLS can be applied to various code structures, it is most often applied to loops. In this case, tasks are typically formed by a set of consecutive iterations.

The rest of this article is organized as follows: First, the article briefly classifies TLS schemes. Then, it describes the two major problems that any TLS scheme has to solve, namely, buffering and managing speculative state, and detecting and handling dependence violations. Next, it describes the initial efforts in TLS, other uses of TLS, and machines that use TLS.

Classification of Thread-Level Speculation Schemes

There have been many proposals of TLS schemes. They can be broadly classified depending on the emphasis on hardware versus software, and the type of target machine.

The majority of the proposed schemes use hardware support to detect cross-task dependence violations that result in task squashes (e.g., [1, 4, 6, 8, 11, 12, 14, 16, 18, 20, 23, 27, 28, 31, 32, 36]). Typically, this is attained by using the hardware cache coherence protocol, which sends coherence messages between the caches when multiple processors access the same memory line. Among all these hardware-based schemes, the majority rely on a compiler or a software layer to identify and prepare the tasks that should be executed in parallel. Consequently, there have been several proposals for TLS compilers (e.g., [9, 19, 33, 34]). Very few schemes rely on the hardware to identify the tasks (e.g., [1]).

Several schemes, especially in the early stages of TLS research, proposed software-only approaches to TLS (e.g., [7, 13, 25, 26]). In this case, the compiler typically generates code that causes each task to keep shadow locations and, after the parallel execution, checks if multiple tasks have updated a common location. If they have, the original state is restored.

Most proposed TLS schemes target small shared-memory machines of about two to eight processors (e.g., [14, 18, 27, 29]). It is in this range of parallelism that TLS is most cost effective. Some TLS proposals have focused on smaller machines and have extended a superscalar core with some hardware units that execute threads speculatively [1, 20]. Finally, some TLS proposals have targeted scalable multiprocessors [4, 23, 28]. This is a more challenging environment, given the longer communication latencies involved. It requires applications that have significant parallelism that cannot be analyzed statically by the compiler.

Buffering and Managing Speculative State

The state produced by speculative tasks is unsafe, since such tasks may be squashed. Therefore, any TLS scheme must be able to identify such state and, when necessary, separate it from the rest of the memory state. For this, TLS systems use structures, such as caches [4, 6, 12, 18, 28], and special buffers [8, 14, 23, 32], or undo logs [7, 11, 36]. This section outlines the challenges in buffering and managing speculative state. A more detailed analysis and a taxonomy is presented by Garzaran et al. [10].

Multiple Versions of the Same Variable in the System

Every time that a task writes for the first time to a variable, a new version of the variable appears in the

system. Thus, two speculative tasks running on different processors may create two different versions of the same variable [4, 12]. These versions need to be buffered separately, and special actions may need to be taken so that a reader task can find the correct version out of the several coexisting in the system. Such a version will be the version created by the producer task that is the closest predecessor of the reader task.

A task has at most a single version of any given variable, even if it writes to the variable multiple times. The reason is that, on a dependence violation, the whole task is undone. Therefore, there is no need to keep intermediate values of the variable.

Multiple Speculative Tasks per Processor

When a processor finishes executing a task, the task may still be speculative. If the TLS buffering support is such that the processor can only hold state from a single speculative task, the processor stalls until the task commits. However, to better tolerate task load imbalance, the local buffer may have been designed to buffer state from several speculative tasks, enabling the processor to execute another speculative task. In this case, the state of each task must be tagged with the ID of the task.

Multiple Versions of the Same Variable in a Single Processor

When a processor buffers state from multiple speculative tasks, it is possible that two such tasks create two versions of the same variable. This occurs in load-imbalanced applications that exhibit private data patterns (i.e., WAW dependences between tasks). In this case, the buffer will have to hold multiple versions of the same variable. Each version will be tagged with a different task ID. This support introduces complication to the buffer or cache. Indeed, on an external request, extra comparisons will need to be done if the cache has two versions of the same variable.

Merging of Task State

The state produced by speculative tasks is typically merged with main memory at task commit time; however, it can instead be merged as it is being generated. The first approach is called *Architectural Main Memory (AMM)* or *Lazy Version Management*; the second one is called *Future Main Memory (FMM)* or *Eager Version Management*. These schemes differ on whether the main memory contains only safe data (AMM) or it can also contain speculative data (FMM).

In AMM systems, all speculative versions remain in caches or buffers that are kept separate from the coherent memory state. Only when a task becomes nonspeculative can its buffered state be merged with main memory. In a straightforward implementation, when a task commits, all the buffered dirty cache lines are merged with main memory, either by writing back the lines to memory [4] or by requesting ownership for them to obtain coherence with main memory [28].

In FMM systems, versions from speculative tasks are merged with the coherent memory when they are generated. However, to enable recovery from task squashes, when a task generates a speculative version of a variable, the previous version of the variable is saved in a log. Note that, in both approaches, the coherent memory state can temporarily reside in caches, which function in their traditional role of extensions of main memory.

Detecting and Handling Dependence Violations

Basic Concepts

The second aspect of TLS involves detecting and handling dependence violations. Most TLS proposals focus on data dependences, rather than control dependences. To detect (cross-task) data-dependence violations, most TLS schemes use the same approach. Specifically, when a speculative task writes a datum, the hardware sets a Speculative Write bit associated with the datum in the cache; when a speculative task reads a datum before it writes to it (an event called *Exposed Read*), the hardware sets an Exposed Read bit. Depending on the TLS scheme supported, these accesses also cause a tag associated with the datum to be set to the ID of the task.

In addition, when a task writes a datum, the cache coherence protocol transaction that sends invalidations to other caches checks these bits. If a successor task has its Exposed Read bit set for the datum, the successor task has prematurely read the datum (i.e., this is a RAW dependence violation), and is squashed [18].

If the Speculative Write and Exposed Read bits are kept on a per-word basis, only dependences on the same word can cause squashes. However, keeping and maintaining such bits on a per-word basis in caches, network

messages, and perhaps directory modules is costly in hardware. Moreover, it does not come naturally to the coherence protocol of multiprocessors, which operate at the granularity of memory lines.

Keeping these bits on a per-line basis is cheaper and compatible with mainstream cache coherence protocols. However, the hardware cannot then disambiguate accesses at word level. Furthermore, it cannot combine different versions of a line that have been updated in different words. Consequently, cross-task RAW and WAW violations, on both the same word and different words of a line (i.e., false sharing), cause squashes.

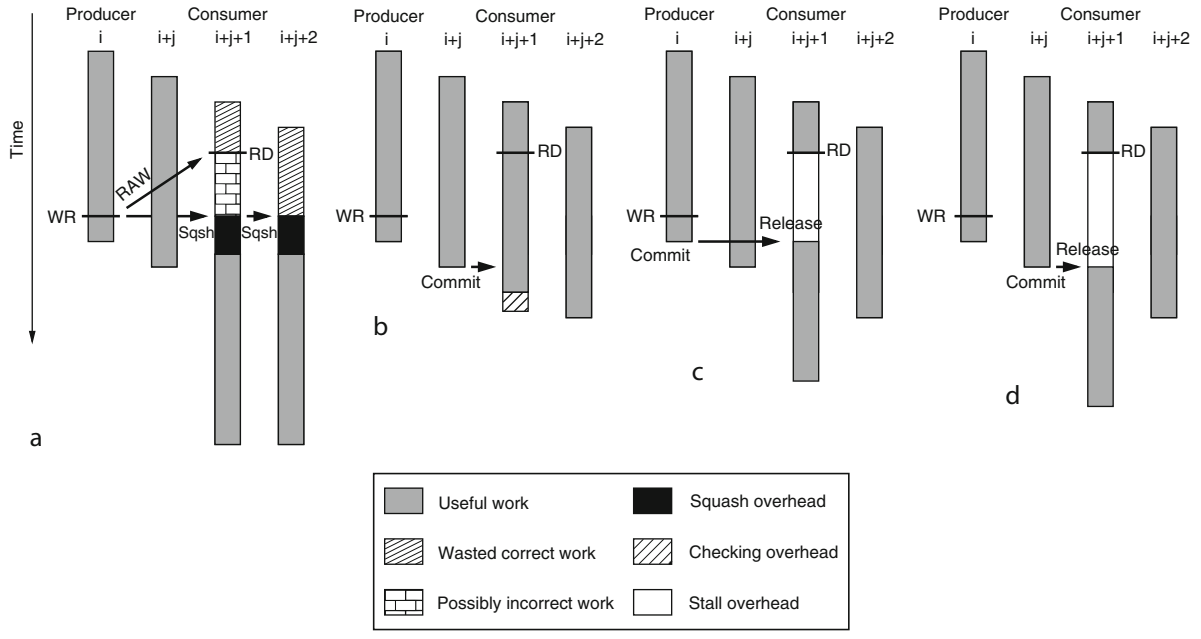
Task squash is a very costly operation. The cost is threefold: overhead of the squash operation itself, loss of whatever correct work has already been performed by the offending task and its successors, and cache misses in the offending task and its successors needed to reload state when restarting. The latter overhead appears because, as part of the squash operation, the speculative state in the cache is invalidated. Figure 3a shows an example of a RAW violation across tasks *i* and *i+j+1*. The consumer task and its successors are squashed.

Techniques to Avoid Squashes

Since squashes are so expensive, there are techniques to avoid them. If the compiler can conclude that a certain pair of accesses will frequently cause a data-dependence violation, it can statically insert a synchronization operation that forces the correct task ordering at runtime.

Alternatively, the machine can have hardware support that records, at runtime, where dependence violations occur. Such hardware may record the program counter of the read or writes involved, or the address of the memory location being accessed. Based on this information, when these program counters are reached or the memory location is accessed, the hardware can try one of several techniques to avoid the violation. This section outlines some of the techniques that can be used. A more complete description of the choices is presented by Cintra and Torrellas [5]. Without loss of generality, a RAW violation is assumed.

Based on past history, the predictor may predict that the pair of conflicting accesses are engaged in false sharing. In this case, it can simply allow the read to proceed and then the subsequent write to execute silently, without sending invalidations. Later, before the



Speculation, Thread-Level. Fig. 3 RAW data-dependence violation that results in a squash (**a**) or that does not cause a squash due to false sharing or value prediction (**b**), or consumer stall (**c** and **d**)

consumer task is allowed to commit, it is necessary to check whether the sections of the line read by the consumer overlap with the sections of the line written by the producer. This can be easily done if the caches have per-word access bits. If there is no overlap, it was false sharing and the squash is avoided. Figure 3b shows the resulting time line.

When there is a true data dependence between tasks, a squash can be avoided with effective use of value prediction. Specifically, the predictor can predict the value that the producer will produce, speculatively provide it to the consumer's read, and let the consumer proceed. Again, before the consumer is allowed to commit, it is necessary to check that the value provided was correct. The timeline is also shown in Fig. 3b.

In cases where the predictor is unable to predict the value, it can avoid the squash by stalling the consumer task at the time of the read. This case can use two possible approaches. An aggressive approach is to release the consumer task and let it read the current value as soon as the predicted producer task commits. The time line is shown in Fig. 3c. In this case, if an intervening task between the first producer and the consumer later writes the line, the consumer will be squashed. A more conservative approach is not to release the consumer task until it becomes nonspeculative. In this case, the presence of multiple predecessor writers will not squash the consumer. The time line is shown in Fig. 3d.

Initial Efforts in Thread-Level Speculation

An early proposal for hardware support for a form of speculative parallelization was made by Knight [16] in the context of functional languages. Later, the Multiscalar processor [27] was the first proposal to use a form of TLS within a single-chip multithreaded architecture. A software-only form of TLS was proposed in the LRPD test [25]. Early proposals of hardware-based TLS include the work of several authors [14, 17, 21, 29, 35].

Other Uses of Thread-Level Speculation

TLS concepts have been used in environments that have goals other than trying to parallelize sequential programs. For example, they have been used to speed up explicitly parallel programs through Speculative Synchronization [22], or for parallel program debugging [24] or program monitoring [37]. Similar concepts to TLS have been used in systems supporting hardware transactional memory [15] and continuous atomic-block operation [30].

Machines that Use Thread-Level Speculation

Several machines built by computer manufacturers have hardware support for some form of TLS – although the specific implementation details are typically not disclosed. Such machines include systems designed for Java applications such as Sun Microsystems' MAJC chip [31] and Azul Systems' Vega processor [2]. The most high-profile system with hardware support for speculative threads is Sun Microsystems' ROCK processor [3]. Other manufacturers are rumored to be developing prototypes with similar hardware.

Bibliography

1. Akkary H, Driscoll M (1998) A dynamic multithreading processor. In: International symposium on microarchitecture, Dallas, November 1998
2. Azul Systems. Vega 3 Processor. http://www.azulsystems.com/products/vega/processor
3. Chaudhry S, Cypher R, Ekman M, Karlsson M, Landin A, Yip S, Zeffer H, Tremblay M (2009) Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's ROCK Processor. In: International symposium on computer architecture, Austin, June 2009
4. Cintra M, Martínez JF, Torrellas J (2000) Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: International symposium on computer architecture, Vancouver, June 2000, pp 13–24
5. Cintra M, Torrellas J (2002) Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In: Proceedings of the 8th High-Performance computer architecture conference, Boston, Feb 2002
6. Figueiredo R, Fortes J (2001) Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocesors. In: Proceedings of the international conference on parallel processing, Valencia, Spain, September 2001
7. Frank M, Lee W, Amarasinghe S (2001) A software framework for supporting general purpose applications on raw computation fabrics. Technical report, MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001
8. Franklin M, Sohi G (1996) ARB: a hardware mechanism for dynamic reordering of memory references. IEEE Trans Comput 45(5):552–571

9. Garcia C, Madriles C, Sanchez J, Marcuello P, Gonzalez A, Tullsen D (2005) Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In: Conference on programming language design and implementation, Chicago, Illinois, June 2005
10. Garzarán M, Prvulovic M, Llabería J, Viñals V, Rauchwerger L, Torrellas J (2005) Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. ACM Trans Archit Code Optim
11. Garzaran MJ, Prvulovic M, Llabería JM, Viñals V, Rauchwerger L, Torrellas J (2003) Using software logging to support multi-version buffering in thread-level speculation. In: International conference on parallel architectures and compilation techniques, New Orleans, Sept 2003
12. Gopal S, Vijaykumar T, Smith J, Sohi G (1998) Speculative versioning cache. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
13. Gupta M, Nim R (1998) Techniques for speculative run-time parallelization of loops. In: Proceedings of supercomputing 1998, ACM Press, Melbourne, Australia, Nov 1998
14. Hammond L, Willey M, Olukotun K (1998) Data speculation support for a chip multiprocessor. In: International conference on architectural support for programming languages and operating systems, San Jose, California, Oct 1998, pp 58–69
15. Herlihy M, Moss E (1993) Transactional memory: architectural support for lock-free data structures. In: International symposium on computer architecture, IEEE Computer Society Press, San Diego, May 1993
16. Knight T (1986) An architecture for mostly functional languages. In: ACM lisp and functional programming conference, ACM Press, New York, Aug 1986, pp 500–519
17. Krishnan V, Torrellas J (1998) Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In: International conference on supercomputing, Melbourne, Australia, July 1998
18. Krishnan V, Torrellas J (1999) A chip-multiprocessor architecture with speculative multithreading. IEEE Trans Comput 48(9):866–880
19. Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J (2006) POSH: A TLS compiler that exploits program structure. In: International symposium on principles and practice of parallel programming, San Diego, Mar 2006
20. Marcuello P, Gonzalez A (1999) Clustered speculative multithreaded processors. In: International conference on supercomputing, Rhodes, Island, June 1999, pp 365–372
21. Marcuello P, Gonzalez A, Tubella J (1998) Speculative multithreaded processors. In: International conference on supercomputing, ACM, Melbourne, Australia, July 1998
22. Martinez J, Torrellas J (2002) Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
23. Prvulovic M, Garzaran MJ, Rauchwerger L, Torrellas J (2001) Removing architectural bottlenecks to the scalability of speculative parallelization. In: Proceedings of the 28th international symposium on computer architecture (ISCA'01), New York, June 2001, pp 204–215
24. Prvulovic M, Torrellas J (2003) ReEnact: using thread-level speculation to debug data races in multithreaded codes. In: International symposium on computer architecture, San Diego, June 2003
25. Rauchwerger L, Padua D (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Conference on programming language design and implementation, La Jolla, California, June 1995
26. Rundberg P, Stenstrom P (2000) Low-cost thread-level data dependence speculation on multiprocessors. In: Fourth workshop on multithreaded execution, architecture and compilation, Monterrey, Dec 2000
27. Sohi G, Breach S, Vijaykumar T (1995) Multiscalar processors. In: International Symposium on computer architecture, ACM Press, New York, June 1995
28. Steffan G, Colohan C, Zhai A, Mowry T (2000) A scalable approach to thread-level speculation. In: Proceedings of the 27th Annual International symposium on computer architecture, Vancouver, June 2000, pp 1–12
29. Steffan G, Mowry TC (1998) The potential for using thread-level data speculation to facilitate automatic parallelization. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
30. Torrellas J, Ceze L, Tuck J, Cascaval C, Montesinos P, Ahn W, Prvulovic M (2009) The bulk multicore architecture for improved programmability. Communications of the ACM, New York
31. Tremblay M (1999) MAJC: microprocessor architecture for java computing. Hot Chips, Palo Alto, Aug 1999
32. Tsai J, Huang J, Amlo C, Lilja D, Yew P (1999) The superthreaded processor architecture. IEEE Trans Comput 48(9):881–902
33. Vijaykumar T, Sohi G (1998) Task selection for a multiscalar processor. In: International symposium on microarchitecture, Dallas, Nov 1998, pp 81–92
34. Zhai A, Colohan C, Steffan G, Mowry T (2002) Compiler optimization of scalar value communication between speculative threads. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
35. Zhang Y, Rauchwerger L, Torrellas J (1998) Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In: Proceedings of the 4th International symposium on high-performance computer architecture (HPCA), Phoenix, Feb 1998, pp 162–174
36. Zhang Y, Rauchwerger L, Torrellas J (1999) Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In: Proceedings of the 5th international symposium on high-performance computer architecture, Orlando, Jan 1999, pp 135–139
37. Zhou P, Qin F, Liu W, Zhou Y, Torrellas (2004) iWatcher: efficient architectural support for software debugging. In: International symposium on computer architecture, IEEE Computer society, München, June 2004

Speculative Multithreading (SM)

►Speculation, Thread-Level

Speculative Parallelization

►Speculation, Thread-Level
►Speculative Parallelization of Loops

Speculative Parallelization of Loops

Lawrence Rauchwerger
Texas A&M University, College Station, TX, USA

Synonyms

Optimistic loop parallelization; Parallelization; Speculative Parallelization; Speculative Run-Time Parallelization; Thread-level data speculation (TLDS); TLS

Definition

Speculative loop (thread) level parallelization is a compiler run-time technique that executes optimistically parallelized loops, verifies the correctness of their execution and, when necessary, backtracks to a safe state for possible re-execution. This technique includes a compiler (static) component for the transformation of the loop for speculative parallel execution as well as a run-time component which verifies correctness and re-executes when necessary.

Discussion

Introduction

The correctness of optimizing program transformations, such as loop parallelization, relies on compiler or programmer analysis of the code. Compiler-performed analysis is preferred because it is more productive and is not subject to human error. However, it usually involves a very complex symbolic analysis which often fails to produce an optimizing transformation. Sometimes the outcome of the code analysis depends on the input values of the considered program block or on computed values, neither of which are available during compilation. Manual code analysis relies on the use of a higher level of semantic analysis which is usually more powerful but not applicable if it depends on run-time computed or input values. To overcome this limitation, the analysis can be (partially) performed at run-time. Run-time analysis can succeed where static analysis fails because it has access to the actual values with which *the program symbolic expressions* are instantiated and thus can make aggressive, instance-specific optimization (e.g., loop parallelization) decisions.

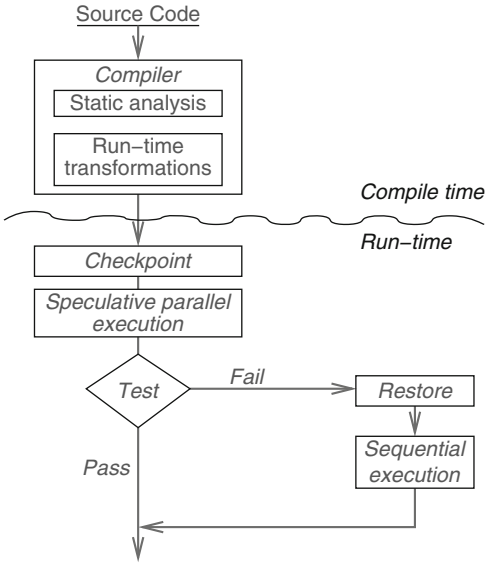
There are essentially two ways of performing the run-time analysis that can (in)validate loop parallelization (and, in general any other optimization):

- Before executing the parallel version of a loop
- During execution of the parallel version of the loop

The analysis that is performed before loop execution can be of various degrees of complexity: from constant time to time proportional to the original loop computation. The parallel execution of a loop before run-time analysis is completed is called speculative (aka optimistic) execution. It represents a speculation because the outcome of the analysis may, in the end, invalidate the (optimistic) parallel execution of the loop (and its results). In this case the state before the speculation has to be restored and the loop is re-executed in a safe manner, e.g., sequentially. Figure 1 shows the global flowchart of the speculative parallelization process.

Fundamentals of Loop Parallelization

A loop can be executed in parallel, without synchronizations, if and only if the desired outcome of the loop does not depend upon the relative execution order of the data accesses across its iterations. This problem has been modeled with the help of data dependence analysis [1, 13, 19, 42, 48]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write – RAW), *anti* (write after read – WAR), and *output* (write after write – WAW). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory-related



Speculative Parallelization of Loops. Fig. 1 Speculative run-time parallelization

dependences, are caused by the reuse of storage for program variables.

When flow data dependences exist between loop iterations, the loop cannot be executed in parallel because the original (sequential) semantics of the loop cannot be preserved. For example, the iterations of the loop in Fig. 2a must be executed in sequential order because iteration $i+1$ needs the value that is produced in iteration i, for $1 \leq i < n$. The simplest and most desired outcome of the data dependence analysis is when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and can be executed in any order, e.g., concurrently. Such loops are known as `DOALL` loops. In the absence of flow dependences, which are fundamental to a program's algorithms, the anti and/or output dependences can be removed through *privatization* (or *renaming*) [40], a very effective loop transformation.

Privatization creates, for each processor cooperating on the execution of the loop, private copies of the program variables that can give rise to anti-or output dependences (see, e.g., [2, 16, 17, 38, 39]). Figure 2b exemplifies a loop that can be executed in parallel after applying the privatization transformation: The anti-dependences between statement `S2` of iteration i and statement `S1` of iteration $i+1$, for $1 \leq i < n/2$, are removed by privatizing the temporary variable `tmp`. This transformation can be applied to a loop variable if it can be proven, statically or dynamically, that every read access to it (e.g., elements of array A) is preceded by a write access to the same variable within the same iteration. Of course, variables that are never written (read only) cannot generate any data dependence. Intuitively, variables are privatizable when they are used as workspace (e.g., temporary variables) *within* an iteration.

A semantically higher level transformation is the parallelization of *reduction* operations. Reductions are operations of the form $x = x \otimes exp$, where $\otimes$ is an associative operator and x does not occur in exp or anywhere else in the loop. A simple, but typical example of a reduction is statement `S1` in Fig. 2c. The operator $\otimes$ is exemplified by the + operator, the access to the array $A(:)$ is a *read, modify, write* sequence, and the function performed by the loop is a prefix sum of the values stored in A. This type of reduction is sometimes called an *update* and occurs quite frequently in programs.

A reduction can be readily transformed into a parallel operation using, e.g., a recursive doubling algorithm [12, 15]. When the operator takes the form $x = x + exp$, the values taken by variable x can be accumulated in private storage followed by a global reduction operation. There are also other, less scalable methods that use unordered critical sections [7, 48]. When the operator is also *commutative*, its substitution with a parallel algorithm can be done with fewer restrictions (e.g., dynamic scheduling of `DOALL` loops can be employed).

Thus, the difficulty encountered by compilers in parallelizing loops with reductions arises not from transforming the loop for parallel execution but from correctly identifying and validating reduction patterns in loops. This problem has been handled at compile time mainly by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement and thus does not cause additional dependences [48].

Compiler Limitation and Run-time Parallelization

In essence, the parallelization of `DO (for)` loops depends on proving that their memory reference

```
do i=1, n
  A(K(i)) = A(K(i)) + A(K(i–1))
  if  (A(K(i)) .eq. 0) then
    B(i) = A(L(i)
  endif
enddo
```
a

```
    do i = 1, n/2
S1:   tmp = A(2*i)
      A(2*i) = A(2*i–1)
S2    A(2*i–1) = tmp
    enddo
```
b

```
do i=1, n
  do j = 1, m
S1:   A(j) = A(j) + exp()
  enddo
enddo
```
c

Speculative Parallelization of Loops. Fig. 2 Examples of representative loops targeted by automatic parallelization

patterns do not carry data dependences or that there are legal transformations that can remove possible data dependences. The burden of proof relies on the static analysis performed by autoparallelizing compilers. However, there are many situations when this is not possible because either symbolic analysis is not powerful enough or the memory reference pattern is input or data dependent. The technique that can overcome such problems is run-time parallelization because, at run-time, all input values are known and the symbolic evaluation of complex expressions is vastly simplified. In this scenario, the compiler generates, conceptually at least, a parallel and a serial version of the loop as well code for the dynamic dependence analysis using the loop's memory references.

If the decision about the serial or parallel character of the loop is (and can be) made before its execution, then the results computed and written by the loop to memory can be considered to be always correct. Such a technique is called "inspector/executor" [31–33] because the memory references are first "inspected" and then executed in a safe manner, whether sequentially or concurrently. In this context, an "inspector" is obtained by extracting a loop slice that generates (and perhaps records) the relevant memory addresses which can then reveal possible loop-carried data dependences. It is important for such "inspectors" to be scalable and not to become serial bottlenecks. If no dependences are found, then the loop can be scheduled (and executed) as a `DOALL`, i.e., all its iterations can be executed concurrently. If dependences are found, then the "executor" of the loop has to enforce them using ordered synchronizations at the memory or iteration level such that sequential semantics is preserved. A frequently used solution [23, 24, 33] to this problem has been to first construct the loop dependence graph from the memory trace obtained by the inspector and then to use it to compute a parallel execution schedule. Finally, the loop is executed in parallel according to the schedule.

The computation of this execution schedule can also be interleaved with the executor [46]. If the reference pattern does not change within a larger scope than the considered loop, then the schedule can be reused, thus reducing the impact of its overhead. There have been several variations of this technique which were reported in [22].

If a loop is executed in parallel *before* its data dependences are uncovered, then it can cause out of order memory references which may lead to incorrect computation. Such an execution model is called *speculative execution*, also known as *optimistic execution*, because its performance is based on the the optimistic assumption that such dependences do in fact not materialize or are quite infrequent. To ensure that even when dependences may occur, the final computation produces sequentially equivalent results, the speculative execution model includes a *restart* from a safe (correct) state mechanism. Such a mechanism implies either saving state (checkpointing) before its speculative modification or writing into a temporary memory which has to be later merged (committed) into the global state of the program.

For example, the references to array *A* in the loop in Fig. 2a depend on some input values stored in array *K* and cannot be statically analyzed by the compiler. An inspector for this loop would analyze the contents of array *K* and decide whether the loop is parallel and then execute it accordingly. A speculative approach is to execute the loop in parallel while at the same time recording all the references to *A*. After the loop has ended, the memory trace is checked for data dependences and, if any are found, the results are discarded and the loop is re-executed sequentially or in some other safer mode. Alternatively, the memory references can be checked as they occur ("on-the-fly") and, if dependences are

detected the execution can be aborted at that point, the program state repaired and the loop restarted in a safe manner.

There are advantages and disadvantages to using either of the run-time parallelization methods. In the "inspector/executor" approach, the inspector does not modify the program state and thus does not require a restart mechanism with its associated memory overhead. On the other hand, a speculative approach may need to discard its results and restart from a safe state. This implies the allocation of additional memory for checkpointing or buffering state. Inspectors always add to the critical path of the program because they have to be inserted serially before the parallelized loop. Speculative parallelization performs the needed inspection of the memory references *during* the parallel execution almost independently from the actual computation and thus can be almost overlapped with it (assuming available resources). On the other hand, the checkpoint or commit phase introduces some overhead which may add to the critical path. What is perhaps the most important feature of speculative parallelization is its general applicability, even when the memory reference pattern is dependent on the computation of the loop. For example, the code snippet in Fig. 3 shows that the reference to array *NUSED* is dependent on its value which in turn, may have been modified in a different iteration (because the indirection array *IHITS* is not known at compile time).

There are two ways to look at speculation: optimistically assuming that a loop is fully parallel and can be executed as a `DOALL` or, pessimistically, assuming that the loop has dependences and must be executed as a DOACROSS, i.e., with synchronizations. When a DOALL is expected, then the overhead of the data checking can be done once, after the speculative loop has finished. If, however, dependences are expected, then they need to be detected early so that the resulting incorrect computation can be minimized. Early detection means frequent memory reference checks, well before the speculative loop has finished. These speculative approaches are bridged (transformed into one another) through the variation of the frequency (granularity) of the memory reference checks from once per loop to once per reference.

A related, but different limitation of compilers is their inability to analyze and parallelize most `while` loops. The reason is twofold:

- The classical loop dependence analysis looks for dependences in a bounded iteration space. However, the upper bound of a `while` loop can only be conservatively established at compile time which results in overly restrictive decisions, possibly inhibiting parallelization.
- The fully parallel execution of a `while` loop without data dependences (e.g., `do` loops with possible premature exits) may not be limited to the original iteration space. Iterations may be executed beyond their sequential upper bound, i.e., "overshoot" and thus incorrectly modify the global state.

A possible solution is to use a speculative parallel execution framework which allows discarding any unnecessary work and its effects. Speculation can also be used to estimate an upper bound of the iteration space of `while` loops.

```
read (IHITS(:))
do k= 1, LST
   j = IHITS(1,k)
   if (NUSED(j).LE.1) then
     NUSED(j)=NUSED(j) –1
   endif
enddo
```

Speculative Parallelization of Loops. Fig. 3 A loop example where only speculative parallelization is possible: array indexes are computed during loop execution

DOALL Speculative Parallelization: The LRPD Test

The optimistic version of speculative parallelization executes a loop in parallel and tests *subsequently* if any data dependences could have occurred. If this validation test fails, then the loop is re-executed in a safe manner, starting from a safe state, e.g., sequentially from a previous checkpoint. This approach, known as the *LRPD Test* (Lazy Reduction and Privatization Doall Test), [25, 27] is sketched in the next paragraph. To qualify more loops as parallel, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.

Consider a `do` loop for which the compiler cannot statically determine the access pattern of a shared array *A* (Fig. 4a). The compiler allocates the shadow arrays for

```
do i=1, 5
   z = A(K(i))
   if (B(i) .eq. .true.) then
      A(L(i)) = z +C(i)
   endif
enddo
        B(1:5) = (1 0 1 0 1)
        K(1:5) = (1 2 3 4 1)
        L(1:5) = (2 2 4 4 2)
```

a

```
doall i=1, 5
   markread(K(i))
   z = A(K(i))
   if (B(i) .eq. .true.) then
      markwrite(L(i))
      A(L(i)) = z +C(i)
   endif
enddoall
```

b

PD test	Shadow arrays 1	2	3	4	*tw*	*tm*
A_w	0	1	0	1	3	2
A_r	1	1	1	1		
A_{np}	1	1	1	1		
$A_w(:) \wedge A_r(:)$	0	1	0	1		
$A_w(:) \wedge A_{np}(:)$	0	1	0	1		

c

Speculative Parallelization of Loops. Fig. 4 `Do` loop (**a**) transformed for speculative execution, (**b**) the `markwrite` and `markread` operations update the appropriate shadow arrays, (**c**) shadow arrays after loop execution. In this example, the test fails

marking the write accesses, A_w, and the read accesses, A_r, and an array A_{np}, for flagging non-privatizable elements. The loop is augmented with code (Fig. 4b) that during the speculative execution will mark the shadow arrays every time A is referenced (based on specific rules). The result of the marking can be seen in Fig. 4c. The first time an element of A is written during an iteration, the corresponding element in the write shadow array A_w is marked. If, during any iteration, an element in A is read, but never written, then the corresponding element in the read shadow array A_r is marked. Another shadow array A_{np} is used to flag the elements of A that *cannot* be privatized: An element in A_{np} is marked if, in any iteration, the corresponding element in A has been written only after it has been read.

A *post-execution analysis*, illustrated in Fig. 4c, determines whether there were any cross-iteration dependencies between statements referencing A as follows. If $any(A_w(:) \wedge A_r(:))$ is true, (*any* returns the "OR" of its vector operand's elements, i.e., $any(v(1:n)) = (v(1) \vee v(2) \vee \ldots \vee v(n))$) then there is at least one flow- or anti-dependence that was not removed by privatizing A (some element is read and written in different iterations). If $any(A_{np}(:))$ is true, then A is not privatizable (some element is read before being written in an iteration). If *tw*, the total number of writes marked during the parallel execution, is not equal to *tm*, the total number of marks computed after the parallel execution, then there is at least one output dependence (some element is overwritten); however, if A is privatizable (i.e., if $any(A_{np}(:))$ is false), then these dependencies were removed by privatizing A.

The addition of an A_{rx} field to the shadow structure and some simple marking logic can extend the previous algorithm to validate parallel reductions at run-time [25, 27].

For this speculative technique two *safe restart* methods can be used [21, 25].

- The compiler either generates a checkpoint of all global, modified variables before the starting the speculative loop or does it, on demand, before a variable is modified the first time.
- The compiler generates code to allocate temporary, private storage for the global, modified variables. If the test fails the private storage is de-allocated, else its contents are merged into the global storage.

The data structures used for the checkpointing and shadowing can be appropriately chosen depending on the dense (e.g., arrays) or sparse reference characteristics of the loop (e.g., hash tables, linked lists) [44].

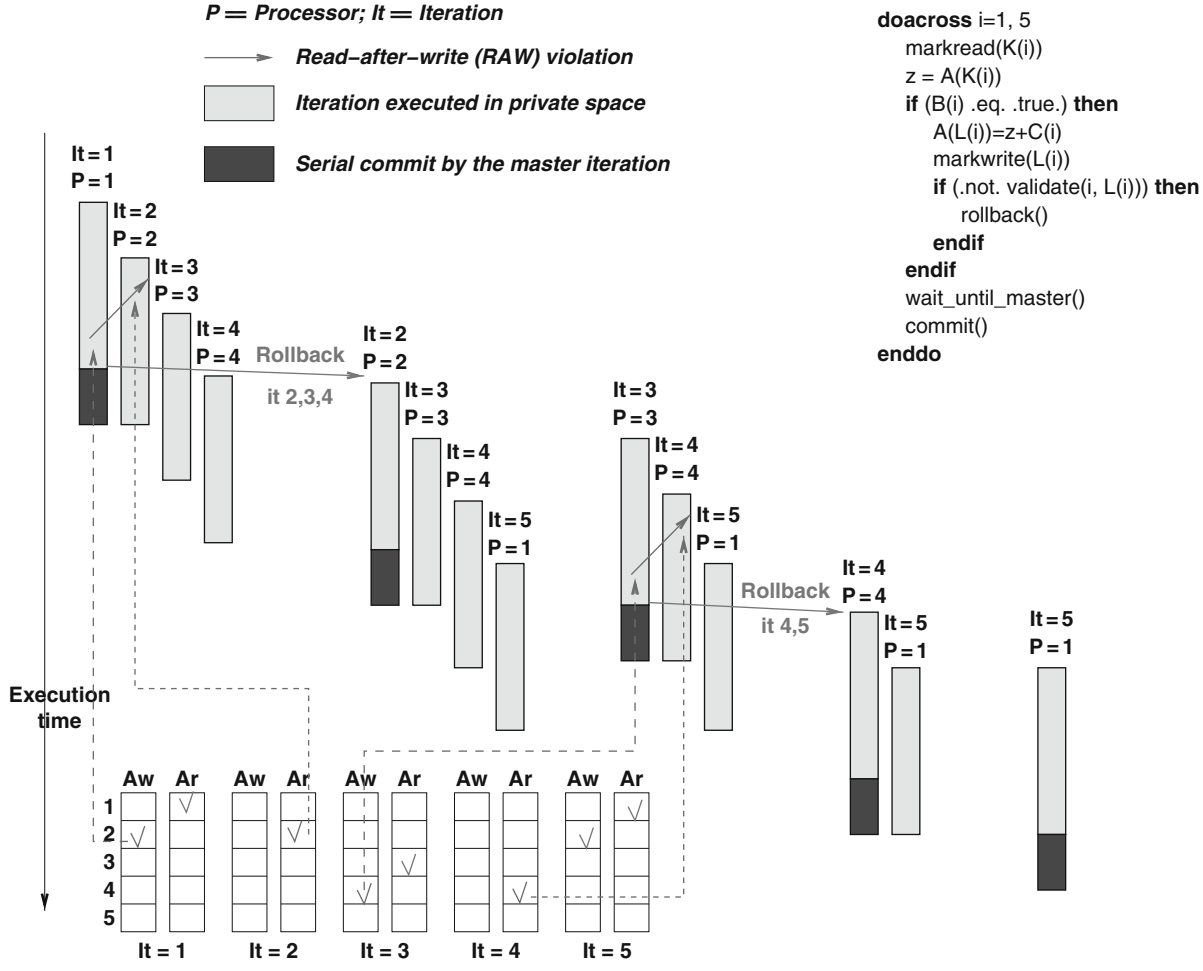
Compiler analysis can reduce the overhead of speculation by establishing equivalence classes (in the data dependence sense) of the interesting memory references and then tracking only one representative per class [44].

The speculative LRPD test has been later modified into the R-LRPD (Recursive LRPD) test [6] to improve its performance for loops with dependences. When a speculative parallel loop is block scheduled, the R-LRPD test can detect the iteration that is the source of the earliest dependence and thus validate the execution of the loop up to it. The remainder of the loop is then speculatively re-executed in a recursive manner until all work has finished, thus guaranteeing, at worst, a serial time complexity.

DOACROSS Speculative Parallelization

If dependences are expected, then speculation should be verified frequently, so that incorrect iterations can be restarted as soon as violations have occurred, thus reducing the overall speculation overhead. Approaches that track dependencies at low level (access/iteration), usually simulate the operation of a cache coherence protocol-based (hardware-based) TLS (Thread Level Speculation). Figure 5 depicts a sliding-window approach: The updates are recorded in per-iteration private storage and are merged (*committed*) to global storage by the oldest executing iteration, referred to as the *master*. Since the master represents a nonspeculative thread, i.e., it cannot be invalidated by any future iteration, its updates are known to be correct and the copy-out operation is safe. Then, the next iteration becomes nonspeculative and can eventually commit its updates, etc.

While implementations are diverse, one approach [4, 28] could be that (1) a read access returns the value written by the closest predecessor iteration (*forwarding*), or the nonspeculative value if no such write exists, and (2) a write access to memory location `l` signals a flow dependence violation if a successor iteration has read from location `l`. This behavior is achieved by inspecting the shadow vectors A_w and A_r, which record per-iteration write/read accesses. For example, in Fig. 5, the call of the function `validate` by iteration



Speculative Parallelization of Loops. Fig. 5 Serial-commit, sliding-window-based Thread Level Speculation execution. Dashed, *red arrows* starting from A_w/A_r represent the source/sink of flow dependences

1, which writes $A[2]$, detects a flow dependence violation because iteration 2 already read $A[2]$. The master is always correct, so iteration 1 commits its updates, but restarts iterations 2, 3, and 4. Similarly, iterations 3 and 4 are the source and sink of another flow dependence violation.

While maintaining per-iteration shadow vectors is prohibitively expensive in many cases, a sliding window of size C will reduce the memory overhead to more manageable levels. In particular, since only C consecutive iterations may execute concurrently, only $O(C)$ sets of shadow vectors need to be maintained, and then recycled.

Speculative DOACROSS methods are best suited for loops with more frequent data dependences because they can detect dependences earlier thus reducing wasted computation. However, verifying the dependence violation for each memory reference can be quite expensive because it requires global synchronizations. Furthermore, the merge (commit) phase is done in iteration order which constitutes a serial bottleneck. Loops with frequent dependences are not candidates for scalable parallelization regardless of the methods used to detect and process them. They cause, aside from lack of parallelism, frequent back tracking which rapidly degrades performance to possible negative levels.

While Loop Speculative Parallelization

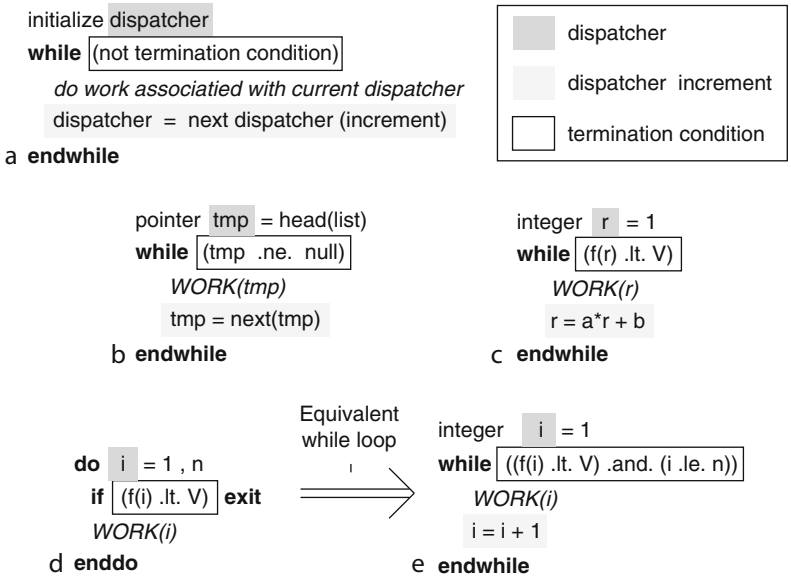
`While` loops and `do` loops with conditional premature exits arise frequently in practice and techniques for extracting their available parallelism [5, 26] are highly desirable. In the most general form, a `while` loop can be defined as a loop that includes at least one *recurrence*, a *remainder*, and at least one *termination condition*. The dominating recurrence, which precedes the rest of the computation is called the *dispatching recurrence*, or simply the *dispatcher* (Fig. 6a).

Sometimes the termination conditions form part of one of the recurrences, but they can also occur in the remainder, e.g., *conditional exits* from `do` loops. Assuming, for simplicity, that the *remainder* is fully parallelizable, there are two potential problems in the parallelization of `while` constructs:

- *Evaluating the recurrences.* If the recurrences cannot be evaluated in parallel, then the iterations of the loop must be started sequentially, leading in the best case to a pipelined execution.
- *Evaluating the termination conditions.* If the termination conditions (loop exits) cannot be evaluated independently by all iterations, the parallelized `while` loop could continue to execute beyond the point where the original sequential loop would stop, i.e., it can *overshoot*.

Evaluating the recurrences concurrently. In general, the terms of the dispatcher must be evaluated sequentially, e.g., the pointer chasing when traversing a linked list. Because the values of the dispatcher (the pointer) must be evaluated in sequential order, iteration i of the loop cannot be initiated until the dispatcher for iteration $i-1$ has been computed (see Fig. 6b). However, if the dispatching recurrence is associative then it can be evaluated in parallel using, e.g., a parallel prefix algorithm (see Fig. 6c). Better yet, when the dispatcher takes the form of an induction, its values can be computed concurrently by evaluating its symbolic form and thus allowing all iterations of the `while` loop to execute in parallel. A typical example is represented by a `do` loop (see Fig. 6d, e).

Evaluating the termination conditions in parallel. Another difficulty with parallelizing `while` loops is that the termination condition (*terminator*) of the loop may be *overshot*, i.e., the loop would continue to execute beyond its sequential (original) counterpart. The *terminator* is defined as *remainder invariant*, or *RI*, if it is only dependent on the dispatcher and values that are computed outside the loop. If it is dependent on some value computed in the loop, then it is considered to be *remainder variant* or *RV*. If the *terminator* is RV, then iterations larger than the last (sequentially) valid iteration could be performed in a parallel execution of the loop, i.e., iteration i cannot decide if the *terminator* is satisfied in the remainder of some iteration $i' < i$. Overshooting may also occur if the dispatcher is an induction, or an associative recurrence, and the *terminator* is RI. An exception in which overshooting would not occur is if the dispatcher is a monotonic function, and the *terminator* is a threshold on this function, e.g., $d(i) = i^2$, and $tc(i) = (d(i) < V)$, where V is a constant, and $d(j)$ and $tc(j)$ denote the dispatcher and the *terminator*, respectively, for the jth iteration. Overshooting can also



Speculative Parallelization of Loops. Fig. 6 (**a**) The general structure of `while` loops (**b**) Pointer chasing (**c**) Threshold *terminator* with monotonic *dispatcher* (**d**) `DO` loop with premature exit and (**e**) its equivalent `while` loop

be avoided when the dispatcher is a general recurrence, and the *terminator* is RI. For example, the dispatcher *tmp* is a pointer used to traverse a linked list, and the *terminator* is (*tmp* = *null*) (see Fig. 6b). The parallelization potential of the *dispatcher* is summarized by the taxonomy of `while` loops given in Table 1.

Speculative `while` *loop Parallelization*. Executing a loop outside its intended iteration space may result in undesired global state modification which has to be later undone. Such parallel execution can be regarded as *speculative* and is handled similarly to the previously mentioned speculative `do` loop parallelization.

The effects of *overshooting* in speculative parallel *while* loops can be undone after loop termination by restoring the (sequentially equivalent) correct state from a trace of the time (iteration) stamped memory references recorded during speculative execution. This solution may, however, have a large memory overhead for the time-stamped memory trace.

Alternatively, shared variables can be written into temporary storage, and then copied out only if their time stamps are less than or equal to the last valid iteration. There are various techniques for reducing memory overhead that can take advantage of the specific memory reference pattern, e.g., sparse patterns can be stored in hash tables. Further optimizations include *strip mining* the loop, or using a "sliding window" of iterations whose advance can be throttled adaptively.

An alternative to time-stamping is to attempt to extract a slice from the original `while` loop that can precompute the iteration space. When such a transformation is possible (e.g., traversal of a linked list) then the remainder of the `while` loop can be executed as a `doall`.

While loops with statically unknown cross-iteration dependences. When the data dependences of a `While` loop cannot be conclusively analyzed by a static compiler, the loop parallelization can be attempted by combining the LRPD test (applied to the *remainder* loop) with the techniques for *while* loop parallelization described above.

When it can be determined statically (see the taxonomy Table 1) that the parallelized `while` loop will not overshoot, then the shadow variable tracing instrumentation for the LRPD test can be inserted directly into the `while` loop.

In the more complex case when overshooting may occur (see the taxonomy Table 1) the LRPD test can be combined with the `while` loop parallelization methods by augmenting its shadow arrays with the minimum

Speculative Parallelization of Loops. Table 1 A taxonomy of `while` loops and their dispatcher's potential for parallel execution. In the table, *mono* is monotonic, *OV* is overshoot, *P* is parallel, *N* is no, *Y* is yes, and *pp* means parallelizable with a parallel prefix computation. *RV* is remainder variant and *RI* is remainder invariant

Loop terminator	Dispatcher							
	Induction				Recurrence			
	mono		other		associative		general	
	OV	P	OV	P	OV	P	OV	P
RI	N	Y	Y	Y	Y	Y-pp	N	N
RV	Y	Y	Y	Y	Y	Y-pp	Y	N

iteration that marked each element. The post-execution analysis of the LRPD test has to ignore the shadow array entries with minimum time stamps greater than the last valid iteration.

A special and unwelcome situation arises when the termination condition of the `while` loop is dependent (data or control) upon a variable which has **not** been found by the LRPD test to be independent. The speculative parallel execution of such a `while` loop may incorrectly compute its iteration space, or even worse, the termination condition might never be met (an infinite loop). Such loops are very hard to parallelize efficiently.

In general, `while` loops do not usually lend themselves to scalable parallelization due to their inherent nontrivial recurrence and overshooting potential which is expensive to mitigate.

Speculative Parallelization as a Parallel Programming Paradigm

The speculative techniques presented thus far concern themselves with the automatic transformation of sequential loops into parallel ones. They can validate, at run-time, if an instance of a loop executed in parallel does not have potential dependences and thus has *exactly* the same output as its sequential original. This validation does not, and could not, realistically check if the parallel and sequential execution have the same results. However, it imposes conditions on the memory reference trace which can ensure that the parallel loop respects the same dependence graph as the sequential one. In general, the speculative parallelization presented thus far represents a gamble that its results are equal to those of the sequential execution because it executes the same fine-grain algorithm, i.e., the same fine-grain dependence graph as the sequential original loop. This approach has lent itself well to automatic compiler implementation.

There is, however, a higher semantic level, albeit more complex, use of speculative parallelization. Instead of speculating on the execution of a fine-grain dependence graph (constructed from memory references), the programmer can generate the same results by following the correct execution of a coarser, semantically higher level dependence graph. For example, the nodes of such a coarse graph could represent the methods of a container and the edges the order in which they will be invoked. The programmer may also specify high-level properties of the methods involved. For example, the order of "inquire" type method invocations may be declared as interchangeable (commutative) similar with the `read` operations on an array. Other methods may be required to respect the program order, e.g., *add* and *delete*. In general, the cause-and-effect relation of the operations can be user defined. The result is a partial order of operations, a coarse dependence graph. A speculative parallelization of a program at this level of abstraction will ensure that its execution respects the higher level dependence graph. It implements a more relaxed execution model thus possibly improving performance. The Galois system implements this approach to speculative parallel programming [14]. It is based on two key abstractions: (1) *optimistic iterators* that can iterate over *sets* of iterations in any order and (2) a collection of assertions about properties of methods in class libraries. In addition a run-time scheme can detect and undo potentially unsafe operations due to speculation.

The *set iterator* abstracts the possibility selecting for execution elements of a set of iteration in any serial order (similar to the `DOANY` construct [43]). When an iteration executes it may add new iterations to the iteration space, hence allowing for fully dynamic execution. This is inherently possible because the addition of work can be inserted anywhere in the unordered set without changing the semantics of the loop. The *set iterator*

can become an *optimistic iterator* if the generated work can create conflicts at the iteration level. To exploit parallelism, the system relies on the semantic commutativity property of methods which must be declared by the programmer through special class interface method declarations. Finally, when the program executes, the methods are invoked like transactions (in isolation) and the commutativity property of the sequence of invoked methods is asserted. If it fails, then it is possible that operations have been executed in an illegal order and therefore have to be undone. The execution is rolled back using *anti-methods*, i.e., methods that undo the effects of the offending method. For example, an *add* operation can be undone by a *delete* operation. The entire system is reminiscent of Jefferson's Virtual Time system [10].

Speculating about higher level operations using higher level abstractions allows the exploitation of algorithmic parallelism otherwise impossible to exploit at the memory reference level. It requires, however, the user to reason more about the employed algorithm.

Future Directions

Run-time parallelization in general, and speculative parallelization in particular, are likely to be essential techniques for both automatic and manual parallelization. Speculation has its drawbacks: The success rate of speculation is rather unpredictable and thus affects performance in a nondeterministic manner. Speculation may waste resources, including power, when it does not produce a speedup. However, given the ubiquitous parallelism encountered today, it is sometimes the only avenue of performance improvement.

It would be of great benefit if meaningful statistics about the success rate of speculative parallelization could be found. Speculation will continue to be used for manual parallelization of irregular programs [14], at least until good parallel algorithms will be found. In this case, high-level speculation is likely to produce better results and result in more expressive programs!.

Speculative parallelization has been integrated into the Hybrid Analysis compiler framework [29, 30]. This framework seamlessly integrates static and dynamic analysis to extract the minimum sufficient conditions that need to be evaluated at run-time in order to validate a parallel execution of a loop. By carefully staging these sufficient conditions in order of their run-time complexity, the compiler often minimizes the need for a full speculative parallelization, thus reducing the nondeterminism of the code performance.

In conclusion, speculative parallelization is a globally applicable parallelization technique that can make the difference between a fully and a partially parallelized program, i.e., between scalable and non-scalable performance.

Related Entries

►Debugging
►Dependences
►Dependence Analysis
►Parallelization, Automatic
►Race Conditions
►Run Time Parallelization
►Speculation, Thread-Level

Bibliographic Notes and Further Reading

Speculative run-time parallelization has been first mentioned in the context of processes of parallel discrete event simulations by D. Jefferson [10]. In his *virtual time* concurrent processes are launched asynchronously and are tagged with their logical time stamp. When such processes communicate, they compare time stamps to check if their order respects the logical clock of the program (i.e., if data dependences are not violated). If a violation is detected, then anti-messages will recursively undo the effects of the incorrect computations and restart from a safe point.

The LRPD test, i.e., the speculative parallelization of loops, was introduced in [27]. It has later been modified to parallelize loops with dependences [6]. Compiler optimizations [21, 44] have lowered its overhead. The Hybrid Analysis framework has integrated the LRPD test in a compiler framework [30].

A significant amount of later work [4, 11, 20] has followed the hardware based approach to speculative parallelization presented in [8, 36, 37, 45].

Other related work reduces communication overhead via a master–slave model [47] in which the master executes an optimistic (fast) approximation of the code, while the slaves verify the master's correctness. Another framework [3, 41] exploits method-level parallelism for Java applications. The design space of speculative

parallelization has been further widened by allowing tunable memory overhead [18] by mapping more than one memory reference to the same "shadow" structure, but with the penalty of generating false dependence violations.

Software Transactional Memory [9, 35] can be and often is implemented using speculative parallelization techniques.

Debugging of parallel programs in general and detecting memory reference "anomalies" in particular [34] is a related topic to speculative parallelization and data dependence violation detection.

Bibliography

1. Banerjee U (1988) Dependence analysis for supercomputing. Kluwer, Boston
2. Burke M, Cytron R, Ferrante J, Hsieh W (1989) Automatic generation of nested, fork-join parallelism. J Supercomput 2:71–88
3. Chen MK, Olukotun K (1998) Exploiting method level parallelism in single threaded java programs. In: International conference on parallel architectures and compilation techniques PACT'98, IEEE, Paris, pp 176–184
4. Cintra M, Llanos DR (2003) Toward efficient and robust software speculative parallelization on multiprocessors. In: International conference on principle and practice of parallel computing PPoPP'03, ACM, San Diego, pp 13–24
5. Collard J-F (1994) Space-time transformation of while-loops using speculative execution. In: Scalable high performance computing conference, IEEE, Knoxville, pp 429–436
6. Dang F, Yu H, Rauchwerger L (2002) The R-LRPD test: speculative parallelization of partially parallel loops. In: International parallel and distributed processing symposium, Florida
7. Eigenmann R, Hoeflinger J, Li Z, Padua D (1991) Experience in the automatic parallelization of four perfect-benchmark programs. Lecture notes in computer science 589. Proceedings of the fourth workshop on languages and compilers for parallel computing, Santa Clara, pp 65–83
8. Hammond L, Willey M, Olukotun K (1998) Data speculation support for a chip multiprocessor. In: 8th international conference on architectural support for programming languages and operating systems, San Jose, pp 58–69
9. Herlihy M, Shavit N (1995) The art of multiprocessor programming. Morgan Kaufmann, London
10. Jefferson DR (1985) Virtual time. ACM Trans Program Lang Syst 7(3):404–425
11. Kazi IH, Lilja DJ (2001) Coarsed-grained thread pipelining: a speculative parallel execution model for shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 12(9):952
12. Kruskal C (1986) Efficient parallel algorithms for graph problems. In: Proceedings of the 1986 international conference on parallel processing, University Park, pp 869–876, Aug 1986
13. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M (1981) Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM symposium on principles of programming languages, Williamsburg, pp 207–218
14. Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K, Paul Chew L (2007) Optimistic parallelism requires abstractions. In: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementa-tion, PLDI '07, ACM, New York, pp 211–222
15. Thomson Leighton F (1992) Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufmann, London
16. Li Z (1992) Array privatization for parallel execution of loops. In: Proceedings of the 19th international symposium on computer architecture, Gold Coast, pp 313–322
17. Maydan DE, Amarasinghe SP, Lam MS (1992) Data dependence and data-flow analysis of arrays. In: Proceedings 5th workshop on programming languages and compilers for parallel computing, New Haven
18. Oancea CE, Mycroft A, Harris T (2009) A lightweight in-place implementation for software thread-level speculation. In: International symposium on parallelism in algorithms and architectures SPAA'09, ACM, Calgary, pp 223–232
19. Padua DA, Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. Commun ACM 29:1184–1201
20. Papadimitriou S, Mowry TC (2001) Exploring thread-level speculation in software: the effects of memory access tracking granularity. Technical report, CMU
21. Patel D, Rauchwerger L (1999) Implementation issues of loop-level speculative run-time parallelization. In: Proceedings of the 8th international conference on compiler construction (CC'99), Amsterdam. Lecture notes in computer science, vol 1575. Springer, Berlin
22. Rauchwerger L (1998) Run-time parallelization: its time has come. Parallel Comput 24(3–4):527. Special issues on languages and compilers for parallel comput
23. Rauchwerger L, Amato N, Padua D (1995) Run–time methods for parallelizing partially parallel loops. In: Proceedings of the 9th ACM international conference on supercomputing, Barcelona, Spain, pp 137–146
24. Rauchwerger L, Amato N, Padua D (1995) A scalable method for run-time loop parallelization. Int J Parallel Prog 26(6): 537–576
25. Rauchwerger L, Padua DA (1999) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans Parallel and Distrib Syst 10(2):160–180
26. Rauchwerger L, Padua DA (1995). Parallelizing WHILE loops for multi-processor systems. In: Proceedings of 9th international parallel processing symposium, Santa Barbara
27. Rauchwerger L, Padua DA (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the SIGPLAN 1995 conference on programming language design and implementation, La Jolla pp 218–232
28. Rundberg P, Stenstrom P (2000) Low-cost thread-level data dependence speculation on multiprocessors. In: 4th workshop on multithreaded execution, architecture and compilation, Monterey

29. Rus S, Pennings M, Rauchwerger L (2007) Sensitivity analysis for automatic parallelization on multi-cores. In: Proceedings of the ACM international conference on supercomputing (ICS07), Seattle
30. Rus S, Hoeflinger J, Rauchwerger L (2003) Hybrid analysis: static & dynamic memory reference analysis. Int J Parallel Prog 31(3):251–283
31. Saltz J, Mirchandaney R (1991) The preprocessed doacross loop. In: Schwetman HD (ed) Proceedings of the 1991 international conference on parallel processing, Software, vol II. CRC Press, Boca Raton, pp 174–178
32. Saltz J, Mirchandaney R, Crowley K (1989) The doconsider loop. In: Proceedings of the 1989 international conference on supercomputing, Irakleion, pp 29–40
33. Saltz J, Mirchandaney R, Crowley K (1991) Run-time parallelization and scheduling of loops. IEEE Trans Comput 40(5):603–612
34. Schonberg E (1989) On-the-fly detection of access anomalies. In: Proceedings of the SIGPLAN 1989 conference on programming language design and implementation, Portland, pp 285–297
35. Shavit N, Touitou D (1995) Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on principles of distributed computing, PODC '95, ACM, New York, pp 204–213
36. Sohi GS, Breach SE, Vijayakumar TN (1995) Multiscalar processors. In: 22nd international symposium on computer architecture, Santa Margherita
37. Steffan JG, Mowry TC (1998) The potential for using thread-level data speculation to facilitate automatic parallelization. In: Proceedings of the 4th international symposium on high-performance computer architecture, Las Vegas
38. Tu P, Padua D (1992) Array privatization for shared and distributed memory machines. In: Proceedings 2nd workshop on languages, compilers, and run-time environments for distributed memory machines, Boulder
39. Tu P, Padua D (1993) Automatic array privatization. In: Proceedings 6th annual workshop on languages and compilers for parallel computing, Portland
40. Tu P, Padua D (1995) Efficient building and placing of gating functions. In: Proceedings of the SIGPLAN 1995 conference on programming language design and implementation, La Jolla, pp 47–55
41. Welc A, Jagannathan S, Hosking A (2006) Safe futures for Java. In: International conference object-oriented programming, systems, languages and applications OOP-SLA'06, ACM, New York, pp 439–453
42. Wolfe M (1989) Optimizing compilers for supercomputers. MIT Press, Boston
43. Wolfe M (1992) Doany: not just another parallel loop. In: Proceedings 5th annual workshop on programming languages and compilers for parallel computing, New Haven. Lecture notes in computer science, vol 757. Springer, Berlin
44. Yu H, Rauchwerger L (2000) Run-time parallelization overhead reduction techniques. In: Proceedings of the 9th international conference on compiler construction (CC 2000), Berlin Germany. Lecture notes in computer science vol 1781. Springer, Heidelberg
45. Zhang Y, Rauchwerger L, Torrellas J (1998) Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In: Proceedings of the 4th international symposium on high-performance computer architecture (HPCA), Las Vegas, pp 162–174
46. Zhu C, Yew PC (1987) A scheme to enforce data dependence on large multiprocessor systems. IEEE Trans Softw Eng 13(6): 726–739
47. Zilles C, Sohi G (2002) Master/slave speculative parallelization. In: International symposium on microarchitecture Micro-35, IEEE, Los Alamitos, pp 85–96
48. Zima H (1991) Supercompilers for parallel and vector computers. ACM Press, New York

Speculative Run-Time Parallelization

►Speculation, Thread-Level
►Speculative Parallelization of Loops

Speculative Threading

►Speculation, Thread-Level

Speculative Thread-Level Parallelization

►Speculation, Thread-Level

Speedup

►Metrics

SPIKE

Eric Polizzi
University of Massachusetts, Amherst, MA, USA

Definition

SPIKE is a polyalgorithm that uses many different strategies for solving large banded linear systems

in parallel. Existing parallel algorithms and software using direct methods for banded matrices are mostly based on LU factorizations. In contrast, SPIKE uses a novel decomposition method (i.e., DS factorization) to balance communication overhead with arithmetic cost to achieve better scalability than other methods. The SPIKE algorithm is similar to a domain decomposition technique that allows performing independent calculations on each subdomain or partition of the linear system, while the interface problem leads to a reduced linear system of much smaller size than that of the original one. Direct, iterative, or approximate schemes can then be used to handle the reduced system in a different way depending on the characteristics of the linear system and the parallel computing platform.

Discussion

Introduction

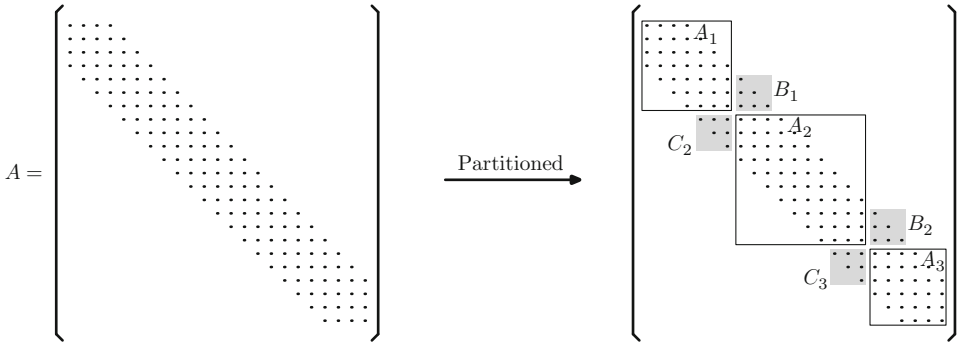
Many science and engineering applications, particularly those involving finite element analysis, give rise to very large sparse linear systems. These systems can often be reordered to produce either banded systems or low-rank perturbations of banded systems in which the width of the band is but a small fraction of the size of the overall problem. In other instances, banded systems can act as effective preconditioners to general sparse systems, which are solved via iterative methods.

Direct methods for solving linear systems $AX = F$ are commonly based on the LU decomposition that represents a matrix A as a product of lower and upper triangular matrices $A = LU$. Consequently, solving $AX = F$ can be achieved by solutions of two triangular systems $LG = F$ and $UX = G$. A parallel LU decomposition for banded linear systems has also been proposed by Cleary and Dongarra in [1] for the ScaLAPACK package [2]. The central idea behind the SPIKE algorithm is a different decomposition for banded linear systems, introduced by A. Sameh in the late 1970s [3], which is ideally suited for parallel implementation as it naturally leads to lower communication cost. Several enhancements and variants of the SPIKE algorithm have since been proposed by Sameh and coauthors in [4–11]. In the case when A is a banded matrix as depicted in Fig. 1, SPIKE is using a direct partitioning in the context of parallel processing.

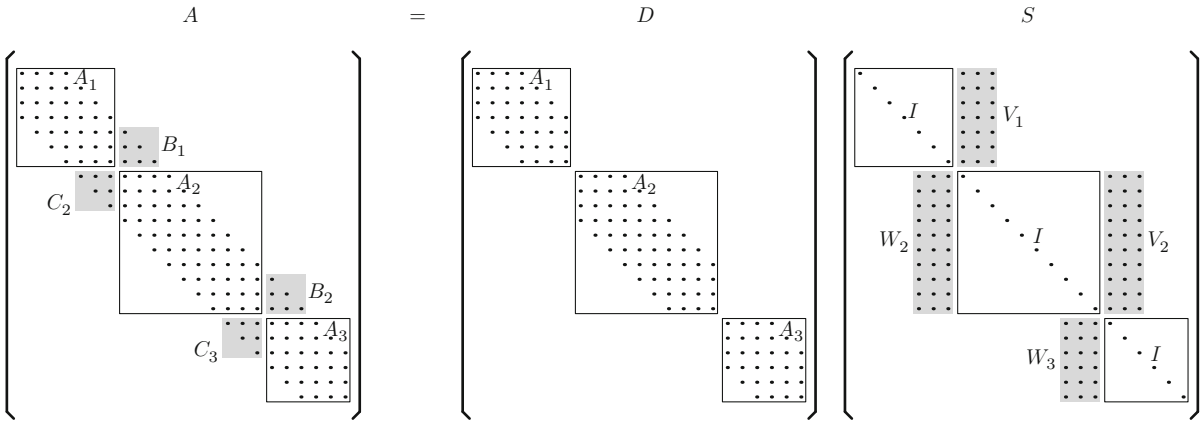
SPIKE relies on the decomposition of a given banded matrix A into the product of a block-diagonal matrix D, and another matrix S which has the structure of an identity matrix with some extra "spikes" (and hence the name of the algorithm). This DS factorization procedure is illustrated in Fig. 2.

Solving $AX = F$ can then be accomplished in two steps:

1. Solution of block-diagonal system $DG = F$. Because D consists of decoupled systems of each of the diagonal block A_i, they can be solved in parallel without requiring any communication between the individual systems.
2. Solution of the system $SX = G$. This system has a wonderful characteristic that it is also decoupled to a large extent. Except for a reduced system (near the interface of each of the identity blocks), the rest are independent from one another. The natural way to tackle this system is to first solve the reduced system via some parallel algorithms that



SPIKE. Fig. 1 A banded matrix with a conceptual partition



SPIKE. Fig. 2 SPIKE factorization where $A = DS, S = D^{-1}A$. The blocks in the block-diagonal matrix D are supposed non-singular

require inter-processor communications, followed by retrieval of the rest of the solution without requiring further inter-processor communications.

The SPIKE Algorithm: Basics

As illustrated in Fig. 1, a $(N \times N)$ banded matrix A can be partitioned into a block tridiagonal form $\{C_j, A_j, B_j\}$, where A_j is the $(n_j \times n_j)$ diagonal block j, and B_j (i.e., C_j) is the $(ku \times ku)$ (i.e., $(kl \times kl)$) right block (i.e., left block). Using p partitions, it comes that n_j is roughly equal to N/p. In order to ease the description of the SPIKE algorithm but without loss of generality, the size off-diagonal blocks are both supposed equal to m ($kl = ku = m$). The size of the bandwidth is then defined by $b = 2m + 1$ where $b << n_j$. Each partition j ($j = 1, \ldots, p$) can be associated to one processor or one node allowing multilevel of parallelism. Using the DS factorization illustrated in Fig. 2, the obtained spike matrix S has a block tridiagonal form $\{W_j, I_j, V_j\}$, where I_j is the $(n_j \times n_j)$ identity matrix, V_j and W_j are the $(n \times m)$ right and left spikes. The spikes V_j and W_j are solutions of the following linear systems:

$$A_j V_j = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ B_j \end{bmatrix}, \quad \text{and} \quad A_j W_j = \begin{bmatrix} C_j \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{1}$$

respectively for $j = 1, \ldots, p-1$ and $j = 2, \ldots, p$.

Solving the system $AX = F$ now consists of two steps:

$$\text{(a) solve} \quad DG = F \tag{2}$$

$$\text{(b) solve} \quad SX = G. \tag{3}$$

The solution of the linear system $DG = F$ in Step (a) yields the modified right-hand side G needed for Step (b). In case of assigning one partition to each processor, Step (a) is performed with perfect parallelism. To solve $SX = G$ in Step (b), one should observe that the problem can be reduced further by solving a system of much smaller size, which consists of the m rows of S immediately above and below each partitioning line. Indeed, the spikes V_j and W_j can also be partitioned as follows

$$V_j = \begin{bmatrix} V_j^{(t)} \\ V_j' \\ V_j^{(b)} \end{bmatrix} \quad \text{and} \quad W_j = \begin{bmatrix} W_j^{(t)} \\ W_j' \\ W_j^{(b)} \end{bmatrix} \tag{4}$$

where $V_j^{(t)}$, V_j', $V_j^{(b)}$, and $W_j^{(t)}$, W_j', $W_j^{(b)}$, are the top m, the middle $n_j - 2m$ and the bottom m rows of V_j and W_j, respectively. Here,

$$V_j^{(b)} = [0 \quad I_m] V_j; \quad W_j^{(t)} = [I_m \quad 0] W_j, \tag{5}$$

and

$$V_j^{(t)} = [I_m \quad 0] V_j; \quad W_j^{(b)} = [0 \quad I_m] W_j. \tag{6}$$

Similarly, if X_j and G_j are the jth partitions of X and G, it comes

$$X_j = \begin{bmatrix} X_j^{(t)} \\ X_j' \\ X_j^{(b)} \end{bmatrix} \quad \text{and} \quad G_j = \begin{bmatrix} G_j^{(t)} \\ G_j' \\ G_j^{(b)} \end{bmatrix}. \tag{7}$$

It is then possible to extract from a block tridiagonal reduced linear system (8) of size $2(p-1)m$, which involves only the top and bottom elements of V_j, W_j, X_j, and G_j. As example, the reduced system obtained for the case of four partitions ($p = 4$) is given by

$$\begin{bmatrix} I_m & V_1^{(b)} & & & & & \\ W_2^{(t)} & I_m & & V_2^{(t)} & & & \\ W_2^{(b)} & & I_m & V_2^{(b)} & & & \\ & & W_3^{(t)} & I_m & & V_3^{(t)} & \\ & & W_3^{(b)} & & I_m & V_3^{(b)} & \\ & & & & W_4^{(t)} & I_m & \end{bmatrix} \begin{bmatrix} X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \\ X_3^{(t)} \\ X_3^{(b)} \\ X_4^{(t)} \end{bmatrix} = \begin{bmatrix} G_1^{(b)} \\ G_2^{(t)} \\ G_2^{(b)} \\ G_3^{(t)} \\ G_3^{(b)} \\ G_4^{(t)} \end{bmatrix}. \tag{8}$$

Finally, once the solution of the reduced system is obtained, the global solution X can be reconstructed from $X_k^{(b)}$ ($k = 1, \ldots, p-1$) and $X_k^{(t)}$ ($k = 2, \ldots, p$) either by computing

$$\begin{cases} X_1' = G_1' - V_1' X_2^{(t)}, \\ X_j' = G_j' - V_j' X_{j+1}^{(t)} - W_j' X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1 \\ X_p' = G_p' - W_j' X_{p-1}^{(b)}, \end{cases} \tag{9}$$

or by solving

$$\begin{cases} A_1 X_1 = F_1 - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_2^{(t)}, \\ A_j X_j = F_j - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_{j+1}^{(t)} - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1 \\ A_p X_p = F_p - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{p-1}^{(b)}. \end{cases} \tag{10}$$

SPIKE: A Hybrid and Polyalgorithm

Multiple options are available for efficient parallel implementation of the SPIKE algorithm depending on the properties of the linear system as well as the architecture of the parallel platform. More specifically, the following stages of the SPIKE algorithm can be handled in several ways resulting in a polyalgorithm:

1. Factorization of the diagonal blocks A_j. Depending on the sparsity pattern of the matrix and the size of the bandwidth, these diagonal blocks could be considered either as dense or sparse within the band. For the dense banded case, a number of strategies based on the LU decomposition of each A_i can be applied here. This include variants such as LU with pivoting, LU without any pivoting but diagonal boosting, as well as a combination of LU and UL decompositions, either with or without pivoting. For the sparse banded case, it is common to use a sparse direct linear system solver to reorder and then factorize the diagonal blocks. However, solving the various linear systems for A_j can also be achieved using an iterative solver with preconditioner. Finally, each partition in the decomposition can be associated with one or several processors (one node), enabling multilevel parallelism.
2. Computation of the spikes. If the spikes V_j and W_j are determined entirely, the reduced system (8) can be solved explicitly and equation (9) can be used to retrieve the entire solution. In contrast, if equation (10) is used to retrieve the solution, the spikes may not be computed but only for the top and bottom ($m \times m$) blocks of V_j and W_j needed to form the reduced system. It should be noted that the determination of the top and bottom spikes is also not explicitly needed for computing the actions of the multiplications with $W_j^{(t)}$, $W_j^{(b)}$, $V_j^{(t)}$, and $V_j^{(b)}$. These latter can be realized "on-the-fly" using $\begin{pmatrix} I_m & 0 \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} C_j$, $\begin{pmatrix} 0 & I_m \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} C_j$, $\begin{pmatrix} I_m & 0 \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} B_j$, $\begin{pmatrix} 0 & I_m \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} B_j$, respectively.
3. Solution scheme for the reduced system. One of the earliest concerns with the SPIKE algorithm for large number of partitions was to propose a reliable and efficient parallel strategy for solving the reduced

system (8). Krylov subspace-based iterative methods have been the first candidates to fulfill this purpose, while giving to SPIKE its hybrid nature. These iterative methods are often used in conjunction with a block Jacobi preconditioner (i.e., diagonal blocks of the reduced system) if the bottom of the V_j spikes and the top of the W_j spikes are computed explicitly. In turn, the matrix-vector multiplication operations of the iterative technique can be done explicitly or implicitly ("on-the-fly"). In order to enhance robustness and scalability for solving the reduced system, two new highly efficient direct methods have been introduced by Polizzi and Sameh in [10, 11]. These SPIKE schemes, which have been named "truncated scheme" for handling diagonally dominant systems, and "recursive scheme" for non-diagonally dominant systems, are presented in the next sections. Here again, a number of different strategies exists for solving the reduced system.

As mentioned above, and in order to minimize memory references, it is sometimes advantageous to factorize the diagonal blocks A_j using LU without any pivoting but adding a diagonal boosting if a "zero-pivot" is detected. Hence, A is not exactly the product DS and rather takes the form $A = DS + R$, where R represents the correction which, even if nonzero, is by design small in some sense. Outer iterations via Krylov subspace schemes or iterative refinement, are then necessary to obtain sufficient accuracy as SPIKE would act on $M = DS$ (i.e., the approximate SPIKE decomposition for M is used as effective preconditioner).

Finally, a SPIKE-balance scheme has also been proposed by Golub, Sameh, and Sarin in [12], for addressing the case where the block diagonal A_j are nearly singular (i.e., ill-conditioned), and when even the LU decomposition with partial pivoting is expected to fail.

The Truncated SPIKE Scheme for Diagonally Dominant Systems

The truncated SPIKE scheme is an optimized version of the SPIKE algorithm with enhanced use of parallelism for handling diagonally dominant systems. These systems may arise from several science and engineering applications, and are defined if the degree of diagonally dominance, dd, of the matrix A is greater than 1, where dd is given by

$$dd = \min \frac{|A_{i,i}|}{\sum_{j \neq i} |A_{i,j}|}. \tag{11}$$

It is possible to show from equation (1), that the magnitude of the elements of the right spikes V_j decay from bottom to top, while the elements of the left spikes W_j decay in magnitude from top to bottom [13, 14]. Since the size n of A_j is much larger than the size m of the blocks B_j and C_j, the bottom blocks of the left spikes $W_j^{(b)}$ and the top blocks of the right spikes $V_j^{(t)}$ can be approximately set equal to zero. In fact, the zero accuracy machine is ensured to be reached either in the case of a pronounced decay (i.e., high value for dd), or for large ratio n_j/m. Thus, it follows that the off-diagonal blocks of the reduced system (8) are equal to zero, and the solution of this new block-diagonal "truncated" reduced system can be obtained by solving $p - 1$ independent $2m \times 2m$ linear systems in parallel:

$$\begin{bmatrix} I_m & V_j^{(b)} \\ W_{j+1}^{(t)} & I_m \end{bmatrix} \begin{bmatrix} X_j^{(b)} \\ X_{j+1}^{(t)} \end{bmatrix} = \begin{bmatrix} G_j^{(b)} \\ G_{j+1}^{(t)} \end{bmatrix}. \tag{12}$$

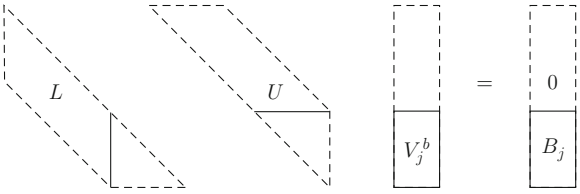
These systems can be solved directly using a block-LU factorization, where the solution steps consist of the following: (a) Form $E = I_m - W_{j+1}^{(t)} V_j^{(b)}$, (b) Solve $EX_{j+1}^{(t)} = G_{j+1}^{(t)} - W_{j+1}^{(t)} G_j^{(b)}$ to obtain $X_{j+1}^{(t)}$, (c) Compute $X_j^{(b)} = G_j^{(b)} - V_j^{(b)} X_{j+1}^{(t)}$.

Solving the reduced system via the truncated SPIKE algorithm for diagonally dominant systems demonstrates then linear scalability with the number partitions. The truncated scheme is often associated with an outer-iterative refinement step to increase the solution accuracy.

Within the framework of the truncated scheme, two other major contributions have also been proposed for improving computing performance and scalability of the factorization stage: (a) a LU/UL strategy, and (b) a new unconventional partitioning scheme.

LU/UL Strategy

The truncated scheme facilitates different new options for the factorization step that make possible to avoid the computation of the entire spikes. As illustrated in



SPIKE. Fig. 3 The bottom of the spike V_j can be computed using only the bottom $m \times m$ blocks of L and U. Similarly, the top of the spike W_j may be obtained if one performs the UL-factorization

Fig. 3, computational solve times can be drastically reduced by using the LU factorization without pivoting on each diagonal block A_j. Obtaining the top block of W_j, however, would still require computing the entire spike with complete forward and backward sweeps. Another approach consists of performing also the UL-factorization of the block A_j without pivoting. Similar to the LU-factorization, this allows obtaining the top block of W_j involving only the top $m \times m$ blocks of the new $\dot{U}$ and $\dot{L}$ matrices. Numerical experiments indicate that the time consumed by this LU/UL strategy is much less than that taken by performing only one LU factorization per diagonal block and generating the entire left spikes.

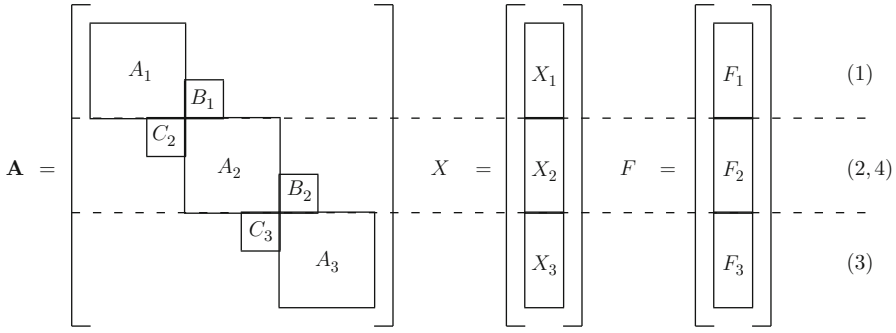
Unconventional Partitioning Schemes

A new partitioning scheme can be introduced to avoid performing both LU and UL factorization for a given A_j ($j = 2, \ldots, p-1$). This scheme acts on a new parallel distribution of the system matrix, which considers less partition than processors. Practically, if k represents an even number of processors (or nodes in the case of multilevel parallelism), the new number of partitions will be equal to $p = (k+2)/2$. The new block matrices A_j, $j = 1, \ldots, p$ can be associated to the first p processors while processors $p+1$ to k hold another copy of the block matrix A_j, $j = 2, \ldots, p-1$. Figure 4 illustrates the new partitioning of the matrix right-hand side and solution, for the case $k = 4$ and $p = 3$.

The diagonal blocks A_j associated to processors 1 to $p-1$ are then factorized using LU without pivoting, while a UL factorization is used for the diagonal blocks associated with processors p to k. In the example of $k = 4$, $L_j U_j \leftarrow A_j$ for $j = 1, 2$ (processors 1, 2), and $\dot{U}_j \dot{L}_j \leftarrow A_j$ for $j = 2, 3$ (processors 4, 3). As described above using the LU/UL strategy, the $V_j^{(b)}$ ($j = 1, \ldots, p-1$) can be obtained with minimal computational efforts via the LU solve step on processors 1 to $p-1$, while the $W_j^{(t)}$ ($j = 2, \ldots, p$) can be obtained in the similar way but now on different processors using the UL solve step. Using this new partitioning scheme, the size of the partitions does increase but the number of arithmetic operations by partition decreases as well as the size of the truncated reduced system. This scheme achieves better balance between the computational cost of solving the subproblems and the communication overhead. In addition to increasing scalability results for large number of processors, the scheme also addresses the "bottleneck" of the small number of processors case as described below.

Speed-Up Performances on Small Number of Processors

One of the main focus in the development of parallel algorithms for solving linear systems aims at achieving linear scalability on large number of processors. However, the emergence of multicore computing platforms in these recent years has brought new emphasis for parallel algorithms on achieving net speedup over the corresponding best sequential algorithms on small number of processors/cores. Clearly, parallel algorithms often inherit extensive preprocessing stages with increased memory references or arithmetics, leading to counter-performances on small number of cores. For parallel banded solvers, in particular, four to eight cores may be usually needed to solve linear systems as fast as the best corresponding sequential solver in LAPACK [15]. It is then important to note that the new partitioning scheme for the truncated SPIKE algorithm has also been designed to address this issue while offering a speedup of two from only two cores. While the two-cores (two-partitions) case can take advantage of a single LU or UL factorization for A_1 and A_2, respectively, the efforts to solve the reduced system become minimal (i.e., as compared to a LU decomposition on the overall system, the number of arithmetic operations is essentially divided by two in the SPIKE factorization and solve stages). When the number of cores increases, and without accounting for the communication costs (which are minimal for the truncated scheme), the



SPIKE. Fig. 4 Illustration of the unconventional partitioning of the linear system for the truncated SPIKE algorithm in the case of 4 processors. A_1 is sent to processor 1, A_2 to processors 2 and 4, and A_3 to processor 3.

speedup are expected ideally equal to the number of partitions: ×2 on two cores, ×3 on four cores, ×5 on eight cores, etc. Thereafter, the SPIKE performance will approach linear scalability as the number of cores increases.

The Recursive SPIKE Scheme for Non-Diagonally Dominant Systems

For non-diagonally dominant systems and large number of partitions, solving the reduced system (8) using Krylov subspace iterative method with or without preconditioner may result in high interprocessor communication cost. Interestingly, the truncated scheme for the two-cores (two-partitions) case is as well applicable for non-diagonally dominant systems since the reduced system (12) contains only one diagonal block. It should be noted, however, that this scheme may necessitate outer-refinement steps, since the diagonal boosting used to handle the "zero-pivot" in the *LU* and *UL* factorization stages, are more likely to appear for non-diagonally dominant systems.

For larger number of partitions, a new direct approach named "recursive" scheme has been proposed for solving the reduced system and enhancing robustness, accuracy, and scalability. This recursive scheme consists of successive iterations of the SPIKE algorithm from systems to reduced systems, resulting in better balance between the costs of computation and communication. This scheme assumes that the original number of (conventional) partitions is given by $p = 2^d$ $(d > 1)$. The bottom and top blocks of the V_j and W_j spikes are then computed explicitly to form the reduced system. In practice, a modified version of the reduced system is preferred which also includes the top block V_1^t and bottom block W_p^b. For the case $p = 4$, the original reduced system (8) is now represented by the following "reduced spike matrix":

$$\left(\begin{array}{cccc|cccc} I_m & V_1^{(t)} & & & & & & \\ & I_m & V_1^{(b)} & & & & & \\ W_2^{(t)} & & I_m & & V_2^{(t)} & & & \\ W_2^{(b)} & & & I_m & V_2^{(b)} & & & \\ \hline & & W_3^{(t)} & & I_m & & V_3^{(t)} & \\ & & W_3^{(b)} & & & I_m & V_3^{(b)} & \\ & & & & W_4^{(t)} & & I_m & \\ & & & & W_4^{(b)} & & & I_m \end{array}\right) \left[\begin{array}{c} X_1^{(t)} \\ X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \\ \hline X_3^{(t)} \\ X_3^{(b)} \\ X_4^{(t)} \\ X_4^{(b)} \end{array}\right] = \left[\begin{array}{c} G_1^{(t)} \\ G_1^{(b)} \\ G_2^{(t)} \\ G_2^{(b)} \\ \hline G_3^{(t)} \\ G_3^{(b)} \\ G_4^{(t)} \\ G_4^{(b)} \end{array}\right]. \tag{13}$$

This reduced spike system matrix contains p partitions with p diagonal block identities. The system can be easily redistributed in parallel using only $p/2$ partitions which are factorized by SPIKE recursively up until obtaining two partitions only. It can be shown [10] that the two partitions case leading to a $2m \times 2m$ linear system presented in (12), constitutes the basic computational kernel of the recursive SPIKE scheme.

The SPIKE Solver: Current and Future Implementation

Since the publications of the first SPIKE algorithm in the late seventies [3, 4], many variations and new schemes have been implemented. In recent years, a comprehensive MPI-Fortran 90 SPIKE package for distributed memory architecture has been developed by the author.

This implementation includes, in particular, all the different family of SPIKE algorithms: recursive, truncated, and on-the-fly schemes. These SPIKE solvers rely on a hierarchy of computational modules, starting with the data locality-rich BLAS level-3, up to the blocked LAPACK [15] algorithms for handling dense banded systems, or up to the direct sparse solver PARDISO [16] for handling sparse banded systems, with SPIKE being on the outermost level of the hierarchy. The package also includes new primitives for banded matrices that make efficient use of BLAS level-3 routines. Those include banded triangular solvers with multiple right-hand sides, banded matrix-matrix multiplications, and *LU*, *UL* factorizations with diagonal boosting strategy.

In addition, the large number of options/decision schemes available for SPIKE created the need for the automatic generation of a sophisticated runtime decision tree "SPIKE-ADAPT" that has been developed by Intel. This adaptive layer indicates the most appropriate version of the SPIKE algorithm capable of achieving the highest performance for solving banded systems that are dense within the band. The relevant linear system parameters in this case are system size, number of nodes/processors to be used, bandwidth of the linear system, and degree of diagonal dominance. SPIKE and SPIKE-ADAPT have been regrouped into one package, named "Intel Adaptive Spike-Based Solver," which has been released to the public in June 2008 on the Intel whatif web site [17].

The SPIKE package also includes a SPIKE-PARDISO scheme for addressing banded linear systems with large sparse bandwidth while offering a basic distributed version of the current shared memory PARDISO package. The capabilities and domain applicability of the SPIKE-PARDISO scheme have recently been significantly enhanced by Manguoglu, Sameh, and Schenk in [18] to address general sparse systems. In this approach, a weighted reordering strategy is used to extract efficient banded preconditioners that are solved via SPIKE-PARDISO including new specific PARDISO features for computing the relevant bottom and top tips of the spikes.

While the current parallel distributed SPIKE package does offer HPC users a new and valuable tool for solving large-scale problems arising from many areas in science and engineering, the growing size of the number of cores in compute node forestalls distributed programming model (i.e., MPI) for many users. On the other hand, the scalability of the LAPACK banded algorithms on multicore node or SMP is first and foremost dependent on the threaded capabilities of the underlying BLAS routines. A new implementation of the SPIKE solver recently initiated by the author is concerned with a shared memory programming model (i.e., OpenMP) that can consistently match the LAPACK functions for solving banded systems. This SPIKE Open-MP project is expected to offer high efficient threaded alternatives for solving banded linear systems on current and emerging multicore architectures.

Related Entries

►BLAS (Basic Linear Algebra Subprograms)
►Collective Communication
►Dense Linear System Solvers
►Linear Algebra, Numerical
►Load Balancing, Distributed Memory
►Metrics
►PARDISO
►Preconditioners for Sparse Iterative Methods
►ScaLAPACK

Bibliography Notes and Further Reading

As mentioned in the introduction, the main ideas of the SPIKE algorithm has been introduced in the late 1970s [3, 4], since then, many improvements and variations have been proposed [5–12, 18]. In particular, the highly efficient truncated and recursive schemes for solving the reduced system presented here are discussed in more detail in [10, 11]. All the main SPIKE algorithm variations have been regrouped into a comprehensive MPI-based SPIKE solver package in [17], where the associated SPIKE's user guide contains more detailed information on the capabilities of the different SPIKE schemes and their domain of applicability.

Bibliography

1. Cleary A, Dongarra J (1997) Implementation in ScaLAPACK of divide and conquer algorithms for banded and tridiagonal linear systems. University of Tennessee Computer Science Technical Report, UT-CS-97-358
2. Blackford LS, Choi J, Cleary A, Dazevedo E, Demmel J, Dhillon I, et al (1997) ScaLAPACK users guide. Society for Industrial and Appl. Math, Philadelphia
3. Sameh A (1977) Numerical parallel algorithms: a survey. In: Kuck D, Lawrie D, Sameh A (eds) High speed computer and algorithm organization. Academic, New York, pp 207–228

4. Sameh A, Kuck D (1978) On stable parallel linear system solvers. J ACM 25:81–91
5. Sameh A (1983) On two numerical algorithms for multiprocessors. In: Proceedings of NATO adv res workshop on high-speed comp. Series F: computer and systems sciences, vol 7. Springer, Berlin, pp 311–328
6. Lawrie D, Sameh A (1984) The computation and communication complexity of a parallel banded system solver. ACM Trans Math Software 10(2):185–195
7. Dongarra J, Sameh A (1984) On some parallel banded system solvers. Parallel Comput 1:223–235
8. Berry M, Sameh A (1988) Multiprocessor schemes for solving block tridiagonal linear systems. Int J Supercomput Appl 2(3): 37–57
9. Sameh A, Sarin V (1999) Hybrid parallel linear solvers. Int J Comput Fluid Dyn 12:213–223
10. Polizzi E, Sameh A (2006) A parallel hybrid banded system solver: the SPIKE algorithm. Parallel Comput 32(2):177–194
11. Polizzi E, Sameh A (2007) SPIKE: A parallel environment for solving banded linear systems. Comput Fluids 36:113–120
12. Golub G, Sameh V, Sarin V (2001) A parallel balance scheme for banded linear systems. Numer Linear Algebr Appl 8(5):297–316
13. Demko S, Moss WF, Smith PW (1984) Decay rates for inverses of band matrices. Math Comput 43(168):491–499
14. Mikkelsen CCK, Manguoglu M (2008) Analysis of the truncated spike algorithm. SIAM J Matrix Anal Appl 30(4):1500–1519
15. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, et al (1999) LAPACK users guide, 3rd edn. Society for Industrial and Appl. Math, Philadelphia
16. Schenk O, Grtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. J Future Gener Comput Syst 20(3):475–487
17. A distributed memory version of the SPIKE package can be obtained from http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/
18. Manguoglu M, Sameh A, Schenk O (2009) PSPIKE: a parallel hybrid sparse linear system solver. Lecture notes in computer science, vol 5704. Springer, Berlin, pp 797–808

Spiral

Markus Püschel[1], Franz Franchetti[2], Yevgen Voronenko[2]
[1]ETH Zurich, Zurich, Switzerland
[2]Carnegie Mellon University, Pittsburgh, PA, USA

Definition

Spiral is a program generation system (software that generates other softwares) for linear transforms and an increasing list of other mathematical functions. The goal of Spiral is to automate the development and porting of performance libraries. Linear transforms include the discrete Fourier transform (DFT), discrete cosine transforms, convolution, and the discrete wavelet transform. The input to Spiral consists of a high-level mathematical algorithm specification and selected architectural and microarchitectural parameters. The output is performance-optimized code in a high-level language such as C, possibly augmented with vector intrinsics and threading instructions.

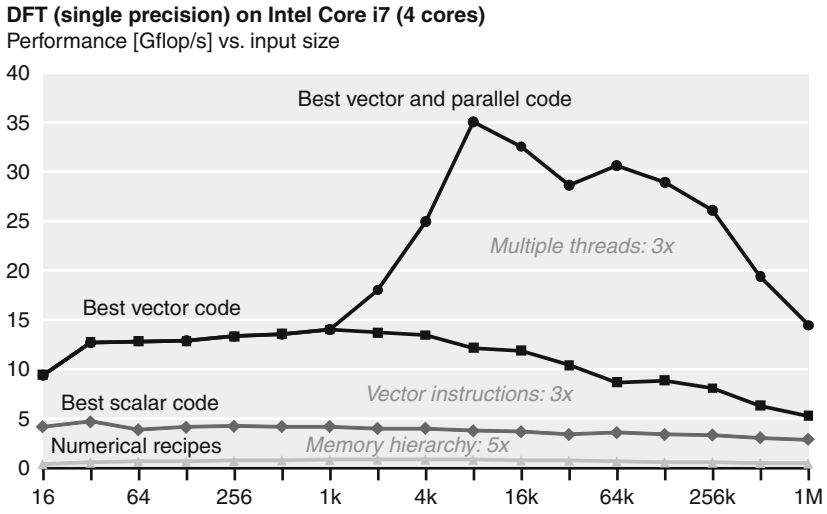
Discussion

Introduction

The advent of computers with multiple cores, SIMD (single-instruction multiple-data) vector instruction sets, and deep memory hierarchies has a dramatic effect on the development of high-performance software. The problem is particularly apparent for functions that perform mathematical computations, which form the core of most data or information processing applications. Namely, on a current workstation, the performance difference between a straightforward implementation of an optimal (minimizing operations count) algorithm and the fastest possible implementation is typically 10–100 times.

As an example, consider Fig. 1, which shows the performance (in gigafloating point operations per second) of four implementations of the discrete Fourier transform for varying input sizes on a quadcore Intel Core i7. Each one uses a fast algorithm with roughly the same operations count. Yet the difference between the slowest and the fastest is 12–35 times. The bottom line is the code from Numerical Recipes [20]. The best standard C code is about five times faster due to memory hierarchy optimizations and constant precomputation. Proper use of explicit vector intrinsics instructions yields another three times. Explicit threading for the four cores, properly done, yields another three times for large sizes.

The plot shows that the compiler cannot perform these optimizations as is true for most mathematical functions. The reason lies in both the compiler's lack of domain knowledge needed for the necessary transformations and the large set of optimization choices with uncertain outcome that the compiler cannot assess. Hence the optimization task falls with the programmer and requires considerable skill. Further, the optimizations are usually platform specific, and hence have to be repeated with every new generation of computers.



Spiral. Fig. 1 Performance of different implementations of the discrete Fourier transform (DFT) and reason for the performance difference (From [10])

Spiral overcomes these problems by completely automating the implementation and optimization processes for the functions it supports. Complete automation means that Spiral produces source code for a given function given only a very high-level representation of the algorithms for this function and a high-level platform description. After algorithm and platform knowledge are inserted, Spiral can generate various types of code including for fixed and general input size, threaded or vectorized.

The approach taken by Spiral is based on the following key principles:

- Algorithm knowledge for a given mathematical function is represented in the form of *breakdown rules* in a *domain-specific language*. Each rule represents a divide-and-conquer algorithm. The language is based on mathematics, and is declarative and platform independent. These properties enable the mapping to various forms of parallelism from algorithm knowledge that is inserted only once. They also enable the derivation of the library structure for general input size implementations by computing the so-called *recursion step closure*.
- Platform knowledge is organized into *paradigms*. A paradigm is a feature of a platform that requires structural optimization and possibly source code extensions. Examples include shared memory parallelism and SIMD vector processing. Each paradigm consists of a set of *parameterized rewrite rules* and *base cases* expressed in the same language as the algorithm knowledge. The base cases constitute a subset of the domain-specific language that maps well to a paradigm. The rewrite rules interact with the breakdown rules to produce algorithms that are base cases, which means they are structurally optimized for the considered paradigm. Examples of parameters include the SIMD vector length or the cacheline size. Paradigms are designed to be composable.
- Spiral uses *empirical search* to automatically explore choices in a feedback loop. This is done by generating candidate implementations and evaluating their performance. Even though theoretically unsatisfying, search enables further optimization for intricate microarchitectural details that may be unknown or are not well understood.

In summary, Spiral integrates techniques from mathematics, programming languages, compilers, automatic performance tuning, and symbolic computation. The entire Spiral system combines aspects of a compiler, generative programming, and an expert system.

The remainder of this section describes the framework underlying Spiral and the inner workings of the actual system. The presentation focuses on linear transforms; extensions of Spiral beyond transforms are briefly discussed in the end.

Algorithm Representation

Linear transforms. A linear transform is a function

$$x \mapsto Mx,$$

where M is a fixed matrix, x is the input vector, and $y = Mx$ the output vector. Different transforms correspond to different matrices M. For simplicity, M is referred to as transform in the following. Most transforms M are square $n \times n$, which implies that x and y are of length n. Most transforms exist for all $n = 1, 2, \ldots$.

Possibly the most well-known transform is the DFT, defined by the $n \times n$ matrix:

$$\mathrm{DFT}_n = \left[\omega_n^{k\ell}\right]_{0 \le k,\ell < n}, \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}.$$

Other examples include the discrete Hartley transform,

$$\mathrm{DHT}_n = \left[\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)\right]_{0 \le k,\ell < n},$$

the discrete cosine transform (DCT) of type 2,

$$\text{DCT-}\mathbf{2}_n = \left[\cos\left(k\left(\ell + \tfrac{1}{2}\right)\pi/n\right)\right]_{0 \le k,\ell < n},$$

as well as other types of discrete cosine and sine transforms, the Walsh–Hadamard transform, the real DFT, the discrete wavelet transform, the inverses and other variants of the preceding transforms, and finite impulse response filters.

Fast transform algorithms: *SPL*. If M is $n \times n$ and has few or no zero entries, then a direct computation of $y = Mx$ requires $O(n^2)$ many operations. However, all the transforms mentioned above have fast algorithms that reduce their complexity below that, typically to $O(n\log(n))$. Every algorithm can be expressed as a factorization of the transform matrix M into a product of sparse matrices. As an example, assume $M = M_1M_2M_3M_4$; then $y = Mx$ can be computed in four steps as

$$t = M_4x, u = M_3t, v = M_2u, y = M_1v.$$

If the M_i are sufficiently sparse, this reduces the operations count.

The sparse matrices occurring in transform algorithms have a structure that can be formally expressed using basic matrices and matrix operators such as the direct sum and the tensor or Kronecker product. This notation forms the basis for the language SPL (signal processing language) explained next.

Basic matrices include the $n \times n$ identity matrix I_n, diagonal matrices $D_n = \mathrm{diag}(a_0, \ldots, a_{n-1})$, the 2×2 butterfly matrix

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

the stride permutation matrix L_k^n, defined for $n = km$ by the underlying permutation

$$\ell_k^n : \; im + j \mapsto jk + i, \; 0 \le i < k, \; 0 \le j < m, \qquad (1)$$

and several others.

Matrix operators include the matrix product, the direct sum

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix},$$

and the tensor product

$$A \otimes B = \left[a_{k,\ell}B\right]_{0 \le k,\ell < n}, \quad \text{for } A = \left[a_{k,\ell}\right]_{0 \le k,\ell < n}.$$

Most important are the tensor products where A or B is the identity:

$$I_n \otimes B = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix},$$

and, for example,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_3 = \begin{bmatrix} aI_3 & bI_3 \\ cI_3 & dI_3 \end{bmatrix} = \begin{bmatrix} a & & & b & & \\ & a & & & b & \\ & & a & & & b \\ c & & & d & & \\ & c & & & d & \\ & & c & & & d \end{bmatrix}.$$

A (partial) description of SPL in Backus–Naur form is provided in Table 1.

Algorithms as SPL breakdown rules. Using SPL, the algorithm knowledge in Spiral is captured by *breakdown rules*. A breakdown rule represents a one-step divide-and-conquer algorithm of a transform. This means the transform is factorized into sparse matrices involving other, typically smaller, transforms.

Spiral. Table 1 A subset of SPL in Backus–Naur form; n, k are positive integers, a_i are real or complex numbers

⟨spl⟩	::= ⟨generic⟩ | ⟨basic⟩ | ⟨transform⟩ |	
	⟨spl⟩ · · · · · ⟨spl⟩ |	(product)
	⟨spl⟩ ⊕ . . . ⊕ ⟨spl⟩ |	(direct sum)
	⟨spl⟩ ⊗ · · · · ⊗⟨spl⟩ |	(tensor product)
	. . .	
⟨generic⟩	::= $\mathrm{diag}(a_0, \ldots, a_{n-1})$ | . . .	
⟨basic⟩	::= $I_n \mid L^n_k \mid F_2 \mid \ldots$	
⟨transform⟩	::= $\mathrm{DFT}_n \mid \mathrm{DHT}_n \mid \mathrm{DCT\text{-}}\mathbf{2}_n \mid \ldots$	

The most well-known example is the general-radix Cooley–Tukey fast Fourier transform (FFT):

$$\mathrm{DFT}_n \to (\mathrm{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k, \quad n = km, \tag{2}$$

where T^n_m is the diagonal matrix of *twiddle factors*. For $n = 16 = 4 \times 4$, the factorization is visualized in Fig. 2 together with the associated data-flow graph. The smaller DFT_4's are boxes of equal shades of gray.

To terminate the recursion, base cases are needed. For example, for two-powers n, a size two base case is sufficient:

$$\mathrm{DFT}_2 \to F_2. \tag{3}$$

A few points are worth noting about this representation of transform algorithms:

- The representation (2) is *point free*, i.e., the input vector is not present.
- The representation (2) is declarative.
- Since the rule (2) is a matrix equation, it can be manipulated using matrix identities. For example, both sides can be inverted or transposed, to obtain an inverse or transposed transform algorithm.
- A breakdown rule may have degrees of freedom. An example is the choice of k in (2).
- A rule like (2) does not specify how to compute the smaller transforms. This implies that rules have to be applied recursively until an algorithm is completely specified. Because of this and the availability of different rules for the same transform, there is a large set of choices. In other words, the relatively few existing rules yield a very large space of possible algorithms. This makes rules a very efficient representation of algorithm knowledge. For example, for $n = 2^\ell$, (2) alone yields $\Theta(5^\ell/\ell^{3/2})$ different algorithms, all with roughly the same operations count.

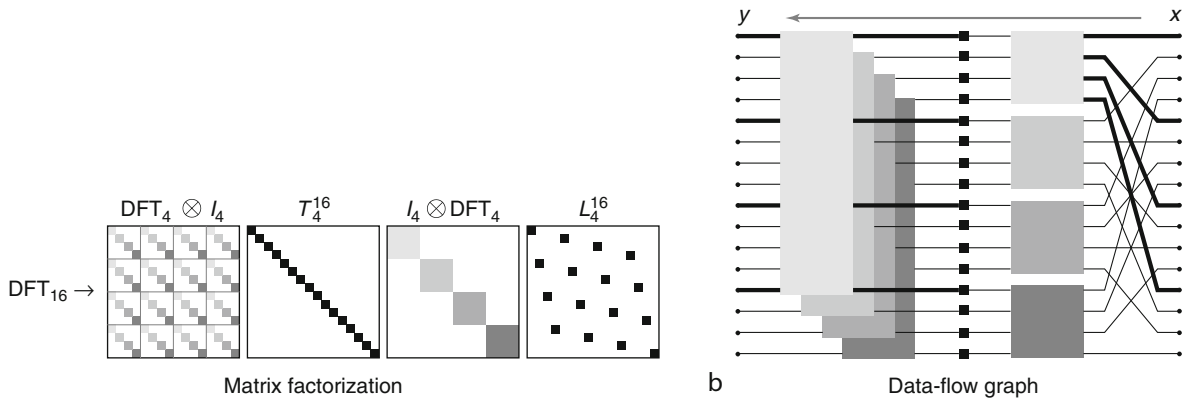
Spiral contains about 200 breakdown rules for about 40 transforms, some of which are auxiliary. The most important rules for the DFT, without complete specification, are shown in Table 2. Note the occurrence of auxiliary transforms.

Spiral Program Generation: Overview

The task performed by Spiral is to translate the algorithm knowledge (represented as in Table 2) for a given transform into optimized source code (we assume C/C++) for a given platform.

The exact approach for generating the code depends on the type of code that has to be generated. The most important distinctions are the following:

- *Fixed input size versus general input size*: If the input size is known (e.g., "DFT of size 4" as shown in Table 3a and b), the algorithm to be used and other decisions can be determined at program generation time and can be inlined. The result is a function containing only loops and basic blocks of straightline code. If the input size is not known, it becomes an additional input and the implementation becomes recursive (Table 3c). The actual algorithm, i.e., recursive computation, is now chosen at runtime once the input size is known.
- *Straightline code versus loop code (fixed input size only)*: Straightline code (Table 3a) is only suitable for small sizes, but can be faster, due to reduced overhead and increased opportunities for algebraic simplifications. Loop code (Table 3b) requires additional optimizations that merge redundant loops.
- *Scalar code versus parallel code*: Code that is parallelized for SIMD vector extensions or multiple cores requires specific optimizations and the use of explicit vector intrinsics or threading directives.



Spiral. Fig. 2 Cooley–Tukey FFT (2) for $16 = 4 \times 4$ as SPL rule and as (complex) data-flow graph (from *right* to *left*). Some lines are bold to emphasize the strided access of the $\mathbf{DFT}_4$s (From [10])

Spiral. Table 2 A selection of breakdown rules representing algorithm knowledge for the DFT. rDFT is an auxiliary transform and has two parameters. RDFT is a version of the real DFT

DFT_n	$\rightarrow$	$(\mathrm{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k$,	(Cooley–Tukey FFT)	$n = km$
DFT_n	$\rightarrow$	$V_n^{-1} (\mathrm{DFT}_k \otimes I_m)(I_k \otimes \mathrm{DFT}_m) V_n$,	(Prime-factor FFT)	$n = km$, $\gcd(k,m) = 1$
DFT_n	$\rightarrow$	$W_n^{-1}(I_1 \oplus \mathrm{DFT}_{p-1}) E_n (I_1 \oplus \mathrm{DFT}_{p-1}) W_n$,	(Rader FFT)	n prime
DFT_n	$\rightarrow$	$B'_n D_m \mathrm{DFT}_m D'_m \mathrm{DFT}_m D''_m B_n$,	(Bluestein FFT)	$n > 2m$
DFT_n	$\rightarrow$	$P^{\top}_{k,2m} \left(\mathrm{DFT}_{2m} \oplus \left(I_{k-1} \otimes_i C_{2m}\, \mathrm{rDFT}_{2m,i/2k}\right)\right) (\mathrm{RDFT}_{2k} \otimes I_m)$,		$n = 2km$
RDFT_n	$\rightarrow$	$\left(P^{\top}_{k,m} \otimes I_2\right) \left(\mathrm{RDFT}_{2m} \oplus \left(I_{k-1} \otimes_i D_{2m}\, \mathrm{rDFT}_{2m,i/2k}\right)\right) (\mathrm{RDFT}_{2k} \otimes I_m)$,		$n = 2km$
$\mathrm{rDFT}_{n,u}$	$\rightarrow$	$L^{2n}_m \left(I_k \otimes_i \mathrm{rDFT}_{2m,(i+u)/k}\right) (\mathrm{rDFT}_{2k,u} \otimes I_m)$,		$n = 2km$

Spiral. Table 3 Code types

(a) Fixed input size, unrolled

```
void dft_4(cpx
*Y, cpx *X){
  cpx s, t, t2, t3;
  t = (X[0] + X[2]);
  t2 = (X[0] - X[2]);
  t3 = (X[1] + X[3]);
  s = _I_*(X[1] - X[3]);
  Y[0] = (t + t3);
  Y[2] = (t - t3);
  Y[1] = (t2 + s);
  Y[3] = (t2 - s);
}
```

(b) Fixed input size, looped

```
void dft_4(cpx *Y, cpx *X){
  cpx T[4];
  cpx W[2] = {1, _I_};
  for(int i = 0; i <= 1; i++) {
    cpx w = W[i];
    T[2*i] = (X[i] + X[i+2]);
    T[2*i+1] = w*(X[i] - X[i+2]);
  }
  for(int j = 0; j <= 1; j++) {
    Y[j] = T[j] + T[j+2];
    Y[2+j] = T[j] - T[j+2];
  }
}
```

(c) General input size library, recursive

```
struct dft : public Env{
  dft(int n); // constructor
  void compute(cpx *Y, cpx *X);
  int _rule, f, n;
  char *_dat;
  Env *ch1, *ch2;
};

void dft::compute(cpx *Y, cpx *X){
  ch2->compute(Y, X, n, f, n, f);
  ch1->compute(Y, Y, n, f, n, n/f);
}
```

The program generation process is explained in the next four sections corresponding to four different code types of increasing difficulty. The order matches the historic development, since for each move to the next code type at least one new idea had to be introduced. The types and main ideas (in parentheses) are

- Fixed input size straightline code (SPL, breakdown rules, feedback loop)

- Fixed input size loop code (Σ-SPL, loop merging)
- Fixed input size parallel code (paradigms, tagged rewriting)
- General input size code (recursion step closure, parameterization)

Spiral generates code for fixed input size transforms (first three bullets), as shown in Fig. 3. The input is the transform symbol (e.g., "DFT") and the size (e.g., "128"). The output is a C function that computes the transform ($y = \mathrm{DFT}_{128}\, x$ in this case). Depending on code type, not all blocks in Fig. 3 may be used.

The block diagram for the general input size code is shown later.

Fixed Input Size: Straightline Code

Given as input to Spiral is a transform symbol ("DFT") and the input size. The program generation does not need the parallelization and loop optimization blocks. Further, no Σ-SPL is needed, which means the SPL-to-Σ-SPL block and the Σ-SPL-to-code block are joined to one SPL-to-code block.

Algorithm generation. Spiral uses a rewrite system that recursively applies the breakdown rules (e.g., Table 2) to generate a complete SPL algorithm for the transform. As mentioned before, there are many choices due to the choice of rule and the degree of freedom in some rules (e.g., k in (2)).

SPL to C code and optimization. The SPL expression is then compiled into actual C code using the internal SPL compiler, which recursively applies the translation rules sketched in Table 4.

All loops are unrolled and code-level optimizations are applied. These include array scalarization, constant propagation, and algebraic simplification.

Performance evaluation. The runtime of the resulting code is measured and fed into the search block that controls the algorithm generation.

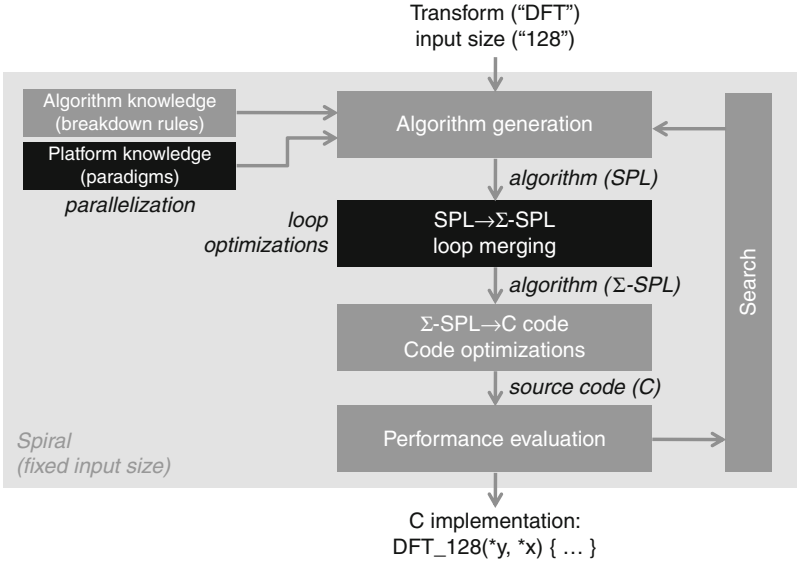
Search. The search drives a feedback loop that generates and evaluates different algorithms to find the fastest. Dynamic programming has proven to work best in many cases, but other techniques including evolutionary search or bandit-based Monte Carlo exploration have also been studied.

Fixed Input Size: Loop Code

The approach to generating straightline code can also be used to generate loop code (Table 4 yields loops), but the code will be inefficient.

The problem: Loop merging. To illustrate the problem, consider the SPL expression

$$(I_4 \otimes F_2)L_4^8.$$



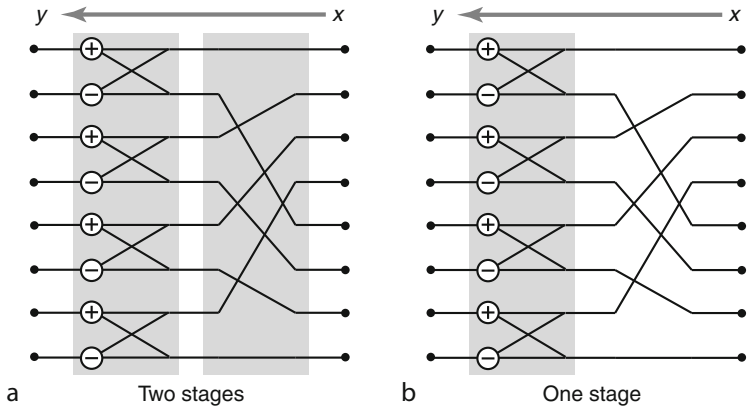
Spiral. Fig. 3 Spiral program generator for fixed input size functions. For straightline code, no Σ-SPL is needed and SPL is translated directly into C code

Spiral. Table 4 Translation of SPL to code. The subscript of A, B specifies the (square) matrix size. `x[b:s:e]` denotes (Matlab style) the subvector of x starting at `b`, ending at `e`, and extracted at stride `s`. D is a diagonal matrix, whose diagonal elements are stored in an array with the same name

SPL expression S	Pseudo code for $y = Sx$
$A_n B_n$	`<code for: t = Bx>` `<code for: y = At>`
$I_m \otimes A_n$	`for (i=0; i<m; i++)` `  <code for: y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])>`
$A_m \otimes I_n$	`for (i=0; i<n; i++)` `  <code for: y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])>`
D_n	`for (i=0; i<n; i++)` `  y[i] = D[i]*x[i];`

L^{km}_k	`for (i=0; i<k; i++)`
	`  for (j=0; j<m; j++)`
	`  y[i*m+j] = x[j*k+i];`

F_2	`y[0] = x[0] + x[1];` `y[1] = x[0] - x[1];`



Spiral. Fig. 4 The loop merging problem for $(I_4 \otimes F_2)L^8_4$

The application of Table 4 yields the code visualized in Fig. 4a:

```
// Input: double x[8], output: y[8]
double t[8];
for(int i=0; i<4; i++) {
   for (int j=0; j<2; j++) {
       t[i*2+j] = x[j*4+i];
   }
}
for (int j=0; j<4; j++) {
   y[2*j]   = t[2*j] + t[2*j+1];
   y[2*j+1] = t[2*j] - t[2*j+1];
}
```

This is known to be suboptimal since the permutation (first loop) can be fused with the subsequent computation loop, thus eliminating one pass through the data (Fig. 4b):

```
//Input: double x[8], output: y[8]
for (int j=0; j<4; j++) {
   y[2*j]   = x[j] + x[j+4];
   y[2*j+1] = x[j] - x[j+4];
}
```

This transformation cannot be expressed in SPL and, in the general case, is difficult to perform on C code. To solve this problem, Σ-SPL was developed, an extension

of SPL that can express loops. The loop merging is then performed by rewriting Σ-SPL expressions.

Σ-SPL. Σ-SPL adds four basic components to SPL:

1. Index mapping functions
2. Scalar functions
3. Parameterized matrices
4. Iterative sum $\sum$

These are defined next.

An integer interval is denoted by $\mathbb{I}_n = \{0, \ldots, n-1\}$, and an index mapping function f with domain $\mathbb{I}_n$ and range $\mathbb{I}_N$ is denoted by

$$f^{n\to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto f(i).$$

An example is the stride function

$$h_{b,s}^{n\to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto b + is, \quad \text{for } s|N. \tag{4}$$

Permutations are written as $f^{n\to n} = f^n$, such as the stride permutation in (1).

A scalar function $f : \mathbb{I}_n \to \mathbb{C};\ i \mapsto f(i)$ maps an integer interval to the domain of complex or real numbers, and is abbreviated as $f^{n\to\mathbb{C}}$. Scalar functions are used to describe diagonal matrices.

Σ-SPL adds four types of parameterized matrices to SPL (gather, scatter, permutation, diagonal):

$$G(f^{n\to N}),\ S(f^{n\to N}),\ P(f^n),\ \text{and}\ \operatorname{diag}\left(f^{n\to\mathbb{C}}\right).$$

Their translation into actual code (which also defines the matrices) is shown in Table 5. For example,

$$G(h_{0,1}^{n\to N}) = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad S(h_{0,1}^{n\to N}) = G(h_{0,1})^\top.$$

Spiral. Table 5 Translation of $\sum$-SPL to code

$\sum$-SPL expression S	Code for $y = Sx$
$G(f^{n\to N})$	`for(i=0; i<n; i++) y[i] = x[f(i)];`
$S(f^{n\to N})$	`for(i=0; i<n; i++) y[f(i)] = x[i];`
$P(f^n)$	`for(i=0; i<n; i++) y[i] = x[f(i)];`
$\operatorname{diag}\left(f^{n\to\mathbb{C}}\right)$	`for(i=0; i<n; i++) y[i] = f(i)*x[i];`
$\sum_{i=0}^{k-1} A_i$	`for(i=0; i<k; i++) <code for: y = A_i * x>`

Finally, Σ-SPL adds the iterative matrix sum

$$\sum_{i=0}^{n-1} A_i$$

to represent loops. The A_i are restricted such that no two A_i have a nonzero entry in the same row.

The following example shows how ⊗ is converted into a sum. A is assumed to be $n \times n$, and domain and range in the occurring stride functions are omitted for simplicity.

$$\begin{aligned} I_k \otimes A &= \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & \\ & & \\ & & \end{bmatrix} + \cdots + \begin{bmatrix} & & \\ & & \\ & & A \end{bmatrix} \\ &= S(h_{0,1})AG(h_{0,1}) + \ldots \\ &\quad + S(h_{(k-1)n,1})AG(h_{(k-1)n,1}) \\ &= \sum_{i=0}^{k-1} S(h_{in,1})AG(h_{in,1}) \end{aligned}$$

Intuitively, the conversion to Σ-SPL makes the loop structure of $y = (I_k \otimes A)x$ explicit. In each iteration i, $G(\cdot)$ and $S(\cdot)$ specify how to read and write a portion of the input and output, respectively, to be processed by A.

Loop merging using Σ-SPL and rewriting. Using Σ-SPL, the loop merging problem identified before in the example $(I_4 \otimes F_2)L_4^8$ is solved by the loop optimization block in Fig. 3 as follows:

$$\begin{aligned} (I_4 \otimes F_2)L_4^8 &\to \left(\sum_{i=0}^{3} S(h_{2i,1})F_2G(h_{2i,1})\right)P(\ell_4^8) \\ &\to \sum_{i=0}^{3} \left(S(h_{2i,1})F_2G(\ell_4^8 \circ h_{2i,1})\right) \\ &\to \sum_{i=0}^{3} \left(S(h_{2i,1})F_2G(h_{i,2})\right) \end{aligned}$$

The first step translates SPL into Σ-SPL. The second step performs the loop merging by composing the permutation ℓ_4^8 with the index functions of the subsequent gathers. The third step simplifies the resulting index functions. After that, actual C code is generated using Table 5.

Besides the added loop optimizations block in Fig. 3, the program generation for loop code operates iteratively exactly as for straightline code.

Fixed Input Size: Parallel Code

As was illustrated in Fig. 1, for compute functions, compilers usually fail to optimally (or at all) exploit the parallelism offered by a platform. Hence, the task falls

with the programmer, who has to leave the standard C programming model and insert explicit threading or OpenMP loops for shared memory parallelism and so-called intrinsics for vector instruction sets. However, doing so in a straightforward way does not necessarily yield good performance.

The problem: Algorithm structure. To illustrate the problem, consider a target platform with four cores that share a cache with a cache block size of two complex numbers.

The first goal is to obtain parallel code with four threads for $I_4 \otimes F_2$ visualized in Fig. 5a. The computation is data parallel; hence, the loop suggested in Table 4 can be replaced, for example, by an OpenMP parallel loop. Note that each processor "owns" as working set exactly one cache block; hence, the parallelization will be efficient.

Now consider again the SPL expression $(I_4 \otimes F_2)L_4^8$ visualized in Fig. 5b. The computation is again data parallel, but the access pattern has changed such that always two processors access the same cache block. This produces false sharing, which triggers the cache coherency protocol and reduces performance. The problem is obviously the permutation L_4^8. Since the rules (e.g., those in Table 2) contain many, and various, permutations, a straightforward mapping to parallel code will yield highly suboptimal performance. To solve this problem inside Spiral, another rewrite system is introduced to restructure algorithms before mapping to parallel code. The restructuring will be different for different forms of parallelism, called paradigms.

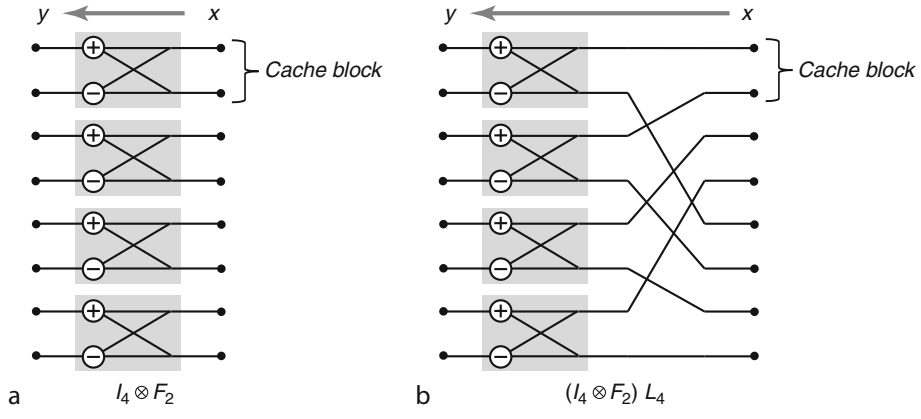
Paradigms and tagged rewriting. A *paradigm* in Spiral is a feature of the target platform that requires structural optimization. Typically, a paradigm is a form of parallelism. Examples include shared memory parallelism (SMP) and SIMD parallelism. A paradigm may be parameterized, for example, by the vector length ν for SIMD parallelism. In Spiral, a paradigm manifests itself by another rewrite system provided by the additional parallelism block in Fig. 3 (and backend extensions in the Σ-SPL to C code block to produce the actual code).

The goal of the new rewrite system is to structurally optimize a given SPL expression into a form that can be efficiently mapped to a given paradigm. The rewrite system is built from three main components:

- *Tags* encode the paradigm and relevant parameters. Examples include the tags "vec(ν)" for SIMD vector extensions and the tag "smp(p, μ)" for SMP. The meaning of the parameters is explained later.
- *Base cases* are SPL constructs that can be mapped well to a given paradigm. As illustrated above, one example is any $I_p \otimes A_n$ for p-way SMP.
- *Tagged rewrite rules* are mathematical identities that translate general SPL expressions toward base cases. An example is the rule (assuming $p|n$)

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p,\mu)} \rightarrow \underbrace{L_m^{mn}}_{\text{smp}(p,\mu)} \left(I_p \otimes (I_{n/p} \otimes A_m)\right) \underbrace{L_n^{mn}}_{\text{smp}(p,\mu)} .$$

The rule extracts the p-way parallel loop (base case) $I_p \otimes (I_{n/p} \otimes A_m)$ from $A_m \otimes I_n$. The stride permutations L_m^{mn} and L_n^{mn} are handled by further rewriting.



Spiral. Fig. 5 Mapping SPL constructs to four threads. Each thread computes one F_2. Both computations are data parallel, but (**a**) produces no false sharing, whereas (**b**) does

Example: *SMP*. For SMP, the tag $\mathrm{smp}(p, \mu)$ contains the number of processors p and the cache block size μ. Base cases include $I_p \otimes A_n$ and $P \otimes I_\mu$, where P is any permutation. $P \otimes I_\mu$ moves data in blocks of size μ; hence false sharing is avoided. From these, other base cases can be built recursively as captured by the sketched grammar in Table 6.

Some SMP rewrite rules are shown in Table 7. Note that the rewriting is not unique, and not every sequence of rules terminates. Once all tags disappear, the rewriting terminates.

Example: *SIMD*. For SIMD, the tag $\mathrm{vec}(\nu)$ contains only the vector length ν. The most important base case is $A_n \otimes I_\nu$, which can be mapped to vector code by generating scalar code for A_n and replacing every operation by its corresponding ν-way vector operation. Other base cases include $L^{2\nu}_\nu$, $L^{2\nu}_2$, and $L^{\nu^2}_\nu$, which are generated automatically from the instruction set [9]. Similar to Table 6, the entire set of vector base cases is specified by a grammar recursively built from the above special constructs.

Parallelization by rewriting. In Spiral, parallelization adds the new parallelization block in Fig. 3. The parallelization rules are applied interleaved with the breakdown rules to generate SPL algorithms that have the right structure for the desired paradigm. For example, for the DFT it may operate as follows:

$$
\begin{aligned}
&\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{smp}(p,\mu)} \rightarrow \underbrace{\left((\mathrm{DFT}_m \otimes I_n) T^{mn}_n (I_m \otimes \mathrm{DFT}_n) L^{mn}_m\right)}_{\mathrm{smp}(p,\mu)} \\
&\quad \cdots \\
&\quad \rightarrow \underbrace{\left(\mathrm{DFT}_m \otimes I_n\right)}_{\mathrm{smp}(p,\mu)} \underbrace{T^{mn}_n}_{\mathrm{smp}(p,\mu)} \underbrace{\left(I_m \otimes \mathrm{DFT}_n\right)}_{\mathrm{smp}(p,\mu)} \underbrace{L^{nm}_m}_{\mathrm{smp}(p,\mu)} \\
&\quad \cdots \\
&\quad \rightarrow \left(\left(L^{mp}_m \otimes I_{n/p\mu}\right) \otimes I_\mu\right) \left(I_p \otimes \left(\mathrm{DFT}_m \otimes I_{n/p}\right)\right) \\
&\qquad \left(\left(L^{mp}_p \otimes I_{n/p\mu}\right) \otimes I_\mu\right) T^{mn}_m \left(I_p \otimes \left(I_{m/p} \otimes \mathrm{DFT}_n\right)\right) \\
&\qquad \left(I_p \otimes L^{mn/p}_{m/p}\right) \left(\left(L^{pn}_p \otimes I_{m/p\mu}\right) \otimes I_\mu\right)
\end{aligned}
$$

First, Spiral applies the breakdown rule (2). Then the parallelization rules transform the resulting SPL expression in several steps. Note how the final expression has only access patterns (permutations) of the form $P \otimes I_\mu$ and all computations are in the form $I_p \otimes A$ (and the diagonal T^{mn}_m). The smaller DFTs can be expanded in different ways, for example, by rewriting for SIMD. Further choices are used for search.

The remaining operation of Spiral including Σ-SPL conversion and search proceeds as before.

Spiral. Table 6 $\mathrm{smp}(p, \mu)$ base cases in Backus–Naur form; n is a positive integer, a_i are real or complex numbers

$\langle \mathrm{smp} \rangle$	::=	$\langle \mathrm{generic} \rangle \mid \langle \mathrm{basic} \rangle \mid$	
		$\langle \mathrm{smp} \rangle \cdot \cdots \cdot \langle \mathrm{smp} \rangle \mid$	(product)
		$\langle \mathrm{smp} \rangle \oplus \ldots \oplus \langle \mathrm{smp} \rangle \mid$	(direct sum)
		$I_n \otimes \langle \mathrm{smp} \rangle \mid$	(tensor product)
		$\ldots$	
$\langle \mathrm{generic} \rangle$	::=	$\mathrm{diag}(a_0, \ldots, a_{n-1}) \mid \ldots$	
$\langle \mathrm{basic} \rangle$	::=	$I_p \otimes A_n \mid P \otimes I_\mu \mid \ldots$	

Spiral. Table 7 Examples of $\mathrm{smp}(p, \mu)$ rewrite rules

$\underbrace{AB}_{\mathrm{smp}(p,\mu)}$	$\rightarrow$	$\underbrace{A}_{\mathrm{smp}(p,\mu)} \underbrace{B}_{\mathrm{smp}(p,\mu)}$
$\underbrace{A_m \otimes I_n}_{\mathrm{smp}(p,\mu)}$	$\rightarrow$	$\underbrace{\left(L^{mp}_m \otimes I_{n/p}\right)\left(I_p \otimes \left(A_m \otimes I_{n/p}\right)\right)\left(L^{mp}_p \otimes I_{n/p}\right)}_{\mathrm{smp}(p,\mu)}$
$\underbrace{L^{mn}_m}_{\mathrm{smp}(p,\mu)}$	$\rightarrow$	$\begin{cases} \underbrace{\left(I_p \otimes L^{mn/p}_{m/p}\right)}_{\mathrm{smp}(p,\mu)} \underbrace{\left(L^{pn}_p \otimes I_{m/p}\right)}_{\mathrm{smp}(p,\mu)} \\ \underbrace{\left(L^{pm}_m \otimes I_{n/p}\right)}_{\mathrm{smp}(p,\mu)} \underbrace{\left(I_p \otimes L^{mn/p}_m\right)}_{\mathrm{smp}(p,\mu)} \end{cases}$
$\underbrace{I_m \otimes A_n}_{\mathrm{smp}(p,\mu)}$	$\rightarrow$	$I_p \otimes \left(I_{m/p} \otimes A_n\right)$
$\underbrace{\left(P \otimes I_n\right)}_{\mathrm{smp}(p,\mu)}$	$\rightarrow$	$\left(P \otimes I_{n/\mu}\right) \otimes I_\mu$

General Input Size

An implementation that can compute a transform for arbitrary input size is fundamentally different from one for fixed input size (compare Table 3b and c). If the input size n is fixed, for example, $n = 4$, the computation is

```
(x,y) -> dft_4(y,x)
```

and all decisions such as the choice of recursion until base cases are reached can be made at implementation time. In an equivalent implementation (called library) for general input size n,

```
(n,x,y) -> dft(n,y,x)
```

the recursion is fixed only once the input size is known. Formally, the computation now becomes

```
n -> ((x,y) -> dft(n,y,x))
```

which is an example of function currying. A C++ implementation is sketched in Table 3c, where the two steps would take the form

```
dft * f = new dft(n); // initialization
f->compute(y, x);     // computation
```

The first step determines the recursion to be taken using search or heuristics, and precomputes the twiddle factors needed for the computation. The second step performs the actual computation. The underlying assumption is that the cost of the first step is amortized by a sufficient number of computations. This model is used by FFTW [15] and the libraries generated by Spiral.

To support the above model, the implementation needs recursive functions. The major problem is that the optimizations introduced before operate in nontrivial ways across function boundaries, thus creating more functions than expected. The challenge is to derive these functions automatically.

The problem: *Loop merging across function boundaries*. To illustrate the problem, consider the Cooley–Tukey FFT (2). A direct recursive implementation would consist of four steps corresponding to the four matrix factors in (2). Two of the steps would call smaller DFTs:

```
void dft(int n, cpx *y, cpx *x) {
   int k = choose_factor(n);
   int m = n/k;
   cpx *t1 = Permute x with L(n,k);
   // t2 = (I_k tensor DFT_m)*t1
   for(int i=0; i<k; ++i)
      dft(m, t2 + m*i, t1 + m*i);
   // t3 = T^n_m*t2, f() computes
   // diagonal entries of T
   for(int i=0; i<n; ++i)
      t3[i] = f(i) * t2[i];
   // y = (DFT_k tensor I_m)*t3,
   //cannot call dft() recursively,
   //need strided I/O
   for(int i=0; i<m; ++i)
      dft_stride(k, m, y + i, t3 + i);
}
// to be implemented
void dft_stride(int n, int stride,
   cpx *Y, cpx *X);
```

Note how even this simple implementation is not self-contained. A new function `dft_stride` is needed that accesses the input in a stride and produces the output at the same stride (see the data flow in Fig. 2).

However, as explained before, loops should be merged where possible. For fixed size code, Spiral would merge the first loop with the second, and the third loop with the fourth, using Σ-SPL rewriting. The same can be done in the general size recursive implementation, but the merging crosses function boundaries:

```
void dft(int n, cpx *y, cpx *x) {
   int k = choose_factor(n);
   //t1 = (I_k tensor DFT_m)L(n,k)*x
   for(int i=0; i < k; ++i)
      dft_iostride(m, k, 1, t1 + m*i,
        x + m*i);
   // y = (DFT_k tensor I_m) T^n_m
   // diagonal entries of T are now
   //precomputed in precomp_f[]
   for(int i=0; i < m; ++i)
      dft_scaled(k, m, precomp_f[i],
        y + i, t1 + i);
}


// to be implemented
void dft_iostride(int n, int istride,
int ostride, cpx *y, cpx *x);
void dft_scaled(int n, int stride,
cpx *d, cpx *y, cpx *x);
```

Now there are two additional functions: `dft_iostride` reads at a stride and writes at a different stride, and `dft_scaled` first scales the input and then performs a DFT at a stride.

So at least three functions are needed with different signatures. However, the two additional functions are also implemented recursively, possibly spawning new functions. Calling these functions *recursion steps*, the main challenge is to automatically derive the complete set of recursion steps needed, called the "recursion step closure." Further, for each recursion step in the closure, the signature has to be derived.

Recursion step closure by Σ-SPL rewriting. Spiral derives the recursion step closure using Σ-SPL and the same rewriting system that is used for loop merging.

For example, the two additional recursion steps in the optimized implementation above are automatically obtained from (2) as follows. Recursion steps are marked by overbraces.

$$
\begin{aligned}
\overbrace{\mathrm{DFT}_n} &\rightarrow (\overbrace{\mathrm{DFT}_{n/k}} \otimes I_k) T^n_k (I_{n/k} \otimes \overbrace{\mathrm{DFT}_k}) L^n_{n/k} \\
&\rightarrow \left(\sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\mathrm{DFT}_{n/k}} G(h_{i,k}) \right) \\
&\quad \mathrm{diag}(f) \left(\sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\mathrm{DFT}_k} G(h_{jk,1}) \right) P(\ell^n_{n/k}) \\
&\rightarrow \sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\mathrm{DFT}_{n/k}} \mathrm{diag}(f \circ h_{i,k}) G(h_{i,k}) \\
&\quad \sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\mathrm{DFT}_k} G(h_{j,n/k}) \\
&\rightarrow \overbrace{\sum_{i=0}^{k-1} S(h_{i,k}) \mathrm{DFT}_{n/k} \mathrm{diag}(f \circ h_{i,k}) G(h_{i,k})} \\
&\quad \overbrace{\sum_{j=0}^{n/k-1} S(h_{jk,1}) \mathrm{DFT}_k G(h_{j,n/k})}
\end{aligned}
\tag{5}
$$

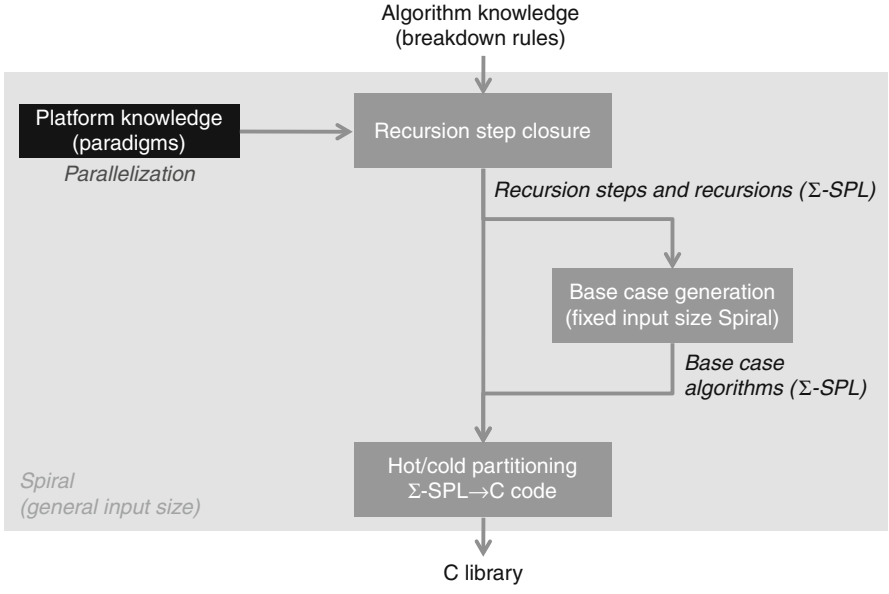
The first step applies the breakdown rule (2). The second step converts to Σ-SPL. The third step performs loop merging as explained before. The fourth step expands the braces to include the context. The two expressions under the braces correspond to the two functions `dft_iostride` and `dft_scaled`. The process is now repeated for the expression under the braces until closure is reached. In this example, only one additional function is needed, i.e., the recursion step closure consists of four mutually recursive functions. The derivation of the recursion steps also yields a Σ-SPL specification of the actual recursion, i.e., their implementation by a recursive function (e.g., (5) for DFT_n).

For the best performance, the braces may be extended to also include the loop represented by the iterative sum. Moving the loop into the function enables better C/C++ compiler optimizations.

If the implementation is vectorized or parallelized, the initial breakdown rules are first rewritten as explained before and then the closure is computed. The size of the closure is typically increased in this case.

Program generation for general input size: Overview. The overall process is visualized in Fig. 6. The input to Spiral is now a (sufficient) set of breakdown rules for a given transform or transforms. The rules are parallelized if desired, using the appropriate paradigms; then the recursion step closure is computed, which also yields the actual recursions.

The resulting recursion steps need base cases for termination. These are generated using the algorithm generation block from the fixed input size Spiral (Fig. 3)



Spiral. Fig. 6 Spiral program generator for general input size libraries

for a range of small sizes (e.g., $n \leq 32$) to improve performance. These, the recursion steps, and the recursions are fed into the final block to generate the final library. Among other junctions, the block performs the *hot/cold partitioning* that determines which parameters in a recursion step are precomputed during initialization and which become parameters of the actual compute function. Finally, the actual code is generated (which now includes recursive functions) and integrated into a common infrastructure to obtain a complete library.

Many details are omitted in this description and are provided in [29, 30].

Extensions

A major question is whether the approach taken by Spiral can be extended beyond the domain of linear transforms, while maintaining both the basic principles outlined in the introduction and the ability to automatically perform the necessary transformations and reasoning. First progress in this direction was made in [7] with the introduction of the operator language (OL). OL generalizes SPL by considering operators that may be nonlinear and may have more than one vector input or output. Important constructs such as the tensor product are generalized to operators. First results on program generation for functions such as radar imaging, Viterbi decoding, matrix multiplication, and the physical layer functions of wireless communication protocols have already been developed.

Related Entries

►ATLAS (Automatically Tuned Linear Algebra Software)
►FFT (Fast Fourier Transform)
►FFTW

Bibliographic Notes and Further Reading

Spiral is based on early ideas on using tensor products to map FFT algorithms to parallel supercomputers [16]. The first paper describing SPL and the SPL compiler is [32]. See also [14] for basic block optimizations for transforms. The first complete basic Spiral system including SPL algorithm generation and search was presented in [23], with a more extensive treatment in [24] and probably the best overview paper [22], which fully develops SPL for a variety of transforms. The path to complete automation in the transform domain continued with Σ-SPL and loop merging [11], the introduction of rewriting systems for SIMD vectorization [8, 13] and base case generation [9], SMP parallelization [12], and distributed memory parallelization [2, 3]. The final step to generating general size, parallel, adaptive libraries was made in [29, 30]. The generated libraries are modeled after FFTW [15], which is written by hand but uses generated basic blocks [14].

The most important extensions of Spiral are the following. Extensions to generate Verilog for field-programmable gate-arrays (FPGAs) are presented in [18, 19]. Search techniques other than dynamic programming are developed in [5, 26]. The use of learning to avoid search was studied in [6, 25]. Finally, [4, 7, 17] make the first steps toward extending Spiral beyond the transform domain including the first OL description. The Spiral project website with more information and all publications is given in [1].

A good introduction to FFTs using tensor products is given in the books [27, 28]. A comprehensive overview of algorithms for Fourier/cosine/sine transforms is given in [21, 31]. A good introduction to mapping FFTs to multicore platforms is given in [10].

Bibliography

1. Spiral project website. www.spiral.net
2. Bonelli A, Franchetti F, Lorenz J, Püschel M, Ueberhuber CW (2006) Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: International symposium on parallel and distributed processing and application (ISPA), Lecture notes in computer science, vol 4330. Springer, Berlin, pp 818–832
3. Chellappa S, Franchetti F, Püschel M (2009) High performance linear transform program generation for the Cell BE. In: Proceedings of the high performance embedded computing (HPEC), Lexington, 22–23 September 2009
4. de Mesmay F, Chellappa S, Franchetti F, Püschel M (2010) Computer generation of efficient software Viterbi decoders. In: International conference on high performance embedded architectures and compilers (HiPEAC), Lecture notes in computer science, vol 5952. Springer, Berlin, pp 353–368
5. de Mesmay F, Rimmel A, Voronenko Y, Püschel M (2009) Bandit-based optimization on graphs with application to library performance tuning. In: International conference on machine learning (ICML), ACM international conference proceedings series, vol 382. ACM, New York, pp 729–736

6. de Mesmay F, Voronenko Y, Püschel M (2010) Offline library adaptation using automatically generated heuristics. In: International parallel and distributed processing symposium (IPDPS)
7. Franchetti F, de Mesmay F, McFarlin D, Püschel M (2009) Operator language: a program generation framework for fast kernels. In: IFIP working conference on domain specific languages (DSL WC), Lecture notes in computer science, vol 5658. Springer, Berlin, pp 385–410
8. Franchetti F, Püschel M (2002) A SIMD vectorizing compiler for digital signal processing algorithms. In: International parallel and distributed processing symposium (IPDPS). pp 20–26
9. Franchetti F, Püschel M (2008) Generating SIMD vectorized permutations. In: International conference on compiler construction (CC), Lecture notes in computer science, vol 4959. Springer, Berlin, pp 116–131
10. Franchetti F, Püschel M, Voronenko Y, Chellappa S, Moura JMF (2009) Discrete Fourier transform on multicore. IEEE Signal Proc Mag 26(6):90–102
11. Franchetti F, Voronenko Y, Püschel M (2005) Formal loop merging for signal transforms. In: Programming languages design and implementation (PLDI). ACM, New York, pp 315–326
12. Franchetti F, Voronenko Y, Püschel M (2006) FFT program generation for shared memory: SMP and multicore. In: Supercomputing (SC). ACM, New York
13. Franchetti F, Voronenko Y, Püschel M (2006) A rewriting system for the vectorization of signal transforms. In: High performance computing for computational science (VECPAR), Lecture notes in computer science, vol 4395. Springer, Berlin, pp 363–377
14. Frigo M (1999) A fast Fourier transform compiler. In: Proceedings of the programming language design and implementation (PLDI). ACM, New York, pp 169–180
15. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93(2):216–231
16. Johnson J, Johnson RW, Rodriguez D, Tolimieri R (1990) A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. IEEE Trans Circ Sys 9:449–500
17. McFarlin D, Franchetti F, Moura JMF, Püschel M (2009) High performance synthetic aperture radar image formation on commodity architectures. Proc SPIE 7337:733708
18. Milder PA, Franchetti F, Hoe JC, Püschel M (2008) Formal datapath representation and manipulation for implementing DSP transforms. In: Design automation conference (DAC). ACM, New York, pp 385–390
19. Nordin G, Milder PA, Hoe JC, Püschel M (2005) Automatic generation of customized discrete Fourier transform IPs. In: Design automation conference (DAC). ACM, New York, pp 471–474
20. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1992) Numerical recipes in C: the art of scientific computing, 2nd edn. Cambridge University Press, Cambridge
21. Püschel M, Moura JMF (2008) Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. IEEE Trans Signal Proces 56(4):1502–1521
22. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N (2005) SPIRAL: code generation for DSP transforms. Proc IEEE (Special Issue on Program Generation, Optimization, and Adaptation) 93(2):232–275
23. Püschel M, Singer B, Veloso M, Moura JMF (2001) Fast automatic generation of DSP algorithms. In: International conference on computational science (ICCS), Lecture notes in computer science, vol 2073. Springer, Berlin, pp 97–106
24. Püschel M, Singer B, Xiong J, Moura JMF, Johnson J, Padua D, Veloso M, Johnson RW (2004) SPIRAL: a generator for platform-adapted libraries of signal processing algorithms. J High Perform Comput Appl 18(1):21–45
25. Singer B, Veloso M (2001) Learning to generate fast signal processing implementations. In: International conference on machine learning (ICML). Morgan Kaufmann, San Francisco, pp 529–536
26. Singer B, Veloso M (2001) Stochastic search for signal processing algorithm optimization. In: Supercomputing (SC). ACM, New York, p 22
27. Tolimieri R, An M, Lu C (1997) Algorithms for discrete Fourier transforms and convolution, 2nd edn. Springer, Berlin
28. Van Loan C (1992) Computational framework of the fast Fourier transform. SIAM, Philadelphia
29. Voronenko Y (2008) Library generation for linear transforms. Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University
30. Voronenko Y, de Mesmay F, Püschel M (2009) Computer generation of general size linear transform libraries. In: International symposium on code generation and optimization (CGO). IEEE Computer Society, Washington, DC, pp 102–113
31. Voronenko Y, Püschel M (2009) Algebraic signal processing theory: Cooley–Tukey type algorithms for real DFTs. IEEE Trans Signal Proces 57(1):205–222
32. Xiong J, Johnson J, Johnson RW, Padua D (2001) SPL: a language and compiler for DSP algorithms. In: Programming languages design and implementation (PLDI). ACM, New York, pp 298–308

SPMD Computational Model

Frederica Darema
National Science Foundation, Arlington, VA, USA

Definition of the Subject

The Single Program – Multiple Data (SPMD) parallel programming paradigm is premised on the concept that all processes participating in the execution of a program work cooperatively to execute that program, but at any given instance different processes may execute different instruction-streams, and act on different data

and on different sections in the program, and whereby these processes dynamically self-schedule themselves, according to the program workflow and through synchronization constructs embedded in the application program. SPMD programs comprise of serial, parallel, and replicate sections.

Introduction

The (*SPMD)* Single Program-Multiple Data model [1–6] is premised on the concept that all processes participating in the (parallel) execution of a program work cooperatively to execute this program, but at any given instance, through synchronization constructs embedded in the application program, different processes may execute different instruction-streams, and act on different data and on different sections in the program; thus the name of the model: Single Program – Multiple Data. The model was proposed by the author in January 1984 [1], as a means for expressing and enabling parallel execution of applications on highly parallel MIMD computational platforms. The term "single-program" was used for emphasis on the parallel execution of a given program (i.e., concurrent execution of tasks in a given program), in distinction from the environments of that time, where OS-level concurrent tasks would (concurrently) execute different programs on the multiprocessors of that time (e.g., IBM 3081). The initial motivation for the SPMD model was to enable the expression of the (then) high degrees of parallelism supported by the IBM Research Parallel Processor Prototype (RP3) [7] parallel computer system. The model was first implemented in the Environment for Parallel Execution (EPEX) [2–6] programming environment (one of the first general-purpose parallel programming environments, and of course the first to implement SPMD). This entry is an excerpt of a more comprehensive treatise on the SPMD [8] which puts the motivation for the SPMD in the context of the landscape of the computer platforms and software support of the early- to mid-1980s, and discusses experiences, effectiveness, and impact of the SPMD in expediting adoption of parallelism in that time frame and as it has influenced derivative programming environments in the intervening 25 years.

The SPMD model fostered a new approach to parallel programming, differing than the Fork & Join (or Master–Slave) model predominantly pursued by others at around the 1984 timeframe. In SPMD, all processes that participate in the parallel execution of a program commence execution of the program, and each process is dynamically self-scheduled and selects work-tasks to execute (self-allocated work), based on parallelization directives embedded in the program and according to the program workflow. In the SPMD model, a process represents a separate instantiation of the program, and processes participating in the cooperative execution of a program (also referred to as parallel processes) execute distinct instruction streams in a coordinated way through these embedded parallelization directives (also referred to as synchronization directives). With respect to parallel execution, in general a program consists of sections that are executed by one process (*serial sections*) and sections that can be executed by multiple cooperating processes (i.e., *parallel sections*, that can be executed by several processes in a cooperative and concurrent manner, and *replicate sections*, where the computations are executed by every process; parallel sections may include one or more levels of nested parallel sections). Regardless of the section considered, the beginning and end of serial and of parallel sections (and nested parallel sections) are points of synchronization of the processes (*synchronization points*), that is, points of coordination and control of the execution path of the participating processes. The definition in [1, 2]: "*all processes working together will execute the very same program,*" uses the word "same" to emphasize concurrent execution of a given program by these ("multiple") processes, and not to imply that all these processes execute identical instruction streams (as it has been interpreted by some). In expressing parallelism through the SPMD model, the flow of control is distinguished in two classes: the *parallel* flow of control is the flow of control followed by the processes while executing a serial or a parallel section in the program; the *global flow of control* is the flow of control followed by the processes as they step through the synchronization points of the program, according to the program workflow. In parallel execution with the SPMD model, the participating processes follow a different *parallel flow of control*, but all the processes follow the same *global flow of control*.

Conceptually, the SPMD model has been from the outset a general model, enabling to express parallelism for concurrent execution of distinct instruction streams and allowing application-level parallelization control

and dynamic scheduling of work-tasks ready to execute (with process self-scheduling). The model is able to support general MIMD task-level parallelism, including nested parallelism, and is applicable to a range of parallel architectures (those considered at the time the model was proposed and other parallel and distributed architectures that have subsequently appeared) in more efficient and general ways than the then proposed alternate parallel programming models, such as those based on Fork-and-Join, SIMD, and data-parallel and systolic-arrays approaches. In fact SPMD allows combining with and implementing such models as parallel execution models invoked under the SPMD rubric. Applied to parallel architectures supporting shared memory (for example RP3) the SPMD is a global-view programming model. From its inception, SPMD allowed efficient implementations of expressing parallelism, through parallelization directives inserted in the initial serial application programs, the ensuing "parallelized" program compiled with the standard serial compliers. The parallelization directives enabled application-level dynamic scheduling and control of the parallel execution, with efficient runtime implementations, requiring minimal interaction with the OS, avoiding (the heavy) OS level-synchronization overheads, and without requiring new, parallel OS services. These were important considerations in the 1984 time frame, the SPMD showed early-on that it allowed the parallelization of non-trivial programs; many were production-level application programs, for example, from the areas of applied physics, aerospace, and design automation. Thus the *SPMD* expedited adoption of parallelism in the mid-1980s, as the model enabled to determine that it was not difficult to express parallelism, and map and execute such non-trivial applications on parallel machines, and thus exploit parallelism and do so flexibly and adaptively, and with efficiency.

SPMD enabled creating parallel versions of FORTRAN and C programs, and also enabled the parallel execution of such programs without the need for new parallel languages. To date, there are many environments that have implemented *SPMD*, primarily for science and engineering parallel computing, but also for commercial applications. The most widely used today being MPI [9] for distributed multiprocessing systems or *"message-passing"* systems, and later used also for shared memory systems; the predecessor of MPI being PVM [10], which appeared in the late-1980s as one of the first popular implementations of SPMD for "message-passing" systems. The SPMD model through its implementation in MPI is widely used for exploiting today's heterogeneous complex parallel and distributed computational platforms, including computational grids [11], which embody many types of processors, multiple levels of memory hierarchy, and multiple levels of networks and communication, and which may employ a combination of SPMD as well as Fork-and-Join programming environments. Other influential programming environments also based on SPMD include OpenMP [12] for shared-memory parallel programming in FORTRAN, C, and C++; Titanium [13] for parallelization of Java programs; and Split-C [14] and Unified Parallel C (UPC) [15] for parallel execution of C programs.

The SPMD Model

In the SPMD model, a parallel program conceptually consists of serial, parallel (including nested parallel), and replicate sections. The program data are distinguished into application data and synchronization data; synchronization data are shared among the parallel processes, and application data are distinguished into private and into shared data. In the following are discussed how these parallelization aspects are treated in SPMD:

- *Serial sections* are executed by one process (either a designated process, or more generally, and typically, the first one to arrive at that section). The other processes that arrive at this serial section, through the "*serial-section synchronization directives*" are directed to bypass its execution; there, they either wait at the end of the section for its execution to complete, or they may proceed to seek and execute available work-task(s) in subsequent section(s), if that is allowed by the program workflow dependencies (and with appropriate synchronizations imposed – *soft* and *hard* barriers, discussed later on in their EPEX implementation).
- *Parallel sections* are executed concurrently by processes that arrive at the beginning of such a section. These processes, through "*parallel-section synchronization directives*" dynamically self-schedule themselves and get allocated with the next available parallel work-task to execute. As each of these

processes arriving in a parallel section is self-assigned a work - task, and as each process completes its task of work, these processes can seek the next available work-task to execute in that parallel section; or if all work tasks in the parallel section have been allocated to processes, the remaining process(es) can proceed to the end of that parallel section, and either wait there till all the work of the parallel section is completed, or continue-on to the next section of the program or other subsequent sections of the program as allowed by the program workflow dependencies (and with appropriate synchronizations imposed – *soft* and *hard* barriers, discussed later on in their EPEX implementation). Parallel loops in programs are the predominant form of a parallel section and thus major targets of parallel execution; however, SPMD also supports other forms of general task parallelism. Later in this section nested parallelism support in SPMD is discussed.

- *Replicate sections* of the program are sections allowed to be executed by all processes (it is the default mode of execution that was envisioned when the SPMD was first proposed). Such sections involve parts of the computation (typically small portion of the total amount of the computation) and where allowing all processes to replicate the execution of the computation is more efficient than having one process execute the section (while the others wait) and then make the result of the computation available to other processes (as shared data or as communicated messages). This is applicable, for example, where replicate execution avoids: the serialization overhead, the busy-waiting and polling a synchronization semaphore by the other processes (with potential additional overhead due for example contention in this semaphore, and also potential network contention), and the overhead of making the resulting data available to the other process (e.g., in shared memory architectures, potential of contention upon access of the resulting data placed in shared memory; or network contention upon broadcasting the results to the other processes through message passing, in logically distributed, "message-passing" architectures). Especially for message-passing environments, such overheads can be rather high, and in such cases the approach of replicate computations can be a more efficient alternative for certain portions of the computation.
- *Shared and Private Data*: These refer to application data and synchronization data. For parallel machine architectures which support shared as well as private memory, parallelism is expressed by considering two types of data: *shared data* and *private data*; each of the parallel processes having read and write capability on the shared data, while private data are exclusive to each process (and typically the private data of a process would reside in the local memory of the processor on which the process executes). A*pplication shared data* and *synchronization data* are declared as *shared data*. SPMD was originally proposed for architectures supporting a mix of shared and (logically local – private) memory, and the approach followed in the original SPMD implementation was for the default to be the *private data* (to each parallel process); this decision was made (by the author) on the basis that it is easier (especially for the user) to identify the application shared data and the synchronization data, rather than explicitly defining the private data as other approaches have done. For message-passing architectures, where the SPMD was also applied, the *application shared data* are partitioned into per process private data, and "sharing" of such data is effected through "messages" exchanged between parallel processes, as needed during the course of the computation. It is the belief of this author that expressing parallelism through message-passing is more challenging than supporting parallelism through a "shared-memory plus private memory" architecture support, and that is also the case with the kinds of programming models for expressing parallelism (more on this in later discussion on MPI).
- *I/O*: The SPMD model allows general flexibility in handling I/O. Any of the parallel processes executing a parallel program are allowed to perform I/O, and (as discussed later-on in the context of the EPEX programming environment) all processes can have read/write access to any of the parallel program's files. One can argue that the typical approach would be for I/O to be treated as a serial action, although that is not necessarily always so. For example, upon

commencing execution of a program, one process typically would be assigned to read the input data file, and perform the program state initialization. However, there are many other instances in the program execution, for example, within parallel sections, where it may be more efficient to allow each process to read data from a file, and as needed write data back into a file. In this case, for flexibility, random-access file structure would be more desirable, than sequential record files; it is then not necessary to attach a file to a given process, which is a possible but more restrictive approach.

In the early implementations of the SPMD, where processes were implemented by heavy-weight mechanisms (such as VMs – Virtual Machines), to enable efficient parallel execution, all processes were created in the beginning of the program execution, and this has been applied to most SPMD implementations since then. Noted, however, that the SPMD model does not preclude by principle the case where additional processes can be created and participate in the program execution; this was allowed through EPEX implemented directives, allowing such additional processes to skip the already executed portion of the entire computation. However, at the time SPMD was proposed and within the systems then available for the implementation of the model, creation of such processes and inheriting program state was expensive, unlike capabilities that were enabled later on, like creation of light-weight threads.

Nesting in SPMD can be implemented by allowing enough processes to enter an (outer) parallel loop, for example, allocate several processes – a group of processes – to a given outer loop iteration, allowing only one of these processes to execute the outer loop portion (for example, with appropriate directives treating it as a "nested serial") and then allow all processes in this group to execute in parallel the inner loop. In fact this author experimented with such approaches, as well as a hybrid of SPMD combined with Fork-and-Join capabilities to implement nesting, but they were never implemented in a production way, due to the overheads of spawning VM/SP virtual processes, but also because in the mid-1980s timeframe the degrees of parallelism in the applications (e.g., dimensionality of the outer parallel loops in a program) exceeded the numbers of processors in the commercial and in the prototype systems of that time, so exploiting nesting was more of an intellectual endeavor rather than a practical need. With threading capabilities, other implementations can be used to support more elegantly execution of nested parallel computations, for example, by each process spawning threads to execute the inner iterations of the loop, like for example in OpenMP. Later in this entry is discussed the need for nested or multilevel threading capabilities, gang scheduling of threads, partially shared data and active data-distribution notions, in the context of multi-level parallelization hierarchies that will be encountered in the emerging and future parallel architectures.

The SPMD model does not restrict how many processes can run on each parallel processor. While often one process executes per processor, there are cases where multiple processes may be spawned to execute per processor, for example, to mask memory latency, shared memory contention, I/O, etc. Also SPMD does not require a given process to be attached to a given processor; however, the typical approach is to bind a process (or a thread to a processor), to avoid cost of migration, context-switch, and better exploitation of (local data) cache; however, there are situations where, as long as there is no thrashing, a process may be moved, for example, to improve performance, and in other cases for fault tolerance or recovery [16, 17].

Other SPMD features are provided as illustrative examples of the EPEX programming environment discussed next.

The EPEX Programming Environment – Implementation of the SPMD Model

The first implementation of SPMD was in the EPEX environment [2–5], initially implemented on IBM multiprocessors like the dual processor IBM/3081 and, later, the six-processor IBM/3090 vector multiprocessor (through MVS/XA [18]). The IBM/3081 through the VM/SP OS [19] supported execution of multiple virtual machines (VMs), running in time-shared mode (on each node of the 3081). This capability was used to simulate parallel execution by a large number of processors working together on a single program, each simulated parallel processor and its local memory (correspondingly for each of the parallel processes) being simulated by a VM (a virtual machine). Although in

practice the 3081 hardware supported 2-way parallelism (and the 3090 hardware supported 6-way parallelism), the simulation environment created was used to simulate up to 64 parallel processors; of course, in principle the environment could simulate even higher numbers of processors. To simulate execution on parallel platforms, like RP3, the VM/SP OS support of the Writable Shared Segments (VM/WSS) capability was used, which allowed to establish a portion of the virtual machines' memory as shared across a set of VMs (in read and write modalities); this feature allowed to simulate the shared-memory of a parallel system (like RP3), while the private memory of each virtual machine simulated the local memory of an RP3 processor, and the private memory of the corresponding process. The VM/WSS was also used to emulate message-passing environments, by using the WSS-based (virtual) shared memory as a "mailbox" for messages, and thus allowing to also experiment and to demonstrate the use of SPMD as a model also supporting message-passing parallel computation. These capabilities were used to create a simulation platform for the EPEX programming environment, and were also used for the development of other simulation tools for analysis of execution on various parallel machines with shared-plus-local memory systems as well as for "message - passing" distributed systems. While over the last two decades most parallel systems have been used with MPI (a message-passing environment), it behooves elaborating on shared-plus-local memory systems because the emerging multicore-based architectures are expected to support such memory organizations within the likely set of memory hierarchies.

Initially, parallelization of programs under EPEX was enabled by implementing the "serial," "parallel," and "barrier" parallelization directives through synchronization subroutine calls inserted in the program; the first EPEX implementation was applied to parallelization of FORTRAN programs, and shared data declared explicitly, through FORTRAN Shared COMMON statements. Shortly thereafter, the synchronization subroutine calls were replaced by corresponding parallelization macros, with the development of a preprocessor (source-to-source translator from macros to subroutine calls). Denoting the parallelization directives through macros, and likewise for the declaration of application shared data, allowed converting serial programs to their parallel counterparts, more easily and elegantly. Additional synchronization constructs were also expressed through other appropriate macros. All these parallelization macros inserted in the program were then expanded by the "EPEX Preprocessor" into the parallel program version, by inserting the SPMD parallelization subroutines (specifically referred to here is the EPEX FORTRAN Preprocessor source-to-source translator [20–22]; later a preprocessor for C was also developed [23]). Then the ensuing "parallelized program" was compiled through a standard serial FORTRAN or C compiler. By '87–'88 the SPMD-based EPEX system had been installed in ten IBM sites (including IBM Scientific Centers, such as those in Palo Alto, CA-USA and Rome, Italy), three National and International Labs (LANL, ANL, and CERN); two industrial partners (MartinMarietta and Grumman), and eight universities.

The syntax and utilization of the initial EPEX implementation is presented in [2–5]. The macros implementing SPMD and the EPEX preprocessor are presented in detail in [6, 19–21] which document in more detail the preprocessor–based EPEX environment. Here, for reference, an illustrative sample of the set of the EPEX macros is given with a brief description of their use:

- The traditional FORTRAN DO-loop was designated through the:

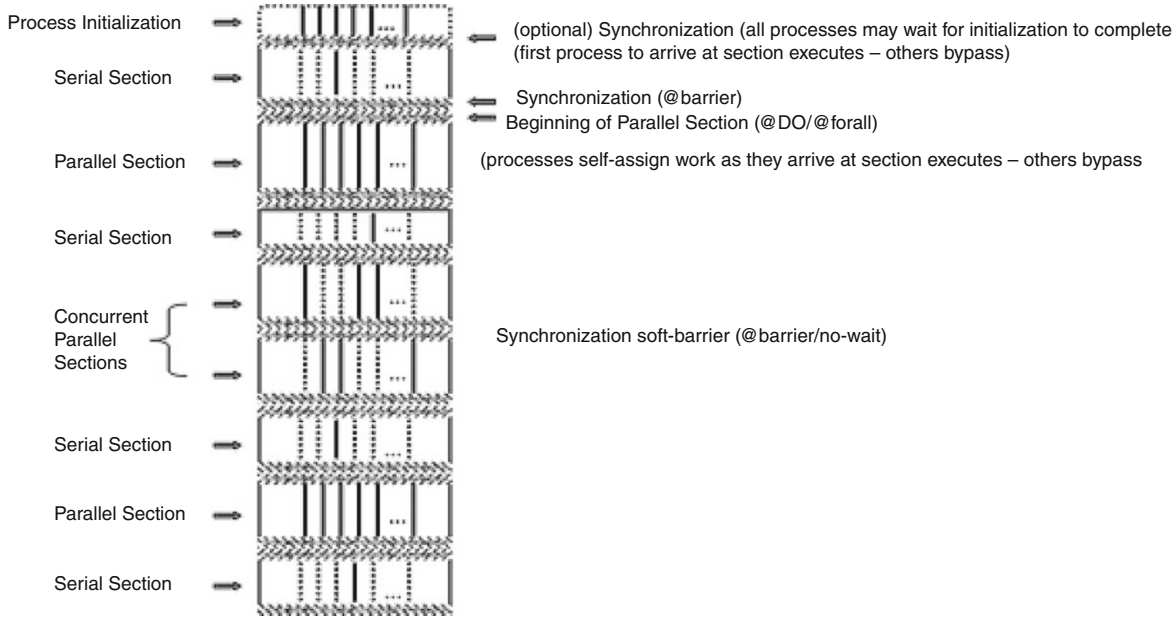
```
@DO [stmt#|label] index = n1, n2, [n3]
      [CHUNK = chunksize]
        … loop body …
[stmt#] @ENDO [label] [WAIT | NOWAIT]
```

  Processes entering the @DO get self-assigned with the next available loop-iteration(s), through synchronization routines utilizing the Fetch-and-Add (F&A) [24] RP3 primitive (implemented in the simulated environment through the Compare&Swap instruction); processes that arrive after all the work in the loop has been allocated or all the work in the loop has been completed proceed to the end of the section (@ENDO); there they may wait for the loop execution to be completed, or proceed to subsequent section(s), as allowed by the program workflow dependencies. The CHUNK option allows processes to get assigned more than one iterations at a time, thus decreasing the overhead of parallelism; the WAIT option requires processes to wait for the

execution of the preceding section (in this case the loop body) to be completed before continuing to the next section of the program; the NOWAIT option allowed processes that reach the end of the section to continue execution of subsequent section(s), thus allowing multiple concurrent sections of the program (or subroutines) to be executed concurrently, as allowed by dependencies in the program workflow.

- The serial section is bounded by the @SERBEG and @SEREND macros. The default is for the first process to encounter the @SERBEG to execute the section, the other processes proceed to @SEREND, with [WAIT | NOWAIT] options similar to the ones for a parallel loop. The environment also allows designating a given process to execute a given serial section.
- The @WAITFOR macro can be inserted at any point of the program and designates that processes cannot go beyond that point, until *a logical condition* specified by the argument of the macro is satisfied; for instance, this construct can be used for imposing various kinds of soft barriers – for example, for the processes to wait until execution of the preceding section(s) is completed.
- The @BARRIER macro can be inserted at any point of the program and forces all processes participating in the execution of the program to arrive at that point of the program (hard barrier).
- The parallel processes are endowed with an identity [@MYNUM]; the MYNUM for each process can be a parameterized assignment. This property can be used in several ways: for example, for designating a given process to perform I/O to a file if that is desired, or execute a given section (e.g., a serial section) that can be designated to be executed by a given process (of course in general this is not mandatory, as the default option for a section, and more generally the next available work-task is to be executed on a "first-arrives-executes" basis).
- @SHARED [application shared data: parameters, arrays] allows the user to designate the application shared data; by default all other data of the application are assumed as private to each parallel process; @SHARED [synchronization data] are created automatically by the preprocessor and designated all the shared data used by the synchronization constructs.

A schematic of an example of SPMD execution is shown in Fig. 1.



SPMD Computational Model. Fig. 1 Schematic of an example of SMPD MIMD task-parallel execution

- Handling of I/O: In [25, 26] are discussed implementations of ideas for parallel I/O presented in the previous section, which allowed I/O by multiple processes, and not necessarily serializing I/O. A file could be opened, written into and closed, and then made available to other processes. This could be done at the end of a parallel section or at the end of an outmost iteration of the program, depending on the problem needs; such implementations might be enabled via a barrier at the end of the outermost loop, or if no barrier was imposed the file maybe accessed (in read-write mode) by another process (which may need to spin-wait if the file had not been closed by the previous process). The @MYNUM designation allowed to tie a given process to a file if that was desirable. The advent of newer file-systems (such as GPFS [27, 28]) allows more flexible mechanisms for parallel I/O in SPMD programs, and parallel I/O is supported in programming environments, like OpenMP. Furthermore, as mass storage technologies evolve from disk to solid-state and nano-devices, it is possible that parallel I/O could be implemented through other than file-based methods (perhaps something akin to a "DBMS-like" I/O management).

Through the EPEX parallelization directives, the participating processes executing cooperatively the program step through the synchronization points in the program, and are dynamically allocated the next work-task or execution path to take. In that sense all participating processes follow the same global flow of control in the program. In executing serial or parallel sections, each process is allocated the next available work-task, and thus they may execute different instructions at any given time and act on different parts of the program (including potentially different procedures or subroutines), as consistent with the workflow dependencies in the program. A given process may follow a different parallel flow of control during different passes of an outer iteration of the entire program. The synchronization routines provided in EPEX allowed executing outer iterations of the entire program (that is repeated execution of the serial and parallel sections of the program) without necessarily imposing the need for the programmer to introduce explicit blocking at the end of the outer iteration of the program.

The EPEX programming environment allowed to parallelize and simulate the parallel execution of a large number of applications, to understand their characteristics, and assess the efficacy and effectiveness of parallelism. In the 1984 – 1986 time-span, over 40 applications were parallelized with SPMD, and executed in parallel under the EPEX programming environment. The set of these applications (mostly numeric, but also non-numeric) included: Fluid Dynamics (e.g., advanced turbulent-flow programs [29], and shallow water and heart blood-flow problems); Radiation Transport (Discrete Ordinates methods and Monte-Carlo); Design Automation (Chip placement and Routing by Simulated Annealing [30–33]; and Fault simulation); Seismic, Reservoir Modeling, Weather Modeling, Pollution Propagation; Physics Applications (Band Structure, Spin-Lattice, Shell Model); Applied Physics and Chemistry Applications (Molecular Dynamics); Epidemic Spread; Computer Graphics and Image Processing (e.g., ray-tracing); Numerical and non-numeric Libraries (FFTs, Linear Solvers, Eigen-solvers, sorting), etc.

Advancing into the Future: Directions, Opportunities, Challenges, and Approaches

Emerging Computational Platforms and Emerging Applications Systems. Increasingly, large-scale distributed systems are deploying as their building blocks high-performance multicore processors (and in combination with special-purpose processor chips, like GPUs), as are the emerging petaflops and future hexaflops platforms. In fact, it is also conceivable that this kind of heterogeneity of processors will be eventually embedded in the multicore chip itself. Such systems, with 1,000s to 100s of thousands of tightly coupled nodes, will be enabled as Grids-in-a-Box (GiBs). Computational platforms include both high-end systems as well as globally-distributed, meta-computing, heterogeneous, networked and adaptive platforms, ranging from assemblies of networked workstations, to networked supercomputing clusters or combinations thereof, together with their associated peripherals such as storage and visualization systems. All these hardware platforms will have potentially not only multiple levels of processors, but also multiple levels of memory hierarchies (at the cache, main memory and storage

levels), and multiple levels of interconnecting networks, with multiple levels of latencies (variable at inter-node and intra-node levels) and bandwidths (differing for different links, differing based on traffic). Moving into the exascale domain, the challenges are augmented as we are faced with prospects of addressing billion-way concurrency, and in the presence of heterogeneity of processing units in a processing node, increasing degrees in the multiple levels of memory hierarchies, and multiple levels of interconnects hierarchies; these are the Grids-in-a-Box, referred to earlier-on, with significantly added complexity, because of the wider range of granularity, needing to expose concurrency from the fine grain to many more levels of coarser grain. The questions range from: how to express parallelism and optimally map and execute applications on such platforms, to how to enable load balancing, and at what level (or levels) is load balancing applied. It becomes evident that static parallelization approaches are inadequate, that one needs programming models that allow expressing parallelism so that it can be exploited dynamically, for load balancing and hiding latency, and for expressing dynamic flow control and synchronization possibly at multiple levels. The ideas of dynamic runtime compiler [34] capabilities discussed below are becoming more imperative.

Furthermore, new application paradigms (e.g., DDDAS – Dynamic Data Driven Applications Systems [35, 36]) that have emerged over the last several years and which entail dynamic on-line integration and feed-back and control between the computations of complex application models and measurement aspects of the application system, leading to *SuperGrids* [37] of integrated computational and instrumentation platforms, as well as other sensory devices and control platforms. In the context of this entry on programming environments and programming models, the implications are that these environments and models will need to support seamlessly execution of applications encompassing dynamically integrated high-end computing with real-time data-acquisition and control components and requirements.

New Programming Environments and Runtime-Support Technologies Such environments require technology approaches which break down traditional barriers in existing software components in the application development support and runtime layers, to deliver QoS in application execution. That is, together with new programming models, new compiler technology is needed, such as the runtime-compiler system (RCS [38]), where part of the compiler becomes embedded in the runtime and where the compiler interacts with the system monitoring and resource managers, as well as performance models of the underlying hardware and software, using such capabilities for optimizing the mapping of the application on the underlying complex platform(s). This runtime-compiler system is aware of the heterogeneity in the underlying architecture of the platforms, such as multi-level hierarchy of processing nodes, memories, and interconnects, with differing architecture, memory organization, and latencies, and will link to appropriately selected components (*dynamic application composition*) to generate consistent code at runtime. Representative examples of developing runtime-compiler capabilities are given in [39] and [40]. Together with the "runtime-compiler" capabilities [33], called for novel approaches and substantial enhancements in computational models to actualize the distributed applications software, including user-provided *assists* to facilitate and enhance the runtime-compiler's ability to analyze task and data dependencies in the application programs, resolve dependencies, and dynamically optimize mapping across a complex set of processing nodes and memory structure of distributed platforms (such as Grids and GiBs, multicore and GPU-based), with multiple levels of processors and processing nodes, interconnects, and memory hierarchies. It is the thesis of this entry that the models needed should facilitate the RCS (runtime compiling system) to map applications, without requiring detailed resource management specifications by the user and without requiring specification of data location (e.g., proximity – or PGAS). Rather the programming models should incorporate advanced concepts such as "*active data-distribution,*" that is the user specifies that these data are candidates for *active or runtime distribution*, and the RCS determines at runtime how to map them, determines partial sharing, copy/move between memory hierarchies, through dynamic adaptive resource management, decoupled execution and data location/placement, memory consistency models, and multithreaded hierarchical concurrency. Such capabilities may be materialized through a hybrid combination of language, library models, development of OS-supported "hierarchical threading" capabilities and partial sharing of data approaches.

In order to adequately support the future parallel systems, it is advocated here that the new software technologies need to adopt a more integrated view of the architectural layers and software components of a computing system (hardware/software co-design), consisting of the applications, the application support environments (languages, compilers, application libraries, linkers, run-time support, security, visualization, etc.), operating systems (scheduling, resource allocation and management, etc), computing platform architectures, processing nodes and network layers, and also support systems encompassing the computational and application measurement systems. Furthermore, such environments need to include approaches for engineering such systems, at the hardware, systems software, and at the applications levels, so that they execute with optimized efficiency with respect to runtime, quality of service, performance, power utilization, fault tolerance, and reliability. Such capabilities require robust software frameworks, encompassing the systems software layers and the application layers and cognizant of the underlying hardware platform resources.

Summary

This entry has provided an overview of the SPMD model, its origin and its initial implementations, its effectiveness and impact in expediting adoption of parallelism in the mid-80s, and its use as a vehicle for exploiting parallel and distributed computational platforms for the intervening 25 years. As we consider what are the possible new parallel programming models for new and future computing platforms, as well as present and emerging compute-, data-intensive, dynamically integrated applications executing on such platforms, it is becoming imperative to advance programming models and programming environments by building from past experiences and opening new directions through synergistic approaches of models, advanced runtime-compiler methods, user assists, multi-level threading OS services, and ensuring that the approaches and technologies developed are created in the context of end-to-end software architectural frameworks.

Acknowledgment

The IBM RP3 Project became the ground for my inspiration and work on the SPMD model; I'm forever grateful for being part of the RP3 team and will always value my collaborations with the RP3 team.

Bibliography

1. Darema-Rogers F (1984) IBM Internal Communication, Jan 1984
2. Darema-Rogers F, George D, Norton VA, Pfister G (1984) A VM parallel environment. Proceedings of the IBM Kingston Parallel Processing Symposium, 27–29 Nov 1984 (IBM Confidential)
3. Darema-Rogers F, George DA, Norton VA, Pfister GF (1985) A VM based parallel environment. IBM research report, RC11225
4. Darema-Rogers F, George DA, Norton VA, Pfister GF (1985) Environment and system interface for VM/EPEX. IBM research report R11381, 9/19/1985
5. Darema-Rogers F, George DA, Norton VA, Pfister GF (1985) Using a single-program-multiple-data model for parallel execution of scientific applications. SIAM Conference on Parallel Processing for Scientific Computing, November, 1985, and IBM research report R11552, 11/19/1985
6. Darema F, George DA, Norton VA, Pfister GF (1988) A single-program-multiple-data computational model for EPEX/-FORTRAN. Parallel Comput 7:11–24 (received April 1987 upon publication release by IBM - IBM Technical Disclosure Bulletin 29(9) February 1987)
7. Pfister G et al (1984) The research parallel processor prototype (RP3). Proceedings of the IBM Kingston Parallel Processing Symposium, 27–29 Nov 1984 (IBM Confidential); and Proceedings of the ICPP, August 1985
8. Darema F Historical and future perspectives on the SPMD computational model. – Forthcoming Publication
9. MPI standard – draft released by the MPI Forum. http://www.mcs.anl.gov/Projects/mpi/standard.html
10. PVM – Parallel Visrtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html
11. Foster I, Kesselman C (eds) (1999) The grid: blueprint for a new computing infrastructure. Morgan Kaufmann
12. OpenMP. http://openmp.org/wp/
13. Yelick KA, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger PN, Graham SL, Gay D, Colella P, Aiken A (1998) Concurrency: practice and experience. vol 10, No. 11–13, September-November. An earlier version was presented at the Workshop on Java for High-Performance Network Computing, Palo Alto, CA, Feb 1998
14. Culler DE, Arpaci-Dusseau AC, Goldstein SC, Krishnamurthy A, Lumetta S, Eicken T, Yelick KA (1993) Parallel programming in Split-C. SC, pp 262–273
15. El-Ghazawi T, Carlson W, Sterling T, Yelick KA (2005) UPC: distributed shared-memory programming. Wiley, Hoboken
16. Bronevetsky G, Marques D, Pingali K, Stodghill P (2000) Automated application-level checkpointing of MPI programs. ACM SIGPLAN 38(10):84–94
17. Bronevetsky G, Marques D, Pingali K, Szwed P, Schulz M (2004) Application-level checkpointing for shared memory programs. ACM Comp Ar 32(5):235–247
18. George DA MVS/XA EPEX – Environment for parallel execution. IBM research report RC 13158, 9/28/87
19. VM/System Product (VM/SP). http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3081.html (and MVS/XA also supported)

20. Stone JM, Darema-Rogers F, Norton VA, Pfister GF Introduction to the VM/EPEX FORTRAN Preprocessor. IBM research report RC 11407, 9/30/85
21. Stone JM, Darema-Rogers F, Norton VA, Pfister GF The VM/EPEX FORTRAN Preprocessor Reference. IBM research report RC 11408, 9/30/85
22. Bolmarcich T, Darema-Rogers F Tutorial for EPEX/FORTRAN Program Parallelization and Execution. IBM research report RC12515, 2/18/87
23. Whet-Ling C, Norton A (1986) VM/EPEX C preprocessor user's manual. Version 1.0. Technical report RC 12246, IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1986
24. Gottlieb A, Kruskal CP (1981) Coordinating parallel processors: a partial unification. Computer Architecture News, pp 16–24, October 1981
25. Darema-Rogers F I/O capabilities in the VM/EPEX system. IBM research report, RC 12219, 10/9/86
26. Darema F (1987) Applications environment for the IBM research parallel processor prototype (RP3). IBM research report RC 12627, 3/27/87; and Proceedings of the International Conference on Supercomputing (ICS), (published by Springer, Athens, Greece, June 1988
27. GPFS: http://www.almaden.ibm.com/StorageSystems/projects/gpfs
28. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. (pdf). Proceedings of the FAST'02 Conference on File and Storage Technologies. USENIX, Monterey, California, USA, pp 231–244. ISBN 1-880446-03-0. http://www.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf. Accessed 18 Jan 2008
29. ARC3D: Pulliam TH Euler and thin layer navier stokes codes: ARC2D, ARC3D. Computational fluid dynamics, A workshop held at the University of Tennessee Space Institute, UTSI publ. E02-4005-023-84, 1984 {other related application programs parallelized included: SIMPLE and HYDRO-1}
30. Darema-Rogers F, Kirkpatrick S, Norton VA (1987) Parallel techniques for chip placement by simulated annealing. Proceedings of the International Conference on Computer-Aided Design. pp 91–94
31. Darema F, Kirkpatrick S, Norton VA (1987) Simulated annealing on shared memory parallel systems. IBM Journal of R&D 31:391–402
32. Jayaraman R, Darema F Error analysis of parallel simulated annealing techniques. Proceedings of the ICCD'88, Rye, NY, 10/3-5/88
33. Greening D, Darema F Rectangular spatial decomposition methods for parallel simulated annealing. IBM research report, RC14636, 5/2/89, and in the Proceedings of the International Conference on Supercomputing '89. Crete-Greece
34. Darema F (2009) Report on cyberifrastructures of cyber-applications-systems & cyber-systems-software, submitted for external publication
35. DDDAS. www.cise.nsf.gov/dddas
36. DDDAS. www.dddas.org
37. Darema F (2005) Grid computing and beyond: the context of dynamic data driven applications systems. Proc IEEE (Special Issue on Grid Computing) 93(3):692–697
38. Darema F (2000) New software architecture for complex applications development and runtime support. Int J High-Perform Comput (Special Issue on Programming Environments, Clusters, and Computational Grids For Scientific Computing) 14(3)
39. GRADS project. http://www.hipersoft.rice.edu/grads/
40. Adve VS, Sanders WH A compiler-enabled model- and measurement-driven adaptation environment for dependability and performance. – http://www.perform.csl.illinois.edu/projects/newNSFNGS.html, LLVL compiler – http://llvlm.org

SSE

►AMD Opteron Processor Barcelona
►Intel Core Microarchitecture, x86 Processor Family
►Vector Extensions, Instruction-Set Architecture (ISA)

Stalemate

►Deadlocks

State Space Search

►Combinatorial Search

Stream Processing

►Stream Programming Languages

Stream Programming Languages

Ryan Newton
Intel Corporation, Hudson, MA, USA

Synonyms

Complex event processing; Event stream processing; Stream processing

Definition

Stream Programming Languages are specialized to the processing of data streams. A key feature of stream programming languages is that they structure programs not as a series of instructions, but as a graph of computation *kernels* that communicate via data streams. Stream programming languages are implicitly parallel and contain data, task, and pipeline parallelism. They offer some of the best examples of high-performance, fully automatic parallelization of implicitly parallel code.

Discussion

Overview

Stream processing is found in many areas of computer science. Accordingly, it has received distinct and somewhat divergent treatments. For example, most work on stream processing in the context of compilers, graphics, and computer architecture has focused on streams with predictable data-rates, such as in digital signal processing (DSP) applications. These can be modeled using *synchronous dataflow*, are deterministic (being a subclass of Kahn process networks), and are especially attractive as parallel programming models. Specialized languages for dataflow programming go at least as far back the SISAL (Streams and Iteration in a Single Assignment Language) in 1983. Somewhat different are *streaming databases* that are concerned with asynchronous (unpredictable) streams of timestamped events. Typical tasks include searching for temporal patterns and relating streams to each other and to data in stored tables, which may be termed "complex event processing."

This entry considers a broad definition of stream processing, requiring only that kernel functions and data streams be the dominant means of computation and communication. This includes both highly restrictive models – statically known, fixed graph and data-rates – and less restrictive ones. More restrictions allow for better performance and more automatic parallelism, whereas less restrictive models handle a broader class of applications.

Even this broad definition has necessarily blurry boundaries. For example, while stream-processing models typically allow only pure *message passing* communication, they can be extended to allow various forms of shared memory between kernel functions. General purpose programming languages equipped with libraries for stream processing are likely to fall into this category, as are streaming databases that include shared, modifiable tables.

Example Programming Models

This entry classifies stream programming designs along two major axes: *Graph Construction Method* and *Messaging Discipline*. The following three fictional programming models will be used to illustrate the design space:

- StreamStatic: a language with a statically known, fixed graph of kernel functions, as well as statically known, fixed data-rates on edges.
- StreamDynamic: a general purpose language that can manipulate streams as first-class values, constructing and destroying graphs on the fly and allowing arbitrary access to shared memory.
- StreamDB: Kernel graphs that are changed dynamically through transactions. Includes shared memory in the form of modifiable tables. More restricted and with different functionality than StreamDynamic.

Snippets of code corresponding to these three languages appear in Figs. 1–3, respectively. Kernels, being nothing more than functions, are usually defined in a manner resembling function definitions. In StreamDynamic the same language is used for graph construction as for kernel definition, whereas StreamStatic employs a distinct notation for graph topologies. Only in StreamDB are the definitions of kernels nonobvious. Whenever a new stream is defined from an

```
# metadata: pop 1 push 1
stream out AddAccum(stream in) {
  state { int s = 0; }
  execute() {
    s++;
    out.push(s + in.pop());
  }
}

// A static graph topology, in literal
   textual notation:
IntIn -> AddAccum -> IntOutput;
```

Stream Programming Languages. Fig. 1 A kernel function in StreamStatic might appear as in the above pseudocode, explicitly specifying the amount of data produced and consumed (pushed and popped) by the kernel during each invocation. This particular kernel carries a state that persists between invocations

```
-- Streams and tables coexist
CREATE TABLE Prices (
   Id     string PRIMARY KEY,
   Price  double);
CREATE INPUT STREAM PriceIncrs (
   Id        string,
   Increase  double);
-- A 'kernel' is constructed by building new streams from old
CREATE STREAM NormedPriceIncrs AS
   SELECT Id, Increase / 10.0
   FROM PriceIncrs;
-- Table modified based on streaming data:
UPDATE Prices USING NormedPriceIncrs
   SET Price = Prices.Price + NormedPriceIncrs.Increase,
   WHERE Prices.Id == NormedPriceIncrs.Id;
```

Stream Programming Languages. Fig. 2 A pseudocode example of StreamDB code

```
  // A function that creates $N$ copies of a kernel
  fun pipeline(int n, Function fn, Stream S) {
    if (n==0) return S;
    else pipeline(n-1, fn, map(fn,S));
  }

// Graph wiring is implicit in program execution:
pipeline(10, AddAccum, OrigStream)
```

Stream Programming Languages. Fig. 3 A pseudocode example of StreamDynamic code

old one with a `SELECT` statement, the expressions used in the `SELECT` statement form the body of a new kernel function. However, in streaming databases, much more functionality is built into the primitive operators, supporting, for example, a variety of windowing and grouping operations on streaming data.

Messaging StreamStatic assumes that each kernel specifies exactly how many inputs it consumes on each inbound edge, and outputs it produces on outbound edges. StreamDynamic instead uses asynchronous event streams and allows *nondeterministic merge*, which interleaves the elements of two streams in the real-time order in which they occur. StreamDB is similar to StreamDynamic, but timestamps all stream elements at their source, and then maintains a deterministic semantics with respect to those explicit timestamps.

Methods for Graph Wiring

Our three fictional languages use different mechanisms for constructing graphs. StreamStatic uses a literal textual encoding of the graph (Fig. 1). StreamDB also uses a direct encoding of a graph in the text of a query, but in the form of named streams with explicit dependencies (Fig. 2). In contrast, in StreamDynamic the graph is implicit, resulting from applying operators such as `map` to stream objects (Fig. 3).

There are other common graph construction methods not represented in these fictional languages: first, GUI-based construction of graphs, as found in LabView[12] and StreamBase[1]; second, an API for adding edges in a general purpose language, along the lines of "`connect(X,Y);`". Compared to implicit graph wiring via the manipulation of stream values (as seen in StreamDynamic or in real systems such as FlumeJava[2]), APIs of this kind are more explicit about edges and usually more verbose.

Finally, an advanced technique for constructing stream graphs is *metaprogramming*. Metaprogramming, or staged execution, simply refers to any program that generates another program. In the case of streaming, the most common situation is that a static kernel graph is desired, yet the programmer wishes to use loops and abstraction in the construction of (potentially complex) graphs. Therefore the need for

metaprogramming in streaming is analogous to hardware description languages, which also have a static target (logic gates) but need abstraction in the source language.

While it would be possible to write a program to generate the textual graph descriptions used by, for example, StreamStatic, a much more disciplined form of metaprogramming can be provided by stream-processing DSLs (domain specific languages), which can integrate the type-checking of the metalanguage and the target kernel language. Indeed, this is the approach taken by DSLs such as WaveScript[9] and to a lesser extent StreamIt[].

Parallelism in Stream Programs

Once constructed, a kernel graph exposes *pipeline*, *task*, and *data* parallelism. This terminology, used by the StreamIt authors [4] among others, distinguishes between producer/consumer parallelism (pipeline) and kernels lacking that relationship, such as the siblings in a fork-join pattern (task parallelism). Data parallelism in this context refers specifically to the ability to execute a kernel simultaneously on multiple elements in a single stream. Data parallelism is not always possible with a stateful kernel function – equivalent to a loop-carried dependence.

Stream programs expose abundant parallelism. It is the job of a scheduler or program optimizer to manage that parallelism and map it onto hardware. Fully automatic solutions to this problem remain difficult, but less difficult in a restrictive stream programming model than many other programming models. The key advantage is that stream programs define independent kernels with local data access and explicit, predictable communication.

This advantage enables stream programming models like StreamStatic to target a wide range of hardware architectures, including traditional cache-based CPUs (both multicore and vector parallelism), as well as GPUs, and architectures with software-controlled communication, such as the RAW tiled architecture [11] or the Cell processor. For example, the DSL StreamIt[4] (which subsumes StreamStatic) is an example of a programming language that has targeted all these platforms as well as networked clusters of workstations.

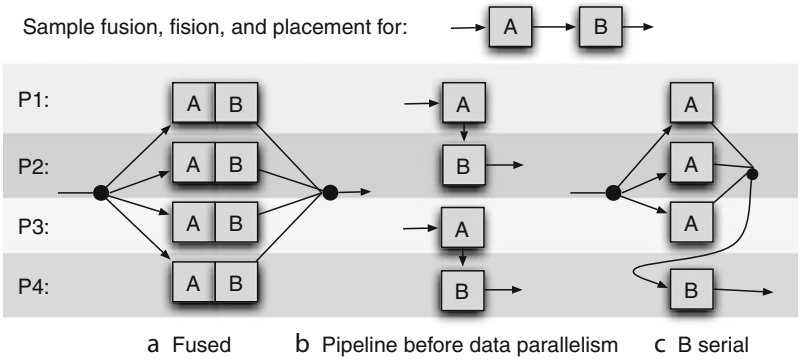
Optimization and Scheduling

An effective stream-processing implementation must accomplish three things: adjust granularity, choose between sources of parallelism, and place computations. Granularity here refers to how much data is processed in bulk, as well as whether or not kernels are combined (fused). Second, because stream programs include multiple kinds of parallelism, it is necessary to balance pipeline/task parallelism and data parallelism. Finally, placement and ordering presents a variant of the traditional task-scheduling problem.

Consider the two-kernel pipeline shown in Fig. 4. Even the simple composition of two kernels exposes several trade-offs. Three possible placements of kernels A and B onto four processors are shown in the figure. Placement (a) illustrates a common scenario: fusing kernels and leveraging data parallelism rather than pipeline parallelism. Yet choosing pipeline parallelism (placement (b)) could be preferable if, for example, A and B's combined working sets exceed the cache capacity of a processor. (Note that in a steady state streams provide enough data to keep A and B simultaneously executing – software pipelining.) Finally, placement (c) illustrates another common scenario – a kernel carries state and cannot be parallelized. In this case, kernel B must be executed serially, so it must not be fused with A or it would serialize A.

Note that there is a complicated interplay between batch size, kernel working set, and fusion/placement. One simple model for kernel working set assumes its size is linear in the size of its input x: that is, $mx + k$ where k represents memory that must be read irrespective of batch size. A large k might, for example, suggest the selection of placement (b) in Fig. 4 because if $k = 0$ batch size could be tuned to make placement (a) attractive. But a phase-ordering problem arises: Solving either batch size or fusion/fission independently (rather than simultaneously) can sacrifice opportunities to achieve optimal placement.

A related complication is that while the choices made by a stream-processing system could take place either statically or dynamically, granularity adjustment is more difficult to perform dynamically. For example, while dynamic work-stealing on a shared-memory computer can provide load balance and place kernels onto processors, it cannot deal with fine-grained kernels



Stream Programming Languages. Fig. 4 Sample placements of two kernels, A and B onto processors P1–P4

that perform only a few FLOPS per invocation. That said, as long as kernels are compiled to handle batches of data, the exact size of batches can be adjusted at runtime.

Program Transformation

If optimizations are performed statically by a compiler, they can take the form of source-to-source transformations. The most important optimizations fall into the following four categories:

- High-level / Algebraic – domain - specific optimizations, for example, canceling an FFT and an inverse FFT (see Haskell [8], WaveScript [9])
- Batch-processing / Execution Scaling – choosing batch size
- Fusion – combining and inlining kernels
- Fission / Data Parallelism – choosing N-ways to split a kernel

StreamStatic, for example, could follow the example of StreamIT and perform fusion, fission, and scaling as source-to-source program transformations, resulting in a number of kernels that match the target number of processors and leaving only a one-to-one placement problem to be solved by a simulated annealing optimization process. To achieve load balance during this process, StreamIt happens to use static work-estimation for kernels, but profiling is another option. In fact, given more dynamic data-rates, as in StreamDynamic and StreamDB, profile-based optimization and auto-tuning techniques are a good way to enable some of the above static program transformations. (WaveScript takes this approach [9].) To our knowledge no systems attempt to apply high-level / algebraic optimizations dynamically.

Algorithms

Dynamic systems along the lines of StreamDynamic tend to use work-stealing, as do most of today's popular task-scheduling frameworks. Further, many streaming databases use heuristics for kernel/operator migration to achieve load balance in a distributed setting. On the other hand, streaming models with fixed data-rates permit static scheduling strategies and much work has focused there. Scheduling predictable stream programs bears some resemblance to the well-studied problem of mapping a coarse-grained task-graph onto a multiprocessor (surveyed in [6]). But there is a notable difference: stream programs run continuously and are usually scheduled for throughput (with some exceptions [10]). Likewise, traditional graph-partitioning methods can be applied, but they do not, in general, capture all the degrees of freedom present in stream programs, whose graphs are not fixed, but are subject to transformations such as data-parallel fission of kernels.

An overview of scheduling work on stream processing specifically can be found in [13], focusing on embedded and real-time uses. The ideal would be an algorithm that simultaneously optimizes all aspects of a streaming program. A recent paper [5] proposed one such system that uses integer linear programming to simultaneously perform fission and processor placement for StreamIt programs.

Data Structures and Synchronization

Static and dynamic scheduling approaches employ different data structures. Dynamic approaches targeting shared memory multiprocessors are likely to rely on concurrent data structures including queues. In contrast, a completely static schedule typically requires no synchronization on data structures. Rather it may repeat a fixed schedule, using a global barrier synchronization at the end of each iteration of the schedule.

Relation to Other Parallel Programming Models

Functional Reactive Programming

Like StreamDynamic, FRP enables general purpose functional programs (usually in Haskell) to manipulate streams of values. FRP in particular deals not with discrete streams of elements, but with semantically continuous *signals*. Sampling of signals is performed by the runtime, or separately from the main definition of programs. Most FRP implementations are not high-performance, but parallel implementations and staged implementations that generate efficient code exist.

Relation to Data-parallel Models

With the increasing popularity of the *MapReduce* paradigm, there is a lot of interest in programming models for massively parallel manipulation of data. The FlumeJava library [2] (built on Google's MapReduce) bears a lot of resemblance to a stream-processing system. The user composes parallel operations on collections, and FlumeJava applies fusion transformations to pipelines of parallel operations. Similar systems like Cascade and Dryad explicitly construct dataflow graphs. These systems have a different application focus, target hardware, and scale than most of the work on stream processing. Further, they focus on batch processing of data in parallel, not on continuous execution.

Relation to Fork-Join Shared-Memory Parallelism

Fork-join parallelism in the form of parallel subroutine calls and parallel loops, as found in Cilk [7] and OpenMP [3], are attractive because of their relative ease of incorporation into legacy codes. In contrast, stream processing requires that a program be factored into kernels and that communication become explicit. Once a program is ported, however, it is perhaps easier to assure that it is correct and deterministic. Stream-processing languages provide a complete programming model encompassing computation and communication. In contrast, fork-join models treat the issues of control-flow and work-decomposition, but do not directly address data decomposition or communication.

Future Directions

This entry has described a family of programming models that have already accomplished a lot. Unfortunately, the systems described above have seen little application to industrial stream-processing problems. Most stream-processing codes are written in general purpose languages without any special support for stream processing. A major future challenge is to increase adoption of the body of techniques developed for stream processing.

Libraries, rather than stream-programming languages may have an advantage in this respect. Further, stream programming systems that are *wide spectrum* may prove desirable in the future – models that can reproduce the best results of more restrictive programming models, while also allowing graceful degradation into more general programming, therefore not confining the user strictly.

Related Entries

►Cell Processor

Bibliography

1. http://www.streambase.com/
2. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N (2010) Flumejava: easy, efficient data-parallel pipelines. In: PLDI '10: proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 363–375
3. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared memory programming. IEEE Comp Sci Eng 5(1):46–55
4. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGOPS Oper Syst Rev 40(5):151–162
5. Kudlur M, Mahlke S (2008) Orchestrating the execution of stream programs on multicore platforms. In: PLDI '08 proceedings of

the 2008 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 114–124
6. Kwok Y-K, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4):406–471
7. Leiserson CE (2009) The *cilk* + + concurrency platform. In: DAC '09 proceedings of the 46th annual design automation conference. ACM, New York, pp 522–527
8. Liu H, Cheng E, Hudak P (2009) Causal commutative arrows and their optimization. In: ICFP '09 proceedings of the 14th ACM SIGPLAN international conference on functional programming. ACM, New York, pp 35–46
9. Newton RR, Girod LD, Craig MB, Madden SR, Morrisett JG (2008) Design and evaluation of a compiler for embedded stream programs. In: LCTES '08 proceedings of the 2008 ACM SIGPLAN-SIGBED conference on languages, compilers, and tools for embedded systems. ACM, New York, pp 131–140
10. Pillai PS, Mummert LS, Schlosser SW, Sukthankar R, Helfrich CJ (2009) Slipstream: scalable low-latency interactive perception on streaming data. In: NOSSDAV '09 proceedings of the 18th international workshop on network and operating systems support for digital audio and video. ACM, New York, pp 43–48
11. Taylor MB, Lee W, Miller J, Wentzlaff D, Bratt I, Greenwald B, Hoffmann H, Johnson P, Kim J, Psota J, Saraf A, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A (2004) Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams. In: ISCA '04 proceedings of the 31st annual international symposium on Computer architecture. IEEE Computer Society, Washington, DC, p 2
12. Travis J, Kring J (2006) LabVIEW for everyone: graphical programming made easy and fun, 3rd edn. Prentice Hall, Upper Saddle River
13. Wiggers MH (2009) Aperiodic multiprocessor scheduling for real-time stream processing applications. Ph.D. thesis, University of Twente, Enschede, The Netherlands

Strong Scaling

►Amdahl's Law

Suffix Trees

Amol Ghoting, Konstantin Makarychev
IBM Thomas. J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Position tree

Definition

The suffix tree is a data structure that stores all the suffixes of a given string in a compact tree-based structure. Its design allows for a particularly fast implementation of many important string operations.

Discussion

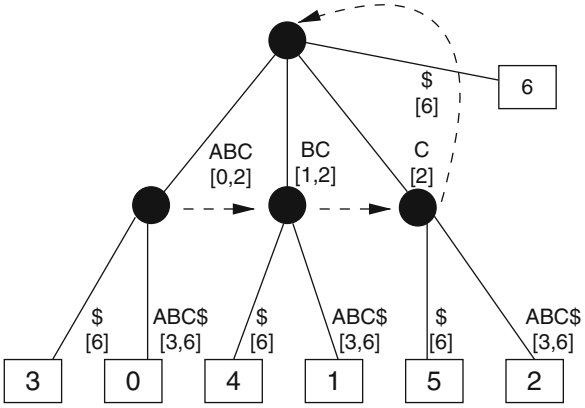
Introduction

The suffix tree is a fundamental data structure in string processing. It exposes the internal structure of a string in a way that facilitates the efficient implementation of a myriad of string operations. Examples of these operations include string matching (both exact and approximate), exact set matching, all-pairs suffix-prefix matching, finding repetitive structures, and finding the longest common substring across multiple strings [12].

Let A denote a set of characters. Let $S = s_0, s_1, \ldots, s_{n-1}, \$$, where $s_i \in A$ and $\$ \notin A$, denote a \$ terminated input string of length $n + 1$. The *ith* suffix of S is the substring $s_i, s_{i+1}, \ldots, s_{n-1}, \$$. The suffix tree for S, denoted as T, stores all the suffixes of S in a tree structure. The tree has the following properties:

1. Paths from the root node to the leaf nodes have a one-to-one relationship with the suffixes of S. The terminal character \$ is unique and ensures that no suffix is a proper prefix of any other suffix. Therefore, there are as many leaf nodes as there are suffixes.
2. Edges spell nonempty strings.
3. All internal nodes, except the root node, have at least two children. The edge for each child node begins with a character that is different from the starting character of its sibling nodes.
4. For an internal node v, let $l(v)$ denote the substring obtained by traversing the path from the root node to v. For every internal node v, with $l(v) = x\alpha$, where $x \in A$ and $\alpha \in A^*$, we have a pointer known as a suffix link to an internal node u such that $l(u) = \alpha$.

An instance of a suffix tree for a string $S = ABCABC\$$ is presented in Fig. 1. Each edge in a suffix tree is represented using the start and end indices of the corresponding substring in S. Therefore, even though a suffix tree represents n suffixes (each with at most n characters) for a total of $\Omega(n^2)$ characters, it only requires $O(n)$ space [31].



Suffix Trees. Fig. 1 Suffix tree for $S = ABCABC\$$ [0123456]. Internal nodes are represented using circles and leaf nodes are represented using rectangles. Each leaf node is labeled with the index of the suffix it represents. The *dashed arrows* represent the suffix links. Each edge is labeled with the substring it represents and its corresponding edge encoding

Applications

Over the past few decades, the suffix tree has been used for a spectrum of tasks ranging from data clustering [33] to data compression [3]. The quintessential usage of suffix trees is seen in the bioinformatics domain [2, 5, 6, 12, 17, 19, 24] where it is used to effectively evaluate queries on biological sequence data sets. A hallmark of suffix trees is that in many cases it allows one to process queries in time proportional to the size of the query rather than the size of string.

Suffix Tree Construction

Serial Suffix Tree Construction Algorithms

Algorithms due to Weiner [32], McCreight [22], and Ukkonen [31] have shown that suffix trees can be built in linear space and time. These algorithms afford a linear time construction by employing suffix links. Ukkonen's algorithm is more recent and popular because it is easier to implement than the other algorithms. It is an $O(n)$, in-memory construction algorithm. The algorithm is based on the simple, but elegant observation that the suffixes of a string $S_i = s_0, s_1, \ldots, s_i$ can be obtained from the suffixes of string $S_{i-1} = s_0, s_1, \ldots, s_{i-1}$ by catenating symbol s_i at the end of each suffix of S_{i-1} and by adding the empty suffix. The suffixes of the whole string $S = S_n = s_0, s_1, \ldots, s_n$ can then be obtained by first expanding the suffixes of S_0 into the suffixes of S_1 and so on, until the suffixes of S_n are obtained from the suffixes of S_{n-1}. This translates into a suffix tree construction that can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of suffix links, which provide a mechanism for quickly locating suffixes, the suffix tree can be expanded by simply adding the $(i + 1)$th character to the leaves of the suffix tree built on the previous i characters. The algorithm thus relies on suffix links to traverse through all of the sub-trees in the main tree, expanding the outer edges for each input character.

Parallel Suffix Tree Construction

Parallel suffix tree construction has been extensively studied in theoretical computer science. Apostolico et al. [1], Sühleyman et al. [28], and Hariharan [14] proposed theoretically efficient parallel algorithms for the problem based on the PRAM model. For example, Hariharan [14] showed how to execute McCreight's [22] algorithm concurrently on many processors. These algorithms, however, are designed for the case when the string and the suffix tree fit in main memory. Since the memory accesses of these algorithms exhibit poor locality of reference [8], these algorithms are inefficient when either the string or the suffix tree do not fit in main memory. On many real-world data sets, this is often the case.

The aforementioned problem has been theoretically resolved by Farach-Colton et al. [8]. Farach-Colton et al. [8] proposed a divide-and-conquer algorithm that operates as follows. First, construct a suffix tree for suffixes starting at odd positions. To do so, sort all pairs of characters at positions $2i - 1$ and $2i$ and replace each pair with its index in the sorted list. This gives us a string of length $\lceil n/2 \rceil$ with characters in a bigger alphabet. Recursively construct the suffix tree for this string. The obtained tree is essentially the suffix tree for suffixes starting at odd positions in the original string. Then, given the "odd" suffix tree construct an "even" suffix tree of suffixes starting at even positions. Finally, merge the trees. The details of the Farach-Colton et al. [8] algorithms are very complex, and there have not been any successful implementations. However, the doubling approach has been successfully used in practice for constructing suffix arrays

(see [4, 7], and section Suffix Arrays). While the authors do not explicitly provide a parallel algorithm, they indicate that the sort and merge phases do lend themselves to a parallel implementation. While the aforementioned algorithms provide theoretically optimal performance, for most algorithms, memory accesses exhibit poor locality of reference. As a consequence, these algorithms are grossly inefficient when either the tree or the string does not fit in main memory. To tackle this problem, this past decade has seen several research efforts that target large suffix tree construction. Algorithms that have been developed in these efforts can be placed in two categories: ones that require the input string to fit in main memory and ones that do not have this requirement. Many of these efforts only provide and evaluate serial algorithms for suffix tree construction. However, by design, as will be discussed, they do indeed lend themselves to a parallel implementation.

Practical Parallel Algorithms for In-Core Strings

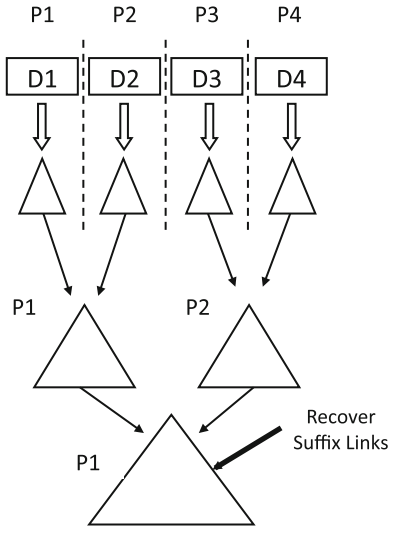
Hunt et al. [13] presented the very first approach to efficiently build suffix trees that do not fit in main memory. The approach drops the use of suffix links in favor of better locality of reference. Typically, the suffix tree is an order of magnitude larger than the string being indexed. As a result, for large input strings, the suffix tree cannot be accommodated in main memory. The method first finds a set of prefixes so as to partition the suffix tree into sub-trees (each prefix corresponds to a sub-tree) that can be built in main memory. The number of times a prefix occurs in a string is used to bound the size of the suffix sub-tree. The approach iteratively increases the size of the prefix until a length is reached where all the suffix sub-trees will fit in memory. Next, for each of these prefixes, the approach builds the associated suffix sub-tree using a scan of the data set. Essentially, for each suffix with the prefix, during insertion, one finds a path in the partially constructed suffix sub-tree that shares the longest common prefix (lcp) with this suffix, and branches from this path when no more matching characters are found. The worst case complexity of the approach is $O(n^2)$, but it exhibits $O(n \log n)$ average case complexity. This algorithm can be parallelized by distributing the prefixes to the different processors in a round-robin fashion and having each processor build one or more suffix sub-trees of the suffix tree. Kalyanaraman et al. [18] presented a parallel generalized suffix tree construction algorithm that in some ways builds upon Hunt's approach to realize a scalable parallel suffix tree construction. A generalized suffix tree is a suffix tree for not one but a group of strings. While the algorithm was presented in the context of the genome assembly problem, the proposed approach is not tied to genome assembly and can be applied else where. The approach first sorts all suffixes based on their w-length prefixes, where (like Hunt et al.'s approach) w is picked to ensure that the associated suffix sub-trees will fit in main memory. Suffixes with the same prefix are then assigned to the a single bucket. The buckets are then partitioned across the processors such that load is approximately balanced. Each processor then builds a sub-tree of the final suffix tree using a depth-first tree building approach. This is accomplished by sorting all the suffixes in the local bucket and then inserting them in sorted order. The end result in a distributed representation of the generalized suffix tree as a collection of sub-trees.

Japp introduced top-compressed suffix trees [15] that improves upon Hunt's approach by introducing a pre-processing stage to remove the need for repeated scans of the input sequence. Moreover, the author employed partitioning and optimized the insertion of suffixes into partitions using suffix links. The method is based on a new linear time construction algorithm for "sparse" suffix trees, which are sub-trees of the whole suffix tree. The new data structures are called the paged suffix tree (PST) and the distributed suffix tree (DST), respectively. Both tackle the memory bottleneck by constructing sub-trees of the full suffix tree independently and are designed for single processor and distributed memory parallel computing environments, respectively. The standard operations on suffix trees of biological importance are shown to be easily translatable to these new data structures.

Practical Parallel Algorithms for Out-of-Core Strings

The above mentioned parallel algorithms scale well when the input string fits in main memory. Many real-world input strings (like the human genome), however, do not fit in main memory. Researchers have developed a variety of algorithms to handle this situation. The earlier solutions for this problem follow what is known as

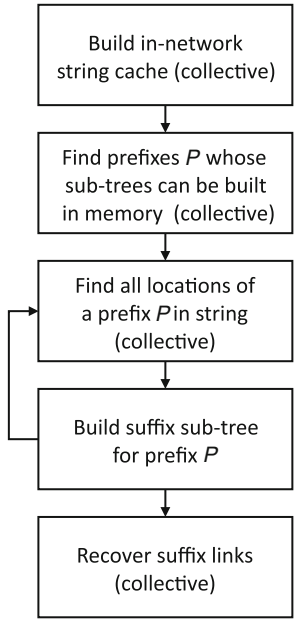
the partition-and-merge methodology. The approach is illustrated in Fig. 2. ST-MERGE [30], proposed by Tian et al. partitions the input string and constructs a suffix tree for each of these partitions in main memory. These suffix trees are then merged to create the final suffix tree. Merging two suffix trees involves finding matching suffixes in the two trees that share the longest common prefix, reusing this path in the new merged tree, and splitting this path when no more matching characters are found. Suffix link recovery can them be accomplished using a postprocessing step. The approach can be parallelized as follows. During the partition phase, each processor can be assigned block of the input string and it can build a suffix tree for the partition. During the merge phase, a processor can be assigned a pair of suffix trees where it is responsible for merging a pair of trees. One can realize a binary merge tree to build the final suffix tree. TRELLIS [26], due to Phoophakdee and Zaki, is similar in flavor but differs in the following regards. First, the approach finds a set of variable-length prefixes such that the corresponding suffix sub-trees will fit in main memory. Second, it partitions the input string and constructs a suffix tree for each partition in main memory (like ST-MERGE) and stores the sub-trees for each prefix determined in the first step, separately, on disk. Finally, it merges all the sub-trees associated with each prefix to realize the final set of suffix sub-trees. By



Suffix Trees. Fig. 2 Illustration of the partition and merge approach

design, TRELLIS ensures that each of the suffix sub-trees (built using the merge operation) will fit in main memory. TRELLIS can be parallelized using an approach similar to the parallelization of ST-MERGE.

Ghoting and Makarychev [10] studied the performance of the "partition-and-merge" approach for very large input strings and showed that the working set for these algorithms scales linearly with the size of the input string. This results in a lot of *random disk I/O* during the merge phase (the authors of TRELLIS do mention this as well [26]) when indexing strings that do not fit in main memory. One can even argue that for very large strings, the performance of the "partition-and-merge" way will converge to that of Hunt's approach [13], as merging sub-trees in tantamount to inserting suffixes into a partially constructed suffix tree. To address this challenge, Ghoting and Makarychev presented a serial algorithm WAVEFRONT and its parallelization P-WAVEFRONT [10, 11]. P-WAVEFRONT is the first parallel algorithm that can build a suffix tree for an input string that does not fit in main memory. P-WAVEFRONT diverges from the partition-and-merge methodology to suffix tree construction. Leveraging the structure of suffix trees, the algorithm builds a suffix tree by simultaneously tiling accesses to both the input string and the partially constructed suffix tree. Steps for P-WAVEFRONT are presented in Fig. 3. First, an in-network string cache is built by distributing the input strings across all processors in a round-robin fashion. The algorithm assumes that the input string can fit in collective main memory. This step uses collective I/O to ensure efficient reading of the input string. Next, like TRELLIS, a set of variable length prefixes are found in parallel such that the associated suffix sub-trees fit in memory. These prefixes are then distributed across all processors in a round-robin fashion. The following prefix location discovery phase is used to find the location of each prefix being processed using a collective procedure. During this step, each processor is responsible for finding locations of all prefixes (not just its own) in a partition of the input string and these are collective exchanged with other processors such that each processor has locations of its prefix in the entire input string. Finally, the suffix subtree for each prefix is built in a tiled and iterative manner by processing a pair of blocks of the input string at a time. The end result is an algorithm that can index very large input strings and at the same time maintain a



Suffix Trees. Fig. 3 Steps of P-wavefront

bounded working set size and a fixed memory footprint. The proposed methodology was applied to the suffix link recovery process as well, realizing an end-to-end I/O-efficient solution.

Suffix Arrays

Closely related to suffix trees is the suffix array [23]. Suffix array is the alphabetically sorted list of all suffixes of a given string *S*. For example, the suffix array of the string $S = ABCABC\$$ is as follows

Index	Suffix	lcp
6	$	0
3	ABC$	0
0	ABCABC$	3
4	BC$	0
1	BCABC$	2
5	C$	0
2	CABC$	1

The suffix array only contains pointers to the suffixes in the original string *S*. Thus, in the example above, the suffix array is the first column of the table, that is, the array $\{6, 3, 0, 4, 1, 5, 2\}$. Given a suffix tree one can obtain a suffix array in linear time by performing a depth-first search (DFS) (see Fig. 1). Suffix arrays are often used instead of suffix trees. The main advantage of suffix arrays is that they have a more compact representation in the memory. Besides the pointers to the suffixes, suffix arrays can also be augmented to contain lengths of the longest common prefixes (lcps) between adjacent strings (see the third column in the example above). In which case, given a suffix array it is easy to reconstruct the suffix tree. Essentially, the longest common prefix corresponds to the lowest common ancestor (lca) in the tree. Relative to suffix trees, the main drawback is that query processing using suffix arrays can be more time consuming as most queries are processed in time that is function of the size of the input string and not the size of the query (as was the case with suffix trees). Researchers have developed several linear time algorithms for directly building suffix arrays without pre-building a suffix tree [16, 21].

Futamura et al. [9] presented the first algorithm for parallel suffix array construction. The approach is similar to the one by Kalyanaraman et al. [18] that was described in section Practicel Parallel Algorithms for In-Core Strings. The approach first partitions the suffixes into buckets based on their *w*-length prefixes. These buckets are then distributed across the processors, where the suffixes in each bucket are sorted locally to obtain a portion of the suffix array. The final suffix array can then be realized by concatenating these distributed suffix arrays.

Algorithms for constructing suffix arrays in external memory have been extensively studied and compared by Crauser and Ferragina [4] and by Dementiev et al. [7]. Kärkkäinen et al. [21] proposed the DC3 algorithm, which is based on a similar approach as the Farach et al. [8]divide-and-conquer algorithm for suffix trees (see section Parallel Suffix Tree Construction). However, instead of dividing suffixes in the input string into suffixes starting at odd and even positions, it divides them in three groups depending on the remainder of the suffix position modulo 3. Surprisingly, this change significantly simplifies the algorithm. A variant of DC3, called pDC3, has been implemented by Kulla and Sanders [20]. Their study as well as the study of Dementiev et al. [7] indicate that DC3 and pDC3 are currently

the most efficient algorithms for constructing suffix arrays.

Related Entries

►Bioinformatics
►Genome Assembly

Bibliographic Notes and Further Reading

Research on parallel indexing for sequence/string data sets is still in its infancy. The primary reason for this is that until 2005, sequence data sets were not growing at a rapid pace and most problems could be handled in main memory. However, over the past few years, with advances in sequencing technologies, sequence databases have reached gigantic proportions. For instance, aggressive DNA sequencing efforts have resulted in the GenBank sequence database surpassing the 100 Gbp (one bp (base pair) is one character in the sequence) mark [25], with sequences from over 165,000 organisms. Further complicating the issue is the fact that researchers not only need the ability to index a single large genome, but a group of large genomes. For example, consider the area of comparative genomics [27] where one is interested in comparing different genomes, be they from the same or different species. Here researchers may be interested in comparing the genomes of individuals that are prone to a specific type of cancer to those that are not susceptible. In this case, we need to efficiently build a suffix tree for a group of large genomes, as and when needed. These trends suggest the parallel indexing technology for sequence data sets will be extremely important in the coming decade.

Much of this entry focused on parallel construction of suffix trees and suffix arrays. We would like to point the reader to Dan Gusfield's book [12] for a larger overview of applications. There has not been much work on parallelization of query processing using suffix trees and suffix arrays.

Another important direction is the design of parallel cache-conscious algorithms for multi-core processors so as to effectively utilize the memory hierarchy on modern processors. Tsirogiannis and Koudas looked at the problem of parallelizing the partition-and-merge approach on chip-multiprocessor architectures [29] and developed a cache-conscious algorithm for suffix tree construction. Such directions will become increasingly important in the near future given the tendency of packing many cores on a single processor.

Bibliography

1. Apostolico A, Iliopoulos C, Landau G, Schieber B, Vishkin U (1988) Parallel construction of a suffix tree with applications. Algorithmica 3(1–4):347–365
2. Bray N, Dubchak I, Pachter L (2003) AVID: a global alignment program. Genome research 13(1):97–102
3. Burrows M, Wheeler D (1994) A block sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation. Palo Alto, California
4. Crauser A, Ferragina P (2008) A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica 32(1):1–35
5. Delcher A, Kasif S, Fleischmann R, Peterson J, White O, Salzberg S (1999) Alignment of whole genomes. Nucleic Acids Res 27(11):2369–2376
6. Delcher A, Phillippy A, Carlton J, Salzberg S (2002) Fast algorithms for large-scale genome alignment and comparison. Nucleic Acids Res 30(1)
7. Dementiev R, Kärkkäinen J, Mehnert J, Sanders P (2008) Better external memory suffix array construction. J Exp Algorithmics (JEA) 12:3–4
8. Farach-Colton M, Ferragina P, Muthukrishnan S (2000) On the sorting-complexity of suffix tree construction. J ACM 47(6): 987–1011
9. Futamura N, Aluru S, Kurtz S (2001) Parallel suffix sorting. In: Proceedings 9th international conference on advanced computing and communications. Citeseer, pp 76–81
10. Ghoting A, Makarychev K (2009) Indexing genomic sequences on the IBM Blue Gene. In: SC '09: proceedings of the conference on high performance computing networking, storage and analysis. ACM, New York, pp 1–11
11. Ghoting A, Makarychev K (2009) Serial and parallel methods for I/O efficient suffix tree construction. In: Proceedings of the ACM international conference on management of data. ACM, New York
12. Gusfield D (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, Cambridge
13. Hariharan R (1994) Optimal parallel suffix tree construction. In: Proceedings of the symposium on theory of computing. ACM, New York
14. Hunt E, Atkinson M, Irving R (2001) A database index to large biological sequences. In: Proceedings of 27th international conference on very large databases. Morgan Kaufmann, San Francisco
15. Japp R (2004) The top-compressed suffix tree: a disk resident index for large sequences. In: Proceedings of the bioinformatics workshop at the 21st annual british national conference on databases

16. Kalyanaraman A, Emrich S, Schnable P, Aluru S (2007) Assembling genomes on largescale parallel computers. J Parallel Distr Comput 67(12):1240–1255
17. Kärkkäinen J, Sanders P, Burkhardt S (2006) Linear work suffix array construction. J ACM 53(6):918–936
18. Ko P, Aluru S (2005) Space efficient linear time construction of suffix arrays. J Discret Algorithms 3(2–4):143–156
19. Kulla F, Sanders P (2006) Scalable parallel suffix array construction. In: Recent advances in parallel virtual machine and message passing interface: 13th European PVM/MPI User's Group Meeting, Bonn, Germany, 17–20 September, 2006: proceedings. Springer, New York, p 22
20. Kurtz S, Choudhuri J, Ohlebusch E, Schleiermacher C, Stoye J, Giegerich R (2001) Reputer: the manifold applications of repeat analysis on a genome scale. Nucleic Acids Res 29(22):4633–4642
21. Kurtz S, Phillippy A, Delcher A, Smoot M, Shumway M, Antonescu C, Salzberg S (2004) Versatile and open software for comparing large genomes. Genome Bio 5:(R12)
22. Manber U, Myers G (1990) Suffix arrays: a new method for on-line string searches. In: Proceedings of the first annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 319–327
23. McCreight E (1976) A space-economical suffix tree construction algorithm. J ACM 23(2)
24. Meek C, Patel J, Kasetty S (2003) Oasis: an online and accurate technique for localalignment searches on biological sequences. In: Proceedings of 29th international conference on very large databases
25. NCBI. Public collections of DNA and RNA sequence reach 100 gigabases, 2005. http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html.
26. Phoophakdee B, Zaki M (2007) Genome-scale disk-based suffix tree indexing. In: Proceedings of the ACM international conference on management of data. ACM, New York
27. Rubin GM, Yandell MD, Wortman JR, Gabor Miklos GL, Nelson CR, Hariharan IK, Fortini ME, Li PW, Apweiler R, Fleischmann W, Cherry JM, Henikoff S, Skupski MP, Misra S, Ashburner M, Birney E, Boguski MS, Brody T, Brokstein P, Celniker SE, Chervitz SA, Coates D, Cravchik A, Gabrielian A, Galle RF, Gelbart WM, George RA, Goldstein LS, Gong F, Guan P, Harris NL, Hay BA, Hoskins RA, Li J, Li Z, Hynes RO, Jones SJ, Kuehl PM, Lemaitre B, Littleton JT, Morrison DK, Mungall C, O'Farrell PH, Pickeral OK, Shue C, Vosshall LB, Zhang J, Zhao Q, Zheng XH, Zhong F, Zhong W, Gibbs R, Venter JC, Adams MD, Lewis S (2000) Comparative genomics of the eukaryotes. Science 287(5461):2204–2215
28. Sahinalp SC, Vishkin U (1994) Symmetry breaking for suffix tree construction. In: STOC '94: proceedings of the twenty-sixth annual ACM symposium on Theory of computing ACM, New York, pp 300–309
29. Tian Y, Tata S, Hankins R, Patel J (2005) Practical methods for constructing suffix trees. VLDB J 14(3):281–299
30. Tsirogiannis D, Koudas N (2010) Suffix tree construction algorithms on modern hardware. In: EDBT '10: Proceedings of the 13th international conference on extending database Technology. ACM, New York, pp 263–274
31. Ukkonen E (1992) Constructing suffix trees on-line in linear time. In: Proceedings of the IFIP 12th work computer congress on algorithms, software, architecture: information processing. North Holland Publishing Co., Amsterdam
32. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of 14th annual symposium on switch and automata theory. IEEE Computer Society, Washington, DC
33. Zamir O, Etzioni O (1998) Web document clustering: a feasibility demonstration. In: Proceedings of 21st international conference on research and development in information retrieval. ACM, New York

Superlinear Speedup

►Metrics

SuperLU

Xiaoye Sherry Li[1], James Demmel[2], John Gilbert[3], Laura Grigori[4], Meiyue Shao[5]
[1]Lawarence Berkeley National Laboratory, Berkeley, CA, USA
[2]University of California at Berkeley, Berkeley, CA, USA
[3]University of California, Santa Barbara, CA, USA
[4]Laboratoire de Recherche en Informatique Universite Paris-Sud 11, Paris, France
[5]Umeå University, Umeå, Sweden

Synonyms

Sparse gaussian elimination

Definition

SuperLU is a general-purpose library for the solution of large, sparse, nonsymmetric systems of linear equations using direct methods. The routines perform LU decomposition with numerical pivoting and solve the triangular systems through forward and back substitution. Iterative refinement routines are provided for improved backward stability. Routines are also provided to equilibrate the system, to reorder the columns to preserve sparsity of the factored matrices, to estimate the condition number, to calculate the relative backward error, to estimate error bounds for the refined solutions,

and to perform threshold-based incomplete LU factorization (ILU), which can be used as a preconditioner for iterative solvers. The algorithms are carefully designed and implemented so that they achieve excellent performance on modern high-performance machines, including shared-memory and distributed-memory multiprocessors.

Discussion

Introduction

SuperLU consists of a collection of three related ANSI C subroutine libraries for solving sparse linear systems of equations $AX = B$. Here A is a square, nonsingular, $n \times n$ sparse matrix, and X and B are dense $n \times nrhs$ matrices, where *nrhs* is the number of right-hand sides and solution vectors. The LU factorization routines can handle non-square matrices. Matrix A need not be symmetric or definite; indeed, SuperLU is particularly appropriate for matrices with very unsymmetric structure. All three libraries use variations of Gaussian elimination optimized to take advantage of both sparsity and the computer architecture, in particular memory hierarchies (caches) and parallelism [5, 17]. All three libraries can be obtained from the following URL:

`http://crd.lbl.gov/~xiaoye/SuperLU/`

The three libraries within SuperLU are as follows:

- *Sequential SuperLU* (`SuperLU`) is designed for sequential processors with one or more levels of caches [3]. Routines for both complete and threshold-based incomplete LU factorizations are provided [20].
- *Multithreaded SuperLU* (`SuperLU_MT`) is designed for shared-memory multiprocessors, such as multicore, and can effectively use 16–32 parallel processors on sufficiently large matrices in order to speed up the computation [4].
- *Distributed SuperLU* (`SuperLU_DIST`) is designed for distributed-memory parallel machines, using MPI for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices [19].

Table 1 summarizes the current status of the software. All the routines are implemented in C, with parallel extensions using Pthreads or OpenMP for shared-memory programming, or MPI for distributed-memory programming. Fortran interfaces are provided in all three libraries.

SuperLU. Table 1 SuperLU software status

	Sequential SuperLU	SuperLU_MT	SuperLU_DIST
Platform	Serial	Shared-memory	distributed-memory
Language (with Fortran interface)	C	C + Pthreads or OpenMP	C + MPI
Data type	Real/complex Single/double	Real/complex Single/double	Real/comples Double

Overall Algorithm

The kernel algorithm in SuperLU is sparse Gaussian elimination, which can be summarized as follows:

1. Compute a *triangular factorization* $P_r D_r A D_c P_c = LU$. Here D_r and D_c are diagonal matrices to equilibrate the system, and P_r and P_c are *permutation matrices*. Premultiplying A by P_r reorders the rows of A, and postmultiplying A by P_c reorders the columns of A. P_r and P_c are chosen to enhance sparsity, numerical stability, and parallelism. L is a unit lower triangular matrix ($L_{ii} = 1$) and U is an upper triangular matrix. The factorization can also be applied to non-square matrices.
2. Solve $AX=B$ by evaluating $X=A^{-1}B=(D_r^{-1}P_r^{-1}LUP_c^{-1}D_c^{-1})^{-1}B=D_c\left(P_c(U^{-1}(L^{-1}(P_r(D_rB))))\right)$. This is done efficiently by multiplying from right to left in the last expression: Scale the rows of B by D_r. Multiplying P_rB means permuting the rows of D_rB. Multiplying $L^{-1}(P_rD_rB)$ means solving *nrhs* triangular systems of equations with matrix L by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_rD_rB))$ means solving triangular systems with U.

In addition to exact, or complete, factorization, SuperLU also contains routines to perform incomplete factorization (ILU), in which sparser and so cheaper approximations to L and U are computed, which can be

used as a general-purpose preconditioner in an iterative solver.

The simplest implementation, used by the *simple driver* routines in SuperLU and SuperLU_MT, consists of the following steps:

Simple Driver (assumes $D_r = D_c = I$)

1. *Choose P_c to order the columns of A* to increase the sparsity of the computed L and U factors, and hopefully increase parallelism (for SuperLU_MT). Built-in choices are described later.
2. *Compute the LU factorization of AP_c*. SuperLU and SuperLU_MT can perform dynamic pivoting with row interchanges during factorization for numerical stability, computing P_r, L, and U at the same time.
3. *Solve the system* using P_r, P_c, L, and U as described above (where $D_r = D_c = I$).

The simple driver subroutines for double precision real data are called `dgssv` and `pdgssv` for `SuperLU` and `SuperLU_MT`, respectively. The letter `d` in the subroutine names means double precision real; other options are `s` for single precision real, `c` for single precision complex, and `z` for double precision complex. `SuperLU_DIST` does not include this simple driver.

There is also an *expert driver* subroutine that can provide more accurate solutions, compute error bounds, and solve a sequence of related linear systems more economically. It is available in all three libraries.

Expert Driver

1. *Equilibrate* the matrix A, that is, compute diagonal matrices D_r and D_c so that $\hat{A} = D_r A D_c$ is "better conditioned" than A, that is, $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than A^{-1} is to perturbations in A.
2. *Preorder the rows of $\hat{A}$ (`SuperLU_DIST` only)*, that is, replace $\hat{A}$ by $P_r\hat{A}$ where P_r is a permutation matrix. This step is called "static pivoting," and is only done in the distributed-memory algorithm, which allows scaling to more processors.
3. *Order the columns of $\hat{A}$*, to increase the sparsity of the computed L and U factors, and increase parallelism (for `SuperLU_MT` and `SuperLU_DIST`). In other words, replace $\hat{A}$ by $\hat{A}P_c^T$ in `SuperLU` and `SuperLU_MT`, or replace $\hat{A}$ by $P_c\hat{A}P_c^T$ in SuperLU_DIST, where P_c is a permutation matrix.
4. *Compute the LU factorization of $\hat{A}$*. `SuperLU` and `SuperLU_MT` can perform dynamic pivoting with row interchanges for numerical stability. In contrast, `SuperLU_DIST` uses the order computed by the preordering step (Step 2), and replaces tiny pivots by larger values for stability; this is corrected by Step 6.
5. *Solve the system* using the computed triangular factors.
6. *Iteratively refine the solution*, again using the computed triangular factors. This is equivalent to Newton's method.
7. *Compute error bounds*. Both forward and backward error bounds are computed.

The expert driver subroutines for double precision real data are called `dgssvx`, `pdgssvx`, and `pdgssvx` for `SuperLU`, `SuperLU_MT`, and `SuperLU_DIST`, respectively.

The driver routines are composed of several lower level computational routines for computing permutations, computing LU factorization, solving triangular systems, and so on. For large matrices, the LU factorization step takes most of the time, although choosing P_c to order the columns can also be time consuming.

Common Features of the Three Libraries

Supernodes in the Factors

The factorization algorithms in all three libraries use unsymmetric supernodes [3], which enable the use of higher level BLAS routines with higher flops-to-byte ratios, and so higher speed. A supernode is a range $(r : s)$ of columns of L with the triangular block just below the diagonal being full, and the same nonzero structure below the triangular block. Matrix U is partitioned rowwise by the same supernodal boundaries. But due to the lack of symmetry, the nonzero pattern of U consists of dense column segments of different lengths.

Sparse Matrix Data Structure

The principal data structure for a matrix is `SuperMatrix`, defined as a C structure. This structure contains two levels of fields. The first level defines the three storage-independent properties of a matrix: mathematical type, data type, and storage type. The second level points to the actual storage used to store the compressed matrix. Specifically, matrix A is stored

in either column-compressed format (aka Harwell–Boeing format), or row-compressed format (i.e., A^T stored in column-compressed format) [1]. Matrices B and X are stored as a single dense matrix of dimension $n \times nrhs$ in column-major order, with output X overwriting input B. In `SuperLU_DIST`, A and B can be either replicated or distributed across all processes. The factored matrices L and U are stored differently in `SuperLU/SuperLU_MT` and `SuperLU_DIST`, to be described later.

`Options` Input Argument

The `options` is an input argument to control the behaviour of the libraries. The user can tell the solvers how the linear systems should be solved based on some known characteristics of the system. For example, for diagonally dominant matrices, choosing the diagonal pivots ensures stability; there is no need for numerical pivoting (i.e., P_r can be an identity matrix). In another situation where a sequence of matrices with the same sparsity pattern needs to be factorized, the column permutation P_c (and also the row permutation P_r, if the numerical values are similar) needs to be computed only once, and reused thereafter. In these cases, the solvers' performance can be much improved over using the default parameter settings.

Performance-Tuning Parameters

All three libraries depend on having an optimized BLAS library to achieve high performance [7, 8]. In particular, they depend on matrix–vector multiplication or matrix–matrix multiplication of relatively small dense matrices arising from the supernodal structure. The block size of these small dense matrices can be tuned to match the "sweet spot" of the BLAS performance on the underlying architecture. These parameters can be altered in the inquiry routine `sp_ienv`.

Example Programs

In the source code distribution, the `EXAMPLE/` directory contains several examples of how to use the driver routines, illustrating the following usages:

- Solve a system once
- Solve different systems with the same A, but different right-hand sides
- Solve different systems with the same sparsity pattern of A
- Solve different systems with the same sparsity pattern and similar numerical values of A

Except for the case of one-time solution, all the other examples can reuse some of the data structures obtained from a previous factorization, hence, save some time compared to factorizing A from scratch. The users can easily modify these examples to fit their needs.

Differences Between `SuperLU/` `SuperLU_MT` and `SuperLU_DIST`

Numerical Pivoting

Both sequential `SuperLU` and `SuperLU_MT` use *partial pivoting with diagonal threshold*. The row permutation P_r is determined dynamically during factorization. At the jth column, let a_{mj} be a largest entry in magnitude on or below the diagonal of the partially factored A: $|a_{mj}| = \max_{i \geq j} |a_{ij}|$. Depending on a threshold u ($0.0 \leq u \leq 1.0$) selected by the user, the code may use the diagonal entry a_{jj} as the pivot in column j as long as $|a_{jj}| \geq u\,|a_{mj}|$ and $a_{jj} \neq 0$, or else use a_{mj}. If the user sets $u = 1.0$, a_{mj} (or an equally large entry) will be used as the pivot; this corresponds to the classical partial pivoting. If the user has ordered the matrix so that choosing diagonal pivots is particularly good for sparsity or parallelism, then smaller values of u tend to choose those diagonal pivots, at the risk of less numerical stability. Selecting $u = 0.0$ guarantees that the pivot on the diagonal will be chosen, unless it is zero. The code can also use a user-input P_r to choose pivots, as long as each pivot satisfies the threshold for each column. The backward error bound *BERR* measures how much stability is actually lost.

It is hard to get satisfactory execution speed with partial pivoting on distributed-memory machines, because of the fine-grained communication and the dynamic data structures required. `SuperLU_DIST` uses a *static pivoting* strategy, in which P_r is chosen before factorization and based solely on the values of the original A, and remains fixed during factorization. A maximum weighted matching algorithm and the code `MC64` developed by Duff and Koster [6] is currently employed. The algorithm chooses P_r to maximize the product of the diagonal entries, and chooses D_r and D_c simultaneously so that each diagonal entry of $P_r D_r A D_c$ is ± 1 and each off-diagonal entry is bounded by 1 in magnitude. On the basis of empirical evidence, when

this strategy is combined with diagonal scaling, setting very tiny pivots to larger values, and iterative refinement, the algorithm is as stable as partial pivoting for most matrices that have occurred in the actual applications. The detailed numerical experiments can be found in [19].

Sparsity-Preserving Reordering

For unsymmetric factorizations, preordering for sparsity is less understood than that for Cholesky factorization. Many unsymmetric ordering methods use symmetric ordering techniques, either minimum degree or nested dissection, applied to a symmetrized matrix (e.g., A^TA or $A^T + A$). This attempts to minimize certain upper bounds on the actual fills. Which symmetrized matrix to use strongly depends on how the numerical pivoting is performed.

In sequential `SuperLU` and `SuperLU_MT`, where partial pivoting is used, an A^TA–based ordering algorithm is preferrable. This is because the nonzero pattern of the Cholesky factor R in $A^TA = R^TR$ is a superset of the nonzero pattern of the L^T and U in $P_rA = LU$, for any with row interchanges [9]. Therefore, a good symmetric ordering P_c on A^TA that preserves the sparsity of R can be applied to the columns of A, forming AP_c^T, so that the LU factorization of AP_c^T is likely to be sparser than that of the original A.

In `SuperLU_DIST`, an a priori row permutation P_r is computed to form P_rA. With fixed P_r, an $(A^T + A)$-based ordering algorithm is preferrable. This is because the symbolic Cholesky factor of $A^T + A$ is a much tighter upper bound on the structures of L and U than that of A^TA when the pivots are chosen on the diagonal. Note that after P_c is chosen, a symmetric permutation $P_c(P_rA)P_c^T$ is performed so that the diagonal entries of the permuted matrix remain the same as those in P_rA, and they are larger in magnitude than the off-diagonal entries. Now the final row permutation is P_cP_r.

In all three libraries, the user can choose one of the following ordering methods by setting the `options.ColPerm` option:

- NATURAL: Natural ordering
- MMD_ATA: Multiple Minimum Degree [21] applied to the structure of A^TA
- MMD_AT_PLUS_A: Multiple Minimum Degree applied to the structure of $A^T + A$
- METIS_ATA: `MeTiS` [14] applied to the structure of A^TA
- METIS_AT_PLUS_A: `MeTiS` applied to the structure of $A^T + A$
- PARMETIS: `ParMeTiS` [15] applied to the structure of $A^T + A$
- COLAMD: Column Approximate Minimum Degree [2]
- MY_PERMC: Use a permutation P_c supplied by the user as input

COLAMD is designed particularly for unsymmetric matrices when partial pivoting is needed, and does not require explicitly forming A^TA. It usually gives comparable orderings as MMD on A^TA, and is faster.

The purpose of the last option MY_PERMC is to be able to reap the results of active research in the ordering methods. Recently, there is much research on the orderings based on graph partitioning. The user can invoke those ordering algorithms separately, and then input the ordering in the permutation vector for P_c. The user may apply them to the structures of A^TA or $A^T + A$. The routines `getata()` and `at_plus_a()` in the file `get_perm_c.c` can be used to form A^TA or $A^T + A$.

Task Ordering

The Gaussian elimination algorithm can be organized in different ways, such as left-looking (fan-in) or right-looking (fan-out). These variants are mathematically equivalent under the assumption that the floating-point operations are associative (approximately true), but they have very different memory access and communication patterns. The pseudo-code for the left-looking blocking algorithm is given in Algorithm 1.

Algorithm 1 *Left-looking Gaussian elimination*

for *block* $K = 1$ **to** N **do**
(1) *Compute* $U(1:K-1, K)$
(*via a sequence of triangular solves*)
(2) *Update* $A(K:N, K) \leftarrow$ A(K:N, K)
$-L(1:N, 1:K-1) \cdot U(1:K-1, K)$
(*via a sequence of calls to GEMM*)
(3) *Factorize* $A(K:N, K) \rightarrow L(K:N, K)$
(*may involve pivoting*)
end for

SuperLU and SuperLU_MT use the left-looking algorithm, which has the following advantages:

- In each step, the sparsity changes are restricted within the Kth block column instead of the entire trailing submatrix, which makes it relatively easy to accommodate dynamic compressed data structures due to partial pivoting.
- There are more memory *read* operations than *write* operations in Algorithm 1. This is better for most modern cache-based computer architectures, because write tends to be more expensive in order to maintain cache coherence.

The pseudo-code for the right-looking blocking algorithm is given in Algorithm 2.

Algorithm 2 *Right-looking Gaussian elimination*

for *block* $K = 1$ **to** N **do**

(1) *Factorize* $A(K:N,K) \rightarrow L(K:N,K)$
(*may involve pivoting*)

(2) *Compute* $U(K,K+1:N)$
(*via a sequence of triangular solves*)

(3) *Update* $A(K+1:N,K+1:N) \leftarrow$
$A(K+1:N,K+1:N) - L(K+1:N,K)\cdot$
$U(K,K+1:N)$
(*via a sequence of calls to GEMM*)

end for

SuperLU_DIST uses right-looking algorithm mainly for scalability consideration.

- The sparsity pattern and data structure can be determined before numerical factorization because of static pivoting.
- The right-looking algorithm fundamentally has more parallelism: at step (3) of Algorithm 2, all the GEMM updates to the trailing submatrix are independent and so can be done in parallel. On the other hand, each step of the left-looking algorithm involves operations that need to be carefully sequenced, which requires a sophisticated pipelining mechanism to exploit parallelism across multiple loop steps.

Parallelization and Performance

SuperLU_MT is designed for parallel machines with shared address space, thus there is no need to partition the matrices. Matrices A, L, and U are stored in separate compressed formats. The parallel elimination uses an asychronous and barrier-free scheduling algorithm to schedule two types of parallel tasks to achieve a high degree of concurrency. One such task is factorizing the independent panels in the disjoint subtrees of the column elimination tree [10]. Another task is updating a panel by previously computed supernodes. The scheduler facilitates the smooth transition between the two types of tasks, and maintains load balance dynamically. In symbolic factorization, a non-blocking algorithm is used to perform depth-first search and symmetric pruning in parallel. The code achieved over tenfold speedups on a number of earlier SMP machines with 16 processors [4]. Recent evaluation shows that SuperLU_MT performs very well on current multithreaded, multicore machines; it achieved over 20-fold speedup on a 16 core, 128 thread Sun VictoriaFalls [16].

The design of SuperLU_DIST is drastically different from SuperLU/SuperLU_MT. Many design choices were made by the need for scaling to a large process count. The input sparse matrix A is divided by block rows, with each process having one block row represented in a local row-compressed format. This format is user-friendly and is compatible with the input interface of much other distributed-memory sparse matrix software. The factored L and U matrices, on the other hand, are distributed by a two-dimensional block cyclic layout using supernodal structure for block partition. This distribution ensures that most (if not all) processors are engaged in the right-looking update at each block elimination step, and also ensures that interprocess communication is restricted among process row set or column set. The right-looking factorizations use elimination DAGs to identify task and data dependencies, and a one step look-ahead scheme to overlap communication with computation. A distributed parallel symbolic factorization algorithm is designed so that there is no need to gather the entire graph on a single node, which largely increases memory scalability [12]. SuperLU_DIST has achieved 50- to 100-fold speedups with sufficiently large matrices, and over half a Teraflops factorization rate [18].

Future Directions

There are still many open problems in the development of high performance algorithm and software for sparse direct methods. The current architecture trend shows that the Chip Multiprocessor (CMP) will be the basic building block for computer systems ranging from laptops to extreme high-end supercomputers. The core count per chip will quickly increase from four to eight today to tens in the near future. Given the limited per-chip memory and memory bandwidth, the standard parallelization procedure based on MPI would suffer from serious resource contention. It becomes essential to consider a hybrid model of parallelism at the algorithm level as well as the programming level. The quantitative multicore evaluation of SuperLU shows that the left-looking algorithm in `SuperLU_MT` consistently outperforms the right-looking algorithm in `SuperLU_DIST` on a single node of the recent CMP systems, mainly because the former incurs much less memory traffic [16]. One new design is to combine the two algorithms – partitioning the matrix into larger panels, performing left-looking intra-chip elimination and right-looking inter-chip elimination.

A second new direction is to exploit the property of low numerical rank in many discretized operators so that a specialized Gaussian elimination algorithm can be designed. For example, it has been shown recently that *semi-separable structures* occur in the pivoting block and the corresponding Schur complement of each block factorization step. Thus, using the compressed, semi-separable representation throughout the entire factorization leads to an approximate factorization algorithm that has nearly linear time and space complexity [13, 22]. For many discretized elliptic PDEs, this approximation is sufficiently accurate and can be used as an optimal direct solver. For more general problems, the factoriation can be used as an effective preconditioner. Because of its asymptotically lower data volume compared with conventional algorithms, the amount of memory-to-processor and inter-processor communication is smaller, making it more amenable to a scalable implementation.

Another promising area is to extend the new communication avoiding dense LU and QR factorizations to sparse factorizations [11]. In conventional algorithms, the panel factorization of a tall-skinny submatrix requires a sequence of fine-grained message transfers, and often lies on the critical path of parallel execution. The new method employs a divide-and-conquer scheme for this phase, which has asymptotically less communication. This method should also work for sparse matrices. In particular, the new pivoting strategy for LU can replace the static pivoting currently used in `SuperLU`, leading to a more stable and scalable solver.

Related Entries

►Chaco
►LAPACK
►METIS and ParMETIS
►Mumps
►PARDISO
►PETSc (Portable, Extensible Toolkit for Scientific Computation)
►Preconditioners for Sparse Iterative Methods
►Reordering
►ScaLAPACK
►Sparse Direct Methods

Bibliography

1. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, van der Vorst H (1994) Templates for the solution of linear systems: building blocks for the iterative methods. SIAM, Philadelphia, PA
2. Davis TA, Gilbert JR, Larimore S, Ng E (2004) A column approximate minimum degree ordering algorithm. ACM Trans Math Softw 30(3):353–376
3. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH (1999) A supernodal approach to sparse partial pivoting. SIAM J Matrix Anal Appl 20(3):720–755
4. Demmel JW, Gilbert JR, Li XS (1999) An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J Matrix Anal Appl 20(4):915–952
5. Demmel JW, Gilbert JR, Li XS (1999) SuperLU users' guide. technical report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. http://crd.lbl.gov/~xiaoye/SuperLU/. Last update: September 2007
6. Duff IS, Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J Matrix Anal Appl 20(4):889–901
7. BLAS Technical Forum (2002) Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard I. Int J High Perform Comput Appl 16:1–111

8. BLAS Technical Forum (2002) Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard II. Int J High Perform Comput Appl 16:115–199
9. George A, Liu J, Ng E (1988) A data structure for sparse QR and LU factorizations. SIAM J Sci Stat Comput 9:100–121
10. Gilbert JR, Ng E (1993) Predicting structure in nonsymmetric sparse matrix factorizations. In: George A, Gilbert JR, Liu JWH (eds) Graph theory and sparse matrix computation. Springer-Verlag, New York, pp 107–139
11. Grigori L, Demmel J, Xiang H (2008) Communication-avoiding Gaussian elimination. In: Supercomputing 08, Austin, TX, November 15–21, 2008.
12. Grigori L, Demmel JW, Li XS (2007) Parallel symbolic factorization for sparse LU with static pivoting. SIAM J Sci Comput 29(3):1289–1314
13. Gu M, Li XS, Vassilevski P (2010) Direction-preserving and schur-monotonic semi-separable approximations of symmetric positive definite matrices. SIAM J Matrix Anal Appl 31(5):2650–2664
14. Karypis G, Kumar V (1998) MeTiS – a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices – version 4.0. University of Minnesota, September 1998. http://www-users.cs.umn.edu/ karypis/metis/. Accessed 2010
15. Karypis G, Schloegel K, Kumar V (2003) ParMeTiS: Parallel graph partitioning and sparse matrix ordering library – version 3.1. University of Minnesota. http://www- users.cs.umn.edu/~karypis/metis/parmetis/. Accessed 2010
16. Li XS (2008) Evaluation of sparse factorization and triangular solution on multicore architectures. In: Proceedings of VECPAR08 8th international meeting high performance computing for computational science, Toulouse, France, June 24–27, 2008
17. Li XS (Sept 2005) An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans Math Softw 31(3):302–325
18. Li XS (2009) Sparse direct methods on high performance computers. University of California, Berkeley, CS267 Lecture Notes
19. Li XS, Demmel JW (June 2003) SuperLU DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans Math Softw 29(2):110–140
20. Li XS, Shao M (2011) A supernodal approach to imcomplete LU factorization with partial pivoting. ACM Trans Math Softw 37(4)
21. Liu JWH (1985) Modification of the minimum degree algorithm by multiple elimination. ACM Trans Math Softw 11:141–153
22. Xia J, Chandrasekaran S, Gu M, Li XS (2009) Superfast multifrontal method for large structured linear systems of equations. SIAM J Matrix Anal Appl 2008, 31(3):1382–1411

Supernode Partitioning

►Tiling

Superscalar Processors

Wen-mei Hwu
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Multiple-instruction issue; Out-of-order execution processors

Definition

A superscalar processor is designed to achieve an execution rate of more than one instruction per clock cycle for a single sequential program.

Discussion

Introduction

Superscalar processor design typically refers to a set of techniques that allow the central processing unit (CPU) of a computer to achieve a throughput of more than one instruction per cycle while executing a single sequential program. While there is not a universal agreement on the definition, superscalar design techniques typically include parallel instruction decoding, parallel register renaming, speculative execution, and out-of-order execution. These techniques are typically employed along with complementing design techniques such as pipelining, caching, branch prediction, and multi-core in modern microprocessor designs.

A typical superscalar processor today is the Intel Core i7 processor based on the Nehalem microarchitecture. There are multiple processor cores in a Core i7 design, where each processor core is a superscalar processor. The processor performs parallel decoding on IA (X86) instructions, performs parallel register renaming to map the X86 registers used by these instructions to a larger set of physical registers, performs speculative execution of instructions beyond conditional branch instruction and potential exception causing instructions, and allows instructions to execute out of their program specified order while maintaining the appearance of in-order completion. These techniques are accompanied and supported by pipelining, instruction caching, data caching, and branch prediction in the design of each processor core.

Instruction-Level Parallelism and Dependences

Superscalar processor design assumes the existence of instruction-level parallelism, a phenomenon that multiple instructions can be executed independently of each other at each point in time. Instruction-level parallelism arises due to the fact that instructions in an execution phase of a program often read and write different data, thus their executions do not affect each other.

In Fig. 1, Instruction A is a memory load instruction that forms its address from the contents of register 1 (r1), accesses the data in the address location, and deposits the accessed data into register 2 (r2). Instruction B is a memory load instruction that forms its address by adding value 4 to the contents of register 1 (r1), accesses the data in the address location, and deposits the accessed data into register 3 (r3). Instructions A and B can be executed independently of each other. Neither of their execution results is affected by the other. With sufficient execution resources, a superscalar processor can execute A and B together and achieve an execution rate of more than one instruction per clock cycle.

Note that Instruction C cannot be executed independently of Instruction A or Instruction B. This is because Instruction C uses the data in register 2 (r2) and register 3 (r3) as its input. These data are loaded from memory by Instruction A and Instruction B. That is, the execution of Instruction C *depends* on that of instruction A and Instruction B. Such dependences limit the amount of instruction-level parallelism that exists in a program.

In general, *data dependences* arise in programs due to the way instructions read from and write into register and memory storage locations. Data dependencies occur between instructions in three forms. We use the code example in Fig. 1 to illustrate these forms of data dependences:

1. Data flow dependency: the destination register of Instruction A is the same as one of the source registers of Instruction C, and C follows A in the sequential program order. In this case, a subsequent instruction consumes the execution result of a previous instruction.
2. Data antidependency: one of the source operands of Instruction C is the same as the destination register of Instruction D, and D follows C in the sequential program order. In this case, Instruction C should receive result of Instruction B. However, if Instruction D is executed too soon, C may receive execution result of Instruction D, which is too new. In this case, a subsequent instruction overwrites one of the source registers of a previous instruction.
3. Data output dependency: the destination register of Instruction B is the same as the destination register of Instruction D, and Instruction D follows B in the sequential program order. In this case, a subsequent instruction overwrites the destination register of a previous instruction. If D is executed too soon, its result could be overwritten by Instruction B, leaving a result which is too old in the destination register. Subsequent instructions would be using stale results.

A superscalar processor uses register renaming and out-of-order execution techniques to detect and enhance the amount of instruction-level parallelism between instructions so that it can execute multiple instructions per clock cycle. These techniques ensure that all instructions acquire the appropriate input value in the presence of all the parallel execution activities and data dependences. They also make sure that the output values of instructions are reflected correctly in the processor registers and memory locations.

```
A   r2 ← Load MEM[r1+0]
B   r3 ← Load MEM[r1+4]
C   r2 ← r2+r3
D   r3 ← Load MEM[r1+8]
```

Superscalar Processors. Fig. 1 Code example for register access data dependences

Register Renaming

Register renaming is a technique that eliminates register antidependences and output dependences in order to increase instruction parallelism. A register renaming mechanism provides a physical or implementation register file that is larger than the architectural register files. For example, the IA (X86) architecture specifies 8 general-purpose registers whereas the register

renaming mechanism of a superscalar processor typically provides 32 or more physical registers. At any time, each architectural register is mapped to one or more of the physical registers. By mapping, an architectural register is mapped to multiple physical registers, one can eliminate apparent antidependences and output dependences between instructions.

For example, a register renaming mechanism may map architectural register r3 to physical register pr103 for the destination operand of Instruction B and source of Instruction C in Fig. 1. That is, Instruction B will deposit its result to pr103 and Instruction C will fetch its second input operand from pr103. As long as the producer of the data and all the consumers of the data are redirected to the same physical register, the execution results will not be affected.

Let's further assume that architecture register r3 is mapped to physical register pr105 for the destination operand of Instruction D. That is, Instruction D will deposit its execution results to pr105. As long as all subsequent uses of the data produced by Instruction D are redirected to pr105 for their input, their execution results will remain the same.

The reader should see that the antidependence between Instruction C and Instruction D has been eliminated by the register renaming mechanism. Since Instruction C will go to pr103 for its input whereas Instruction D will deposit its result into pr105, Instruction D will no longer overwrite the input for Instruction C no matter how soon it is executed. As a result, we have eliminated a dependence constraint between Instruction C and Instruction D.

The reader should also notice that the output dependence between Instruction B and Instruction D has been eliminated by the register renaming mechanism. Since Instruction B deposits its result into pr103 and Instruction D deposits its result into pr105, B will no longer overwrite the result of D no matter how soon D is executed. As long as subsequent Instructions that use architecture register r3 are also directed to pr105, they will see the updated results rather than stale results even though B could be executed after D.

By eliminating the register antidependence between C and D and output dependence between B and D, Instruction D can now be executed in parallel with Instructions A and B. This increases the level of instruction-level parallelism. This is the reason why register renaming has become an essential mechanism for superscalar processor design.

Speculative Execution

A major limitation on the amount of instruction-level parallelism is uncertainties in program execution sequence. There are two major sources of such uncertainty. The first is conditional and indirect branch instructions. Conditional branches are used to implement control constructs such as if-then-else statements and loops in high-level language programs. Depending on the condition values, the instructions to be executed after a conditional branch can be either from the next sequential location or from a target location specified by the branch offset. Indirect branches use the contents of a register or memory location as the address of the next instruction to execute. Indirect branches are used to implement procedure return statements, case statements, and table-based procedural calls in high-level language programs.

While conditional and indirect branch instructions serve essential purposes in implementing high-level languages, they introduce uncertainties for execution. The classic approach to addressing this problem is branch prediction, where a mechanism is used to predict the next instructions to execute when conditional and indirect branches are encountered during program execution. However, prediction alone is not sufficient. One needs a means to speculatively execute the instructions on the predicted path of execution and recover from any incorrect predictions.

The second source of uncertainty in program execution is exception conditions. Modern computers use exceptions to support the implementation of virtual memory management, memory protection, and rare execution condition handling. For example, the Instruction A in Fig. 1 may trigger a page fault in its execution and require operating system service to bring in its missing load data. The execution needs to be able to resume cleanly after the page fault is handled by the operating system. This requirement means that the instructions after a load instruction, or any instruction that can potentially cause exceptions, cannot change the execution state in a way that prevents the execution from restarting at the exception causing instruction.

Like in the case of branch prediction, one can "predict" that the exception conditions do not occur and assume that the execution will simply continue down the current path. However, one needs a means to recover the state when an exception indeed occurs so that the execution can correctly restart from the exception-causing instruction.

Speculative execution is a mechanism that allows processors to fetch and execute instructions down a predicted path and to recover if the prediction is incorrect. In general, these mechanisms use buffers to keep both the original state and the recent updates to the state. The updated state is used during the speculative execution. The original state is used if the processor needs to recover from an incorrect prediction. Since conditional and indirect branches occur frequently, one in every four to five instructions on average, the level of instruction level parallelism a superscalar processor can exploit would be extremely low without speculative execution. This is why speculative execution has become an essential mechanism in superscalar processor design. The most popular methods for recovering from incorrect branch predictions are reorder buffer (ROB) and checkpointed register file. The most popular method for recovering from exception causing instructions is reorder buffer.

A retirement mechanism in speculative execution attempts to make the effects of instruction execution permanent. An instruction is eligible for retirement when all instructions before it have retired. An eligible instruction is then checked if it has caused any exceptions or incorrect branch prediction. If the instruction does not incur any exception or incorrect branch prediction, it can commit its execution result into an *architectural state*. Otherwise, the instruction triggers a recovery and the state of the processor is restored to a previous architectural state.

The capacity of reorder buffers and checkpoint buffers defines the notion of *instruction window*, a collection of consecutive instructions that are actively processed by a superscalar processor. At any point in time, the instruction window starts with the oldest instruction that has not completed execution and ends with the youngest instruction that has started execution. The larger the instruction window, the more hardware is needed to keep track of the execution status of all instructions in the window and the information needed to recover the processor state if anything goes wrong with the execution of the instructions in the window.

Out-of-Order Execution

In a superscalar processor, instructions are fetched according to their sequential program order. However, this may not be best order of execution. For example, in Fig. 1, Instruction D is fetched after Instruction C. However, Instruction C cannot execute until Instructions B and Instruction D completes their execution. On the other hand, with register renaming, Instruction D does not have any dependence on Instructions A, Instruction B, or Instruction C. Therefore, a superscalar processor "reorders" the execution of Instruction C and Instruction D so that Instruction D can produce results for its consumers as soon as possible.

An out-of-order execution mechanism provides buffering for Instructions that need to wait for their input data so that some of their subsequent instructions can proceed with execution. A popular method used in out-of-order execution is the Tomasulo's Algorithm that was originally used in the IBM 360/91. The method maintains *reservation stations*, hardware buffers that allow the instructions to wait for their input operands.

Brief Early History

The most popular out-of-order execution mechanism in modern superscalar processors is Tomasulo's algorithm [1] designed by Bob Tomasulo and used in the IBM 360/91 floating point unit in 1967. The out-order-execution mechanism was abandoned in later IBM machines partly due to the concern of the problems it introduces to virtual memory management.

The Intel Pentium processor [2] is an early superscalar design that fetches and executes multiple instructions at each clock cycle. It did not employ register renaming or speculative execution and was able to exploit only a very limited amount of instruction-level parallelism.

Register renaming in superscalar processor design started with the Register Alias Table (RAT) by Patt et al. [3]. To this day, the register renaming structure used in Intel superscalar processors are still called RAT.

Simth and Plaszkun proposed reorder buffers and history buffers for recovering from exceptions in highly pipelined processors [4]. Hwu et al. [5] extended the concepts with checkpointing and incorporated these speculative execution mechanisms with Tomasulo's algorithm.

In 1985, Patt et al. [3] proposed a comprehensive superscalar design that incorporates register renaming, speculative execution, out-of-order execution, along with parallel instruction decode and branch prediction. This is the first comprehensive academic design of superscalar processors. In 1987, Sohi and Vijapeyam [6] proposed a unified reservation station design. These designs were later adopted and refined by Intel to create the Pentium Pro Processor, the first commercially successful superscalar processor design [7].

Recent superscalar processors include MIPS R10000, Intel Pentium 4, IBM Power 6, AMD Athlon, and ARM Cortex. Interested readers should refer to Hennessy and Patterson [8] for more detailed treatment and more recent history on superscalar processor design.

Bibliography

1. Tomasulo R (1967) An efficient algorithm for exploiting multiple arithmetic units. IBM J ResDev 11(1):8–24
2. Case B (1993) Intel reveals pentium implementation details. Microprocessor Report 29 Mar 1993
3. Patt Y, Hwu W-M, Shebanow M (1985) HPS, a new microarchitecture: rationale and introduction. In: Proceedings of the 18th annual workshop on microprogramming, Pacific Grove, pp 103–108
4. Smith J, Pleszkun A (1985) Implementation of precise interrupts in pipelined processors. In: Proceedings of the 12th international symposium on computer architecture, Boston
5. Hwu W-M, Patt Y (1987) Checkpoint repair for out-of-order execution machines. In: Proceedings of the 14th international symposium on computer architecture, Pittsburgh
6. Sohi G, Vajapeyam S (1987) Instruction issue logic for high-performance, interruptable pipelined processors. In: Proceedings of the 14th international symposium on computer architecture, New York
7. Colwell R (2005) Pentium chronicles – the people, passion, and politicc behind Intel's Lanmark chips. Wiley-IEEE Computer Society, ISBN 978-0-47-173617-2
8. Hennessy J, Patterson D (2007) Computer architecture – a quantitative approach, 4th edn. Morgan Kauffman, San Francisco, ISBN 978-0-12-370490-0

SWARM: A Parallel Programming Framework for Multicore Processors

David A. Bader[1], Guojing Cong[2]
[1]Georgia Institute of Technology, Atlanta, GA, USA
[2]IBM, Yorktown Heights, NY, USA

Definition

Soft**W**are and **A**lgorithms for **R**unning on **M**ulti-core (SWARM) is a portable open-source parallel library of basic primitives for programming multicore processors. SWARM is built on POSIX threads that allows the user to use either the already developed primitives or direct thread primitives. SWARM has constructs for parallelization, restricting control of threads, allocation and deallocation of shared memory, and communication primitives for synchronization, replication and broadcast. Built on these techniques, it contains a higher-level library of multicore-optimized parallel algorithms for list ranking, comparison-based sorting, radix sort, and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [3], graph decomposition [8], breadth-first-search [9], tree contraction [10], and maximum parsimony [7].

Motivation

For the last few decades, software performance has improved at an exponential rate, primarily driven by the rapid growth in processing power. However, performance improvements can no longer rely solely on Moore's law. Fundamental physical limitations such as the size of the transistor and power constraints have now necessitated a radical change in commodity microprocessor architecture to multicore designs. Dual and quad-core processors from Intel [13] and AMD [2] are now ubiquitous in home computing. Also, several novel architectural ideas are being explored for high-end workstations and servers [15, 16]. Continued software performance improvements on such novel multicore systems now require the exploitation of concurrency at the algorithmic level. Automatic methods for detecting concurrency from sequential codes, for example

with parallelizing compilers, have had only limited success. SWARM was introduced to fully utilize multicore processors.

On multicore processors, caching, memory bandwidth, and synchronization constructs have a considerable effect on performance. In addition to time complexity, it is important to consider these factors for algorithm analysis. SWARM assumes the multicore model that can be used to explain performance on systems such as Sun Niagara, Intel, and AMD multicore chips. Different models [6] are required for modeling heterogeneous multicore systems such as the Cell architecture.

Model for Multicore Architectures

Multicore systems have a number of processing cores integrated on to a single chip [2, 11, 13, 15, 16]. Typically, the processing cores have their own private L_1 cache and share a common L_2 cache [13, 16]. In such a design, the bandwidth between the L_2 cache and main memory is shared by all the processing cores. Figure 1 shows the simplified architectural model.

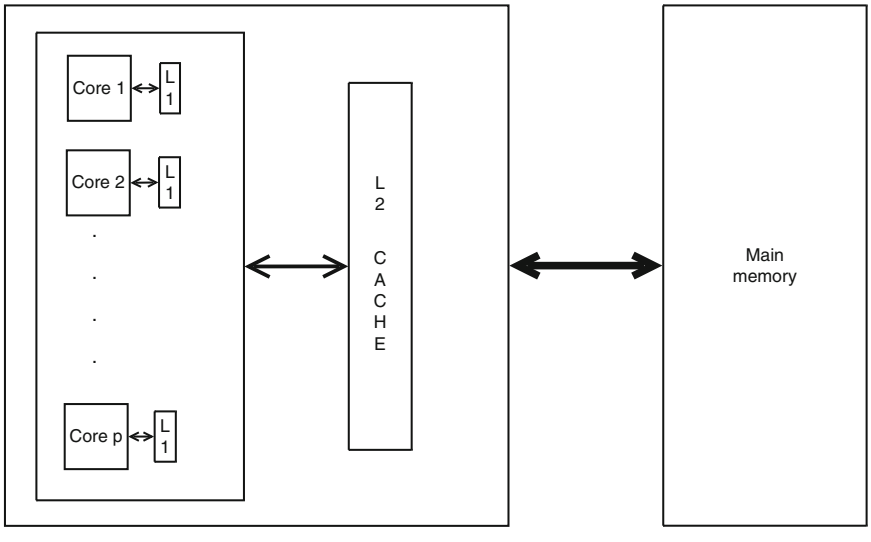
Multicore Model

The multicore model (MCM) consists of p identical processing cores integrated onto a single chip. The processing cores share an L_2 cache of size C, and the memory bandwidth is σ.

1. *Let $T(i)$ denote the local time complexity of the core i for $i = 1, \ldots, p$. Let $T = \max_i T(i)$.*
2. *Let B be the total number of blocks transferred between L_2 cache and the main memory. The requests may arise out of any processing core.*
3. *Let L be the time required for synchronization between the cores. Let $N_S(i)$ be the total number of synchronizations required on core i for $i = 1, \ldots, p$. Let $N_S = \max_i N_S(i)$.*

Then the complexity under the multicore model can be represented by a triple $\langle T, B \cdot \sigma^{-1}, N_S \cdot L \rangle$. The complexity of an algorithm will be represented by the dominant element in this triple.

The model proposed above is in many ways similar to the Helman-JáJá model for symmetric multiprocessor (SMP) systems [12], with a few important differences. In the case of SMPs, each processor typically has a large L_2 cache and dedicated bandwidth to main memory, whereas in multicore systems, the shared memory bandwidth will be an important consideration. Thus, SWARM explicitly models the cache hierarchy, and count the number of block transfers between the cache and main memory in a manner similar to Aggarwal and Vitter's external memory model [1].



SWARM: A Parallel Programming Framework for Multicore Processors. Fig. 1 Architectural model for multicore systems

SWARM targets three primary issues that affect performance on multicore systems:

1. *Number of processing cores*: Current systems have two to eight cores integrated on a single chip. Cores typically support features such as simultaneous multithreading (SMT) or hardware multithreading, which allow for greater parallelism and throughput. In future designs, up to 100 cores can exist on a single chip.
2. *Caching and memory bandwidth*: Memory speeds have been historically increasing at a much slower rate than processor capacity [14]. Memory bandwidth and latency are important performance concerns for several scientific and engineering applications. Caching is known to drastically affect the efficiency of algorithms even on single processor systems [17, 18]. In multicore systems, this will be even more important due to the added bandwidth constraints.
3. *Synchronization*: Implementing algorithms using multiple processing cores will require synchronization between the cores from time to time, which is an expensive operation in shared memory architectures.

Case Study: Merge Sort

Algorithm 1

In the first algorithm, the input array of length N is equally divided among the p processing cores so that each core gets N/p elements to sort. Once the sorting phase is completed, there are p sorted sub-arrays, each of length N/p. Thereafter the merge phase takes place. A p-way merge over the runs will give the sorted array. Each processor individually sorts its elements using some *cache-friendly* algorithm. This approach does not try to minimize the number of blocks transferred between the L_2 cache and main memory.

Analysis. Since the p processors are all sorting their respective elements at the same time, the L_2 cache will be shared by all the cores during the sorting phase. Thus, if the size of the L_2 cache is C, then effectively each core can use just a portion of the cache with size C/p. Assuming the input size is larger than the cache size, the cache misses will be p times that if only a single core were sorting. Also the bandwidth between the cache and shared main memory is also shared by all the p cores, and this may be a bottleneck.

The time complexity of each processor is:

$$T_c(\text{sort}) = \frac{N}{p} \cdot \log\left(\frac{N}{p}\right)$$

During the merge phase:

$$T_c(\text{merge}) = N \cdot \log(p)$$

$$T_c(\text{total}) = \frac{N}{p}\log\left(\frac{N}{p}\right) + N\log(p)$$

Algorithm 2

This algorithm divides the given array of length N into blocks of size M where M is less that C, the size of the L_2 cache. Each of such N/M blocks is first sorted using all p cores. This is the sorting phase. When the sorting phase is completed, the array consists of N/M runs each of length M. During the merge phase, p blocks are merged at a time. This process is repeated till a single sorted array is arrived. Thus, the merge phase is carried out $\log_p\left(\frac{N}{M}\right)$ times.

Analysis. This algorithm is very similar to the I/O model merge sort [1]. Thus, this algorithm is optimal in terms of transfers between main memory and L_2 cache. However, it will have slightly higher-computational complexity. The p cores sort a total of N/M blocks of size M. Assuming the use of a split-and-merge sort for sorting the block of M elements, thus, during the sorting phase, the time per core is:

$$\begin{aligned} T_c(\text{sort}) &= \frac{N}{M} \cdot \frac{M}{p}\log\left(\frac{M}{p}\right) + \frac{N}{M} \cdot M\log(p) \\ &= \frac{N}{p}\log\left(\frac{M}{p}\right) + N\log(p) \end{aligned} \quad (1)$$

During any merge phase, if blocks of size S are being merged p at a time, the complexity per core is $\frac{N}{Sp} \cdot Sp\log(p) = N\log(p)$. There are $\log_p\left(\frac{N}{M}\right)$ merge phases, thus

$$T_c(\text{merge}) = N\log(p) \cdot \log_p\left(\frac{N}{M}\right)$$

$$T_c(\text{total}) = \frac{N}{p}\log\left(\frac{M}{p}\right) + N\log(p)\left(1 + \log_p\left(\frac{N}{M}\right)\right)$$

Comparison. Algorithm 1 clearly has better time complexity than algorithm 2. However, algorithm 2 is optimal in terms of transfers between L_2 cache and shared

main memory. Algorithm analysis using this model captures computational complexity as well as memory performance.

Programming in SWARM

A typical SWARM program is structured as follows:

```
int main (int argc, char **argv)
{
    SWARM_Init(&argc, &argv);
    /* sequential code  */
    ....
    ....
    /* parallelize a routine using
    SWARM */
    SWARM_Run(routine);
    /* more sequential code */
    ....
    ....
    SWARM_Finalize();
}
```

In order to use the SWARM library, the programmer needs to make minimal modifications to existing sequential code. After identifying compute-intensive routines in the program, work can be assigned to each core using an efficient multicore algorithm. Independent operations such as those arising in *functional parallelism* or *loop parallelism* can be typically threaded. For functional parallelism, this means that each thread acts as a functional process for that function, and for loop parallelism, each thread computes its portion of the computation concurrently. Note that it might be necessary to apply loop transformations to reduce data dependencies between threads.

SWARM contains efficient implementations of commonly used primitives in parallel programming.

Data parallel. The SWARM library contains several basic "`pardo`" directives for executing loops concurrently on one or more processing cores. Typically, this is useful when an independent operation is to be applied to every location in an array, for example element-wise addition of two arrays. `Pardo` implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores. By default, `pardo` uses block partitioning of the loop assignment values to the threads, which typically results in better cache utilization due to the array locations on lefthand side of the assignment being owned by local caches more often than not. However, SWARM explicitly provides both block and cyclic partitioning interfaces for the pardo directive.

```
/* example: partitioning a "for"
   loop among the cores */
pardo(i, start, end, incr) {
    A[i] = B[i] + C[i];
}
```

Control. SWARM control primitives restrict which threads can participate in the context. For instance, the control may be given to a single thread on each core, all threads on one core, or a particular thread on a particular core.

```
THREADS: total number of execution
threads
MYTHREAD: the rank of a thread,
from 0 to THREADS-1
/* example: execute code on
   thread MYTHREAD */
on_thread(MYTHREAD) {
    ....
    ....
}
/* example: execute code on
   one thread */
on_one_thread {
    ....
    ....
}
```

Memory management. SWARM provides two directives `SWARM_malloc` and `SWARM_free` that, respectively, dynamically allocate a shared structure and release this memory back to the heap.

```
/* example: allocate a shared array
   of size n */
A=(int*)SWARM_malloc(n*sizeof(int),
TH);
/* example: free the array A */
SWARM_free(A);
```

Barrier. This construct provides a way to synchronize threads running on the different cores.

```
/* parallel code */
```

```
    ....
    ....
/* use the SWARM Barrier for
   synchronization */
SWARM_Barrier();
/* more parallel code */
    ....
    ....
```

Replicate. This primitive uniquely copies a data buffer for each core.

Scan (**reduce**). This performs a prefix (reduction) operation with a binary associative operator, such as addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR. `allreduce` replicates the result from reduce for each core.

```
/* function signatures */
int SWARM_Reduce_i(int myval,
            reduce_t op,
            THREADED);
double SWARM_Reduce_d(double
            myval,
            reduce_t op,
            THREADED);

/* example: compute global sum,
using
partial local values from each
core */
sum = SWARM_Reduce_d(mySum, SUM,
TH);
```

Broadcast. This primitive supplies each processing core with the address of the shared buffer by replicating the memory address.

```
/* function signatures */
int SWARM_Bcast_i    (int myval,
                      THREADED);
int* SWARM_Bcast_ip  (int* myval,
                      THREADED);
char SWARM_Bcast_c   (char myval,
                      THREADED);
```

Apart from the primitives for computation and communication, the thread-safe parallel pseudo-random number generator SPRNG [19] is integrated in SWARM.

Algorithm Design and Examples in SWARM

The SWARM library contains a number of techniques to demonstrate key methods for programming on multi-core processors.

- The ***prefix-sum*** algorithm is one of the most useful parallel primitives and is at the heart of several other primitives, such as array compaction, sorting, segmented prefix-sums, and broadcasting; it also provides a simple use of balanced binary trees.
- ***Pointer-jumping*** (or path-doubling) iteratively halves distances in a list or graph. It is used in numerous parallel graph algorithms, and also as a sampling technique.
- Determining the root for each tree node in a rooted-directed forest is a crucial step in handling equivalence classes – such as detecting whether or not two nodes belong to the same component; when the input is a linked list, this algorithm also solves the parallel prefix problem.
- An entire family of techniques of major importance in parallel algorithms is loosely termed ***divide-and-conquer*** – such techniques decompose the instance into smaller pieces, solve these pieces independently (typically through recursion), and then merge the resulting solutions into a solution to the original instance. These techniques are used in sorting, in almost any tree-based problem, in a number of computational geometry problems (finding the closest pair, computing the convex hull, etc.), and are also at the heart of fast transform methods such as the FFT. The `pardo` primitive in SWARM can be used for implementing such a strategy.
- A variation of the above theme is the ***partitioning strategy***, in which one seeks to decompose the problem into independent subproblems – and thus avoid any significant work when recombining solutions; quicksort is a celebrated example, but numerous problems in computational geometry can be solved efficiently with this strategy (particularly problems involving the detection of a particular configuration in three- or higher-dimensional space).
- Another general technique for designing parallel algorithms is ***pipelining***. In this approach, waves of concurrent (independent) work are employed to achieve optimality.

Built on these techniques, SWARM contains a higher-level library of multicore-optimized parallel algorithms for list ranking, comparison-based sorting, radix sort and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [3], graph decomposition [8], breadth-first-search [9], tree contraction [10] and maximum parsimony [7].

Related Entries

►Cilk
►OpenMP
►Parallel Skeletons

History

The SWARM programming framework is a descendant of the symmetric multiprocessor (SMP) node library component of SIMPLE [4].

Bibliography

1. Aggarwal A, Vitter J (1988) The input/output complexity of sorting and related problems. Commun ACM 31:1116–1127
2. AMD Multi-Core Products (2006), http://multicore.amd.com/en/Products/
3. Bader DA, Cong G (2004) A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proceedings of the international parallel and distributed processing symposium (IPDPS 2004), Santa Fe, NM, April 2004
4. Bader DA, JáJá J (1999) SIMPLE: a methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). J Parallel Distrib Comput 58(1):92–108
5. Bader DA (2006) SWARM: a parallel programming framework for multicore processors, https://sourceforge.net/projects/multicore-swarm
6. Bader DA, Agarwal V, Madduri K (2007) On the design and analysis of irregular algorithms on the Cell processor: a case study of list ranking. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA
7. Bader DA, Chandu V, Yan M (2006) ExactMP: an efficient parallel exact solver for phylogenetic tree reconstruction using maximum parsimony. In: Proceedings of the 35th International Conference on Parallel Processing (ICPP), Columbus, OH, August 2006
8. Bader DA, Illendula AK, Moret BME, Weisse-Bernstein N (2001) Using PRAM algorithms on a uniform-memoryaccess shared-memory architecture. In: Brodal GS, Frigioni D, Marchetti-Spaccamela A (eds) Proceedings of the 5th international workshop on algorithm engineering (WAE 2001), volume 2141 of lecture notes in computer science. Springer-Verlag, Århus, Denmark, pp 129–144
9. Bader DA, Madduri K (2006) Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: Proceedings of the 35th international conference on parallel processing (ICPP), IEEE Computer Society, Columbus, OH, August 2006
10. Bader DA, Sreshta S, Weisse-Bernstein N (2002) Evaluating arithmetic expressions using tree contraction: a fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In: Sahni S, Prasanna VK, Shukla U (eds) Proceedings of the 9th international conference on high performance computing (HiPC 2002), volume 2552 of lecture notes in computer science. Bangalore, India, Springer-Verlag, December 2002, pp 63–75
11. Barroso LA, Gharachorloo K, McNamara R, Nowatzyk A, Qadeer S, Sano B, Smith S, Stets R, Verghese B (2000) Piranha: a scalable architecture based on single-chip multi-processing. SIGARCH Comput Archit News 28(2):282–293
12. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99), volume 1619 of lecture notes in computer science, Springer-Verlag, Baltimore, MD, January 1999, pp 37–56
13. Multi-Core from Intel – Products and Platforms (2006) http://www.intel.com/multi-core/products.htm
14. International Technology Roadmap for Semiconductors (2004), http://itrs.net, 2004 update
15. Kahle JA, Day MN, Hofstee HP, Johns CR, Maeurer TR, Shippy D (2005) Introduction to the cell multiprocessor. IBM J Res Dev 49(4/5):589–604
16. Kongetira P, Aingaran K, Olukotun K (2005) Niagara: a 32-way multithreaded Spare processor. IEEE Micro 25(2):21–29
17. Ladner R, Fix JD, LaMarca A (1999) The cache performance of traversals and random accesses. In: Proceedings of the 10th annual symposium discrete algorithms (SODA-99), ACM-SIAM, Baltimore, MD, pp 613–622
18. Ladner RE, Fortna R, Nguyen B-H (2002) A comparison of cache aware and cache oblivious static search trees using program instrumentation. In: Fleischer R, Meineche-Schmidt E, Moret BME (eds) Experimental algorithms, volume 2547 of lecture notes in computer science, Springer-Verlag, Berlin Heidelberg, pp 78–92
19. Mascagni M, Srinivasan A (2000) Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. ACM Trans Math Softw 26(3):436–461

Switch Architecture

José Flich
Technical University of Valencia, Valencia, Spain

Synonyms

Router architecture

Definition

The switch architecture defines the internal organization and functionality of components in a switch. The switch is in charge of forwarding units of information from the input ports to the output ports.

Discussion

High-performance computing systems, like clusters and massively parallel processors (MPPs), rely on the use of an efficient interconnection network. As the number of end nodes increases to thousands, or even larger sizes, the network becomes a key component since it must provide low latencies and high bandwidth at a moderate cost and power consumption. The basic components of a network are switches/routers, links, and network interfaces. The interconnection network efficiency largely depends on the switch design.

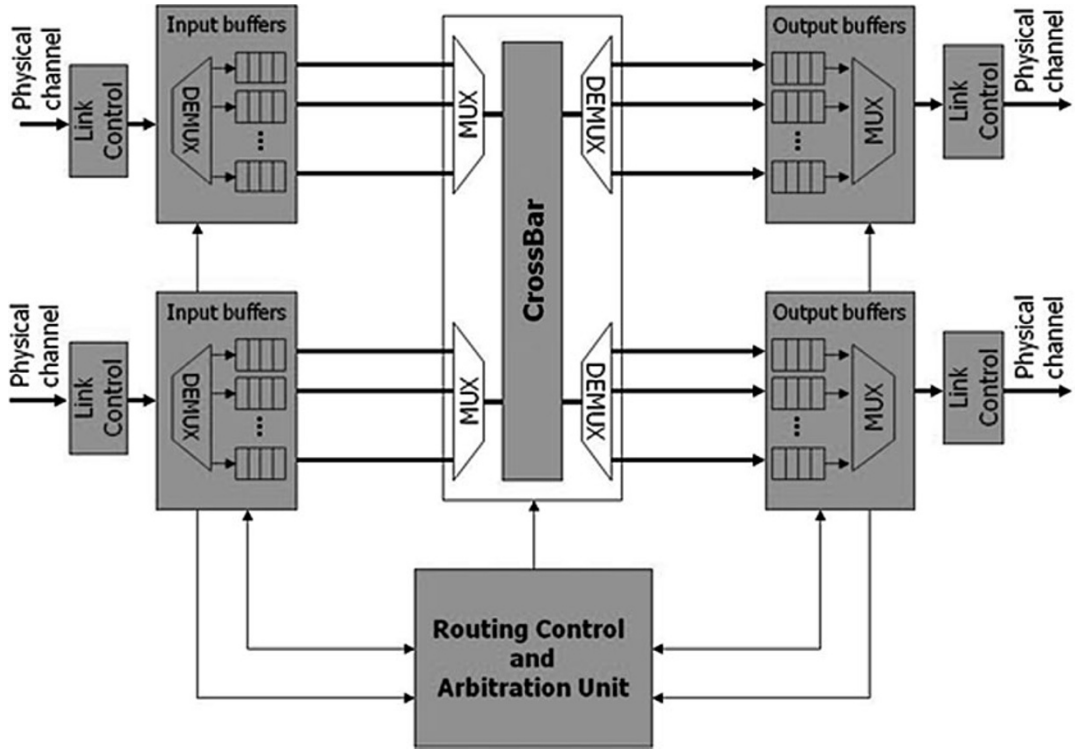
Prior to defining and describing the switch architecture concept, it is worth differentiating between router and switch. Typically, the router is the basic component in a network that forwards messages from a set of input ports to a set of output ports. The router usually negotiates paths and computes the output ports messages need to take. Therefore, the router has some intelligence and adapts to the varying conditions of the network. Indeed, the router concept comes from Wide Area Networks (WANs) where the switching devices must be smart enough to adapt to the varying topological conditions. In high-performance interconnection networks, typically found in cluster cabinets, connecting massively parallel processors, and nowadays even inside a chip, switching devices are also needed. However, differently from the WAN environment, these devices, although have some intelligence and can make critical decisions, they have no capabilities to negotiate the paths and to adapt to the varying conditions. Indeed, usually the topology is expected not to change, thus no need for such negotiation. This is the reason why these devices are also known as switches rather than routers. However, both terms are used with no clear differentiation by the community and thus, both become acceptable. This entry is related to switch architecture, although it can be seen also as *router architecture*.

The switch architecture defines the internal organization and functionality of a switch or router. Basically, the switch has a set of input ports where data messages come in and a set of output ports where data messages are delivered. The way internal components are used and connected between them is defined by the switch architecture.

The switch architecture largely depends on the switching technique (see ►Switching Techniques) used; thus, we may find switch architectures implementing *store and forward switching* that greatly vary from switches implementing *wormhole switching*. Most of the current modern routers and switches, used in high-performance networks, implement *cut-through switching*, or some variant. This entry focuses on such architectures, mainly in wormhole (WH) switches and virtual cut-through (VCT) switches. Although there are basic differences between them, that affect the switch architecture, there are commonalities as both rely on the same form of switching. In the next section, a canonical switch architecture including all the commonalities found in current switches is provided. Then, different components are reviewed and alternative switch organizations are described.

Canonical Switch Architecture

Figure 1 shows the organization of a basic switch architecture. The switch is made up of a set of identical input ports, connected to input physical channels. Each input port uses a link control module to adapt messages coming through the physical channel to the internal switch. Each message is then stored in a buffer at the input port. The buffer can be made of different queues, each one usually associated to a virtual channel. Each message is then routed (computing the appropriate output port to take) at the Routing control unit. Once the output port is computed, the message may cross the internal crossbar of the switch, thus reaching the output port. To do so, the message has to win the access to the output port since different messages may compete for the same port. In addition, the message has to compete to get access to an available virtual channel. Both are resolved by the arbitration unit. Once the access is granted, the message crosses the crossbar and is written at the corresponding queue at the output port. Each message, again, needs to compete with other messages in order to win the access to the physical channel. Prior to reaching the physical channel the message passes through the link controller logic.



Switch Architecture. Fig. 1 Basic switch architecture

As an alternative view, the resources found in the switch can be divided into two separate blocks: the *data plane (or datapath)* and the *control plane*. Basically, the data plane is the set of resources messages use while being forwarded (storage and movement) through the switch. The control plane is made of the resources used to make decisions at the switch, like the routing control unit and the arbitration unit.

Alternative Switch Architectures

Taking this basic switch architecture as the starting point, several alternative architectures can be derived, some of them better suited for a particular switching mechanism. One of the key issues in the switch architecture is the location of the buffering resources. Regarding this, switch architectures can be classified as:

Output-Queued (OQ) switch architecture. In this architecture buffers only exist at the output ports and thus no buffering exists at the input port. Thus, whenever a message arrives at the switch it must be sent directly to the appropriate output port. An $N \times N$ OQ switch requires only N memories, one per output port. As packets are mapped to the memory associated with the requested output port, HOL blocking is totally eliminated (see ▶Congestion Management Entry); thus, this organization achieves maximum switch efficiency. However, internal speedup is required to handle the worst-case scenario without dropping messages, allowing all the input ports to transmit packets to the same output port at the same time. In particular, output queues must either implement multiple write ports or use a higher clock frequency. A speedup of N is required in OQ switches in the worst case. Unfortunately, providing such internal speedup is not always viable.

Input-Queued (IQ) switch architecture. Buffers only exist at input ports and not at output ports. In this architecture the required internal bandwidth does not increase with the number of ports and the switch can be designed with the same bandwidth as the link. However, a switch designed with this organization may face low performance due to contention/congestion at the output ports. It is well known that such switches achieve only 58% of maximum efficiency under uniformly distributed requests [3]. This is mainly

due to the HOL blocking problem. One solution to eliminate the HOL blocking issue in IQ switches is the use of N queues at every input port, mapping the incoming message to a queue associated with the requested output port. This technique is known as Virtual Output Queuing (VOQ) [4]. However, it increases the queue requirements quadratically with the number of ports. Therefore, as the number of ports increases, this solution becomes too expensive.

Combined-Input-Output-Queued (CIOQ) switch architecture. With this organization, the contention/congestion problem found in IQ switches is alleviated or even eliminated, since messages can be also stored at the output side of the switch. In this architecture some moderate internal speedup is used, as the internal bandwidth is higher than the aggregate link bandwidth. A speedup of two is usually enough to compensate the performance drop produced by HOL blocking. Speedup can be implemented by using internal datapaths with higher transmission frequencies or wider transmission paths. However, as the external link bandwidth increases, sustaining the speedup may become difficult.

Buffered-Crossbar (BC) switch architecture. This organization uses a memory at every crossbar crosspoint. An input link is connected to N memories, each one connected to a different output port. By design, the BC organization implements internal speedup, as many inputs can forward a packet to the same output at the same time. Additionally, such memory organization eliminates the HOL blocking (every packet is mapped to the memory associated with the requested output port). As a consequence, the BC organization requires low-cost arbiters per output port. However, the problem with such organization is that the number of memories increases quadratically with the number of ports (N^2), thus limiting scalability.

When focusing on the switching device (the crossbar) different alternatives exist. The solution used in the basic example is the most frequently implemented one, where a crossbar connecting every input to every possible output is used. The crossbar has inherited bandwidth as it is able to connect an input to every output (broadcast communication). There are, however, other solutions, like using a centralized buffer and using a bus.

In addition, the connection of the input ports to the crossbar, and the crossbar to the output ports can be implemented in different ways. In the example multiplexers and demultiplexers are used at both sides of the crossbar. This is done to reduce the crossbar complexity (that increases quadratically with the number of ports). Different alternative configurations arise when these devices are removed from one or both sides of the crossbar. In the case multiplexers are removed, the input speedup of the switch is increased, since different messages can be forwarded through the crossbar from the same input port. If demultiplexers are removed then the output speedup of the switch is increased, since different messages can be written to the same output port at the same time. Obviously, output buffers are required when implementing output speedup.

Input Buffer Organization

An important component of the switch architecture is the buffer organization, mostly when virtual channels are implemented. The way memory resources are assigned to queues will impact on the performance of the switch. This issue is highly related to the flow control protocol (see ►Flow Control). Buffer partitioning can be designed in several ways. The first one is to combine all the buffers across the entire switch (a single memory). In that case, there is no need for a switching element. The benefit of this approach is the flexibility in dynamically allocating memory across the input ports. The problem, however, is the high memory bandwidth required.

The second way to organize buffers is by partitioning them by physical input ports and dynamically assigning them to virtual channels of the same input port. The third way is to partition the buffers by virtual channel, providing a separate memory for each virtual channel at each port. This, however, may lead to high cost and poor memory utilization.

Another important aspect of buffer organization is the way flits are stored. Such memories require data structures to keep track of where flits are located, and to manage free slots. Two buffer organizations are mostly used: circular buffers and linked lists. Circular buffers are used when memory is statically assigned to a queue (virtual channel) and have low implementation overheads. However, a linked list is used when memory is assigned dynamically and has, in turn, higher implementation overhead.

Pipelined Organization

It is common to design a switch as a pipelined datapath. Such designs allow for high clock frequencies of the switch and, thus, high throughput. On every cycle, a different flit (see ►Flow Control Entry) from the same message may be processed at each stage. Typically, five stages are conceived for the switches:

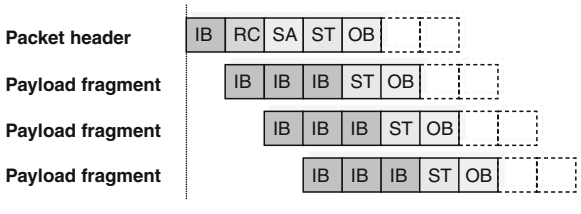
- Input Buffer (IB) stage, where the flit is stored at the input port of the switch.
- Routing computation (RC) stage, where the output port is computed for the message.
- Switch allocation (SA) stage, where the access to the output port is granted among all the requesting messages. Also, selection of the virtual channel to use is performed.
- Switch transversal (ST) stage, where the flit crosses the crossbar and reaches the output.
- Output Buffer (OB) stage, where the flit is stored at the output port of the switch.

In the ideal case (no contention experienced within the switch) the flit advances through the pipelined switch architecture as shown in Fig. 2. The figure shows the case for a switch implementing virtual cut-through switching where arbitration is performed once per packet.

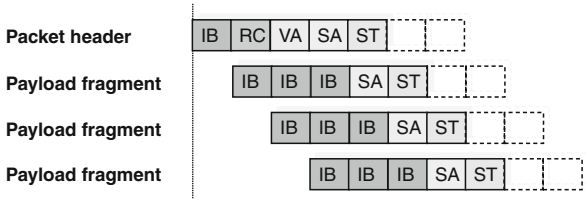
The first flit is the header flit of the message. At the first cycle the header flit is stored at the input port (IB stage) in a virtual channel. The header includes the identifier of the virtual channel to use. At the second cycle, the header flit is processed at the RC stage so the output port for the message is computed. The output port identifier is associated with the input virtual channel, as this output port will be used by all the flits of the message. At the same cycle the second flit of the message (payload flit) is stored at the input port (IB stage). At the third cycle the SA stage is executed to get access to the requested output port. A valid virtual channel for the message at the next switch is also requested. On success (there is an available virtual channel and the output port is granted) the header flit crosses the crossbar at the next cycle, followed in the following cycles by the rest of payload flits. All the flits use the same virtual channel and output port. Notice also that resources (access to the output port) are granted per message, so flit multiplexing in the crossbar may not occur. Upon crossing the crossbar, the flits are stored at the output port in the corresponding queue (OB stage). The tail flit of a message will have a different treatment since the connection of the input port to the output port will be broken. Notice that the RC and SA stages are performed only once per packet.

A typical wormhole switch architecture differs in some stages. First, it is not common to have a buffer at the output port, thus using an IQ switch approach. Second, virtual channel allocation and port allocation are usually performed in separate stages. Therefore, a new stage appears, referred to as Virtual channel allocation (VA) used only for header flits, and the SA stage only intended for requesting the output port. Also, output port usually is granted flit by flit. Figure 3 shows an example of a five-stage pipelined wormhole switch.

The pipeline design may experience stalls. Stalls may happen due to different reasons (a virtual channel is not available in the SA stage, the RC stage does not find a free output port, ...). In all these cases, the switch must be designed accordingly. Also, the flow control (see ►Flow Control) must be aware of the stalls and thus backpressure the previous switch to avoid buffer overflows. For a detailed analysis of pipelined switches and stall treatments, the reader is referred to [1] (Chapter 16).



Switch Architecture. Fig. 2 Typical five-stage pipelined switch design for a virtual cut-through switch

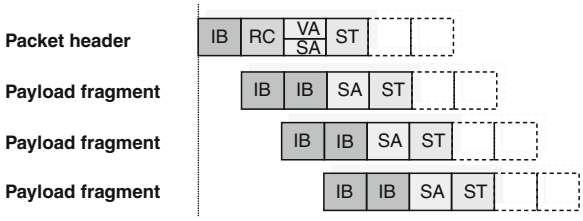


Switch Architecture. Fig. 3 Typical five-stage pipelined switch architecture for a wormhole switch

This basic pipelined architecture can be enhanced with different techniques, most of them trying to reduce the number of stages, thus also reducing the delay of traversing a switch. Routers with three or even two stages can be found in the literature. Also, single-cycle switches are common in some environments, like networks-on-chip (NoCs) for low-end systems-on-chip (SoCs). Obviously, such reduction in the number of stages should be achieved by not incurring in an excessive increase of the cycle time. Two basic procedures exist to reduce the number of stages, the first one by using speculation and performing actions in parallel, and the second one by performing computations ahead of time to remove the operation from the critical path.

Virtual channel allocation can be performed speculatively and in parallel with switch allocation in wormhole switching. Notice that this is done only for header flits. If both resources are obtained (the virtual channel is successfully assigned and the output port is granted) then the wormhole switch saves one cycle. If any of the two fails (or both fail) then the pipeline stalls and at the next cycle the operation is repeated. Figure 4 shows the case.

A further reduction in cycles would be to speculatively send the flit through the crossbar to the output port (ST) at the same time. To achieve this, internal speedup is required. Finally, the output port computation can be performed at the previous switch and stored at the head flit. So, when the flit reaches the next switch the output port to take is already computed and thus, there is no need for the RC stage. In that case, the output port computed for the next switch can be done in parallel with the VA stage, thus further reducing the pipeline depth.



Switch Architecture. Fig. 4 Four-stage pipelined wormhole switch design

High-Radix Switch Architectures

As identified in [5], during the last decades, the pin bandwidth of a switch has increased exponentially (from 64 Mb/s of the Torus routing chip in 1986 to the 1Tb/s of the recent Velio 3003). This is due to the increase in the signaling speed and the increase in the number of signals. In this sense, high-radix switches with narrow channels are able to achieve lower packet latencies than low-radix switches with wide channels. The explanation is simple. With high-radix switches the hop count in the network decreases. Additional benefits from high-radix switches are a lower cost and lower power consumption (as the total number of switches and links to build a network is reduced). Following this trend, there are some proposals for high-radix switch architectures, [5–7].

However, designing high-radix switches presents major challenges. The most important one is to keep a high switch efficiency with an affordable cost. The cost of a high-radix switch will largely depend on three key components: memory resources, arbiter logic, and internal connection logic. Depending on the location of memories in the switch, different switch organizations (memory and crossbar capabilities and their interconnects) have been used. In some of them, the number of memories increases quadratically with the number of ports. Also, arbiters and crossbars must cope with more candidates and connections, and for that reason become expensive. As an example, in on-chip networks, the use of high-radix switches is not appropriate due to the increase in power consumption and reduction in switch-operating frequency [8].

Related Entries

►Flow Control
►Switching Techniques

Bibliographic Notes and Further Reading

Two basic books exist for interconnection networks. Both describe in detail the concept of switch architecture. The first one, [2] describes a wide range of routers, some with wormhole switching (Intel Teraflops router,

Cray 3TD and 3TE routers, Reliable router and SGI spider), others with virtual cut-through switching (Chaos router, Arctic router, R2 router, Alpha 21364 router), and others with circuit switching (Intel iPSC Direct Connect Module). Also, the book describes the Myrinet switch.

The other book [1], provides as a case study the Alpha 21364 router and the IBM Colony router.

In on-chip networks building an efficient switch is critical. In addition to delay constraints, designing an on-chip switch has also power consumption and area limitations. In [9] basic design rules for building an on-chip router are provided. Also, literature is populated with many switch/router architectures for on-chip networks.

Bibliography

1. Dally W, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, CA
2. Duato J, Yalamanchili S, Ni N (2002) Interconnection networks: an engineering approach. Morgan Kaufmann, San Francisco, CA
3. Karol MJ et al (1987) Input versus output queueing on a space-division packet switch. IEEE Trans Commun COM-35(12):1347–1356
4. Tamir Y, Frazier GL (1988) High-performance multi-queue buffers for vlsi communications switches. SIGARCH Comput Archit News 16(2):343–354
5. Kim J, Dally WJ, Towles B, Gupta AK (2005) Microarchitecture of a high-radix router. In: 32nd Annual International Symposium on Computer Architecture (ISCA '05), Madison, WI, pp 420–431
6. Scott S, Abts D, Kim J, Dally WJ (2006) The blackwidow high-radix clos network. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA), The Washington, DC, June 2006
7. Mora G, Flich J, Duato J, López P, Baydal E, Lysne O (2006) Towards and efficient switch architecture for high-radix switches. In: Proceedings of ANCS 2006, San Jose, CA
8. Pullini A, Angiolini F, Murali S, Atienza D, De Micheli G, Benini G (2007) Bringing NOCs to 65 nm. IEEE Micro 27(5):75–85
9. de Micheli G, Benini L (2006) Networks on chips: technology and tools. Morgan Kaufmann, San Francisco, CA

Switched-Medium Network

►Buses and Crossbars

Switching Techniques

Sudhakar Yalamanchili
Georgia Institute of Technology, Atlanta, GA, USA

Definition

Switching techniques determine how messages are forwarded through the network. Specifically, these techniques determine how and when buffers and switch ports of individual routers are allocated and released and thereby the timing with which messages or message components can be forwarded to the next router on the destination path.

Discussion

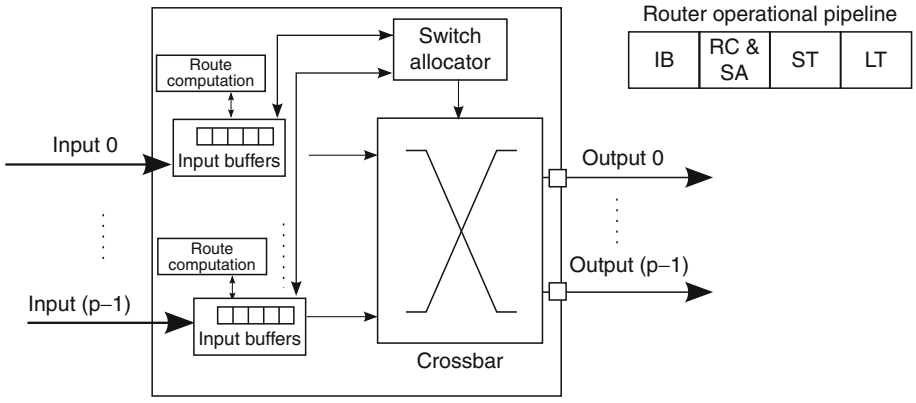
Introduction

This section introduces basic switching techniques used within the routers of multiprocessor interconnection networks. Switching techniques determine *when* and *how* messages are forwarded through the network. These techniques determine the granularity and timing with which resources such as buffers and switch ports are requested and released and consequently determine the blocking behavior of routing protocols that utilize them in different network topologies. As a result, they are key determinants of the deadlock properties of routing protocols. Further, their relationship to flow control protocols and traffic characteristics significantly impact the latency and bandwidth characteristics of the network.

A Generic Router Model

Switching techniques are understood in the context of routers used in multiprocessor interconnection networks. A simple generic router architecture is illustrated in Fig. 1. This router microarchitecture presents to messages a four stage pipeline comprised of the following stages.

- *Input Buffering (IB)*: Message data is received into the input buffer.
- *Route Computation (RC) and Switch Allocation (SA)*: Based on the message destination, a switch output port is computed, requested, and allocated.



Switching Techniques. Fig. 1 A generic router model

- *Switch Traversal (ST)*: Message data traverses the switch to the output buffer.
- *Link Traversal (LT)*: The message data traverses the link to the next router.

The end-to-end latency experienced by a message depends on how the switching techniques interact with this pipeline. This generic router architecture has been enhanced over the years with virtual channels [1], speculative operation [2, 3], flexible arbiters, effective channel and port allocators [4], deeper pipelines, and a host of buffer management and implementation optimizations (e.g., see [5, 6]). The result has been a range of router designs with different processing pipelines.

For the purposes of this discussion, here it is assumed that all of the pipeline stages take the same time – one cycle. This is adequate to define, distinguish, and compare properties of basic switching techniques. The following section addresses some basic concepts governing the operation and implementation of switching techniques based on the generic router model shown in Fig. 1. The remainder of the section is devoted to a detailed presentation of alternative switching techniques.

Basic Concepts

The performance and behavior of switching techniques are enabled by the low-level flow control protocols used for the synchronized transfer of data between routers. Flow control determines the granularity with which data is moved through the network and consequently when routing decisions can be made, when switching operations can be initiated, and how (at what granularity) data is transferred.

Flow control is the synchronized transfer of a unit of data between a sender and a receiver and ensures the availability of sufficient buffering at the receiver to avoid the loss of data. Selection of the unit of data transfer is based on a few simple concepts. A message is partitioned into fixed-length *packets*. Packets are individually routable units of data and are comprised of control bits packaged as the header and data bits packaged as the body. Packets are typically terminated with some bits for error detection such as a checksum. The header contains destination information used by the routers to select the onward path through the network. The packet as a whole is partitioned into fixed size units corresponding to the unit of transfer across a link or across a router and is referred to as a flow control digit or *flit* [7]. A flit becomes the unit of buffer management for transmission across a physical channel. Many schemes have been developed over the years for flow control including on-off, credit-based, sliding window, and flit reservation. The physical transfer of a flit across a link may in fact rely on synchronized transfers of smaller units of information. For example, consider flit sizes of 4 bytes and a physical channel width of 8 bits. The transfer of a flit across the physical link requires the synchronized transfer of 8-bit quantities referred to as a physical digit or *phit* [4] using phit-level flow control. In contrast to flits which represent units of buffer management, phits correspond to quantities reflecting a specific physical link implementation. Both flit-level and phit-level flow control are atomic in the sense that in the absence of

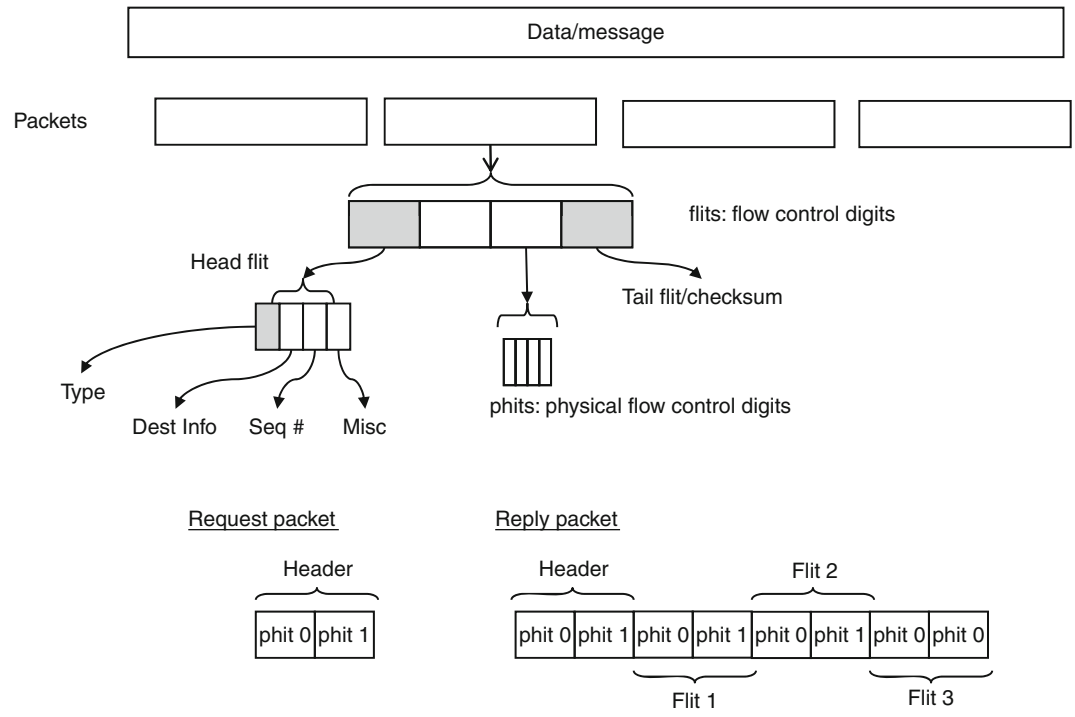
errors, all transfers will complete successfully and are not interleaved with other transfers. While phit sizes between chips or boards tend to be small, e.g., 8–32 bits, phit sizes on-chip can be much larger, e.g., 128 bits. Typical message sizes can range from 8–12 bytes (control messages) to 64–128 bytes (for example cache lines), to much larger sizes in message-passing parallel architectures. The preceding hierarchy of units is illustrated in Fig. 2 along with an example of a packet format [4, 8]. The actual content of a packet header will depend on the specifics of an implementation such as the routing protocol (e.g., destination address), flow control strategy (e.g., credits), use of virtual channels (e.g., virtual channel ID), and fault tolerance strategy (e.g., acknowledgements).

Switching techniques determine when messages are forwarded through the network and differ in the relative timing of flow control operations, route computation, and data transfer. High-performance switching techniques seek as much overlap as possible between these operations while reliable communication protocols may seek less concurrency between these operations in the interest of efficient error recovery. For example, one can wait until the entire packet is received before a request is made for the output port of the router's switch (packet switching). Alternatively, the request can be made as soon as all flits corresponding to the header are received (cut-through switching) but before the rest of the packet has been received.

The remainder of this section will focus on the presentation and discussion of switching techniques under the assumption of a flit-level flow control protocol and a supporting phit-level flow control protocol. It is common to have the flit size the same as the phit size.

Basic Switching Techniques

As one might expect, switching techniques have their roots in traditional digital communication and have evolved over the years to address the unique requirements of multiprocessor interconnection networks. For the purposes of comparison, the no-load latency is computed for an L-bit message. The phit size and flit size are assumed to be equivalent and equal to the physical data channel width of W bits, which is also the width of the internal datapath of the router. The routing header is assumed to be one flit, thus the message size is $L + W$



Switching Techniques. Fig. 2 Basic concepts and an example packet format

bits. A router can make a routing decision in one cycle and a flit can traverse a switch or link in one cycle as described with respect to Fig. 1.

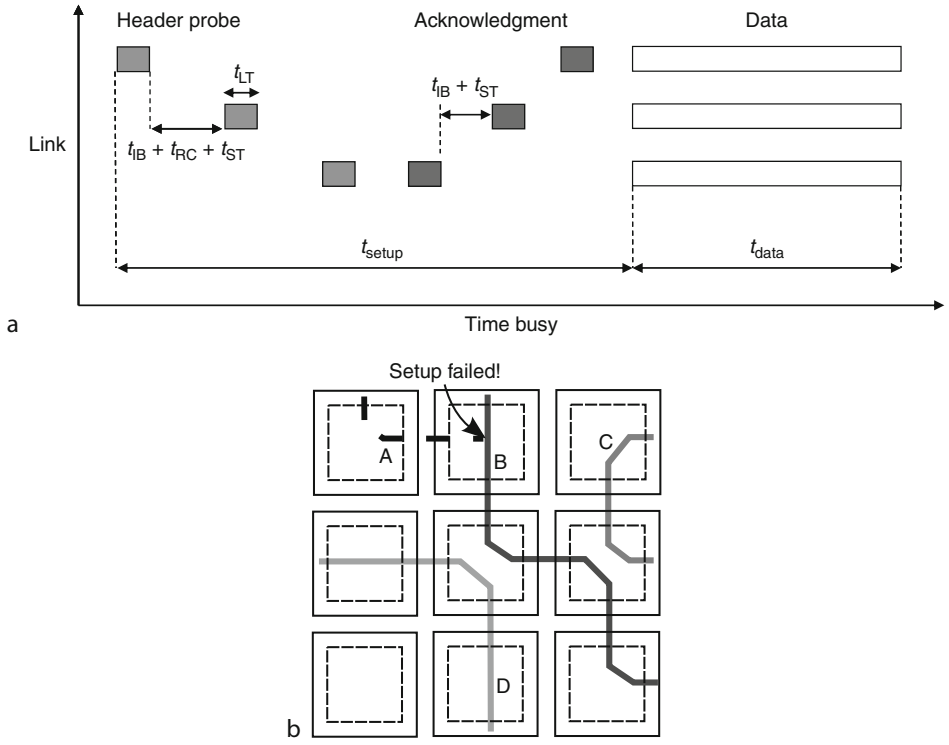
Circuit Switching

Circuit switching evolved from early implementations in telephone switching networks. In circuit switching, a physical path from the source network interface to the destination network interface is reserved prior to the transmission of the data. The source injects a routing packet commonly called a probe into the network. The probe is routed to the destination as specified by the routing protocol reserving the physical channels and switch ports along the way. An acknowledgement is returned to the source node to confirm path reservation. Figure 3b shows several circuits that have been set up and one in the process of being set up. While circuits B, C, D have been set up, circuit A is blocked from being set up by circuit B.

The base no-load latency of a circuit-switched message is determined by the sum of the time to set up a path and the time to transmit data. For a circuit that traverses D routers to the destination and operates at B Hz, the no-load message latency can be represented as follows.

$$\begin{aligned} t_{circuit} &= t_{setup} + t_{data} \\ t_{setup} &= D[t_{RC} + 2(t_{IB} + t_{ST} + t_{LT})] \\ t_{data} &= \frac{1}{B}\left\lceil \frac{L}{W} \right\rceil \end{aligned} \tag{1}$$

The notation in the expression follows the generic router model shown in Fig. 1, and the expression does not include the time to inject the probe into the router at the source (a link traversal) or the time to inject the acknowledgement into the router at the destination. The subscripts correspond to the pipeline stages illustrated in Fig. 1. For example, t_{RC} is the time taken to compute the output port to be traversed by the message at a router. This computation is carried out in the same cycle as the process of requesting and allocating a switch port, and therefore the switch allocation (SA) time is hidden and does not appear in the expression. The terms t_{IB}, t_{ST}, and t_{LT} represent the times for input buffering, switch traversal, and link traversal,



Switching Techniques. Fig. 3 Circuit switching. (**a**) Time-space utilization across two routers, (**b**) an example

respectively, experienced by the probe and the acknowledgement. The factor of 2 in the expression for path setup reflects the forward progress of the probe and the return progress of the acknowledgement – assuming the acknowledgement traverses the same path in the reverse direction. Note the acknowledgements do not have to be routed and therefore do not experience any routing delays at intermediate routers. A time-space diagram in Fig. 3a depicts the setup and operation of a circuit that crosses two routers.

Nominally, the routing probe contains the destination address and additional control bits used by the routing protocol and is buffered at intermediate routers where it is processed to reserve links and set router switch settings. On reaching the destination, an acknowledgement packet is transmitted back to the source. The hardware circuit has been set up and data can be transmitted at the full speed of the circuit. Flow control for data transfer is exercised end-to-end across the circuit and the flow control bandwidth (rate of signaling) should be at least as fast as the transmission speeds to avoid slowing down the hardware circuit. When transmission is complete, a few control bits traversing the circuit from the source release link and switch resources along the circuit. These bits may take the form of a small packet or with suitable channel design and message encoding, may be transmitted as part of the last few bits of the message data. Routing and data transmission functions are disjoint operations where all switches on the source-destination path are set prior to the transmission of any data.

Circuit switching is generally advantageous when messages are infrequent and long compared to the size of the routing probe. It is also advantageous when internode traffic exhibits a high degree of temporal locality. After a path has been set up, subsequent messages to the same destination can be transmitted without incurring the latency of path set-up times. The inter-node throughput is also maximized since once a path has been set up, messages to the destination are not routed and do not block in the network. The disadvantages are the same as those typically associated with most reservation-based protocols. When links and switch ports are reserved for a duration they prevent other traffic from making progress if they need to use any of the resources reserved by the established circuit. In particular, a probe can be blocked during circuit set up while waiting for another circuit to be torn down. The links reserved by the probe up to that point can similarly prevent other circuits from being established.

Wire delays place a practical limit on the speed of circuit switching as a function of system size, motivating the development of techniques to mitigate or eliminate end-to-end wire delay dependencies. One set of techniques evolved around pipelining multiple bits on the wire essentially using a long latency wire as a deeply pipelined transmission medium. Such techniques have been referred to as *wave pipelining* or *wave switching* [9–11]. These techniques maximize wire bandwidth while reducing sensitivity to distance. However, when used in anything other than bit serial channels, pragmatic constraints of signal skew and stable operation across voltage and temperature remain challenges to widespread adoption. Alternatively, both the wire length issue and blocking are mitigated by combining circuit switching concepts with the use of virtual channels. Virtual channel buffers at each router are reserved by the routing probe setting up a pipelined virtual circuit from source to destination in a manner referred to as pipelined circuit switching [12]. This approach increases physical link utilization by enabling sharing across circuits although the available bandwidth to each circuit is now reduced. This approach was also used in the Intel iWarp chip that was designed to support systolic communication through *message pathways*: long-lived communication paths [13]. Rather than set up and remove network paths each time data are to be communicated, paths through the network persist for long periods of time. Special messages called *pathway begin markers* are used to reserve virtual channels (referred to as *logical channels* in iWarp) and set up interprocessor communication paths. On completion of the computation, the paths are explicitly removed by other control messages.

Packet Switching

In circuit switching, routing and data transfer operations are separated. All switches in the path to the destination are set a priori, and all data is transmitted in a burst at the full bandwidth of the circuit. In packet switching, the message data is partitioned into fixed-sized packets. The first few bytes of a packet contain routing and control information and are collectively

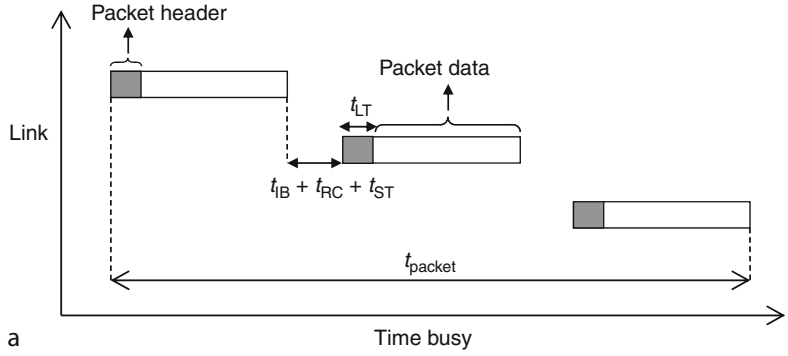
referred to as the *packet header* while the data is contained in the packet body. Each packet can now be independently routed through the network. Flow control between routers is at the level of a complete packet. A packet cannot be forwarded unless buffer space for a complete packet is available at the next router. A packet is then transferred in its entirety across a link to the input buffer of the next router. Only then is the header information extracted and used by the routing and control unit to determine the candidate output port. The switch can now be set to enable the packet to be transferred to the output port and then stored at the next router before being forwarded. This switching technique is also known as *store-and-forward* (SAF) switching. There is no overlap between flow control, routing operations, and data transfer. Consequently, the end-to-end latency of a packet is proportional to the distance between the source and destination nodes.

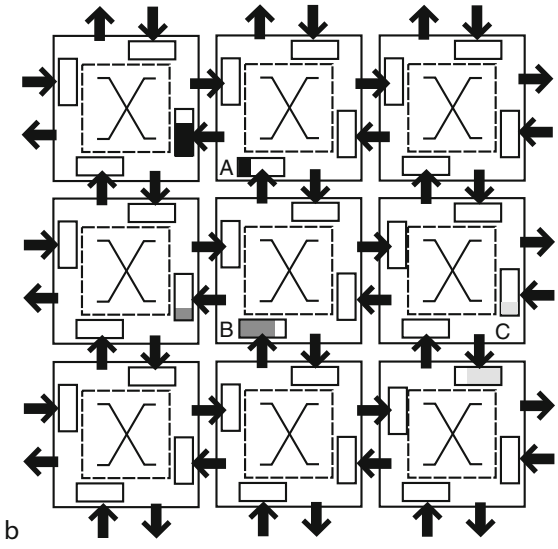
An example of packet switching is illustrated in Fig. 4b (links to local processors are omitted for brevity). Note how packets A, B, and C are in the process of being transferred across a link. Each input buffer is partially full pending receipt of the remainder of the packet before being forwarded.

The no-load latency for a packet transmission across D routers can be modeled as follows.

$$t_{packet} = D\left\{t_{RC} + (t_{IB} + t_{ST} + t_{LT})\left\lceil \frac{L+W}{W} \right\rceil\right\} \qquad (2)$$

This expression does not include the time to inject the packet into the network at the source. The expression follows the router model in Fig. 1 where entire packets must traverse the link or switch at each router.





Switching Techniques. Fig. 4 Packet switching. (**a**) Time-space utilization across three links, (**b**) an example

Therefore end-to-end latency is proportional to the distance between the source and destination.

Packet switching evolved from early implementations in data networks. Relative to circuit switching, packet switching is advantageous when the average message size is small. Since packets only hold resources as they are being used, this switching technique can achieve high link utilization and network throughput. Packets are also amenable to local techniques for error detection and recovery since all data and its associated routing information are encapsulated as a single locally available unit. However, the overhead per data bit is higher – each packet must invest in a header reducing energy efficiency as well as the proportion of physical bandwidth that is accessible to actual data transfer. If a message is partitioned into multiple packets and adaptive routing is employed, packets may arrive at the destination out of order necessitating investments in reordering mechanisms. Packet-switched routers have also been designed with dynamically allocated centralized queues rather than keeping the messages buffered at the router input and output resulting in both cost and power advantages.

Virtual Cut-Through (VCT) Switching

Virtual cut-through (VCT) switching is an optimization for packet switching where in the absence of congestion, packet transfer is pipelined. Like the preceding techniques, VCT has its genesis in packet data networks [14]. Flow control is still at the packet level. However, packet transfer is overlapped with flow control and routing operations as follows. Routing can begin as soon as the header bytes of a packet have arrived at the input buffer and before the rest of the packet has been received. In the absence of congestion, switch allocation and switch traversal can proceed and the forwarding of the packet through the switch as well as flow control requests to the next router can begin. Thus, packet transfer can be pipelined through multiple routers. For example, consider a 128 byte packet with an 8 byte header. After the first 8 bytes have been received, routing decisions and switch allocation can be initiated. If the switch output port is available, then the router can begin forwarding bytes to the output port before the remainder of the packet has arrived and can *cut-through* to the next router. In the absence of blocking in the network, the latency for the header to arrive at the destination network interface is proportional to the distance between the source and destination. Thereafter a phit can exit the network every cycle. If the header is blocked on a busy output channel at an intermediate router, the packet is buffered at the router – a consequence of the fact that flow control is at the level of a packet. Thus, at high network loads, VCT switching behaves like packet switching. An example of VCT at work is illustrated in Fig. 5b. Packet A is blocked by packet B. Note that packet A has enough buffer space to be fully buffered at the local router. Packet B can be seen to be spread across multiple routers as it is pipelined through the network.

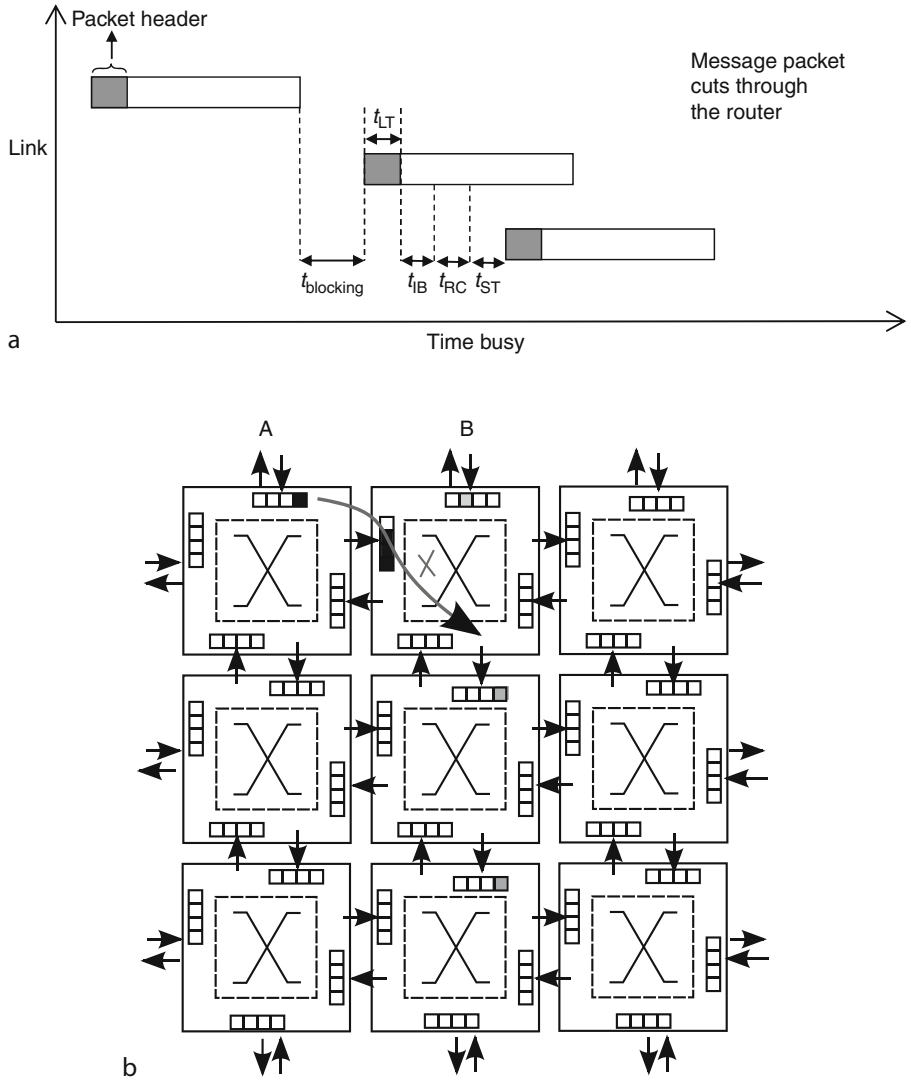
The no-load latency of a message that successfully cuts through D intermediate routers is captured in the following expression.

$$t_{VCT} = D(t_{IB} + t_{RC} + t_{ST} + t_{LT}) + t_{IB}\left\lceil \frac{L}{W} \right\rceil \qquad (3)$$

This expression does not include the time to inject the packet into the network at the source. The first term in the equation is much smaller than the second term (recall that each value of delay is one pipeline cycle). Therefore, under low load conditions, the message latency is approximately proportional to the packet size rather than distance between source and destination. However, when a packet is blocked, the packet is buffered at the router. Thus, at high network loads, the behavior of VCT approximates that of packet switching. At low loads, the performance improves considerably approaching that of wormhole switching which is described next.

Wormhole Switching

The need to buffer complete packets within a router can make it difficult to construct small, compact, and fast routers. In the multiprocessor machines of the 1980s, interconnection networks employed local node memory as storage for buffering blocked packets. This ejection and re-injection of packets incurred significant latency penalties. It was desirable to keep packets in the network. However, there was insufficient buffering within individual routers. Wormhole switching evolved as a small buffer optimization of virtual cut-through where packets were pipelined through routers. The buffers in each router had enough storage for several flits. When a packet header blocks, the message occupies buffers in several routers. For example, consider



Switching Techniques. Fig. 5 Virtual cut-through switching. (**a**) Time-space utilization across three links, (**b**) an example

the message pattern shown in Fig. 6b with routers with one flit buffers. Message A is blocked by message B and occupies buffers across multiple routers leading to secondary blocking across multiple routers. The time-space diagram illustrates how packets are pipelined across multiple routers significantly reducing the sensitivity of message latency to distance.

When it was introduced [7, 15], the pipelined behavior of wormhole switching led to relatively large reductions in message latency at low loads. Further gains derived from not having to eject messages from the network for storage. The use of small buffers also has two physical consequences. First, smaller buffers lead to lower access latency and shorter pipeline stage time (see Fig. 1). The smaller pipeline delay enables higher clock rates and consequently high bandwidth routers, for example, 5 GHz in today's Intel TeraFlops router [16]. Second, the smaller buffers also reduce static energy consumption which is particularly important in the context of on-chip routers. However, as the offered communication load increases, messages block in place occupying buffers across multiple routers and the links between them. This leads to secondary blocking of messages that share any of these links which

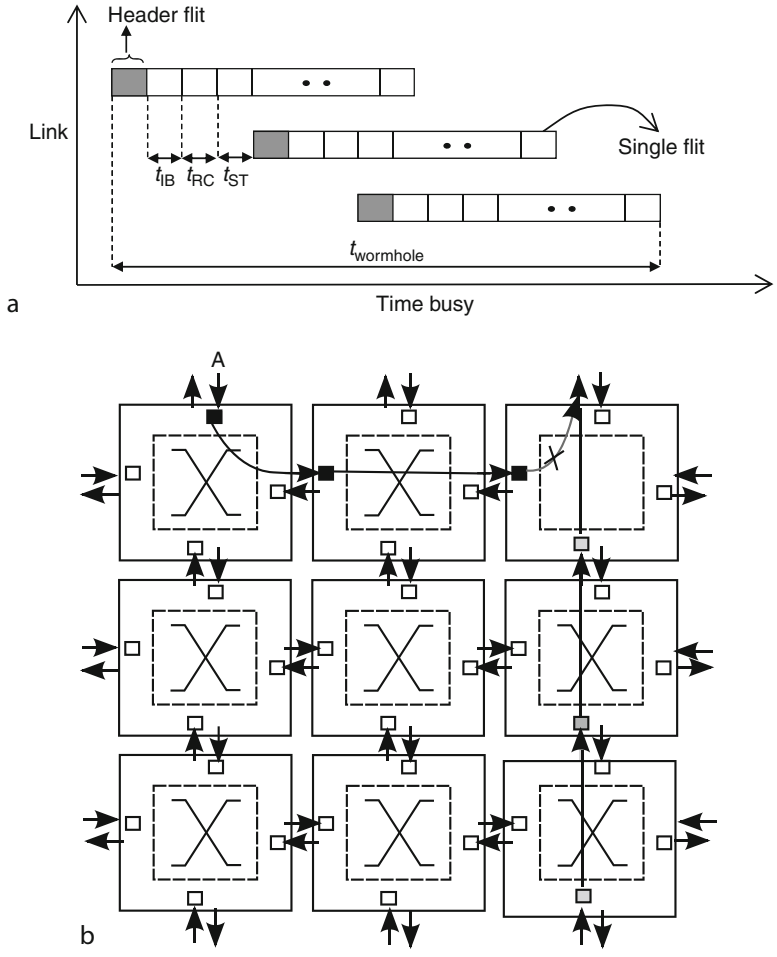
in turn propagates congestion further. The disadvantage of wormhole switching is that blocked messages hold physical channel resources. Routing information is only associated with a few header flits. Data flits have no routing information associated with them. Consequently, when a packet is blocked in place, packet transmission cannot be interleaved over a physical link without additional support (such as virtual channels – see Section on Virtual Channels) and physical channels cannot be shared. The result is the rapid onset of saturation as offered load increases. Virtual channel flow control was introduced to alleviate this problem.

The key deadlock issue is that a single message produces dependencies between buffers across multiple routers. Routing protocols must be designed to ensure that such dependencies are not composed across multiple messages to produce deadlocked configurations of messages. Deadlock freedom requirements for deterministic routing protocols are described in [7] while proofs for adaptive routing protocols are described in [17–19].

The base latency of a wormhole-switched message crossing D routers with the flit size equal to the phit size can be computed as follows.

$$t_{wormhole} = D(t_{IB} + t_{RC} + t_{ST} + t_{LT}) + t_{IB}\left\lceil \frac{L}{W} \right\rceil \quad (4)$$

This expression does not include the time to inject flits into the network at the source. After the first flit arrives at the destination, each successive flit is delivered



Switching Techniques. Fig. 6 Wormhole switching. (**a**) Time-space utilization across three links, (**b**) an example

in successive clock cycles. Thus, for message sizes that are large relative to the distance between sender and destination, the no-load latency is approximately a function of the message size rather than distance.

Several optimizations have been proposed to further improve the performance of wormhole switching. One example is flit reservation flow control was introduced to improve buffer turn-around time in wormhole switching [20]. Deeply pipelined, high speed routers can lead to low buffer occupancy as a consequence of propagation delays of flits over the link and the latency in receiving and processing credits before the buffer can be reused. Flit reservation is a technique that combines some elements of circuit switching (a priori reservations) to improve performance. A control flit advances ahead of the data flits of a message to reserve buffer and channel resources. Note that router pipeline for data flits is much shorter (they do not experience routing and switch allocation delays). As a result, reservations and transfers can be overlapped and buffer occupancy is significantly increased.

Another example that combines the advantages of wormhole switching and packet switching is *buffered wormhole switching (BWS)*. This was proposed and used in IBM's Power Parallel SP systems [21, 22]. In the absence of blocking, messages are routed through the network using wormhole switching. When messages block, 8-flit *chunks* are constructed at the input port of a switch and buffered in a dynamically allocated centralized router memory freeing up the input port for use by other messages. Subsequently, buffered chunks are transferred to an output port where they are converted to a flit stream for transmission across the physical channel. BWS differs from wormhole switching in that flits are not buffered in place. Rather flits are aggregated and buffered in a local memory within the switch and in this respect BWS is similar to packet switching. The no-load latency of a message routed using BWS is identical to that of wormhole-switched messages.

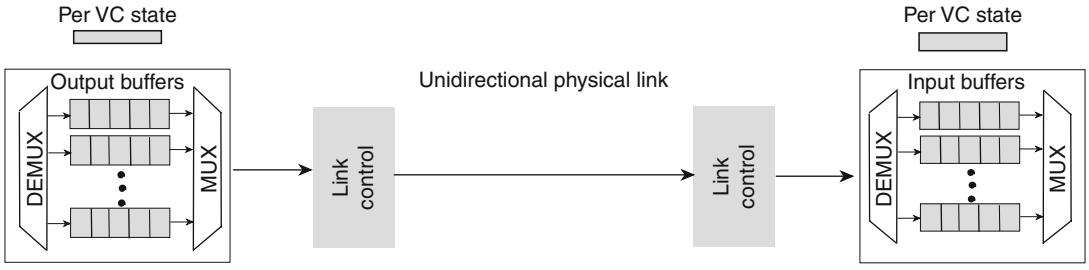
Virtual Channels

An important interconnection architecture function is the use of virtual channel flow control [1]. Each unidirectional virtual channel across a physical link is realized by an independently managed pair of message buffers. Multiple virtual channels are multiplexed across the physical link increasing link utilization and network throughput. Importantly, routing constraints on the use of virtual channels are commonly used to ensure deadlock freedom. The use of virtual channels decouples message flows from the physical links and their use is orthogonal to the operation of switching techniques. Each switching technique is now employed to regulate the flow of packet data within a virtual channel while constraints on virtual channel usage may govern routing decisions at intermediate routers. The microarchitecture pipeline of the routers now includes an additional stage for virtual channel allocation. Virtual channels have been found to be particularly useful for optimizing the performance of wormhole-switched routers ameliorating the consequences of blocking and thus broadening the scope of application of wormhole switching. A simple example of the operation of virtual channel flow control is illustrated in Fig. 7.

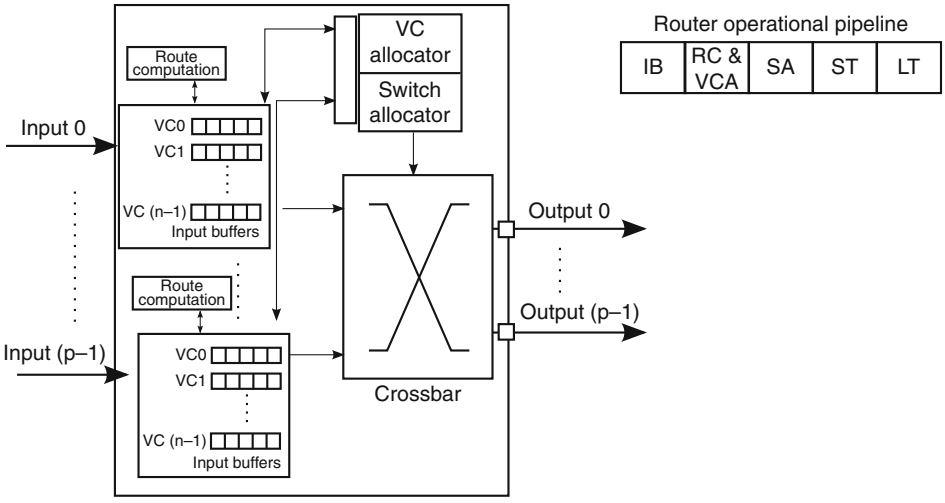
A modified router architecture reflecting the use of virtual channels is shown in Fig. 8. The route computation operation now returns a set of virtual channels that are candidates for forwarding the message (note that the router might be employing adaptive routing). A typical router pipeline is now extended by an extra stage as shown – *virtual channel allocation*. This is implemented prior to requesting an output port of the switch extending the router pipeline of Fig. 1 by one stage.

A Comparison of Switching Techniques

Switching techniques have fundamental differences in their ability to utilize network bandwidth. In packet switching and VCT switching, messages are partitioned into fixed length packets each with its own header. Consequently, the overhead per transmitted data byte of a message is a fixed function of the message length. Wormhole switching supports variable sized messages and consequently overhead per data byte decreases with message size. However, as the network load increases when wormhole-switched messages block, they hold links and resources across multiple routers, wasting physical bandwidth, propagating congestion, and saturating the network at a fraction of the peak. The use of virtual channels in wormhole switching decouples the physical channel from blocked messages improving network utilization but increases the flow control latency across the physical link as well as the complexity of the channel controllers and intra-router switching.



Switching Techniques. Fig. 7 Virtual channel flow control



Switching Techniques. Fig. 8 The generic router with virtual channels

In VCT switching and packet switching, packets are fully buffered at each router and therefore traffic consumes network bandwidth in proportion to network load at the expense of increased amount of in-network buffering.

The latency behavior of packets using different switching techniques exhibits distinct behaviors. At low loads, the pipelined behavior of wormhole switching produces superior latency characteristics. However, saturation occurs at lower loads and the variance in packet latency is higher and accentuated with variance in packet length. The latency behavior of packets under packet switching tends to be more predictable since messages are fully buffered at each load. VCT packets will operate like wormhole switching at low loads and approximate packet switching at high loads where blocking will force packets to be buffered at an intermediate node. Consequently, approaches to provide Quality of Service guarantees (QoS) typically utilize VCT or packet switching. Attempting to control QoS when blocked messages are spread across multiple nodes is by comparison much more difficult.

Reliability schemes are shaped by the switching techniques. Alternative topologies and routing algorithms affect the probability of encountering a failed component. The switching technique affects feasible detection and recovery algorithms. For example, packet switching is naturally suited to link level error detection and retransmission since each packet is an independently routable unit. For the same reason, packets may be adaptively routed around faulty regions of the network. However, when messages are pipelined over several links, error recovery and control becomes complicated. Recall that data flits have no routing information. Thus, errors that occur within a message that is spread across multiple nodes can lead to buffers and channel resources that are indefinitely occupied (e.g., link transceiver failures) and can lead to deadlocked message configurations. Thus, link level recovery must be accompanied by some higher

level layer recovery protocols that typically operate end-to-end.

Finally, it can be observed that the switching techniques exert a considerable influence on the architecture of the router, and as a result, the network performance. For example, flit level flow control enabled pipelined message transfers as well as the use of small buffers. The combination resulted in higher flow control signaling speeds and small compact router pipeline stages that could be clocked at higher speeds. The use of wormhole switching precluded the need for larger (slower) buffers or costly use of local storage at a node – the message could remain the network. This is a critical design point if one considers that the link bandwidth can often exceed the memory bandwidth. Such architectural advances have amplified the performance gained via clock speed advances over several technology generations. For example, consider the difference in performance between the Cosmic cube network [23] that operated at 5 MHz and produced message latencies approaching hundreds of microseconds to milliseconds while the most recent TeraFlops chip from Intel operating at 5 GHz produces latencies on the order of nanoseconds. While performance has increased almost 5 orders of magnitude, the clock speeds have only increased by about 3 orders of magnitude. Much of this performance differential can be attributed to switching techniques and associated microarchitecture innovations that accompany their implementation.

Related Entries

►Collective Communication, Network Support for
►Congestion Management
►Flow Control
►Interconnection Networks
►Networks, Fault-Tolerant
►Routing (Including Deadlock Avoidance)

Bibliographic Notes and Further Reading

Related topics such as flow control and deadlock freedom are intimately related to switching techniques. A combined coverage of fundamental architectural, theoretical, and system concepts and a distillation of key concepts can also be found in two texts [4, 8]. More advanced treatments of these and related topics can be found in papers in most major systems and computer architecture conferences with the preceding texts contributing references to many seminal papers in the field.

Acknowledgments

Assistance and feedback from Mitchelle Rasquinha, Dhruv Choudhary, and Jeffrey Young are gratefully acknowledged.

Bibliography

1. Dally WJ (1992) Virtual-channel flow control. IEEE Trans Parallel Distrib Syst 3(2):194–205
2. Peh L-S, Dally WJ (2001) A delay model for router microarchitectures. IEEE Micro 21:26–34
3. Peh L-S, Dally WJ (2001) A delay model and speculative architecture for pipelined routers. In: Proceedings of the 7th international symposium on high-performance computer architecture, Nuevo Leone
4. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufman, San Francisco
5. Choi Y, Pinkston TM (2004) Evaluation of queue designs for true fully adaptive routers. J Parallel Distrib Comput 64(5):606–616
6. Mullins R, West A, Moore S (2004) Low-latency virtual-channel routers for on-chip networks. In: Proceedings of the 31st annual international symposium on computer architecture, Munchen
7. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. IEEE Trans Comput C-36(5):547–553
8. Duato J, Yalamanchili S, Ni L (2003) Interconnection networks: an engineering Approach. Morgan Kaufmann, San Francisco
9. Flynn M (1995) Computer architecture: pipelined and parallel processor design. Jones & Bartlett, Boston, pp 63–140
10. Duato J et al (1996) A high performance router architecture for interconnection networks. In: Proceedings of the 1996 international conference on parallel processing, Bloomington, vol I, August 1996, pp 61–68
11. Scott SL, Goodman JR (1994) The impact of pipelined channels on k-ary n-cube networks. IEEE Trans Parallel Distrib Syst 5(1):2–16
12. Gaughan PT et al (1996) Distributed, deadlock-free routing in faulty, pipelined, direct interconnection networks. IEEE Trans Comput 45(6):651–665
13. Borkar S et al (1988) iWarp: an integrated solutionto high-speed parallel computing. In: Proceedings of supercomputing ′88, Orlando, November 1988, pp 330–339
14. Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. Comp Networks 3(4):267–286
15. Dally WJ, Seitz CL (1986) The torus routing chip. J Distrib Comput 1(3):187–196
16. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a teraflops processor. IEEE Micro 27(5): 51–61

17. Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Trans Parallel Distrib Syst 4(12): 1320–1331
18. Duato J (1995) A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. IEEE Trans Parallel Distrib Syst 6(10):1055–1067
19. Duato J (1996) A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. IEEE Trans Parallel Distrib Syst 7(8):841–854
20. Peh L-S, Dally WJ (2000) Flit reservation flow control. In: Proceedings of the 6th international symposium on high-performance computer architecture, Toulouse, France, January 2000, pp 73–84
21. Stunkel CB et al (1994) Architecture and implementation of vulcan. In: Proceedings of the 8th international parallel processing symposium, Cancun, Mexico, pp 266–274
22. Stunkel CB et al (1994) The SP1 high-performance switch. In: Proceedings of the scalable high performance computing conference, Knoxville, pp 150–157
23. Seitz C (1985) The cosmic cube. Commun ACM 28(1):22–23

Symmetric Multiprocessors

►Shared-Memory Multiprocessors

Synchronization

Michael L. Scott
University of Rochester, Rochester, NY, USA

Synonyms

Fences; Multiprocessor synchronization; Mutual exclusion; Process synchronization

Definition

Synchronization is the use of language or library mechanisms to constrain the ordering (interleaving) of instructions performed by separate threads, to preclude orderings that lead to incorrect or undesired results.

Discussion

In a parallel program, the instructions of any given thread appear to occur in sequential order (at least from that thread's point of view), but if the threads run independently, their sequences of instructions may interleave arbitrarily, and many of the possible interleavings may produce incorrect results. As a trivial example, consider a global counter incremented by multiple threads. Each thread loads the counter into a register, increments the register, and writes the updated value back to memory. If two threads load the same value before either stores it back, updates may be lost:

```
                c == 0
Thread 1:                    Thread 2:
    r1 := c
                                 r1:= c
    ++r1
                                 ++r1
    c := r1
                                 c := r1
                c == 1
```

Synchronization serves to preclude invalid thread interleavings. It is commonly divided into the subtasks of *atomicity* and *condition synchronization*. Atomicity ensures that a given sequence of instructions, typically performed by a single thread, appears to all other threads as if it had executed indivisibly – not interleaved with anything else. In the example above, one would typically specify that the load-increment-store instruction sequence should execute atomically.

Condition synchronization forces a thread to wait, before performing an operation on shared data, until some desired precondition is true. In the example above, one might want to wait until all threads had performed their increments before reading the final count.

While it is tempting to suspect that condition synchronization subsumes atomicity (make the precondition be that no other thread is currently executing a conflicting operation), atomicity is in fact considerably harder, because it requires *consensus* among all competing threads: they must all agree as to which will proceed and which will wait. Put another way, condition synchronization delays a thread until some locally observable condition is seen to be true; atomicity is a property of the system as a whole.

Like many aspects of parallel computing, synchronization looks different in shared-memory and message-passing systems. In the latter, synchronization is generally subsumed in the message-passing methods; in a shared-memory system, it typically employs a separate set of methods.

Shared-memory implementations of synchronization can be categorized as *busy-wait* (*spinning*), or *scheduler-based*. The former actively consume processor cycles until the running thread is able to proceed. The latter deschedule the current thread, allowing the processor to be used by other threads, with the expectation that future activity by one of those threads will make the original thread runnable again. Because it avoids the cost of two context switches, busy-wait synchronization is typically faster than scheduler-based synchronization when the expected wait time is short and when the processor is not needed for other purposes. Scheduler-based synchronization is typically faster when expected wait times are long; it is *necessary* when the number of threads exceeds the number of processors (else quantum-long delays or even deadlock can occur). In the typical implementation, busy-wait synchronization is built on top of whatever hardware instructions execute atomically. Scheduler-based synchronization, in turn, is built on top of busy-wait synchronization, which is used to protect the scheduler's own data structures (see entries on *Scheduling Algorithms* and on *Processes, Tasks, and Threads*).

Hardware Primitives

In the earliest multiprocessors, *load* and *store* were the only memory-access instructions guaranteed to be atomic, and busy-wait synchronization was implemented using these. Modern machines provide a variety of atomic *read-modify-write* (RMW) instructions, which serve to *update* a memory location atomically. These significantly simplify the implementation of synchronization. Common RMW instructions include:

Test-and-set (l) sets the Boolean variable at location l to *true*, and returns the previous value.

Swap (l, v) stores the value v to location l and returns the previous value.

Atomic-ϕ (l, v) replaces the value o at location l with $\phi(o, v)$ for some simple arithmetic function ϕ (add, sub, and, etc.).

Fetch-and-ϕ (l, v) is like atomic-ϕ, but also returns the previous value.

Compare-and-swap (l, o, n) inspects the value v at location l, and if it is equal to o, replaces it with n. In either case, it returns the previous value, from which one can deduce whether the replacement occurred.

Load-linked (l) and store-conditional (l, v). The first of these returns the value at location l and "remembers" l. The second stores v to l if l has not been modified by any other processor since a previous load-linked by the current processor.

These instructions differ in their expressive power. Herlihy has shown [9] that compare-and-swap (CAS) and load-linked / store-conditional (LL/SC) are *universal* primitives, meaning, informally, that they can be used to construct a *non-blocking* implementation of any other RMW operation. The following code provides a simple implementation of fetch-and-ϕ using CAS.

```
val old := *l;
loop
    val new := phi(old);
    val found := CAS(l, old, new);
    if (old == found) break;
    old := found;
```

If the test on line 5 of this code fails, it must be because some other thread successfully modified `*l`. The system as a whole has made forward progress, but the current thread must try again.

As discussed in the entry on Non-blocking Algorithms, this simple implementation is *lock-free* but not *wait-free*. There are stronger (but slower and more complex) non-blocking implementations in which each thread is guaranteed to make forward progress in a bounded number of its own instructions.

NB: In any distributed system, and in most modern shared memory systems, instructions executed by a given thread are not, in general, guaranteed to be seen in sequential order by other threads, and instructions of any two threads are not, in general, guaranteed to be seen in the same order by all of their peers. Modern processors typically provide so-called *fence* or *barrier* instructions (not to be confused with the barriers discussed under Condition Synchronization below) that force previous instructions of the current thread to be seen by other threads before subsequent instructions of the current thread. Implementations of synchronization

methods typically include sufficient fences that if synchronization method s_1 in thread t_1 occurs before synchronization method s_2 in thread t_2, then all instructions that precede s_1 in t_1 will appear in t_2 to have occurred before any of its own instructions that follow s_2. For more information, see the entry on Memory Models. The remainder of the discussion here assumes that memory is *sequentially consistent*, that is, that instructions appear to interleave in some global total order that is consistent with program order in every thread.

Atomicity

A multi-instruction operation is said to be *atomic* if appears to occur "all at once" from every other thread's point of view. In a sequentially consistent system, this means that the program behaves as if the instructions of the atomic operation were contiguous in the global instruction interleaving. More specifically, in any system, intermediate states of the atomic operation should never be visible to other threads, and actions of other threads should never become visible to a given thread in the middle of one of its own atomic operations.

The most straightforward way to implement atomicity is with a *mutual-exclusion (mutex) lock* – an abstract object that can be *held* by at most one thread at a time. In standard usage, a thread invokes the *acquire* method of the lock when it wishes to begin an atomic operation and the *release* method when it is done. *Acquire* waits (by spinning or rescheduling) until it is safe for the operation to proceed. The code between the acquire and release (the body of the atomic operation) is known as a *critical section*.

Critical sections that conflict with one another (typically, that access some common location, with at least one section writing that location) must be protected by the same lock. Programming discipline commonly ensures this property by associating data with locks. A thread must then acquire locks for all the data accessed in a critical section. It may do so all at once, at the beginning of the critical section, or it may do so incrementally, as the need for data is encountered. Considerable care may be required to ensure that locks are acquired in the same order by all critical sections, to avoid deadlock. All locks are typically held until the end of the critical section. This *two-phase locking* (all acquires occur before any releases) ensures that the global set of critical section executions remains *serializable*.

Relaxations of Mutual Exclusion

So-called *reader–writer locks* increase concurrency by observing that it is safe for more than one thread to read a location concurrently, so long as no thread is modifying that location. Each critical section is classified as either a reader or a writer of the data associated with a given lock. The *reader_acquire* method waits until there is no concurrent writer of the lock; the *writer_acquire* method waits until there is no concurrent reader *or* writer.

In a standard reader–writer lock, a thread must know, when it first reads a location, whether it will ever need to write that location in the current critical section. In some contexts it may be possible to relax this restriction. The Linux kernel, for example, provides a *sequence lock* mechanism that allows a reader to *abort* its peers and upgrade to writer status. Programmers are required to follow a restrictive programming discipline that makes critical sections "restartable," and checks, before any write or "dangerous" read, to see whether a peer's upgrade has necessitated a restart.

For data structures that are almost always read, and very occasionally written, several operating system kernels provide some variant of a mechanism known as *RCU* (originally an abbreviation for read-copy update). RCU divides execution into so-called *epochs*. A writer creates a new copy of any data structure it needs to update. It replaces the old copy with the new, typically using a single CAS instruction. It then waits until the end of the current epoch to be sure that all readers that might have been using the old copy have completed their critical sections (at which point it can reclaim the old copy, or perform other actions that depend on the visibility of the update). The advantage of RCU, in comparison to locks, is that it imposes *zero overhead* in the read-only case.

For more general-purpose use, *transactional memory* (TM) allows arbitrary operations to be executed atomically, with an underlying implementation based on *speculation* and *rollback*. Originally proposed [10] as a hardware assist for lock-free data structures – sort of a multi-word generalization of LL/SC – TM has seen a flurry of activity in recent years, and several hardware

and software implementations are now widely available. Each keeps track of the memory locations accessed by transactions (would-be atomic operations). When two concurrent transactions are seen to conflict, at most one is allowed to *commit*; the others *abort*, "roll back," and try again, using a fully automated, transparent analogue of the programming discipline required by sequence locks. For further details, see the separate entry on TM.

Fairness

Because they sometimes force multiple threads to wait, synchronization mechanisms inevitably raise issues of *fairness*. When a lock is released by the current holder, which waiting thread should be allowed to acquire it? In a system with reader–writer locks, should a thread be allowed to join a group of already-active readers when writers are already waiting? When transactions conflict in a TM system, which should be permitted to proceed, and which should wait or abort?

Many answers are possible. The choice among conflicting threads may be arbitrary, random, first-come-first-served (FIFO), or based on some other notion of priority. From the point of view of an individual thread, the resulting behavior may range from potential *starvation* (no progress guarantees) to some sort of proportional share of system run time. Between these extremes, a thread may be guaranteed to run eventually if it is continuously ready, or if it is ready infinitely often. Even given the possibility of starvation, the system as a whole may be *livelock-free* (guaranteed to make forward progress) as a result of algorithmic guarantees or pseudo-random heuristics. (Actual livelock is generally considered unacceptable.) Any starvation-free system is clearly livelock free.

Simple Busy-Wait Locks

Several early locking algorithms were based on only loads and stores, but these are mainly of historical interest today. All required $\Omega(tn)$ space for t threads and n locks, and $\omega(1)$ (more-than-constant) time to arbitrate among threads competing for a given lock.

In modern usage, the simplest constant-space, busy-wait mutual exclusion lock is the *test-and-set (TAS) lock*, in which a thread acquires the lock by using a test-and-set instruction to change a Boolean flag from false to true. Unfortunately, spinning by waiting threads tends to induce extreme contention for the lock location, tying up bus and memory resources needed for productive work. On a cache-coherent machine, better performance can be achieved with a "test-and-test-and-set" (TATAS) lock, which reduces contention by using ordinary load instructions to spin on a value in the local cache so long as the lock remains held:

```
type lock = Boolean;

proc acquire(lock *l):
     while (test-and-set(l))
        while (*l) /* spin */ ;

proc release(lock *l):
    *l := false;
```

This lock works well on small machines (up to, say, four processors).

Which waiting thread acquires a TATAS lock at release time depends on vagaries of the hardware, and is essentially arbitrary. Strict FIFO ordering can be achieved with a *ticket lock*, which uses fetch-and-increment (FAI) and a pair of counters for constant space and (per-thread) time. To acquire the lock, a thread atomically performs an FAI on the "next available" counter and waits for the "now serving" counter to equal the value returned. To release the lock, a thread increments its own ticket, and stores the result to the "now serving" counter. While arguably fairer than a TATAS lock, the ticket lock is more prone to performance anomalies on a multiprogrammed system: if any waiting thread is preempted, all threads behind it in line will be delayed until it is scheduled back in.

Scalable Busy-Wait Locks

On a machine with more than a handful of processors, TATAS and ticket locks scale poorly, with time per critical section growing linearly with the number of waiting threads. Anderson [1] showed that exponential backoff (reminiscent of the Ethernet contention-control algorithm) could substantially improve the performance of TATAS locks. Mellor-Crummey and Scott [17] showed similar results for linear backoff in ticket locks (where

a thread can easily deduce its distance from the head of the line).

To eliminate contention entirely, waiting threads can be linked into an explicit queue, with each thread spinning on a separate location that will be modified when the thread ahead of it in line completes its critical section. Mellor-Crummey and Scott showed how to implement such queues in total space $O(t + n)$ for t threads and n locks; their *MCS lock* is widely used in large-scale systems. Craig [4] and, independently, Landin and Hagersten [16] developed an alternative *CLH lock* that links the queue in the opposite direction and performs slightly faster on some cache-coherent machines. Auslander et al. developed a variant of the MCS lock that is API-compatible with traditional TATAS locks [3]. Kontothanassis et al. [14] and He et al. [8] developed variants of the MCS and CLH locks that avoid performance anomalies due to preemption of threads waiting in line.

Scheduler-Based Locks

A busy-wait lock wastes processor resources when expected wait times are long. It may also cause performance anomalies or deadlock in a multiprogrammed system. The simplest solution is to *yield* the processor in the body of the spin loop, effectively moving the current thread to the end of the scheduler's ready list and allowing other threads to run. More commonly, *scheduler-based locks* are designed to *deschedule* the waiting thread, moving it (atomically) from the ready list to a separate queue associated with the lock. The release method then moves one waiting thread from the lock queue to the ready list. To minimize overhead when waiting times *are* short, implementations of scheduler-based synchronization commonly spin for a small, bounded amount of time before invoking the scheduler and yielding the processor. This strategy is often known as *spin-then-wait*.

Condition Synchronization

It is tempting to assume that busy-wait condition synchronization can be implemented trivially with a Boolean flag: a waiting thread spins until the flag is true; a thread that satisfies the condition sets the flag to true. On most modern machines, however, additional fence instructions are required both in the satisfying thread, to ensure that its prior writes are visible to other threads, and in the waiting thread, to ensure that its subsequent reads do not occur until after the spin completes. And even on a sequentially consistent machine, special steps are required to ensure that the compiler does not violate the programmer's expectations by reordering instructions within threads.

In some programming languages and systems, a variable may be made suitable for condition synchronization by labeling it `volatile` (or, in C++'0X, `atomic<>`). The compiler will insert appropriate fences at reads and writes of `volatile` variables, and will refrain from reordering them with respect to other instructions.

Some other systems provide special *event* objects, with methods to set and await them. Semaphores and monitors, described in the following two subsections, can be used for both mutual exclusion and condition synchronization.

In systems with dynamically varying concurrency, the *fork* and *join* methods used to create threads and to verify their completion can be considered a form of condition synchronization. (These are, in fact, the principal form of synchronization in systems like Cilk and OpenMP.)

Barriers

One form of condition synchronization is particularly common in data-parallel applications, where threads iterate together through a potentially large number of algorithmic phases. A *synchronization barrier*, used to separate phases, guarantees that no thread continues to phase $n + 1$ until all threads have finished phase n.

In most (though not all) implementations, the barrier provides a single method, composed internally of an *arrival* phase that counts the number of threads that have reached the barrier (typically via a log-depth tree) and a *departure* phase in which permission to continue is broadcast back to all threads. In a so-called *fuzzy barrier* [6], these arrival and departure phases may be separate methods. In between, a thread may perform any instructions that neither depend on the arrival of other threads nor are required by other threads prior to their departure. Such instructions can serve to "smooth out" phase-by-phase imbalances in the work assigned

to different threads, thereby reducing overall wait time. Wait time may also be reduced by an *adaptive barrier* [7, 19], which completes the arrival phase in constant time after the arrival of the final thread.

Unfortunately, when t threads arrive more or less simultaneously, no barrier implementation using ordinary loads, stores, and RMW instructions can complete the arrival phase in less than $\Omega(\log t)$ time. Given the importance of barriers in scientific applications, some supercomputers have provided special near-constant-time hardware barriers. In some cases the same hardware has supported a fast *eureka* method, in which one thread can announce an event to all others in constant time.

Semaphores

First proposed by Dijkstra in 1965 [5] and still widely used today, *semaphores* support both mutual exclusion and condition synchronization. A *general semaphore* is a nonnegative counter with an initial value and two methods, known as **V** and **P**. The **V** method increases the value of the semaphore by one. The **P** method waits for the value to be positive and then decreases it by one. A *binary semaphore* has values restricted to zero and one (it is customarily initialized to one), and serves as a mutual exclusion lock. The **P** method acquires the lock; the **V** method releases the lock. Programming discipline is required to ensure that **P** and **V** methods occur in matching pairs.

The typical implementation of semaphores pairs the counter with a queue of waiting threads. The **V** method checks to see whether the counter is currently zero. If so, it checks to see whether any threads are waiting in the queue and, if there are, moves one of them to the ready list. If the counter is already positive (in which case the queue is guaranteed to be empty) or if the counter is zero but the queue is empty, **V** simply increments the counter. The **P** method also checks to see whether the counter is zero. If so, it places the current thread on the queue and calls the scheduler to yield the processor. Otherwise it decrements the counter.

General semaphores can be used to represent resources of which there is a limited number, but more than one. Examples include I/O devices, communication channels, or free or full slots in a fixed-length buffer. Most operating systems provide semaphores as part of the kernel API.

Monitors

While semaphores remain the most widely used scheduler-based shared-memory synchronization mechanism, they suffer from several limitations. In particular, the association between a binary semaphore (mutex lock) and the data it protects is solely a matter of convention, as is the paired usage of **P** and **V** methods. Early experience with semaphores, combined with the development of language-level abstraction mechanisms in the 1970s, led several developers to suggest building higher-level synchronization abstractions into programming languages. These efforts culminated in the definition of *monitors* [12], variants of which appear in many languages and systems.

A monitor is a data abstraction (a module or class) with an implicit mutex lock and an optional set of *condition variables*. Each *entry* (method) of the monitor automatically acquires and releases the mutex lock; entry invocations thus exclude one another in time. Programmers typically devise, for each monitor, a program-specific *invariant* that captures the mutual consistency of the monitor's state (data members – fields). The invariant is assumed to be true at the beginning of each entry invocation, and must be true again at the end.

Condition variables support a pair of methods superficially analogous to **P** and **V**; in Hoare's original formulation, these were known as *wait* and *signal*. Unlike **P** and **V**, these methods are *memory-less*: a signal invocation is a no-op if no thread is currently waiting.

For each reason that a thread might need to wait within a monitor, the programmer declares a separate condition variable. When it waits on a condition, the thread releases exclusion on the monitor. The programmer must thus ensure that the invariant is true immediately prior to every wait invocation.

Semantic Details

The details of monitors vary significantly from one language to another. The most significant issues, discussed in the paragraphs below, are commonly known as the *nested monitor problem* and the modeling of signals as *hints vs. absolutes*. More minor issues include language syntax, alternative names for signal and wait, the modeling of condition variables in the type system, and the prioritization of threads waiting for conditions or for access to the mutex lock.

The nested monitor problem arises when an entry of one monitor invokes an entry of another monitor, and the second entry waits on a condition variable. Should the wait method release exclusion on the outer monitor? If it does, there is no guarantee that the outer monitor will be available again when execution is ready to resume in the inner call. If it does not, the programmer must take care to ensure that the thread that will perform the matching signal invocation does not need to go through the outer monitor in order to reach the inner one. A variety of solutions to this problem have been proposed; the most common is to leave the outer monitor locked.

Signal methods in Hoare's original formulation were defined to transfer monitor exclusion directly from the signaler to the waiter, with no intervening execution. The purpose of this convention was to guarantee that the condition represented by the signal was still true when the waiter resumed. Unfortunately, the convention often has the side effect of inducing extra context switches, and requires that the monitor invariant be true immediately prior to every signal invocation. Most modern monitor variants follow the lead of Mesa [15] in declaring that a signal is merely a hint, and that a waiting process must double-check the condition before continuing execution. In effect, code that would be written

```
if (!condition)
    cond_var.wait();
```

in a Hoare monitor is written

```
while (!condition)
    cond_var.wait();
```

in a Mesa monitor. To make it easier to write programs in which a condition variable "covers" a set of possible conditions (particularly when signals are hints), many monitor variants provide a *signal-all* or *broadcast* method that awakens all threads waiting on a condition, rather than only one.

Message Passing

In a system in which threads interact by exchanging messages, rather than by sharing variables, synchronization is generally implicit in the *send* and *receive* methods. A receive method typically blocks until an appropriate message is available (a matching send has been performed). Blocking semantics for send methods vary from one system to another:

Asynchronous send – In some systems, a sender continues execution immediately after invoking a send method, and the underlying system takes responsibility for delivering the message. While often desirable, this behavior complicates the delivery of failure notifications, and may be limited by finite buffering capacity.

Synchronous send – In other systems – notably those based on Hoare's Communicating Sequential Processes (CSP) [13] – a sender waits until its message has been received.

Remote-invocation send – In yet other systems, a send method has both ingoing and outcoming parameters; the sender waits until a reply is received from its peer.

Distributed Locking

Libraries, languages, and applications commonly implement higher-level distributed locks or transactions on top of message passing. The most common lock implementation is analogous to the MCS lock: acquired requests are sent to a *lock manager* thread. If the lock is available, the manager responds directly; otherwise it forwards the request to the last thread currently waiting in line. The release method sends a message to the manager or, if a forwarding request has already been received, to the next thread in line for the lock. Races in which the manager forwards a request at the same time the last lock holder sends it a release are trivially resolved by statically choosing one of the two (perhaps the lock holder) to inform the next thread in line. Distributed transaction systems are substantially more complex.

Rendezvous and Remote Procedure Call

In some systems, a message must be received explicitly by an already existing thread. In other systems, a thread is created by the underlying system to handle each arriving message. Either of these options – *explicit* or *implicit receipt* – can be paired with any of the three send options described above. The combination of remote-invocation send with implicit receipt is often called *remote procedure call* (RPC). The combination of remote-invocation send with explicit receipt is known

as *rendezvous*. Interestingly, if all shared data is encapsulated in monitors, one can model – or implement – each monitor with a *manager* thread that executes entry calls one at a time. Each such call then constitutes a rendezvous between the sender and the monitor.

Related Entries

►Actors
►Cache Coherence
►Concurrent Collections Programming Model
►Deadlocks
►Memory Models
►Monitors, Axiomatic Verification of
►Non-Blocking Algorithms
►Path Expressions
►Processes, Tasks, and Threads
►Race Conditions
►Scheduling Algorithms
►Shared-Memory Multiprocessors
►Transactions, Nested

Bibliographic Notes

The study of synchronization began in earnest with Dijkstra's "Cooperating Sequential Processes" monograph of 1965 [5]. Andrews and Schneider provide an excellent survey of synchronization mechanisms circa 1983 [2]. Mellor-Crummey and Scott describe and compare a variety of busy-wait spin locks and barriers, and introduce the MCS lock [17]. More extensive coverage of synchronization can be found in Chapter 12 of Scott's programming languages text [18], or in the recent texts of Herlihy and Shavit [11] and Taubenfeld [20].

Bibliography

1. Anderson TE (Jan 1990) The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans Parallel Distr Sys 1(1):6–16
2. Andrews GR, Schneider FB (Mar 1983) Concepts and notations for concurrent programming. ACM Comput Surv 15(1):3–43
3. Auslander MA, Edelsohn DJ, Krieger OY, Rosenburg BS, Wisniewski RW (2003) Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 20030200457, submitted 23 Oct 2003
4. Craig TS (Feb 1993) Building FIFO and priority-queueing spin locks from atomic swap. Technical Report 93-02-02, University of Washington Computer Science Department
5. Dijkstra EW (Sept 1965) Cooperating sequential processes. Technical report, Technological University, Eindhoven, The Netherlands. Reprinted in Genuys F (ed) Programming Languages, Academic Press, New York, 1968, pp 43–112. Also available at www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html.
6. Gupta R (Apr 1989) The fuzzy barrier: a mechanism for high speed synchronization of processors. Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, pp 54–63
7. Gupta R, Hill CR (June 1989) A scalable implementation of barrier synchronization using an adaptive combining tree. Int J Parallel Progr 18(3):161–180
8. He B, Scherer III WN, Scott ML (Dec 2005) Preemption adaptivity in time-published queuebased spin locks. Proceeding of the 2005 International Conference on High Performance Computing, Goa, India
9. Herlihy MP (Jan 1991) Wait-free synchronization. ACM Trans Progr Lang Syst 13(1):124–149
10. Herlihy MP, Moss JEB (1993) Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th International Symposium on Computer Architecture, San Diego, CA, May 1993 pp 289–300
11. Herlihy MP, Shavit N (2008) The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington, MA
12. Hoare CAR (Oct 1974) Monitors: an operating system structuring concept. Commun ACM 17(10):549–557
13. Hoare CAR (Aug 1978) Communicating sequential processes. Commun ACM 21(8):666–677
14. Kontothanassis LI, Wisniewski R, Scott ML (Feb 1997) Scheduler-conscious synchronization. ACM Trans Comput Sys 15(1):3–40
15. Lampson BW, Redell DD (Feb 1980) Experience with processes and monitors in Mesa. Commun ACM 23(2):105–117
16. Magnussen P, Landin A, Hagersten E (Apr 1994) Queue locks on cache coherent multiprocessors. Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, pp 165–171
17. Mellor-Crummey JM, Scott ML (Feb 1991) Algorithms for scalable synchronization on sharedmemory multiprocessors. ACM Trans Comput Syst 9(1):21–65
18. Scott ML (2009) Programming Language Pragmatics, 3rd edn. Morgan Kaufmann, Burlington, MA
19. Scott ML, Mellor-Crummey JM (Aug 1994) Fast, contention-free combining tree barriers. Int J Parallel Progr 22(4):449–481
20. Taubenfeld G (2006) Synchronization Algorithms and Concurrent Programming. Prentice Hall, Upper Saddle River

System Integration

►Terrestrial Ecosystem Carbon Modeling

System on Chip (SoC)

►SoC (System on Chip)
►VLSI Computation

Systems Biology, Network Inference in

Jaroslaw Zola[1], Srinivas Aluru[1,2]
[1]Iowa State University, Ames, IA, USA
[2]Indian Institute of Technology Bombay, Mumbai, India

Synonyms

Gene networks reconstruction; Gene networks reverse-engineering

Definition

Inference of gene regulatory networks, also called reverse-engineering of gene regulatory networks, is a process of characterizing, either qualitatively or quantitatively, regulatory mechanisms in a cell or an organism from observed expression data.

Discussion

Introduction

Biological processes in every living organism are governed by complex interactions between thousands of genes, gene products, and other molecules. Genes that are encoded in the DNA are transcribed and translated to form multiple copies of gene products including proteins and various types of RNAs. These gene products coordinate to execute cellular processes – sometimes by forming supramolecular complexes (e.g., ribosome), or by acting in a concerted fashion, e.g., in biochemical or metabolic pathways. They also regulate the expression of genes, often through binding to cis-regulatory sequences upstream of the coding region of the genes, to calibrate gene expression depending on the endogenous and exogenous stimuli carried by, e.g., small molecules.

Gene regulatory networks are conceptual representations of interactions between genes in a cell or an organism. They are depicted as graphs with vertices corresponding to genes and edges representing regulatory interactions between genes (see Fig. 1). Overall, gene regulatory networks are mathematical models to explain the observed gene expression levels. Network inference, or reconstructing, is the process of identifying the underlying network from multiple observations of gene expressions (outputs of the network). To infer a gene network, one relies on experimental data from high-throughput technologies such as microarrays, quantitative polymerase chain reaction, or short-read sequencing, which measure a snapshot of all gene expression levels under a particular condition or in a time series.

Information Theoretic Approaches

Consider a set of n genes $\{g_1, g_2, \ldots, g_n\}$, where for each gene a set of m expression measurements is given. One can represent expression of gene i (g_i) as a random variable $X_i \in \mathcal{X}$, $\mathcal{X} = \{X_1, \ldots, X_n\}$, with marginal probability p_{X_i} derived from some unknown joint probability characterizing the entire system. This random variable is described by observations $\{x_{i,1}, \ldots, x_{i,m}\}$, where $x_{i,j}$ corresponds to the expression level of g_i under condition j. The vector $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m}\rangle$ is called profile of g_i. Given a profile matrix $Y_{n\times m}$, $Y[i,j] = x_{i,j}$, one can formulate network inference problem as that of finding a model that best explains the data in Y.

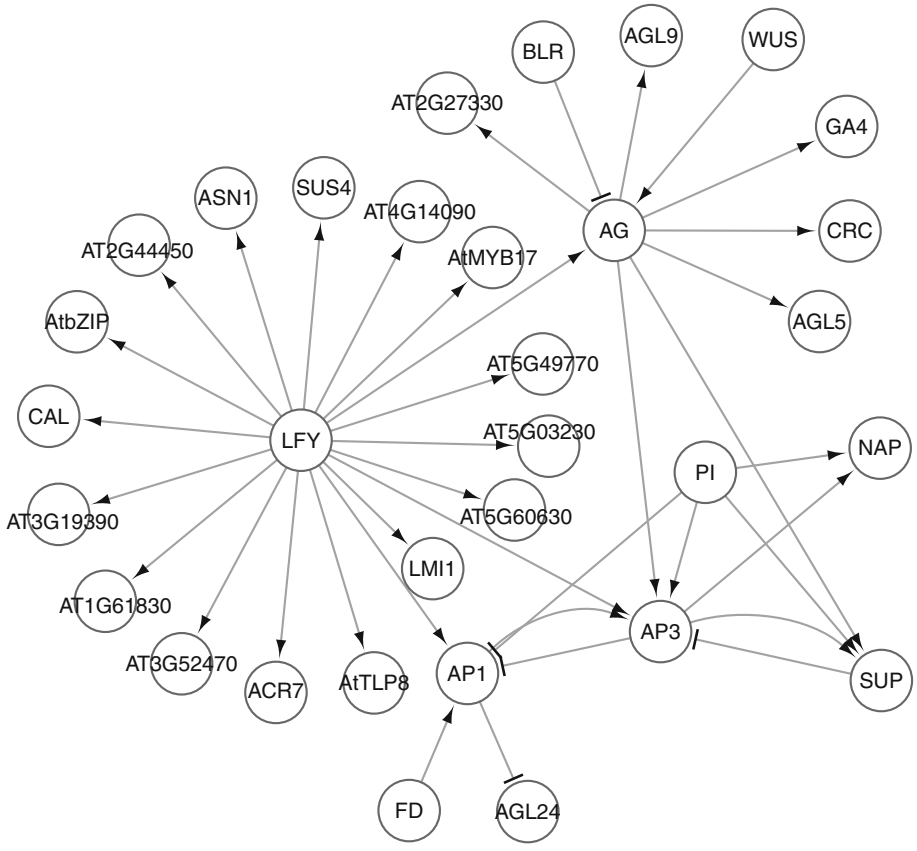
Such formulated problem can be approached using a variety of methods, including Bayesian networks [21] and Gaussian graphical models [20]; one class of methods that has been widely adopted uses the concept of mutual information. These methods [15] operate under the assumption that correlation of expression implies coregulation, and proceed in two main phases: First significant dependencies (connection between two genes) or independencies (lack of connections) are determined by means of computing mutual information for every pair of genes. Then, identification and removal of indirect interactions (e.g., when two genes are coregulated by a third) is performed.

Mutual information is arguably the best measure of correlation between two random variables, and is defined based on entropy $\mathcal{H}$ in the following way:

$$\mathcal{I}(X_i; X_j) = \mathcal{H}(X_i) + \mathcal{H}(X_j) - \mathcal{H}(X_i, X_j),$$

where entropy $\mathcal{H}$ is given by:

$$\mathcal{H}(X) = -\sum p_X(x) \log p_X(x),$$



Systems Biology, Network Inference in. Fig. 1 Example gene regulatory network. Nodes represent genes, *"T"-edges* denote regulation in which a source gene represses expression of the target gene. *Arrow-edges* denote regulation in which a source gene induces expression of the target gene

and p_X defines the probability distribution of X, and $\sum$ is replaced by integral if X is continuous. Mutual information is a symmetric, nonnegative function, and is equal to zero if and only if two random variables are independent.

Application of mutual information for gene network inference poses two significant challenges. As the probability distribution of the random variable describing a gene is unknown, it has to be estimated from the expression profile. Consequently, gene comparison becomes more difficult because even independent expression profiles can result in mutual information greater than zero (owing to sampling and estimation errors). This in turn requires some mechanism to decide if the given mutual information estimate is statistically significant. The second challenge is due to the fact that a typical genome-level network covers thousands of genes, and hence "all-pairs" comparison adds considerably to computational requirements. In practice, several mutual information estimators are available that offer different precision to complexity ratios (e.g., Gaussian kernel estimator, B-spline estimator), and complex statistical techniques are employed to decide if observed mutual information implies dependency.

Although all information theoretic approaches for reverse-engineering depend on "all-pairs" mutual information kernel executed in the first stage, they differ in how they identify indirect interactions in the second stage. For example, in relevance networks [4] the second stage is omitted, in ARACNe [3] and TINGe [22, 23] the Data Processing Inequality concept is used, while CLR algorithm [6] depends on estimates of a likelihood

of obtained mutual information values. Some other methods extend mutual information into conditional mutual information or augment it with feature selection techniques.

Parallel Information Theoretic Approach

Reverse engineering of regulatory networks using mutual information is compute and memory intensive especially if whole-genome (i.e., covering all genes of an organism) networks are considered. Memory consumption arises from the $\Theta(nm)$ size of input data, and from the $\Theta(n^2)$ dense initial network generated in the first phase of the reconstruction algorithm. Adding to this is complexity of mutual information estimators, which for Gaussian kernel estimator for instance is $O(m^2)$. Taking into account that the number of genes typically considered is in the thousands, and at the same time genome-level inference requires that the number of observations m is large, the problem becomes prohibitive for sequential computers.

In [23], Zola et al. proposed a parallel information theory-based inference method that efficiently exploits multiple levels of parallelism inherent to mutual information computations and uses a generalized scheme for pairwise computation scheduling. The method has been implemented in the MPI-based software package called TINGe (Tool Inferring Networks of Genes), along with a version that supports the use of cell accelerators.

The algorithm proceeds in three stages. In the first stage, input expression profiles are rank-transformed and mutual information is computed for each of the $\binom{n}{2}$ pairs of genes, and q randomly chosen permutations per pair. Rank transformation substitutes a gene expression profile with a permutation of $\langle 1, \ldots, m \rangle$ by replacing a gene expression with its rank among all gene expressions within the same expression profile. It has been shown that mutual information is invariant under this transformation [5]. By applying it the algorithm can reduce the total number of mutual information estimations between gene expression vectors and their random permutations [22]. In the second phase, the threshold value above which mutual information is considered to signify dependence is computed, and edges below this threshold are discarded. The threshold is computed by finding the element with rank $(1 - \varepsilon) \cdot q \cdot \binom{n}{2}$ among $q \cdot \binom{n}{2}$ values contributed by permutations generated in the earlier stage, where ε is specifies the desired statistical significance of the corresponding permutation test. Finally, in the third stage data processing inequality is applied with the consequence that if g_i interacts with g_k via some other gene g_j then $\mathcal{I}(X_i; X_k) \leq \min(\mathcal{I}(X_i; X_j), \mathcal{I}(X_j; X_k))$.

The algorithm represents gene network using the standard adjacency matrix $D_{n \times n}$. The input and output data are distributed row-wise among p processors. Each processor stores up to $\lceil \frac{n}{p} \rceil$ consecutive rows of matrix Y and the same number of consecutive rows from matrix D. Matrix D is then partitioned into $p \times p$ blocks of submatrices which are computed in $\lceil \frac{p+1}{2} \rceil$ iterations, where in iteration i processor with rank j computes submatrix $D_{j,(j+i) \mod p}$. To implement the second stage a simple reduction operation is used to find the threshold value followed by pruning of matrix D. Finally, removing of indirect interactions is performed based on streaming of matrix D in $p - 1$ communication rounds, where in iteration i only processors with ranks lower than $p - i$ participate in communication and computation.

In their method, Zola et al. use B-spline mutual information estimator which is implemented to take advantage of SIMD extensions of modern processors. However, any mutual information estimator could be used. Furthermore, they report modification of the first stage of the algorithm that enables execution on cell heterogeneous processors. The method has been used to reconstruct a 15,222 network of the model plant *Arabidopsis thaliana* from 3,137 microarray experiments in 30 minutes on a 2,048 core IBM Blue Gene/L, and in 2 h and 25 min on a 8-node QS20 cell blade cluster.

Approaches Based on Bayesian Networks

Bayesian networks are a class of graphical models that represent probabilistic relationships among random variables of a given domain. Formally, a Bayesian network is a pair (N, P), where P is the joint probability distribution and N is a directed acyclic graph, with vertices representing random variables and edges corresponding to "parent – child" relationship between variables, which encodes the Markov assumption that a node is conditionally independent from its non-descendants, given its parents in N. Under this assumption one can

represent the joint probability as a product of conditional probabilities:

$$P(X_1, \ldots, X_n) = \prod_i P(X_i|\pi_i),$$

where π_i is a set of parents of X_i in N. Given a set of realizations (observations) of random variables one can learn a structure of the Bayesian network that best fits the observed data.

Bayesian networks have been widely employed for reverse-engineering of gene regulatory networks [8, 18, 21]. Following the same formalization as for information theoretic approaches described above, the problem of gene network inference becomes that of learning the structure of the corresponding Bayesian network. Nodes of the network are random variables assigned to genes $\mathcal{X} = \{X_1, \ldots, X_n\}$, expression profiles are realizations of those variables, and a Bayesian network learned from such data represents a gene regulatory network, where π_i is interpreted as a set of regulators of gene g_i. In order to learn the structure of a Bayesian network, a statistically motivated scoring function that evaluates the posterior probability of a network given the input data is typically assumed: $Score(N) = \log P(N|Y)$. To find the optimal network efficiently such a function should be decomposable into individual score contributions $s(X_i, \pi_i)$, i.e.:

$$Score(N) = \sum_i s(X_i, \pi_i).$$

A major difficulty in Bayesian network structure learning is the super exponential search space in the number of random variables – for a set of n variables there exist $\frac{n!2^{\frac{n}{2}(n-1)}}{r \cdot z^n}$ possible directed acyclic graphs, where $r \approx 0.57436$ and $z \approx 1.4881$.

Parallel Exact Structure Learning

Even assuming that the cost of evaluating scoring function is negligible, exhaustive enumeration of all possible network structures remains prohibitive. Although heuristics have been proposed to tackle the problem, e.g., based on simulated annealing, oftentimes reconstructing the optimal network is advantageous as it enables more meaningful conclusions.

Nikolova et al. [17] proposed an elegant parallel exact algorithm for learning Bayesian networks that builds on top of earlier sequential methods [18]. In this approach a network is represented as a permutation of nodes where each node is preceded by its parents. The algorithm identifies the optimal ordering, and a corresponding optimal network, using a dynamic programming approach that operates on the lattice formed on the power set of $\mathcal{X}$ by the partial order "set inclusion." The lattice is organized into $n + 1$ levels, where level $l \in [0, n]$ contains all subsets of size l, and a node at level l has l incoming and $n - l$ outgoing edges. At each node of the lattice $(n - l)$ evaluations of individual scores s have to be performed as a part of dynamic programming search, which next have to be communicated along outgoing edges.

The key component of the approach by Nikolova et al. is the observation that dynamic programming lattice forms an n-dimensional hypercube that can be decomposed on $p = 2^k$ processors into 2^{n-k} k-dimensional hypercubes, each mapping to p processors. These hypercubes can be processed in a pipelined fashion that provides a work optimal algorithm.

The reported method has been used to reverse-engineer regulatory networks using synthetic data with up to 30 genes and 500 microarray observations, and applying Minimum Description Length principle [11] as a scoring function. To reconstruct a network for the largest data it took 1 h and 30 min on 1,024 processors of an IBM BlueGene/L.

Approaches Based on Differential Equations

Under several simplifying assumptions the dynamics of a gene's expression can be modeled as a function of abundance of all other genes and the rate of degradation:

$$\dot{x}_i = f_i(\mathbf{x}) - \lambda(x_i),$$

where f_i is called input function of gene i, x_i is expression of gene i, vector $\mathbf{x}$ represents expression levels of all genes, and λ describes the rate of degradation. One can further assume that input functions are linear and in such cases the dynamics of the entire system can be represented as:

$$\dot{\mathbf{x}} = A \cdot \mathbf{x},$$

where $A_{n \times n}$ is a matrix describing influences of genes on each other (including rates of degradation), i.e., $A[i, j]$ represents the influence of gene g_j on g_i. Consequently

by solving such system of equations one would obtain the underlying gene network represented by matrix A. Unfortunately finding the solution requires a large number of measurements of $\mathbf{x}$ and $\dot{\mathbf{x}}$ since otherwise the system is greatly underdetermined. At the same time obtaining representative measurements is experimentally very challenging and sometimes infeasible.

To overcome this limitation, and to enable linearization of systems with nonlinear gene input functions, Gardner et al. [9] proposed an approach in which m perturbation experiments (i.e., experiments in which expression of selected genes is affected in a controlled way) are performed, and resulting expression profiles are used to write the following system of differential equations:

$$\dot{Y} = AY + U,$$

which at steady state gives:

$$AY = -U.$$

Here $Y_{n\times m}$ is a matrix describing expression of all genes in all experiments, i.e., $Y[i,j]$ describes expression of gene i under perturbation j, and matrix $U_{n\times m}$ represents the effect of perturbations on every gene in all m experiments. Because in the majority of cases $m < n$ and the resulting system remains underdetermined, Gardner et al. assumed that each gene can have at most k regulatory inputs (which is biologically plausible), and then applied multiple regression for every possible combination of k regulators, choosing the one that best fits the data to approximate A.

Parallelization of Multiple Regression Algorithms

The approach of Gardner et al., named Network Identification by Multiple Regression, is computationally prohibitive for networks with more than a few dozen genes as it is infeasible to consider all $\binom{n}{k}$ combinations of regulators, especially that for each gene and each combination of regulators the following expression must be evaluated

$$\hat{\mathbf{a}}_\mathbf{i} = -\mathbf{u}_\mathbf{i} Z^T (ZZ^T)^{-1},$$

to find the combination for which $\hat{\mathbf{a}}_\mathbf{i}$ minimizes the sum squared errors with respect to the observed data. Here, $\mathbf{u}_\mathbf{i}$ represents row of matrix U for gene i, and Z consists of k rows selected from Y that correspond to k selected regulator genes.

In [10], Gregoretti et al. describe parallel implementation of the algorithm that replaces the exact enumeration of all combinations of regulators with a greedy search heuristic that starts with d best candidate regulators that are next iteratively combined with other genes to form the final set of regulators. Furthermore, they observe that it is possible to obtain $\hat{\mathbf{a}}_\mathbf{i}$ by solving the system of linear equations $S\mathbf{a}_\mathbf{i} = -\mathbf{r}$, where S is the symmetric submatrix of YY^T of k rows and columns that correspond to the considered regulators, and $\mathbf{r}$ is the i-th row of Y^T. Because S is positive definite the Cholesky factorization can be used to efficiently solve this system. In practice, the implementation uses the `PPSV` routine, which has multiple parallel implementations. Finally, searching for regulators of a gene can be performed independently for every gene. Consequently, in the parallel version each of p processors is assigned at most $\lceil \frac{n}{p} \rceil$ genes for which it executes the search heuristic.

The main limitation of the multiple regression algorithm is that it requires perturbation data as an input. Because in many cases such data is not available, Gregoretti et al. used synthetic data consisting of 2,500 genes with a single perturbation experiment for each gene. This data has been analyzed on a cluster with 100 nodes, each with dual core Itanium2 processor, and with Quadrics ELAN 4 interconnect in approximately 3 h and 25 min.

Future Directions

The problem of reverse engineering gene regulatory networks is one of many in the broad area of computational systems biology, and parallel processing only recently attracted attention of systems biology researchers. Together with the rapid progress in high-throughput biological technologies one can expect accumulation of massive and diverse data, which will enable more complex and realistic models of regulation. Most likely these models will be evolving such as to enable *in silico* simulation of biological systems, which is one of the goals of the emerging field of synthetic biology. Consequently, parallel processing in its various flavors, ranging from accelerators and multicore

processors to clusters, grids, and clouds, will be necessary to tackle the resulting computational complexity.

Bibliographic Notes and Further Reading

The textbook by Alon [1] and articles by Kitano [12, 13] and Murali and Aluru [16] provide a good general introduction to systems biology. A brief overview of existing approaches and challenges in gene networks inference can be found in [2, 14]. In [7] Friedman gives an introduction to using graphical models for network inference, and Meyer et al. review information theoretic approaches in [15]. The "Dialogue for Reverse Engineering Assessments and Methods" project [19] provides a robust set of benchmark data that can be used to assess the quality of inference methods, and it is a good source of information about developments in the area of networks reconstruction.

Bibliography

1. Alon U (2006) An Introduction to Systems Biology: Design Principles of Biological Circuits. Chapman & Hall/CRC, Boca Raton
2. Bansal M, Belcastro V, Ambesi-Impiombato A, di Bernardo D (2007) How to infer gene networks from expression profiles. Mol Syst Biol 3:78
3. Basso K, Margolin AA, Stolovitzky G, Klein U, Dalla-Favera R, Califano A (2005) Reverse engineering of regulatory networks in human B cells. Nat Genet 37(4):382–390
4. Butte AJ, Kohane IS (2000) Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In Pacific Symposium on Biocomputing, pp 418–429
5. Cover TM, Thomas JA (2006) Elements of Information Theory, 2nd edn. Wiley, New York
6. Faith JJ, Hayete B, Thaden JT, Mogno I, Wierzbowski J, Cottarel G, Kasif S, Collins JJ, Gardner TS (2007) Large-scale mapping and validation of *Escherichia coli* transcriptional regulation from a compendium of expression profiles. PLoS Biol 5(1):e8
7. Friedman N (2004) Inferring cellular networks using probabilistic graphical models. Science 303:799–805
8. Friedman N, Linial M, Nachman I, Pe'er D (2000) Using Bayesian networks to analyze expression data. J Comput Biol 7:601–620
9. Gardner TS, di Bernardo D, Lorenz D, Collins JJ (2003) Inferring genetic networks and identifying compound mode of action via expression profiling. Science 301(5629):102–105
10. Gregoretti F, Belcastro V, di Bernardo D, Oliva G (2010) A parallel implementation of the network identification by multiple regression (NIR) algorithm to reverse-engineer regulatory gene networks. PLoS One 5(4):e10179
11. Grunwald PD (2007) The Minimum Description Length Principle. MIT Press, Cambridge
12. Kitano H (2002) Computational systems biology. Nature 420(6912):206–210
13. Kitano H (2002) Systems biology: a brief overview. Science 295(5560):1662–1664
14. Margolin A, Califano A (2007) Theory and limitations of genetic network inference from microarray data. Ann N Y Acad Sci 1115:51–72
15. Meyer PE, Kontos K, Lafitte F, Bontempi G (2007) Information-theoretic inference of large transcriptional regulatory networks. EURASIP J Bioinform Syst Biol 2007:79879
16. Murali TM, Aluru S (2009) Algorithms and Theory of Computation Handbook, chapter Computational Systems Biology. Chapman & Hall/CRC, Boca Raton
17. Nikolova O, Zola J, Aluru S (2009) A parallel algorithm for exact Bayesian network inference. In IEEE Proceedings of the International Conference on High Performance Computing (HiPC 2009), pp 342–349
18. Ott S, Imoto S, Miyano S (2004) Finding optimal models for small gene networks. In Pacific Symposium on Biocomputing, pp 557–567
19. Prill RJ, Marbach D, Saez-Rodriguez J, Sorger PK, Alexopoulos LG, Xue X, Clarke ND, Altan-Bonnet G, Stolovitzky G (2010) Towards a rigorous assessment of systems biology models: the DREAM3 challenges. PLoS One 5(2):e9202
20. Schafer J, Strimmer K (2005) An empirical Bayes approach to inferring large-scale gene association networks. Bioinformatics 21(6):754–764
21. Yu J, Smith V, Wang PP, Hartemink AJ, Jarvis ED (2004) Advances to Bayesian network inference for generating causal networks from observational biological data. Bioinformatics 20(18):3594–3603
22. Zola J, Aluru M, Aluru S (2008) Parallel information theory based construction of gene regulatory networks. In Proceedings of the International Conference on High Performance Computing (HiPC 2008). LNCS, vol 5375, pp 336–349
23. Zola J, Aluru M, Sarje A, Aluru S (2010) Parallel information-theory-based construction of genome-wide gene regulatory networks. IEEE Trans Parallel Distributed Syst 21(12):1721–1733

Systolic Architecture

►Systolic Arrays

Systolic Arrays

James R. Reinders
Intel Corporation, Hillsboro, OR, USA

Synonyms

Instruction systolic arrays; Processor arrays; Systolic architecture; Wavefront arrays

Definition

A *Systolic Array* is a collection of processing elements, called cells, that implements an algorithm by rhythmically computing and transmitting data from cell to cell using only local communication. Cells of a Systolic Array are arranged and connected in a regular pattern with a design that emphasizes a balance between computational and communicational capabilities. Systolic Arrays have proven particularly effective for real-time applications in signal and image processing.

Systolic Algorithms are algorithms specifically designed to make effective use of Systolic Arrays. Systolic Algorithms have also been shown to make particularly efficient use of many general-purpose parallel computers.

The name *Systolic Arrays* derives from an analogy with the regular pumping of blood by the heart. Systolic, in medical terms, refers to the phase of blood circulation in which the pumping chambers of the heart, ventricles, are contracting forcefully and therefore blood pressure is at its highest.

Discussion

Background: Motivated by Emergence of VLSI

Systolic Arrays, first described in 1978 by H. T. Kung and Charles E. Leiserson, were originally described as a systematic approach to take advantage of rapid advances in VLSI technology and coping with difficulties present in designing VLSI systems. The simplicity and regularity of Systolic Arrays lead to a cheaper VLSI implementation as well as higher chip density.

Systolic Arrays were originally proposed for VLSI implementation of some matrix operations that were shown to have efficient solutions thereafter known as Systolic Algorithms. Systolic Algorithms support high degrees of concurrency while requiring only simple, regular communication and control, which in turn allows for efficient implementation in hardware.

Early VLSI offered high degrees of integration but no speed advantage over the decade-old TTL technology. While tens of thousands of gates could be integrated on a single chip, it appeared that computational speed would only come from the concurrent use of many processing elements. In 1982, H. T. Kung wrote, "Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements."

Furthermore, large-scale designs were pushing the limits of the methodologies of the day. It was observed at the time that the general practice of ad hoc designs for VLSI systems was not contributing to sufficient accumulation of experiences, and errors were often repeated. By providing general guidelines, the concept of a Systolic Array, a general methodology emerged for mapping high-level computations into hardware structures.

Cost-effectiveness emerged as a chief concern with special-purpose systems; costs need to be low enough to justify construction of any device with limited applicability. The cost of special-purpose systems can be greatly reduced if a limited number of simple substructures or building blocks, such as a cell, can be reused repeatedly for the overall design.

The VLSI implications of Systolic Array designs proved to be substantial. Spatial locality divides the time to design a chip by the degree of regularity. Locality avoids the need for long and therefore high capacitive wires, temporal regularity and synchrony reduces control issues, pipelinability yields performance, I/O closeness holds down I/O bandwidth, and modularity allows for parameterized designs.

Special-purpose systems pushed the limits of technology in order to achieve the performance needed to justify their cost. Systolic Arrays embraced VLSI circuit technology to achieve high performance via parallelism, honoring the scarcity of power and resistive delay by using a communication topology devoid of long inter-processor wires. Such communication topologies require chip area that is only linear in the number of processors. Systolic Arrays also embraced the design economics of special-purpose processors by reusing a limited number of cell designs.

Systolic Arrays proved especially well suited for processing data from sensor devices as an attached processor. Stringent time requirements of real-time signal processing and large-scale scientific computation strongly favor special-purpose devices that can be built in a cost-effective and reliable fashion because they deliver the performance needed.

Concept

Unlike general-purpose processors, a Systolic Array is characterized by its regular data flow. Typically, two or more data streams flow through a Systolic Array in various speeds and directions. The crux of the Systolic Array approach is that once a stream of data is formed, it can be used effectively by each processing element it passes. A higher computational throughput is therefore achieved as compared with a general-purpose processor in which its computational speed may be limited by the I/O bandwidth. When a complex algorithm can be decomposed to fine-grained, regular operations, each operation will then be simpler to implement.

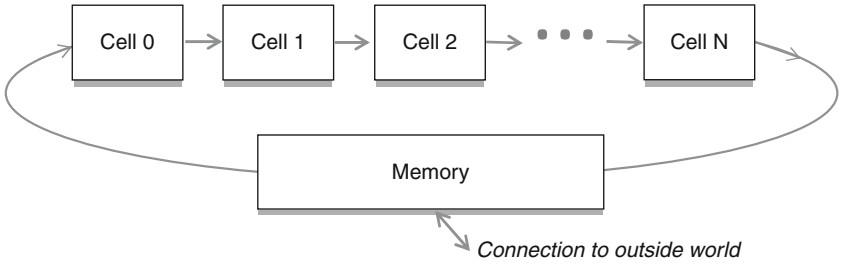
A Systolic Array consists of a set of interconnected cells, each capable of performing at least simple computational operations, and communicating only with cells in close proximity. Simple, regular communication and control structures offer substantial advantages over complicated ones in both design and implementation. Many shapes for an "array" are possible, and have been proposed including triangular, but simple two-dimensional meshes, or tori, have dominated as Systolic Arrays have trended to become more general because of the flexibility and simplicity of meshes and tori (Fig. 1).

Systolic Arrays are specifically designed to address the communication requirements of parallel computer systems by placing an emphasis on strong connections between computation and communication in order to achieve both balance and scaling. Systolic Arrays directly address the importance of the communication system for scalable parallel systems by providing direct paths between the communication system and the computational units. Systolic Algorithms address the need for "balanced algorithms" to best utilize parallelism.

In a Systolic Array, data flows from the computer memory in a rhythmic fashion, passing through many processing cells before it returns to memory, much as blood circulates into and out of the heart. The system works like an assembly line where many people work on the same automobile at different times and many cars are assembled simultaneously. The network for the flow of data can offer different degrees of parallelism, and data flow itself may be at different speeds and multiple directions. Traditional pipelined systems flow only results, whereas a Systolic Array flow includes inputs and partial results.

The essential characteristic of a Systolic Array is an emphasis on balance between computational and communicational capabilities, and the scalability of practical parallel systems. The features of Systolic Arrays, in pursuit of this emphasis on balance and scalability, are generally as follows:

- Spatial regularity and locality: the variety of processing cells is limited, and connections are limited to nearby processors. Cells are not connected via shared busses, which would not scale due to contention. There is neither global broadcasting nor global memory.
- Temporal regularity and synchrony: each cell acts as a finite state transducer. Cells do not need to execute the same program.
- Pipelinability: a design of N cells will exhibit a linear speedup O(N).
- I/O closeness: only cells on the boundaries of the array have access to "the outside world" to perform I/O.
- Modularity/scaling: a larger array can handle a larger instance of a problem than the smaller version of which it is an extension. Replacing a processing cell with an array of cells offers a higher computation throughput without increasing memory bandwidth. This realization of parallelism offers the advantages



Systolic Arrays. Fig. 1 Simple linear systolic array configuration

of both increased performance from increased concurrency and increased designer productivity from component reuse.

Synchronous global clocking is not a requirement of Systolic Arrays despite it being a property of many early implementations. Systolic Arrays are distinguished by their pursuit of both balance and scaling with a design emphasis on strong connections between computation and communication in order to achieve both.

Importance of Interconnect Design

A central issue for every parallel system is how the computational nodes of a parallel computer communicate with other nodes. There are problems that have been dubbed "embarrassingly parallel," where little or no communication is necessary between the multiple processors performing subsets of the problem. For those applications, the computation speed alone will determine the speed of execution on a parallel system. There are numerous important problems that are not embarrassingly parallel, in which communication plays a critical role in determining the effective speed of execution as well as the degree to which the problem can scale to utilize parallelism.

The exploration of "balanced algorithms" looked to find algorithms that scale without bounds. Such algorithms are marked by a constant ratio of computation to communication steps. These algorithms became known as "Systolic Algorithms." While Systolic Algorithms can be mapped to a wide range of hardware, it was found that these algorithms tended to rely on finer and finer-grained communication as machine sizes increased. To efficiently deal with this fine-grained parallelism, communication with little to no overhead is needed.

Systolic Arrays accomplish this goal by providing a method to directly couple computation to communication. In such machines, the design will seek to match the communicational capabilities to the computational capabilities of the machine.

Systolic Arrays are one approach to addressing the communication requirements of parallel systems. Systolic Arrays acknowledge the importance of the communication system for scalable parallel systems and provide direct paths between the communication system and the computational units. The key concepts behind a Systolic Array are the balance between computational and communicational capabilities, and the scalability of practical parallel systems. A balanced design minimizes inefficiencies as measured by underutilization of portions of a system due to stalls and bottlenecks. A scalable design allows for expanding performance by increasing the number of computational nodes in a system. Regularity, an often-noted characteristic of a Systolic Array, is a consequence of the scalability goal and not itself part of the definition of a Systolic Array. Another advantage of regularity is that the system can be scaled by repeating a common subsystem design, a benefit to designers that has reinforced use of regularity as a way to accomplish scalability.

The Systolic Array approach initially led to very rigidly synchronous hardware designs that, while elegant, proved overly constraining for many programming solutions. While balance is desirable, the tight coupling of systolic systems is not always desirable. Designs of more programmable Systolic Arrays worked to preserve the benefits of systolic communication while generalizing the framework to allow for more application diversity.

With tight coupling, any stall in communication will stall computation and vice versa. For many applications, a looser coupling that allowed coupling at a level of data blocks instead of individual data elements was advantageous. Looser coupling also leads to increased ability to overlap communication and computation in practice. As a result, the idea of Systolic Arrays evolved from an academic concept to realization in microprocessors such as the CMU/Intel iWarp.

Variations

Increasing the degree of independence of individual processors in an array adds complexity for flexibility and affects the efficiency and performance of an array. One design consideration is whether individual processors have a local control store or a design where instructions were delivered via the processor interconnects to be executed upon arrival. The latter was sometimes referred to as an Instruction Systolic Array (ISA). Synchronous broadcasting of instructions to an array, as opposed to the flow of instructions via the interconnect, would be inconsistent with the goals of Systolic Array designs because it would introduce long paths and the associated delays. Processors with individual control

stores have been referred to as *programmable* Systolic Arrays, and therefore incur some overhead for the initial loading of the control stores.

As a design methodology for VLSI, silicon implementations were generally hard coded to a large degree once a design was determined. The resulting array performed a fixed function and offered no ability to be loaded with a new algorithm for other functions. Such designs could be optimized to offer the most efficient implementations with the least flexibility for future changes by customizing the cells and the connection networks. More flexible designs offered more design reuse, and thereby offered the opportunity to amortize development costs over multiple uses in exchange for some loss of efficiency, increase in silicon size and generally some loss of array performance.

The most flexible implementations of Systolic Arrays were machines developed principally for academic and research purposes to host the exploration and demonstration of Systolic Algorithms. The Warp and iWarp machines, developed under the guidance of H. T. Kung at Carnegie Mellon University, were such machines.

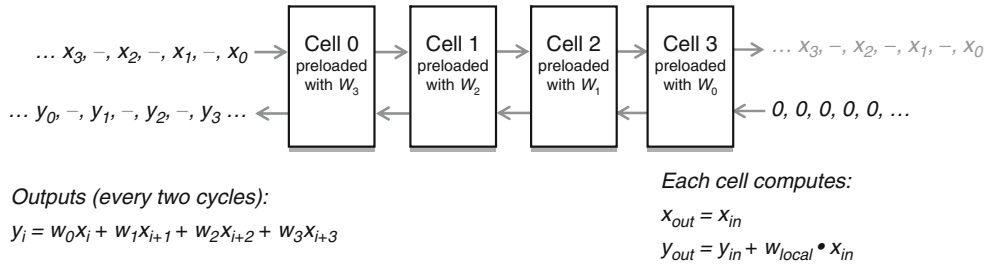
Systolic Algorithms

Systolic Arrays have garnered substantial attention in the development of Systolic Algorithms, which have proven to be efficient at creating solutions suited to the demands of real-time systems in terms of reliability and the ability to meet stringent time constraints. Development of error detections and fault tolerance in Systolic Algorithms has also received substantial attention.

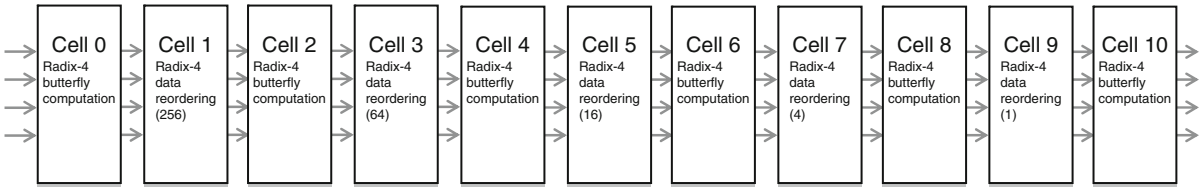
Systolic Algorithms are among the most efficient class of algorithms for massively parallel implementation. A systolic algorithm can be thought of as having two major parts: a cell program and a data flow specification. The cell program defines the local operations of each processing element, while the data flow describes the communication network and its use (Fig. 2).

One nice property of a Systolic Algorithm is that each processor communicates only with a few other processors. It is thus suitable for implementation on a cluster of computers in which we seek to avoid costly global communication operations. Systolic Algorithms require preservation of data ordering within a stream but do not require that streams of data proceed in lockstep as they would have in the earliest hardware implementations of Systolic Arrays.

A Systolic Algorithm will make multiple uses of each input data item, make extensive use of concurrency, rely on only a few simple cell types, and utilize simple and regular data and control flows. Bottlenecks to computational speedups are often caused by limited system memory bandwidths, known as von Neumann bottlenecks, rather than limited processing capabilities (Fig. 3).



Systolic Arrays. Fig. 2 Systolic array implementation for a convolution product



Systolic Arrays. Fig. 3 Systolic array implementation for a 4096-point FFT

Systolic Algorithms have been shown to have wide applicability. They utilize a computational model for a wide range of parallel processing structures not limited to the initially targeted special-purpose needs, and which offer efficient operations on parallel computers in general, not just special-purpose designs. For instance, linear algebra algorithms and FFT algorithms are computationally demanding, especially when high throughput rates are expected, and they display a high degree of regularity. These algorithms are ideal candidates for parallel implementation via Systolic Algorithms.

Systolic Array Machines

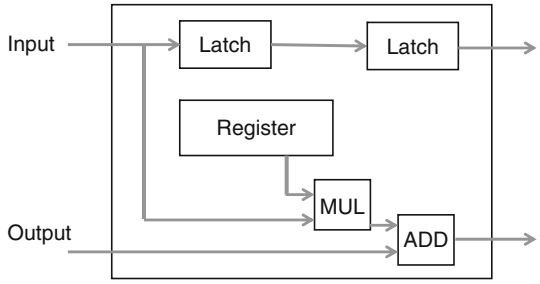
The Colossus Mark II, built in 1944, has been cited as the first digital computer known to have used a technique similar to Systolic Arrays.

In 1978, H. T. Kung and Charles E. Leiserson published their concept of Systolic Arrays. They made systematic use of ideas that were already present in older architectural paradigms in order to respond to the new challenges posed by VLSI technology. This sparked much interest and many designs for Systolic Arrays. While there have been many designs, only a handful have been actually built and put into use. Here is a brief review of key systems in chronological order.

The earliest Systolic Arrays were introduced as a straightforward way to create scalable, balanced, systems for special-purpose computations, by directly embodying a systolic algorithm in the hardware design. These early Systolic Arrays were very limited in applications because the algorithm was expressed in efficient and special-purpose designs. Changing an algorithm implemented as a Systolic Array may be difficult or impossible without redesigning the hardware, particularly the communication interconnects. More general processors used in parallel have been shown to be able to perform a broad range of applications, but are often too large, too slow, or too expensive due to inefficiencies imposed by communication overhead.

Systolic Convolution Chip

The systolic convolution chip was created at CMU in 1979 to solve finite-sized two-dimensional convolutions. All nodes in a system performed the same operation while data flowed through the systems in a completely regular synchronous manner (Fig. 4).



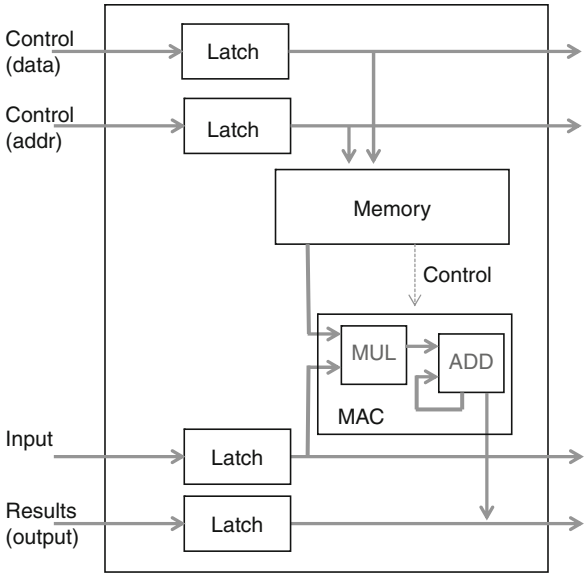
Systolic Arrays. Fig. 4 2-D convolution systolic system cell architecture

ESL Systolic Processor

A Systolic Array to compute convolutions and other signal processing computations was designed and implemented at ESL and operational by 1982. Seven nodes fit on a board measuring 37 cm by 40 cm using discrete high-performance components with each node capable of ten million operations per second. Up to five boards could be linked together in a system, which was attached to a VAX 11/780 host machine. Accessing the capabilities of the Systolic Array from a high-level language was a key innovation of this Systolic Array. A Fortran program viewed capabilities as a function to call with information on the kernel to execute the input data and where to store the output data. The ESL systolic processor was able to greatly expand the application domain for which it was suited by including local memory to hold more kernel elements and a control unit to create addresses to access data from memory. This systolic system could perform 1-D and 2-D convolutions, matrix multiplication, and Fourier and cosine transforms (Fig. 5).

NOSC Systolic Array Test Bed

A programmable Systolic Array, formed using standard off-the-shelf Intel 8051 microprocessors, was built by the Naval Ocean Systems Center (NOSC) from 1981 to 1983. The microprocessor operated as the control unit, while a separate arithmetic unit on the board consumed and output data. The arithmetic processor was directly connected to the communication ports so as to tightly link computations and communications. The NOSC test bed contained 64 nodes arranged in an 8×8 grid. Each node fits on a 6×24-cm board. Each node in the system is connected to five other nodes. The



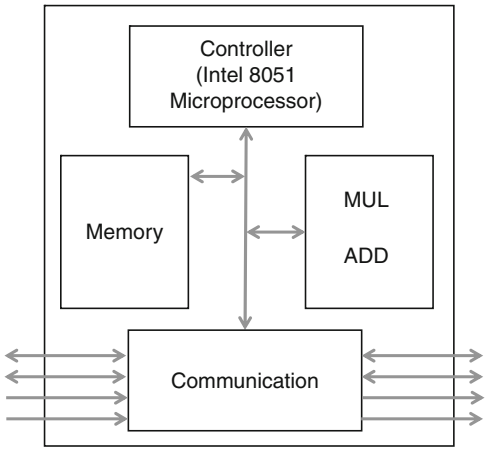
Systolic Arrays. Fig. 5 ESL systolic cell architecture

topology allows for direct implementation of several key Systolic Algorithms such as matrix multiplication by a hexagonal array.

All programming was done in assembly language. Node performance was about 24 K FLOPs for a total performance of 1.5 MFlop for the system. The machine was a test bed for software development, and the design could not extend beyond 64 nodes. NOSC research into Systolic Arrays spanned a number of machines from 1979 to 1991 including the Systolic Array Processor (SAP), the Systolic Linear Algebra Parallel Processor (SLAPP), the Video Analysis Transputer Array (VATA), and the High-Speed Systolic Array Processor (HISSAP) test bed. Subsequent algorithm development moved to iWarp and on to general-purpose machines (Fig. 6).

Programmable Systolic Chip (PSC)

A programmable Systolic Array chip designed at CMU led to a functional nine-node system in 1984. The architecture of the PSC was similar to the NOSC Systolic Array but was organized around three buses instead of one, cells had three input and output ports to attach to other cells. All programming was in assembly language. The machine was used for low-level image processing computations (Fig. 7).



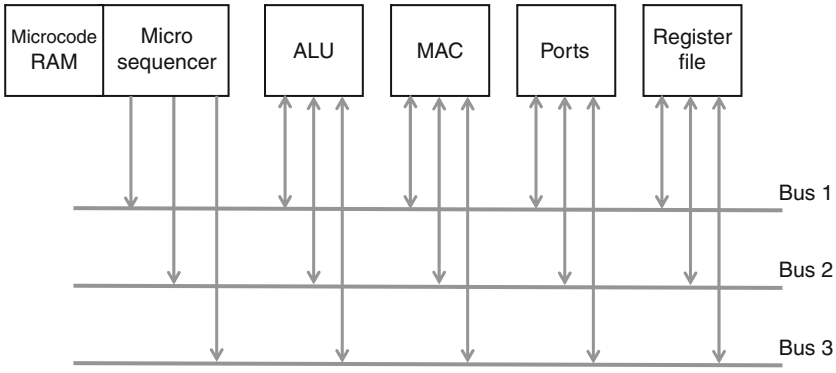
Systolic Arrays. Fig. 6 NOSC test bed cell architecture

Geometric Arithmetic Processor (GAPP)

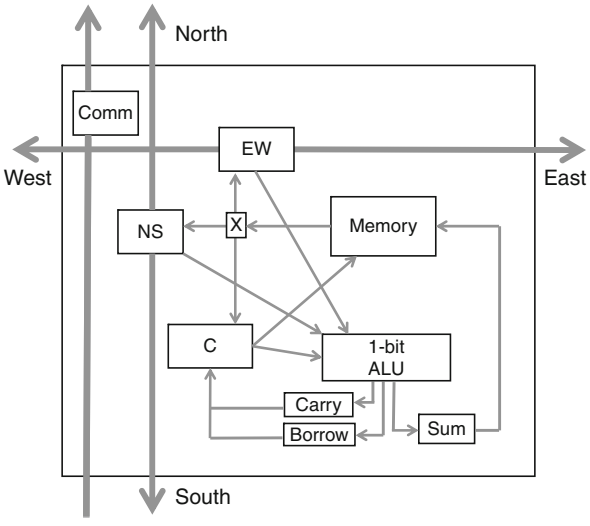
The NCR GAPP programmable Systolic Array is a mesh-connected single-bit cell that communicates directly with neighbors to the North, East, South, and West. It was first implemented as a medium-scale integration (MSI) breadboard in 1982 for a 6 × 12 cell array. GAPP I was a PLA-based 3 × 6 cell chip, GAPP II was a 6 × 12 cell in 3-^{-}m CMOS, and finally a version in two-micron CMOS clocked at 10 MHz with 30 MB/s input and 30 MB/s output bandwidth was sold as NCR45CG72 and could perform 28 million eight-bit additions per second. The use of one-bit data paths and one-bit registers minimize the size of a single cell. Local memory on a node was only 128 bits. The machine broadcast long instruction words to control the machine in pure lockstep SIMD. Programming was originally done in STOIC, a variant of Forth, but later development of an "Ada-like" language compiler eased programming to create important libraries of commonly used functions. VAX, IBM PC-AT, and Sun 3 workstation host support existed. Military and space systems leveraged GAPP for use in image processing, and the construction of the largest array of processing elements of their generation (82,944 processing cells in one system reported in 1988) (Fig. 8).

Warp and iWarp

The Warp project (1984–1993) was a series of increasingly general-purpose programmable Systolic Array



Systolic Arrays. Fig. 7 PSC cell architecture



Systolic Arrays. Fig. 8 Geometric arithmetic cell architecture

systems and related software, created by Carnegie Mellon University (CMU) and developed in conjunction with industrial partners G.E., Honeywell, and Intel with funding from the US Defense Advanced Research Projects Agency (DARPA) (see Warp and iWarp). Warp was a highly programmable Systolic Array computer with a linear array of ten or more cells, capable of performing ten million single-precision floating-point operations per second (10 MFLOPS). A ten-cell machine had a peak performance of 100 MFLOPS. The iWarp machines doubled this performance, delivering 20 MFLOPS single-precision and supporting double-precision floating point at half the performance.

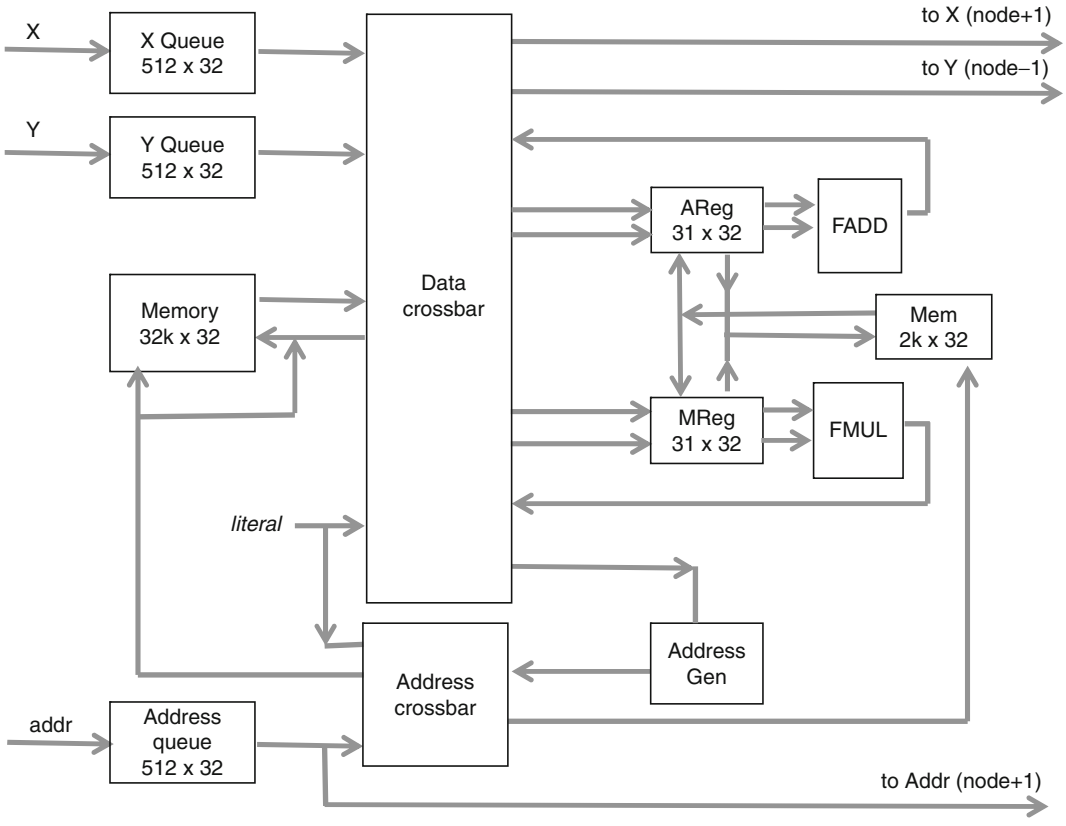
iWarp was based on a full custom VLSI component integrating a 700,000 transistor LIW microprocessor, a network interface, and a switching node into one single chip of $1.2 \times 1.2\,cm$ silicon. The processor dissipated up to 15 watts and was packaged in a ceramic pin grid array with 280 pins. Intel marketed the iWarp with the tag line "Building Blocks for GigaFLOPs." The standard iWarp machines configuration arranged iWarp nodes in a $2m \times 2n$ torus. All iWarp machines included the "back edges" and, therefore, were tori.

Warp and iWarp were programmed using high-level languages and domain-specific program generators. About 20 Warp machines were built, and more than 1,500 iWarp processors were manufactured. Warp and iWarp handled a large number of image and signal processing algorithms (Figs. 9 and 10).

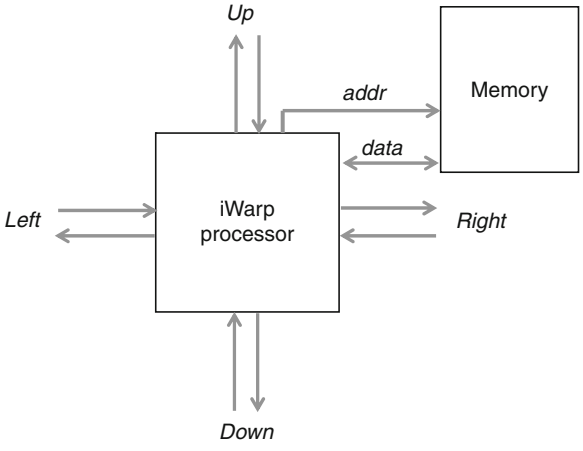
Future Directions

Faced with relatively little growth in transistor speeds coupled with ever-expanding transistor densities in the late 1970s and early 1980s, the use of transistors for many forms for parallelism, from multiple cells to LIW, were urgently explored. Investigations in Systolic Arrays gave rise to fruitful exploration of Systolic Algorithms. Transistor speeds did finally explode, which helped divert interest from parallelism including Systolic Arrays and Systolic Algorithms.

By 2005, once again clock speed gains dramatically slowed, while transistor densities continued growing in accord with Moore's Law as much as they had at the dawn of VLSI. Hardware parallelism, this time as multicore processors, and parallel programming have



Systolic Arrays. Fig. 9 PC-warp cell architecture



Systolic Arrays. Fig. 10 iWarp cell architecture

once again found widespread interest. Single processor designs have become the cells in multicore and many core processor designs, and interconnect debates are being revisited. Interconnection of many processor cores faces the same challenges Systolic Arrays worked to solve the first time the industry found transistor density on the rise without transistor performance moving nearly as quickly, and Systolic Algorithms proved their usefulness via programmability.

Related Entries

►VLIW Processors
►Warp and iWarp

Bibliographic Notes and Further Reading

Application-specific solutions, like the early Systolic Arrays, continue today in the annual *IEEE International Conference on Application-specific Systems, Architectures and Processors*. This conference traces its origins back to the International Workshop on Systolic Arrays, first organized in 1986. It later developed into the International Conference on Application-Specific

Array Processors. With its current title, it was organized for the first time in Chicago, USA, in 1996.

Systolic Arrays were first described in 1978 by H. T. Kung and Charles E. Leiserson [3, 4]. VLSI designs took serious note of the concept and the design principles [4, 6]. Digital systems operating in a synchronous fashion utilizing a central clock predated the description of Systolic Arrays and VLSI by many years, the earliest such machine has been cited as the Colossus [3] built in 1944. Additional reading about actual realizations of Systolic Arrays are available[1, 7–13].

Bibliography

1. Cloud EL (1988) Frontiers of massively parallel computation. In: Proceedings of the 2nd symposium on the frontiers of massively parallel computation, Fairfax, VA, pp 373–381
2. Cragon HG (2003) From fish to colossus: how the German Lorenz cipher was broken at Bletchley park. Cragon Books, Dallas, ISBN: 0-9743045-0-6
3. Frank GA, Greenawalt EM, Kulkarni AV (1982) A systolic processor for signal processing. In: Proceedings of AFIPS '82, June 7–10. ACM, New York, pp 225–231
4. Fisher AL, Kung HT, Monier LM, Dohi Y (1983) Architecture of the PSC: a programmable systolic chip. In: Proceedings of the 10th annual international symposium on computer architecture (ISCA '83), Stockholm, Sweden. ACM, New York, Vol 11, Issue 3, pp 48–53
5. Gross T, O'Hallaron DR (1998) iWarp: anatomy of a parallel computing system. MIT Press, Cambridge, MA, 488 p
6. Kung HT, Leiserson CE (1978) Systolic arrays (for VLSI). In: Duff IS, Stewart GW (eds) Proceedings of sparse matrix proceedings 1978. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, pp 256–282
7. Kung HT, Song SW (1981) A systolic 2-D convolution chip. In: Proceedings of 1981 IEEE computer society workshop on computer architecture for pattern analysis and image database management, 11–13 Nov 1981, Hot Springs, Virginia, pp 159–160
8. Kung HT (1982) Why systolic architectures? IEEE Comput 15(1):37–46
9. Kung SY (1988) VLSI array processors. Prentice Hall, Upper Saddle River
10. Mead C, Conway L (1980) Chap. 8, Highly concurrent systems. In: Introduction to VLSI systems. Addison-Westey series in computer science, Addison-Wesley, Menlo Park, CA, pp 263–332
11. Tirpak FM Jr (1991) Software development on the high-speed systolic array processor (HISSAP): lessons learned. Technical report 1429. Naval Ocean Systems Center, San Diego, CA