H

Half Vector Length

►Metrics

Hang

►Deadlocks

Harmful Shared-Memory Access

►Race Conditions

Haskell

►Glasgow Parallel Haskell (GpH)

Hazard (in Hardware)

►Dependences

HDF5

Quincey Koziol
The HDF Group, Champaign, IL, USA

Synonyms

Hierarchical data format

Definition

HDF5 [1] is a data model, software library, and file format for storing and managing data.

Discussion

Introduction

The HDF5 technology suite is designed to organize, store, discover, access, analyze, share, and preserve diverse, complex data in continuously evolving heterogeneous computing and storage environments. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. The HDF5 library and file format are portable and extensible, allowing applications to evolve in their use of HDF5. The HDF5 technology suite also includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format.

Originally designed within the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign [2], HDF5 is now primarily developed and maintained by The HDF Group [3], a nonprofit organization dedicated to ensuring the sustainable development of HDF technologies and the ongoing accessibility of data stored in HDF files. HDF5 builds on lessons learned from other data storage libraries and file formats, such as the original HDF file format (now known as HDF4 [4]), netCDF [5], TIFF [6], and FITS [7], while adding unique features and extending the boundaries of prior data storage models.

Data Model

HDF5 implements a simple but versatile data model, which has two primary components: groups and datasets. Group objects in an HDF5 file contain a collection of named links to other objects in an HDF5 file. Dataset objects in HDF5 files store arrays of arbitrary element types and are the main method for storing application data.

Groups, which are analogous to directories in a traditional file system, can contain an arbitrary number of uniquely named links. A link can connect a group to another object in the same HDF5 file; include a named path to an object in the HDF5 file, which may not exist currently; or refer to an object in another HDF5 file. Unlike links in a traditional file system, HDF5 links can be used to create fully cyclic directed graph structures. Each group contains one or more B-tree data structures as indices to its collection of links, which are stored in a heap structure within the HDF5 file.

Dataset objects store application data in an HDF5 file as a multidimensional array of elements. Each dataset is primarily defined by the description of how many dimensions its array has and the size of those dimensions, called a "dataspace"; and the description of the type of element to store at each location in the array, called a "datatype."

An HDF5 dataspace describes the number of dimensions for an array, as well as the current and maximum number of elements in each dimension. The maximum number of elements in an array dimension can be specified as "unlimited," allowing an array to be extended over time. An HDF5 dataspace can have multiple dimensions that have unlimited maximum dimensions, allowing that array to be extended in any or all of those dimensions.

An HDF5 datatype describes the type of data to store in each element of an array and can be one of the following classes: integer, floating-point, string, bitfield, opaque, compound, reference, enum, variable-length sequence, and array. These classes generally correspond to the analogous computer science concepts, but the reference and variable-length sequence datatypes are unusual. Reference datatypes contain links to HDF5 objects, allowing HDF5 applications to create datasets that act like groups. The former contain references or pointers to HDF5 objects, allowing HDF5 applications to create datasets that can act as lookup tables or indices. Variable-length sequence datatypes allow a dynamic number of elements of a base datatype to be stored as an element and are one mechanism for creating datasets that represent ragged arrays. All of the HDF5 datatypes can be combined in any arbitrary way, allowing for great flexibility in how an application stores its data.

The elements of an HDF5 dataset can be stored in different ways, allowing an application to choose between various I/O access performance trade-offs.

Dataset elements can be stored as a single sequence in the HDF5 file, called "contiguous" storage, which allows for constant time access to any element in the array and no storage overhead for locating the elements in the dataset. However, contiguous data storage does not allow a dataset to use a dataspace with unlimited dimensions or to compress the dataset elements.

The dataspace for a dataset can also be decomposed into fixed-size sub-arrays of elements, called "chunks," which are stored individually in the file. This "chunked" data storage requires an index for locating the chunks that store the data elements. Datasets that have a data space with unlimited dimensions must use chunked data storage for storing their elements.

Using chunked data storage allows an application that will be accessing sub-arrays of the dataset to tune the chunk size to its sub-array size, allowing for much faster access to those sub-arrays than would be possible with contiguous data storage. Additionally, the elements of datasets that use chunked data storage can be compressed or have other operations, like checksums, etc., applied to them.

The advantages of chunked data storage are balanced by some limitations, however. Using an index for mapping dataset element coordinates to chunk locations in the file can slow down access to dataset elements if the application's I/O access pattern does not line up with the chunk's sub-array decomposition. Furthermore, there is additional storage overhead for storing an index for the dataset, along with extra I/O operations to access the index data structure.

Datasets with very small amounts of element data can store their elements as part of the dataset description in the file, avoiding any extra I/O accesses to retrieve the dataset elements, since the HDF5 library will read them when accessing the dataset description. This "compact" data storage must be very small (less than 64 KB), and may not be used with a dataspace that has unlimited dimensions or when dataset elements are compressed.

Finally, a dataset can store its elements in a different, non-HDF5, file. This "external" data storage method can be used to share dataset elements between an HDF5 application and a non-HDF5 application. As with contiguous data storage, external data storage does not allow a dataset to use a dataspace with unlimited dimensions or compress the dataset elements.

HDF5 also allows application-defined metadata to be stored with any object in an HDF5 file.

These "attributes" are designed to store information about the object they are attached to, such as input parameters to a simulation, the name of an instrument gathering data, etc. Attributes are similar to datasets in that they have a dataspace, a datatype, and elements values. Attributes require a name that is unique among the attributes for an object, similar to link names within groups. Attributes are limited to dataspaces that do not use unlimited maximum dimensions and cannot have their data elements compressed, but can use any datatype for their elements.

Examples of Using HDF5

The following C code example shows how to use the HDF5 library to store a large array. In the example below, the data to be written to the file is a three-dimensional array of single-precision floating-point values, with dimensions of 1024 elements along each axis:

```
1   float data[1024][1024][1024];
2   hid_t file_id, dataspace_id, dataset_id;
3   hsize_t dims[3] = {1024, 1024, 1024};
4
5   < . . .acquire or assign data values. . . >
6
7   file_id = H5Fcreate("example.h5",
    H5F_ACC_TRUNC, H5P_DEFAULT,
    H5P_DEFAULT);
8   dataspace_id = H5Screate_simple(3, dims,
    NULL);
9   dataset_id = H5Dcreate(file_id, "/Float_data",
    H5T_NATIVE_FLOAT, dataspace_id,
    H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
10
11  H5Dwrite(dataset_id, H5T_NATIVE_FLOAT,
    dataspace_id, dataspace_id, H5P_DEFAULT,
    data);
11
12  H5Dclose(dataset_id);
13  H5Sclose(dataspace_id);
14  H5Fclose(file_id);
```

In this example, lines 1–3 declare the variables needed for the example, including the 3-D array of data to store. Line 5 represents the application's process of filling the data array with information. Lines 7–9 create a new HDF5 file, a new dataspace describing a fixed-size three-dimensional array of dimensions 1024 × 1024 × 1024, and a new dataset using a single-precision floating-point datatype and the dataspace created. Line 11 writes the entire 4 GB array to the file in a single I/O operation, and lines 13–15 close the objects created earlier. Several of the calls use H5P_DEFAULT as a parameter, which is a placeholder for an HDF5 property list object, which can control more complicated properties of objects or operations.

The next C code example creates an identically structured file, but adds the necessary calls to open the file with 8 processes in parallel and to perform a collective write to the dataset created.

```
1   float data[512][512][512];
2   hid_t file_id, file_dataspace_id,
    mem_dataspace
    _id, dataset_id, fa_plist_id, dx_plist_id;
3   hsize_t file_dims[3] = {1024, 1024, 1024};
4   hsize_t mem_dims[3] = {512, 512, 512};
5
6   < . . .acquire or assign data values. . . >
7
8   fa_plist_id = H5Pcreate(H5P_FILE_ACCESS);
9   H5Pset_fapl_mpio(fa_plist_id,
    MPI_COMM_WORLD, MPI_INFO_NULL);
10  file_id = H5Fcreate("example.h5", H5F_ACC_
    TRUNC, H5P_DEFAULT, fa_plist_id);
11  H5Pclose(fa_plist_id);
12  file_dataspace_id = H5Screate_simple(3,
    file_dims, NULL);
13  dataset_id = H5Dcreate(file_id, "/Float_data",
    H5T_NATIVE_FLOAT, file_dataspace_id,
    H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
14
15  mem_dataspace_id = H5Screate_simple(3,
    mem_dims, NULL);
```

16	
17	< . . .select process's elements in file dataspace. . . >
18	
19	dx_plist_id = H5Pcreate(H5P_DATASET_XFER);
20	H5Pset_dxpl_mpio(dx_plist_id, H5FD_MPIO_COLLECTIVE);
21	H5Dwrite(dataset_id, H5T_NATIVE_FLOAT, mem_dataspace_id, file_dataspace_id, dx_plist_id, data);
22	
23	H5Pclose(dx_plist_id);
24	H5Dclose(dataset_id);
25	H5Sclose(mem_dataspace_id);
26	H5Sclose(file_dataspace_id);
27	H5Fclose(file_id);

In this updated example, the size of the data array on line 1 has been changed to be only one eighth of the total array size, to allow for each of the eight processes to write a portion of the total array in the file. Lines 8–10 have been updated to create a file access property list, change the file driver for opening the file to use MPI-I/O and collectively open the file with all processes, using the file access property list. Lines 12–13 create the dataset in the file in the same way as the previous example. Line 15 creates a dataspace for each process's portion of the dataset in the file, and line 17 represents a section of code for selecting the part of the file's dataspace that each process will write to (which is omitted due to space constraints). Lines 19–21 create a dataset transfer property list, set the I/O operation to be collective, and perform a collective write operation where each process writes a different portion of the dataset in the file. Finally, lines 23–27 release resources used for the example.

This example shows some ways that property lists can be used to modify the operation of HDF5 API calls, as well as demonstrating a simple example of parallel I/O using HDF5 API calls.

Higher-Level Data Models Built on HDF5

HDF5 provides a set of generic higher-level data models that describe how to store images and tables as datasets and describe the coordinates of dataspace elements. A scientific user community can also use HDF5 as the basis for exchanging data among its members by creating a standardized domain-specific data model that is relevant to their area of interest. Domain-specific data models specify the names of HDF5 groups, datasets and attributes, the dataspace and datatype for the datasets and attributes, etc. Frequently, a domain's user community also creates a "wrapper library" that calls HDF5 library routines while enforcing the domain's standardized data model.

Library Interface

Software applications create, modify, and delete HDF5 objects through an object-oriented library interface that manipulates the objects in HDF5 files. Applications can use the HDF5 library to operate directly on the base HDF5 objects or use a domain-specific wrapper library that operates at a higher level of abstraction. The core software library for accessing HDF5 files is written in C, but library interfaces for the HDF5 data model have been created for many programming languages, including Fortran, C++, Java, Python, Perl, Ada, and C#.

File Format

Objects in the HDF5 data model created by the library interface are stored in files whose structure is defined by the HDF5 file format. The HDF5 file format has many unique aspects, some of which are: a mechanism for storing non-HDF5 formatted data at the beginning of a file, a method of "micro-versioning" file data structures that makes incremental changes to the format possible, and data structures that enable constant-time lookup of data within the file in situations which previously required a logarithmic number of operations.

The HDF5 file format is designed to be flexible and extensible, allowing for evolution and expansion of the data model in an incremental and structured way. This allows new releases of the HDF5 software library to continue to access all previous versions of the HDF5 file format. This capability empowers application developers to create HDF5 files and access data contained within them over very long periods of time.

Tools

HDF5 is distributed with command-line utilities that can inspect and operate on HDF5 files. Operations provided by command-line utilities include copying HDF5 objects from one file to another, compacting internally fragmented HDF5 files to reduce their size, and comparing two HDF5 files to determine differences in the objects contained within them. The latter differencing utility, called "h5diff", is also provided as a parallel computing application that uses the MPI programming interface to quickly compare two files using multiple processes.

Many other applications, both commercial and open source, can access data stored in HDF5 files. Some of these applications include MATLAB [8], Mathematica [9], HDFView [10], VisIt [11], and EnSight [12]. Some of these applications provide generic browsing and modification of HDF5 files, while others provide specialized visualization of domain-specific data models stored in HDF5 files.

Parallel File I/O

Applications that use the MPI programming interface [13] can use the HDF5 library to access HDF5 files in parallel from multiple concurrently executing processes. Internally, the HDF5 library uses the MPI interface for coordinating access to the HDF5 file as well as for performing parallel operations on the file. Efficiently accessing an HDF5 file in parallel requires storing the file on a parallel file system designed for access through the MPI interface.

Two methods of accessing an HDF5 file in parallel are possible: "independent" and "collective." Independent parallel access to an HDF5 file is performed by a process in a parallel application without coordination with or cooperation from the other processes in the application. Collective parallel access to an HDF5 file is performed with all the processes in the parallel application cooperating and possibly communicating with each other.

The following discussion of HDF5 library capabilities describes parallel I/O features in the current release at the time this entry was written, release 1.8.6. The parallel I/O features in the HDF5 library are continually improving and evolving to address the ongoing changes in the landscape of parallel computing. Unless otherwise stated, limitations in the capabilities of the HDF5 library are not inherent to the HDF5 data model or file format and may be addressed in future library releases.

The HDF5 library requires that operations that create, modify, or delete objects in an HDF5 file be performed collectively. However, operations that only open objects for reading can be performed independently. Requiring collective operations for modifying the file's structure is currently necessary so that all processes in the parallel application keep a consistent view of the file's contents.

Reading or writing the elements of a dataset can be performed independently or collectively. Accessing the elements of a dataset with independent or collective operations involves different trade-offs that application developers must balance.

Using independent operations requires a parallel application to create the overall structure of the HDF5 file at the beginning of its execution, or possibly with a nonparallel application prior to the start of the parallel application. The parallel application can then update elements of a dataset without requiring any synchronization or coordination between the processes in the application. However, the application must subdivide the portions of the file accessed from each process to avoid race conditions that would affect the contents of the file. Additionally, accessing a file independently may cause the underlying parallel file system to perform very poorly, due to its lack of a global perspective on the overall access pattern, which prevents the file system from taking advantage of many caching and buffering opportunities available with collective operations.

Accessing HDF5 dataset elements collectively requires that all processes in the parallel application cooperate when performing a read or write operation. Collective operations use the MPI interface within the HDF5 library to describe the region(s) of the file to access from each process, and then use collective MPI I/O operations to access the dataset elements. Collectively accessing HDF5 dataset elements allows the MPI implementation to communicate between processes in the application to determine the best method of accessing the parallel file system, which can greatly improve performance of the I/O operation. However, the communication and synchronization overhead of collective access can also slow down an application's overall performance.

To get good performance when accessing an HDF5 dataset, it is important to choose a storage method that is compatible with the type of parallel access chosen. For example, compact data storage requires all writes to dataset elements be performed collectively, while external data storage requires all dataset element accesses be performed independently.

Collective and independent element access also involves the MPI and parallel file system layers, and those layers add their own complexities to the equation. HDF5 application developers must carefully balance the trade-offs of collective and independent operations to determine when and how to use them.

High performance access to HDF5 files is a strongly desired feature of application developers, and the HDF5 library has been designed with the goal of providing performance that closely matches the performance possible when an application accesses unformatted data directly. Considerable effort has been devoted to enhancing the HDF5 library's parallel performance, and this effort continues as new parallel I/O developments unfold.

Significant Parallel Applications and Libraries That Use HDF5

Many applications and software libraries that use HDF5 have become significant software assets, either commercially or as open source projects governed by a user community. HDF5's stability, longevity, and breadth of features have attracted many developers to it for storing their data, both for sequential and parallel computing purposes.

The first software library to use HDF5 was developed collaboratively by developers at the US Department of Energy's (DOE) Lawrence Livermore, Los Alamos and Sandia National Laboratories. This effort was called the "Sets and Fields" (SAF) library [14] and was developed to give the parallel applications that dealt with large, complex finite element simulations on the highest performing computers of the time a way to efficiently store their data.

Many large scientific communities worldwide have adopted HDF5 for storing data with parallel applications. Some significant examples include the FLASH software for simulating astrophysical nuclear flashes from the University of Chicago [15], the Chombo package for solving finite difference equations using adaptive mesh refinement from Lawrence Berkeley National Laboratory [16], and the open source NeXus software and data format for interchanging data in the neutron, x-ray, and muon science communities [17].

netCDF, PnetCDF, and HDF5

Another significant software library that uses HDF5 is the netCDF library [5], developed at the Unidata Program Center for the University Corporation for Atmospheric Research (UCAR). Originally designed for the climate modeling community, netCDF has since been embraced by many other scientific communities. netCDF has adopted HDF5 as its principal storage method, as of version 4.0, in order to take advantage of several features in HDF5 that its previous file format did not provide, including data compression, a wider array of types for storing data elements, hierarchical grouping structures, and more flexible parallel operations.

Created prior to the development of netCDF-4, parallel-netCDF or "PnetCDF" [18] was developed by Argonne National Laboratory. PnetCDF allows parallel applications to access netCDF format files with collective data operations. PnetCDF does not extend the netCDF data model or format beyond allowing larger objects to be stored in netCDF files. Files written by PnetCDF and versions of the netCDF library prior to netCDF-4 do not use the HDF5 file format and instead store data in the "netCDF classic" format [19].

Future Directions

Both the primary development team at The HDF Group and the user community that has formed around the HDF5 project are constantly improving it. HDF5 continues to be ported to new computer systems and architectures and has its performance improved and errors corrected over time. Additionally, the HDF5 data model is expanding to encompass new developments in the field of high performance storage and computing.

Some improvements being designed or implemented as of this entry's writing include: increasing the efficiency of small file I/O operations through advanced caching mechanisms, finding ways to allow parallel applications to create HDF5 objects in a file with independent operations, and implementing new chunked data storage indexing methods to improve collective access performance.

Additionally, HDF5 continues to lead the scientific data storage field in its adoption of asynchronous file I/O for improved performance, journaled file updates for improved resiliency, and methods for improving concurrency by allowing different applications to read and write to the same HDF5 file without using a locking mechanism.

Related Entries

►File Systems

►MPI (Message Passing Interface)

►NetCDF I/O Library, Parallel

Bibliographic Notes and Further Reading

HDF5 has been under development since 1996, a short history of its development is recorded at The HDF Group web site [20].

Datasets in HDF5 files are analogous to sections of trivial fiber bundles [21], where the HDF5 dataspace corresponds to a fiber bundle's base space, the HDF5 datatype corresponds to the fiber, and the dataset, a variable whose value is the totality of the data elements stored, represents a section through the total space (which is the Cartesian product of the base space and the fiber).

HDF5 datatypes can be very complex and many more details are found in reference [22].

The HDF5 file format is documented in [23].

Many more applications and libraries use HDF5 than are discussed in this entry. A partial list can be found in [24].

Bibliography

1. HDF5, http://www.hdfgroup.org/HDF5/
2. National Center for Supercomputing Application at the University of Illinois at Urbana-Champaign, http://www.ncsa.illinois.edu/
3. The HDF group, http://www.hdfgroup.org/
4. HDF4, http://www.hdfgroup.org/products/hdf4/
5. netCDF, http://www.unidata.ucar.edu/software/netcdf/
6. TIFF, http://partners.adobe.com/public/developer/tiff/index.html
7. FITS, http://fits.gsfc.nasa.gov/
8. MATLAB, http://www.mathworks.com/products/matlab/
9. Mathematica, http://www.wolfram.com/products/mathematica/index.html
10. HDFView, http://www.hdfgroup.org/hdf-java-html/hdfview/
11. VisIt, https://wci.llnl.gov/codes/visit/
12. EnSight, http://www.ensight.com/
13. MPI, http://www.mpi-forum.org/
14. Miller M et al (2001) Enabling interoperation of high performance, scientific computing applications: modeling scientific data with the sets & fields (SAF) modeling system. ICCS – 2001, May, part II, San Francisco. Lecture Notes in Computer Science, vol 2074. Springer, Heidelberg, pp 158–168
15. FLASH, http://flash.uchicago.edu/web/
16. Chombo, https://seesar.lbl.gov/anag/chombo/index.html
17. NeXus, http://www.nexusformat.org/Main_Page
18. PnetCDF, http://trac.mcs.anl.gov/projects/parallel-netcdf
19. netCDF file formats, http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/File-Format.html
20. A history of the HDF group, http://www.hdfgroup.org/about/history.html
21. Fiber bundle definition, http://mathworld.wolfram.com/FiberBundle.html
22. HDF5 user guide, Chapter 6: datatypes, http://www.hdfgroup.org/HDF5/doc/UG/UG_frame11Datatypes.html
23. HDF5 file format specification, http://www.hdfgroup.org/HDF5/doc/H5.format.html
24. Summary of software using HDF5, http://www.hdfgroup.org/products/hdf5_tools/SWSummarybyName.htm

HEP, Denelcor

►Denelcor HEP

Heterogeneous Element Processor

►Denelcor HEP

Hierarchical Data Format

►HDF5

High Performance Fortran (HPF)

►HPF (High Performance Fortran)

High-Level I/O Library

►NetCDF I/O Library, Parallel

High-Performance I/O

►I/O

Homology to Sequence Alignment, From

Wu-Chun Feng[1,2], Heshan Lin[1]
[1]Virginia Tech, Blacksburg, VA, USA
[2]Wake Forest University, Winston-Salem, NC, USA

Discussion

Two sequences are considered to be *homologous* if they share a common ancestor. Sequences are either homologous or nonhomologous, but not in-between [13]. Determining whether two sequences are actually homologous can be a challenging task, as it requires inferences to be made between the sequences. Further complicating this task is the potential that the sequences may *appear* to be related via chance similarity rather than via common ancestry.

One approach toward determining homology entails the use of sequence-alignment algorithms that maximize the similarity between two sequences. For homology modeling, these alignments could be used to obtain the likely amino-acid correspondence between the sequences.

Introduction

Sequence alignment identifies similarities between a pair of biological sequences (i.e., pairwise sequence alignment) or across a set of multiple biological sequences (i.e., multiple sequence alignment). These alignments, in turn, enable the inference of functional, structural, and evolutionary relationships between sequences. For instance, sequence alignment helped biologists identify the similarities between the SARS virus and the more well-studied coronaviruses, thus enhancing the biologists' ability to combat the new virus.

Pairwise Sequence Alignment

There are two types of pairwise alignment: *global alignment* and *local alignment*. Global alignment seeks to align a pair of sequences entirely to each other, i.e., from one end to the other. As such, it is suitable for comparing sequences with roughly the same length, e.g., two closely homologous sequences. Local alignment seeks to identify significant matches between parts of the sequences. It is useful to analyze partially related sequences, e.g., protein sequences that share a common domain.

Many approaches have been proposed for aligning a pair of sequences. Among them, dynamic programming is a common technique that can find optimal alignments between sequences. Dynamic programming can be used for both local alignment and global alignment. The algorithms in both cases are quite similar. The scoring model of an alignment algorithm is given by a substitution matrix and a gap-penalty function. A substitution matrix stores a matching score for every possible pair of letters. A matching score is typically measured by the frequency that a pair of letters occurs in the known homologous sequences according a certain statistical model. Popular substitution matrices include PAM and BLOSUM, an example of which is shown in Fig. 1. A gap-penalty function defines how gaps in the alignments are weighed in alignment scores. For instance, with a linear gap-penalty function, the penalty score grows linearly with the length of a gap. With an affine gap-penalty function, the penalty factors are differentiated for the opening and the extension of a gap.

The following discussion will focus on the basic algorithm and its parallelization of Smith–Waterman, a popular local-alignment tool based on dynamic programming. (For more advanced techniques to compute pairwise sequence alignment, the reader should consult the "Bibliographic Notes and Further Reading" section at the end of this entry.)

Case Study: Smith–Waterman Algorithm

Given two sequences $S_1 = a_1a_2\cdots a_m$ and $S_2 = b_1b_2\cdots b_n$, the Smith–Waterman algorithm uses an m by n scoring matrix H to calculate and track the alignments. A cell $H_{i,j}$ stores the highest similarity score that can be achieved by any possible alignment ending at a_i and b_j. The Smith–Waterman algorithm has three phases: *initialization*, *matrix filling*, and *traceback*.

The initialization phase simply assigns a value of 0 to each of the matrix cells in the first row and the first column. In the matrix-filling phase, the problem of aligning the two whole sequences is broken into smaller subproblems, i.e., aligning partial sequences. Accordingly, a cell $H_{i,j}$ is updated based on the values of its preceding neighbors. For the sake of illustration, the rest of the discussion assumes a linear gap-penalty function

C	9																			
S	–1	4																		
T	–1	1	4																	
P	–3	–1	1	7																
A	0	1	–1	–1	4															
G	–3	0	1	–2	0	6														
N	–3	1	0	–2	–2	0	6													
D	–3	0	1	–1	–2	–1	1	6												
E	–4	0	0	–1	–1	–2	0	2	5											
Q	–3	0	0	–1	–1	–2	0	0	2	5										
H	–3	–1	0	–2	–2	–2	1	1	0	0	8									
R	–3	–1	–1	–2	–1	–2	0	–2	0	1	0	5								
K	–3	0	0	–1	–1	–2	0	–1	1	1	–1	2	5							
M	–1	–1	–1	–2	–1	–3	–2	–3	–2	0	–2	–1	–1	5						
I	–1	–2	–2	–3	–1	–4	–3	–3	–3	–3	–3	–3	–3	1	4					
L	–1	–2	–2	–3	–1	–4	–3	–4	–3	–2	–3	–2	–2	2	2	4				
V	–1	–2	–2	–2	0	–3	–3	–3	–2	–2	–3	–3	–2	1	3	1	4			
F	–2	–2	–2	–4	–2	–3	–3	–3	–3	–3	–1	–3	–3	0	0	0	–1	6		
Y	–2	–2	–2	–3	–2	–3	–2	–3	–2	–1	2	–2	–2	–1	–1	–1	–1	3	7	
W	–2	–3	–3	–4	–3	–2	–4	–4	–3	–2	–2	–3	–3	–1	–3	–2	–3	1	2	11
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W

Homology to Sequence Alignment, From. Fig. 1 BLOSUM62 substitution matrix

where the penalty of a gap is equal to a constant factor g (typically negative) times the length of the gap.

There are three possible alignment scenarios where $H_{i,j}$ is derived from its neighbors: (1) a_i and b_j are associated, (2) there is a gap in sequence S_1, and (3) there is a gap in sequence S_2. As such, the scoring matrix can be filled according to (1). The first three terms in (1) correspond to the three scenarios; the zero value ensures that there are no negative scores. $S(a_i, b_j)$ is the matching score derived by looking up the substitution matrix.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S(a_i, b_j) \\ H_{i-1,j} + g \\ H_{i,j-1} + g \\ 0 \end{cases} \tag{1}$$

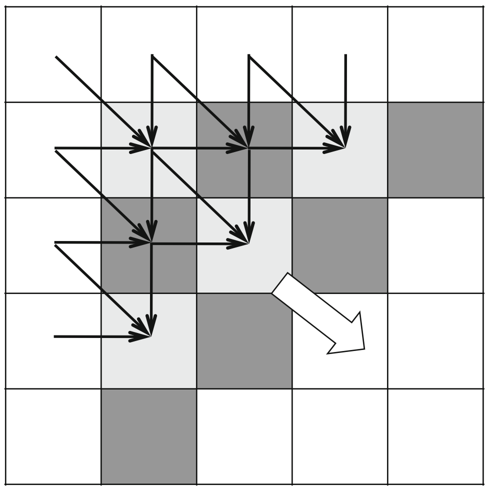
When a cell is updated, the direction from which the maximum score is derived also needs to be stored (e.g., in a separate matrix). After the matrix is filled, a traceback process is used to recover the path of the best alignment. It starts from the cell with the highest score in the matrix and ends at a cell with a value of 0, following the direction information recorded earlier.

The majority of execution time is spent on the matrix-filling phase in Smith–Waterman. Algorithm 1 shows a straightforward implementation of the matrix filling. In the inner loop of the algorithm, the cell calculated in one iteration depends on the value updated in the previous iteration, resulting in a "read-after-write" hazard (see [14] for details), which can reduce the instruction-level parallelism that can be exploited by pipelining, and hence, adversely impact performance. In addition, this algorithm is difficult to directly parallelize because of the data dependency between iterations in the inner loop.

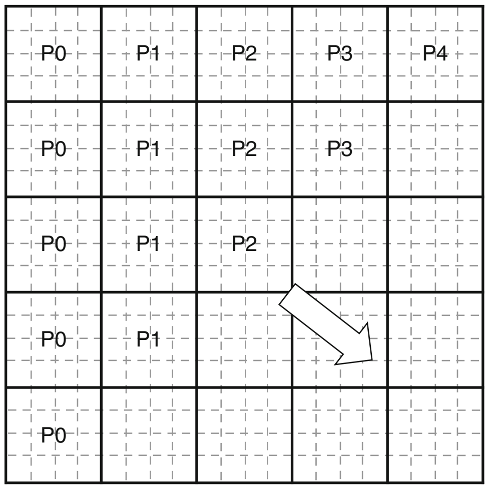
Algorithm 1 Matrix Filling in Smith–Waterman

```
for $i = 1$ to $m$ do
  for $j = 1$ to $n$ do
    $max = H_{i-1,j-1} + S(a_i, b_j) > H_{i-1,j} + g \,?\, H_{i-1,j-1} + S(a_i, b_j) : H_{i-1,j} + g$
    if $H_{i,j-1} + g > max$ then
      $H_{i,j} = H_{i,j-1} + g$
    else
      $H_{i,j} = max$
    end if
  end for
end for
```

As depicted in Fig. 2, the calculation of a particular cell depends on its west, northwest, and north neighbors. However, the updates of individual cells along an anti-diagonal are independent. This observation



Homology to Sequence Alignment, From. Fig. 2 Data dependency of matrix filling



Homology to Sequence Alignment, From. Fig. 3 Tiled implementation

motivates a wavefront-filling algorithm [11], where the matrix cells on an anti-diagonal can be updated simultaneously. That is, because there is no dependency between two adjacent cells along an anti-diagonal, this algorithm greatly reduces read-after-write hazards, and in turn, increases the execution efficiency and ease of parallelization. For example, in a shared-memory environment, individual threads can compute a subset of cells along an anti-diagonal. However, synchronization between threads must occur after computing each anti-diagonal.

Since a scoring matrix is typically stored in "row major" order, as shown in Algorithm 1, the above wavefront algorithm may have a large memory footprint when computing an anti-diagonal, thus limiting the benefits of processor caches. One improvement entails partitioning the matrix into tiles and having each parallel processing unit fill a subset of the tiles, as shown in Fig. 3. By carefully choosing the tile size, data processed by a thread can fit in the processor cache. Furthermore, when parallelized in distributed environments, the tiled approach can effectively reduce internode communication, as compared to the fine-grained wavefront approach, because only elements at the borders of individual tiles need to be exchanged between different compute nodes.

With the wavefront approach, the initial amount of parallelism in the algorithm is low. It gradually increases along each successive anti-diagonal until reaching the maximum parallelism along the longest anti-diagonal, and then monotonically decreases thereafter. A tradeoff needs to be made in choosing the tile size. If the tile size is too large, there is not sufficient parallelism to exploit at the beginning of the wavefront computation, which results in idle resources in systems with a large number processing units. On the other hand, too small a tile size will incur much more synchronization and communication overhead. Nonetheless, the wavefront approach may generate imbalanced workloads on different processors, especially at the beginning and end of the computation. It is worth noting that there is an alternative parallel algorithm that uses prefix-sum to compute the scoring matrix row by row (or column by column) [4], which can generate uniform task distribution among all processors.

The above discussion assumes a simple linear gap-penalty function. In practice, the Smith–Waterman algorithm uses an affine gap-penalty scheme, which requires maintaining three scoring matrices in order to track the gap opening and extension. Consequently, both the time and space usages increase by a factor of three in implementation using affine gap penalties.

Sequence Database Search

With the proliferation of public sequence data, sequence database search has become an important task in sequence analysis. For example, newly discovered sequences are typically searched against a database of sequences with known genes and known functions in order to help predict the functions of the newly discovered sequences. A sequence database-search tool compares a set of query sequences against all sequences in a database with a pairwise alignment algorithm and reports the matches that are statistically significant. Although dynamic-programming algorithms can be used for sequence database search, the algorithms are too computationally demanding to keep up with the database growth. Consequently, heuristic-based algorithms, such as BLAST [2, 3] and FASTA [21], have been developed for rapidly identifying similarities in sequence databases.

BLAST is the most widely used sequence database-search tool. It reduces the complexity of alignment computation by filtering potential matches with common words, called *k*-mers. Specifically, there are four stages in comparing a query sequence and a database sequence.

- Stage 1: The query and the database sequences are parsed into words of length k (k is 3 for protein sequences and 11 for DNA sequences by default). The algorithm then matches words between the query sequence and the database sequence and calculates an alignment score for each matched word, based on a substitution matrix (e.g., BLOSUM62). Only matched words with alignment scores higher than a threshold are kept for the next stage.
- Stage 2: For a high-scoring matched word, ungapped alignment is performed by extending the matched word in both directions. An alignment score will be calculated along the extension. The extension stops when the alignment score stops increasing and slightly drops off from the maximum alignment score (controlled by another threshold).
- Stage 3: Ungapped alignments with scores larger than a given threshold obtained from stage 2 are chosen as seed alignments. Gapped alignments are then performed on the seed alignments using a dynamic-programming algorithm, following both forward and backward directions.
- Stage 4: Traceback is performed to recover the paths of gapped alignments.

BLAST calculates the significance of result alignments using Karlin–Altschul statistics. The Karlin–Altschul theory uses a statistic called the e-value (E) to measure the likelihood that an alignment is resulted from matches by chance (i.e., matches between random sequences) as compared to true homologous relationships. The e-value can be calculated according to (2):

$$E = Kmn \cdot e^{-\lambda S} \tag{2}$$

where K and λ are the Karlin–Altschul parameters, m and n are the query length and the total length of all database sequences, and S is the alignment score. The e-value indicates how many alignments with a score higher than S can be found by chance in the search space, i.e., the multiple of the query sequence length and the database length. The lower the e-value, the more significant is an alignment. The alignment results of a query sequence are sorted in the order of e-value.

A sequence database-search job needs to compute $M \times N$ pairwise sequence alignments, where M and N are the numbers of the query and database sequences, respectively. This computation can be parallelized with a coarse-grained approach, where the alignments of individual pairs of sequences are assigned to different processing units.

Early parallel sequence-search software adopted a *query segmentation* approach, where a sequence-search job is parallelized by having individual compute nodes concurrently search disjoint subsets of query sequences against the whole sequence database. Since the searches of individual query sequences are independent, this *embarrassingly parallel* approach is easy to implement and scales well. However, the size of sequence databases is growing much faster than the memory size of a typical single computer. When the database cannot fit in memory, data will be frequently swapped in and out of the memory when searching multiple queries, thus causing significant performance degradation, because disk I/O is several orders of magnitude slower than memory access. Query segmentation can improve the search throughput, but it does not reduce the response time taken to search a single query sequence.

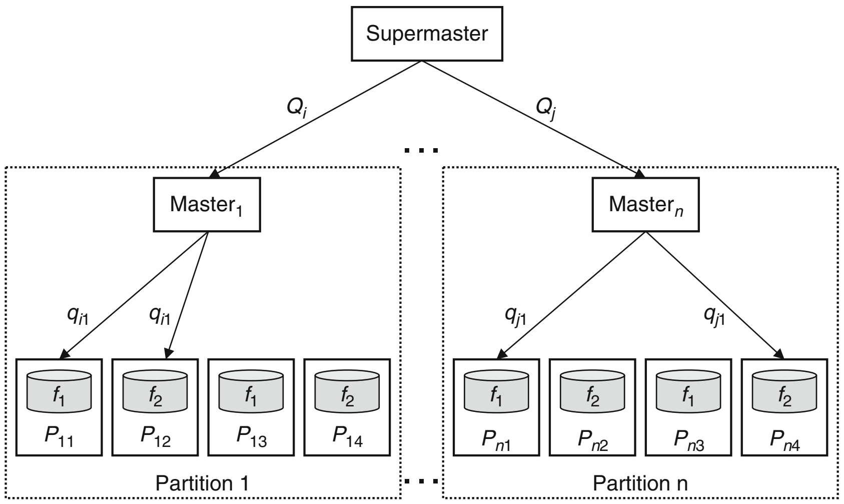
Database segmentation is an alternative parallelization approach, where large databases are partitioned and cached in the aggregate memory of a group of compute nodes. By doing so, the repeated I/O overhead of searching large databases is avoided. Database segmentation also improves the search response time since a compute node searches only a portion of the database. However, database segmentation introduces computational dependencies between individual nodes because the distributed results generated at different nodes need to be merged and sorted to produce the final output. The parallel overhead of merging and sorting increases as the system size grows.

With the astronomical growth of sequence databases, today's large-scale sequence search jobs can be very resource demanding. For instance, BLAST searching in a metagenomics project can consume several millions of processor hours. Massively parallel sequence-search tools, such as mpiBLAST [6, 19, 20] and ScalaBLAST [33], have been developed to accelerate large-scale sequence search jobs on state-of-the-art supercomputers. These tools use a combination of query segmentation and database segmentation to offer massive parallelism needed to scale on a large number of processors.

Case Study: mpiBLAST

mpiBLAST is an open-source parallelization of NCBI BLAST that has been designed for petascale deployment. Adopting a scalable, hierarchical design, mpiBLAST parallelizes a search job via a combination of query segmentation and database segmentation. As shown in Fig. 4, processors in the system are organized into equal-sized *partitions*, which are supervised by a dedicated *supermaster* process. The supermaster is responsible for assigning tasks to different partitions and handling inter-partition load balancing. Within each partition, there is one *master* process and many *worker* processes. The master is responsible for coordinating both computation and I/O scheduling in a partition. The master periodically fetches a subset of query sequences from the supermaster and assigns them to workers, and it coordinates output processing of queries that have been processed in the partition. The sequence database is partitioned into fragments and replicated to workers in the system. This hierarchical design avoids creating scheduling bottlenecks in large systems by distributing scheduling workloads to multiple masters.

Large-scale sequence searches can be highly data-intensive, and as such, the efficiency of data management is critical to the program scalability. For the input data, having thousands of processors simultaneously load database fragments from shared storage may overwhelm the I/O subsystem. To address this, mpiBLAST designates a set of compute nodes as I/O proxies, which read database fragments from the file system in parallel and replicate them to other workers using the broadcasting mechanism in MPI [28, 29] libraries. In addition, mpiBLAST allows workers to cache



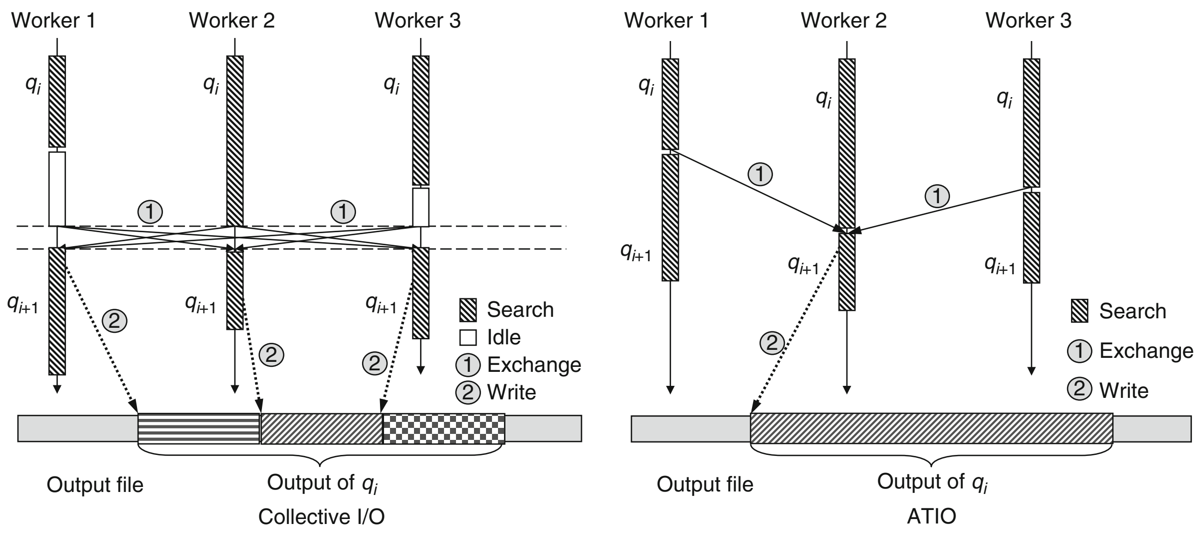
Homology to Sequence Alignment, From. Fig. 4 mpiBLAST hierarchical design. Q_i and Q_j are query batches fetched from the supermaster to masters, and q_{i1} and q_{j1} are query sequences that are assigned by masters to their workers. In this example, the database is segmented into two fragments f_1 and f_2 and replicated twice within each partition assigned database fragments in the memory or local storage, and it uses a task-scheduling algorithm that takes into account data locality to minimize repeated loading of database fragments.

With database segmentation, result alignments of different database fragments are usually interleaved in the global output because those alignments need to be sorted by e-values. Consequently, the output data generated at each worker is noncontiguous in the output file. Straightforward noncontiguous I/O with many seek-and-write operations is slow on most file systems. This type of I/O can be optimized with collective I/O [27, 44, 46], which is available in parallel I/O libraries such as ROMIO [45]. Collective I/O uses a two-phase process. In the first phase, involved processes exchange data with each other to form large trunks of contiguous data, which are stored as memory buffers in individual processes. In the second phase, the buffered data is written to the actual file system. Collective I/O improves I/O performance because continuous data accesses are much more efficient than noncontiguous ones. Traditional collective I/O implementations require synchronization between all involved processes for each I/O operation. This synchronization overhead will adversely impact sequence-search performance when computation is imbalanced across different processes. To address this issue, mpiBLAST introduces a parallel I/O technique called asynchronous, two-phase I/O (ATIO), which allows worker processes to rearrange I/O data without synchronizing with each other. Specifically, mpiBLAST appoints a worker as the write leader for each query sequence. The write leader aggregates output data from other workers via nonblocking MPI communication and carries out the write operation to the file system. ATIO overlaps I/O reorganization and sequence-search computation, thus improving the overall application performance. Figure 5 shows the difference between collective I/O and ATIO within the context of mpiBLAST.

Multiple Sequence Alignment

Multiple sequence alignment (MSA) identifies similarities among three or more sequences. It can be used to analyze a family of related sequences to reveal phylogenetic relationships. Other usages of multiple sequence alignment include detection of conserved biological features and genome sequencing. Multiple sequence alignment can also be global or local.

Like pairwise sequence alignment, multiple sequence alignment can be computed using *dynamic*



Homology to Sequence Alignment, From. Fig. 5 Collective I/O and ATIO. Collective I/O requires synchronization and introduces idle waiting in worker processes. ATIO uses a write leader to aggregate noncontiguous data from other workers in an asynchronous manner *programming* algorithms. Although these algorithms can find optimal alignments, the required resources for these algorithms grow exponentially as the number of sequences increases. Suppose there are N sequences and the average sequence length is L, the time and space complexities are both $O(L^N)$. Thus, it is computationally impractical to use dynamic programming to align a large number of sequences.

Many heuristic approaches have been proposed to reduce the computational complexity of multiple sequence alignment. *Progressive alignment methods* [8, 12, 32, 35, 47] use guided pairwise sequence alignment to rapidly construct MSA for a large number of sequences. These methods first build a phylogenetic tree based on all-to-all pairwise sequence alignments using neighbor joining [40] or UPGMA [43] techniques. Guided by the phylogenetic tree, the most similar sequences are first aligned, the less similar sequences are then progressively added to the initial MSA. One problem with progressive methods is that errors that occur in early aligning stages are propagated to the final alignment results. *Iterative alignment methods* [10, 49] address this problem by introducing "correction" mechanisms during the MSA construction process. These methods incrementally align sequences as progressive methods, but continuously adjust the structure of the phylogenetic tree and previously computed alignments according to a certain objective function.

Case Study: ClustalW

ClustalW [47] is a widely used MSA program based on progressive methods. The ClustalW algorithm includes three stages: (1) distance matrix computation, (2) guided tree construction, and (3) progressive alignment. Due to the popularity of ClustalW, its parallelization has been well studied on clusters [9, 17] and multiprocessor systems [5, 30].

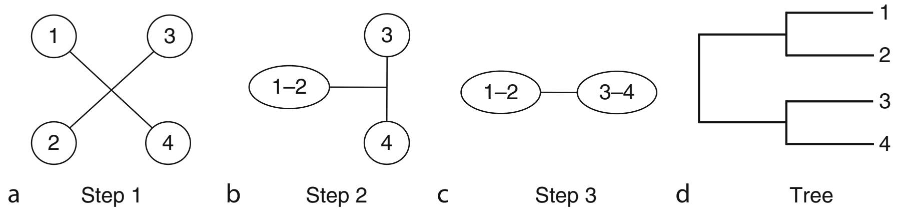
In the first stage, ClustalW computes a distance matrix by performing all-to-all pairwise alignment over input sequences. This requires a total of $\frac{N(N-1)}{2}$ comparison for N sequences since the alignment of a pair of sequences is symmetric. ClustalW allows users to choose between two alignment algorithms: a faster k-mer matching algorithm and a slower but more accurate dynamic programming algorithm. This stage of the algorithm is embarrassingly parallel. Alignments of individual pairs of sequences can be statically assigned or dynamically assigned with a greedy algorithm to different processing units. Additional parallelism can be exploited by parallelizing the alignment of a single pair of sequences, similar to the Smith–Waterman algorithm.

In the second stage, a guided tree is constructed using a neighbor-joining algorithm based on the alignment scores in the distance matrix. Initially all sequences are leaf nodes of a star-like tree. The algorithm iteratively selects a pair of nodes and joins them into a new internal node until there are no nodes left to join, at which point a bifurcated tree is created. Algorithm 2 gives the pseudocode of this process. Suppose there are a total of M nodes at an iteration, there are $\frac{n(n-1)}{2}$ possible pairs of nodes. The pair of nodes that results in the smallest length of branches will be selected to join. An example of neighbor-joining process with four sequences is given in Fig. 6.

In Algorithm 2, the outermost loop cannot be parallelized because one iteration of neighbor joining is dependent on the previous one. However, within an iteration of neighbor joining, the calculation of branch lengths of all pairs of nodes can be performed in parallel. Since the number of available nodes reduces after each joining operation, the parallelism level decreases as the iteration advances in the outermost loop.

Algorithm 2 Construction of Guided Tree

```
while there are nodes to join do
  Let $n$ be the number of current available nodes
  Let $D$ be the current distance matrix
  Let $L_{min}$ be the smallest length of the tree branches
  for $i = 2$ to $n$ do
    for $j = 1$ to $i - 1$ do
      $L(i,j) = (n-2)D_{i,j} - \sum_{k=1}^{n} D_{i,k} - \sum_{k=1}^{n} D_{j,k}$
      if $L(i,j) < L_{min}$ then
        $L_{min} = L(i,j)$
      end if
    end for
  end for
  Combine nodes $i, j$ that result in $L_{min}$ into a new node
  Update distance matrix with the new node
end while
```



Homology to Sequence Alignment, From. Fig. 6 Neighbor-joining example

In the third stage, the sequence is progressively aligned according to the guided tree. For instance, in the tree shown in Fig. 6(d), sequences 1 and 2 are aligned first, followed by 3 and 4. Finally, the alignment results of $< 1 - 2 >$ and $< 3 - 4 >$ are aligned. Note that the alignment on a node can only be performed after both of its children are aligned, but the alignments at the same level of the tree can be performed simultaneously. The level of parallelism of the algorithm depends heavily on the tree structure. In the best case, the guided tree is a balanced tree. At the beginning, all the leaves can be aligned in parallel. The number of concurrent alignments decreases by a half at each higher level toward the root of the tree.

Case Study: T-Coffee

T-Coffee [32] is another popular progressive alignment algorithm. Compared to other typical progressive alignment tools, T-Coffee improves the alignment accuracy by adopting a consistency-based scoring function, which uses similarity information among all input sequences to guide the alignment progress. T-Coffee consists of two main steps: *library generation* and *progressive alignment*. The first step constructs a library that contains a mixture of global and local alignments between *every pair* of input sequences. In the library, an alignment is represented as a list of pairwise constraints. Each constraint stores a pair of matched residues along with an alignment weight, which is essentially a three-tuple $< S_{xi}, S_{yj}, w >$, where S_{xi} is the *ith* residue of sequence S_x and w is the weight. T-Coffee incorporates pairwise global alignments generated by ClustalW as well as top ten nonintersecting local alignments reported by Lalign from the FASTA package [21]. T-Coffee can also take alignment information from other MSA software. The pairwise constraints of a same pair of matched residues from various sources (e.g., global and local alignments) will be combined to remove duplication. Constructing the library requires $\frac{N(N-1)}{2}$ global and local alignments and thus is highly compute-intensive.

After all pairwise alignments are incorporated in the library, T-Coffee performs *library extension*, a procedure that incorporates transitive alignment information to the weighing of pairwise constraints. Basically, for a pair of sequences x and y, if there is a sequence z that aligns to both x and y, then constraints between x and y will be reweighed by combining the weights of the corresponding constraints between x and z as well as y and z. Suppose the average sequence length is L, since for each pair of sequences, the extension algorithm needs to scan the rest of $N - 2$ sequences, and there are at most L constraints between a pair of sequences, the worst computation complexity of library extension is $O(N^3L^2)$.

In the second step, T-Coffee performs progressive alignment guided by a phylogenetic tree built with the neighbor-joining method, similar to the ClustalW algorithm. However, T-Coffee uses the weights in the extended library to align residues when grouping sequences/alignments. Since those weights bear complete alignment information from all input sequences, the progressive alignment in T-Coffee can reduce errors caused by the greediness of classic progressive methods.

Parallel T-Coffee (PTC) is a parallel implementation of T-Coffee in cluster environments [53]. PTC adopts a master–worker architecture and uses MPI to communicate between different processes. As mentioned earlier, the library generation in T-Coffee needs to compute all-to-all global and local alignments. The computation tasks of these alignments are independent of each other. PTC uses guided self-scheduling (GSS) [36] to distribute alignment tasks to different worker nodes. GSS first assigns a portion of the tasks to the workers. Each worker monitors its performance when processing the initial assignments; this performance information is then sent to the master and used for dynamic scheduling for subsequent assignments.

After pairwise alignments are finished, duplicated constraints generated on distributed workers need to be combined. PTC implements this with parallel sorting. Each constraint is assigned to a bucket resident at a worker. Each worker can then concurrently combine duplicated constraints within its own bucket. The constraints in the library are then transformed into a three-dimensional lookup table, with rows and columns indexed by sequences and residues, respectively. Each element in the lookup table stores all constraints for a residue of a sequence. The lookup table will be accessed by all processors during the progressive alignment. PTC evenly distributes the lookup table by rows to all processors and allows table entries to be accessed by other processors through one-sided remote memory access. An efficient caching mechanism is also implemented to improve lookup performance.

During the progressive alignment, PTC schedules tree nodes to a processor according to their readiness; a tree node that has a fewer number of unprocessed child nodes has a higher scheduling priority. For tree nodes that have all child nodes processed, PTC gives higher priority to the ones with shorter estimated execution time. Similar to ClustalW, the parallelism of progress alignment in PTC can be limited if the guided tree is unbalanced. To address this issue, Orobitg et al. proposed a heuristic approach that can construct a more balanced guided tree by allowing a pair of nodes to be grouped if their similarity value is smaller than the average similarity value between all sequences in the distance matrix [34].

Related Entries

►Bioinformatics
►Genome Assembly

Bibliographic Notes and Further Reading

As discussed in the case study of Smith–Waterman, typical sequence-alignment algorithms require $O(mn)$ space and time, where m and n are the lengths of compared sequences. Such a space requirement can be impractical for computing alignments of large sequences (e.g., those with a length of multiple megabytes) on commodity machines. To address this issue, Mayers and Miller introduced a space-efficient alignment algorithm [31] adapted from the Hirschberg technique [15], which was originally developed for finding the longest common subsequence between two strings. By recursively dividing the alignment problem into subproblems at a "midpoint" along the *middle column* of the scoring matrix, Mayers and Miller's approach can find the optimal alignment within $O(m+n)$ space but still $O(mn)$ time. Huang showed that a straightforward parallelization of the Hirschberg algorithm would require more than linear aggregate space [16], i.e., each processor needed to store more than $O\left(\frac{m+n}{p}\right)$ data, where p is the number of concurrent processors. In turn, Huang proposed an improved algorithm that recursively divides the alignment problem at the midpoint along the *middle anti-diagonal* of the scoring matrix. By doing so, Huang's algorithm required only $O\left(\frac{mn}{p}\right)$ space per processor but with an increased time complexity of $O\left(\frac{(m+n)^2}{p}\right)$. Aluru et al. presented an alternative space-efficient parallel algorithm [4] that is more time efficient $\left(O\left(\frac{mn}{p}\right)\right)$ but also consumes more space $\left(O\left(m+\frac{n}{p}\right), m \le n\right)$ than Huang's approach. In their approach, an $O(mn)$ algorithm is first used to partition the scoring matrix into p vertical slices, and the last column of each slice as well as its intersection with the optimal alignment is stored. Each processor then takes a slice and uses a Hirschberg-based algorithm to compute the optimal alignment within the slice. In a subsequent study, Aluru et al. proposed an improved parallel algorithm [37] that requires $O\left(\frac{mn}{p}\right)$ time and $O\left(\frac{m+n}{p}\right)$ space, when $p = O\left(\frac{n}{\log n}\right)$ processors are used. In other words, such a parallel algorithm achieves "optimal" time and space complexities because this algorithm delivers a linear speedup with respect to the best known sequential algorithm.

The computational intensity of sequence-alignment algorithms has motivated studies in parallelizing these algorithms on accelerators. Various algorithms have been accelerated using the SIMD instruction extensions of commodity processors [38, 51], field-programmable gate array (FPGA) [18, 26], Cell Broadband Engine [1, 39, 42], and graphics processing units (GPUs) [22–24, 41, 48, 51, 52]. Developing and optimizing applications on traditional accelerators is much more difficult than on CPUs, which may partially explain why accelerator-based solutions have not been widely adopted even if these solutions have demonstrated very promising performance results. However, the continuing improvement of software environments on commodity GPUs has made them increasingly popular for accelerating sequent alignments. To cope with the astronomical growth of sequence data, cloud-based solutions [7, 25] have also been developed to enable users to tackle large-scale problems with elastic compute resources from public clouds such as Amazon EC2.

Bibliography

1. Aji AM, Feng W, Blagojevic F, Nikolopoulos DS (2008) Cell-SWat: modeling and scheduling wavefront computations on the cell broadband engine. In: CF ’08: Proceedings of the 5th conference on computing frontiers. ACM, New York, pp 13–22
2. Altschul S, Gish W, Miller W, Myers E, Lipman D (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
3. Altschul S, Madden T, Schffer A, Zhang J, Zhang Z, Miller W, Lipman D (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25(17):3389–3402
4. Aluru S, Futamura N, Mehrotra K (2003) Parallel biological sequence comparison using prefix computations. J Parallel Distrib Comput 63:264–272
5. Chaichoompu K, Kittitornkun S, Tongsima S (2006) MT-ClustalW: multithreading multiple sequence alignment. In: International parallel and distributed processing symposium. Rhodes Island, Greece, p 280
6. Darling A, Carey L, Feng W (2003) The design, implementation, and evaluation of mpiBLAST. In: Proceedings of the ClusterWorld conference and Expo, in conjunction with the 4th international conference on Linux clusters: The HPC revolution 2003, San Jose
7. Di Tommaso P, Orobitg M, Guirado F, Cores F, Espinosa T, Notredame C (2010) Cloud-Coffee: implementation of a parallel consistency-based multiple alignment algorithm in the T-Coffee package and its benchmarking on the Amazon Elastic-Cloud. Bioinformatics 26(15):1903–1904
8. Do CB, Mahabhashyam MS, Brudno M, Batzoglou S (2005) ProbCons: probabilistic consistency-based multiple sequence alignment. Genome Res 15(2):330–340
9. Ebedes J, Datta A (2004) Multiple sequence alignment in parallel on a workstation cluster. Bioinformatics 20(7):1193–1195
10. Edgar R (2004) MUSCLE: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics 5(1):113
11. Edmiston EE, Core NG, Saltz JH, Smith RM (1989) Parallel processing of biological sequence comparison algorithms. Int J Parallel Program 17:259–275
12. Feng DF, Doolittle RF (1987) Progressive sequence alignment as a prerequisite to correct phylogenetic trees. J Mol Evol 25(4):351–360
13. Fitch W, Smith T (1983) Optimal sequences alignments. Proc Natl Acad Sci 80:1382–1386
14. Hennessy JL, Patterson DA (2006) Computer architecture: a quantitative approach, 4th edn. Morgan Kaufmann Publishers, San Francisco
15. Hirschberg DS (1975) A linear space algorithm for computing maximal common subsequences. Commun ACM 18: 341–343
16. Huang X (1990) A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. Int J Parallel Program 18:223–239
17. Li K (2003) W-MPI: ClustalW analysis using distributed and parallel computing. Bioinformatics 19(12):1585–1586
18. Li I, Shum W, Truong K (2007) 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). BMC Bioinformatics 8(1):185
19. Lin H, Ma X, Chandramohan P, Geist A, Samatova N (2005) Efficient data access for parallel BLAST. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS’05). IEEE Computer Society, Los Alamitos
20. Lin H, Ma X, Feng W, Samatova NF (2010) Coordinating computation and I/O in massively parallel sequence search. IEEE Trans Parallel Distrib Syst 99:529–543
21. Lipman D, Pearson W (1988) Improved toolsW, HT for biological sequence comparison. Proc Natl Acad Sci 85(8):2444–2448
22. Liu W, Schmidt B, Voss B, Müller-Wittig W (2006) GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment, Chapter 37 In: Robert Y, Parashar M, Badrinath R, Prasanna VK (eds) High performance computing – HiPC 2006. Lecture notes in computer science, vol 4297. Springer, Berlin/Heidelberg, pp 363–374
23. Liu Y, Maskell D, Schmidt B (2009) CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res Notes 2(1):73
24. Liu Y, Schmidt B, Maskell DL (2009) MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In: ASAP ’09: Proceedings of the 2009 20th IEEE international conference on application-specific systems, architectures and processors, Washington, DC. IEEE Computer Society, Los Alamitos, California, USA, pp 121–128
25. Lu W, Jackson J, Barga R (2010) AzureBlast: a case study of cloud computing for science applications. In: 1st workshop on scientific cloud computing, co-located with ACM HPDC 2010 (High performance distributed computing). Chicago, Illinois, USA
26. Mahram A, Herbordt MC (2010) Fast and accurate NCBI BLASTP: acceleration with multiphase FPGA-based prefiltering. In: Proceedings of the 24th ACM international conference on supercomputing. Tsukuba, Ibaraki, Japan
27. May J (2001) Parallel I/O for high performance computing. Morgan Kaufmann Publishers, San Francisco
28. Message Passing Interface Forum (1955) MPI: message-passing interface standard

29. Message Passing Interface Forum (1977) MPI-2 extensions to the message-passing standard
30. Mikhailov D, Cofer H, Gomperts R (2001) Performance optimization of Clustal W: parallel Clustal W, HT Clustal, and MULTICLUSTAL. White Papers, Silicon Graphics, Mountain View
31. Myers EW, Miller W (1988) Optimal alignments in linear space. Comput Appl Biosci (CABIOS) 4(1):11–17
32. Notredame C (2000) T-coffee: a novel method for fast and accurate multiple sequence alignment. J Mol Biol 302(1):205–217
33. Oehmen C, Nieplocha J (2006) ScalaBLAST: a scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. IEEE Trans Parallel Distrib Syst 17(8):740–749
34. Orobitg M, Guirado F, Notredame C, Cores F (2009) Exploiting parallelism on progressive alignment methods. J Supercomput 1–9. doi: 10.1007/s11227-009-0359-5
35. Pei J, Sadreyev R, Grishin NV (2003) PCMA: fast and accurate multiple sequence alignment based on profile consistency. Bioinformatics 19(3):427–428
36. Polychronopoulos CD, Kuck DJ (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36:1425–1439
37. Rajko S, Aluru S (2004) Space and time optimal parallel sequence alignments. IEEE Trans Parallel Distrib Syst 15:1070–1081
38. Rognes T, Seeberg E (2000) Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics 16(8):699–706
39. Sachdeva V, Kistler M, Speight E, Tzeng TK (2008) Exploring the viability of the cell broadband engine for bioinformatics applications. Parallel Comput 34(11):616–626
40. Saitou N, Nei M (1987) The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol Biol Evol 4(4):406–425
41. Sandes EFO, de Melo ACMA (2010) CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. SIGPLAN Not 45(5):137–146
42. Sarje A, Aluru S (2009) Parallel genomic alignments on the cell broadband engine. IEEE Trans Parallel Distrib Syst 20(11): 1600–1610
43. Sneath PH, Sokal RR (1962) Numerical taxonomy. Nature 193:855–860
44. Thakur R, Choudhary A (1996) An extended two-phase method for accessing sections of out-of-core arrays. Sci Program 5(4): 301–317
45. Thakur R, Gropp W, Lusk W (1999) Data sieving and collective I/O in ROMIO. In: Symposium on the frontiers of massively parallel processing. Annapolis, Maryland, USA, p 182
46. Thakur R, Gropp W, Lusk E (1999) On implementing MPI-IO portably and with high performance. In: Proceedings of the sixth workshop on I/O in parallel and distributed systems. Atlanta, Georgia, USA
47. Thompson JD, Higgins DG, Gibson TJ (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 22(22):4673–4680
48. Vouzis PD, Sahinidis NV (2011) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics 27(2):182–188
49. Wallace IM, Orla O, Higgins DG (2005) Evaluation of iterative alignment algorithms for multiple alignment. Bioinformatics 21(8):1408–1414
50. Wozniak A (1997) Using video-oriented instructions to speed up sequence comparison. Comput Appl Biosci 13(2):145–150
51. Xiao S, Aji AM, Feng W (2009) On the robust mapping of dynamic programming onto a graphics processing unit. In: ICPADS ’09: proceedings of the 2009 15th international conference on parallel and distributed systems, Washington, DC. IEEE Computer Society, Los Alamitos, California, USA, pp 26–33
52. Xiao S, Lin H, Feng W (2011) Characterizing and optimizing protein sequence search on the GPU. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium Anchorage, Alaska. IEEE Computer Society, Los Alamitos, California, USA
53. Zola J, Yang X, Rospondek A, Aluru S (2007) Parallel-TCoffee: a parallel multiple sequence aligner. In: ISCA international conference on parallel and distributed computing systems (ISCA PDCS 2007), pp 248–253

Horizon

►Tera MTA

HPC Challenge Benchmark

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

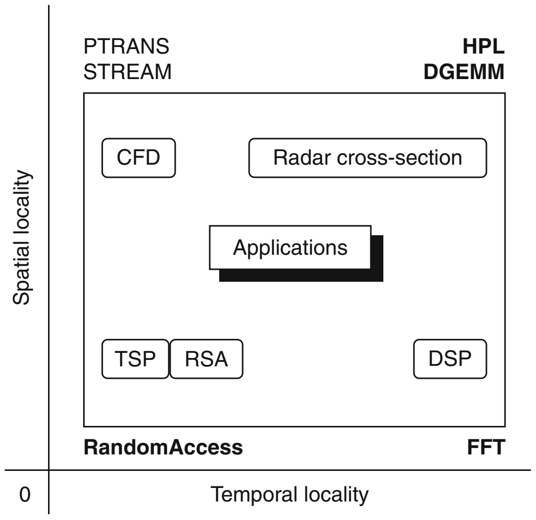
Definition

HPC Challenge (HPCC) is a benchmark that measures computer performance on various computational kernels that span the memory access locality space. HPCC includes tests that are able to take advantage of nearly all available floating point performance: High Performance LINPACK and matrix–matrix multiply allow for data reuse that is only bound by the size of large register file and fast cache. The twofold benefit from these tests is the ability to answer the question how well the hardware is able to work around the "memory wall" and how today's machines compare to the systems of the past as they are cataloged by the LINPACK Benchmark Report [3] and TOP500. HPCC also includes other tests, STREAM, PTRANS, FFT, RandomAccess – when they are combined together they span the memory access locality space. They are able to reveal the growing inefficiencies of the memory subsystem and how they are addressed in the new computer infrastructures. HPCC also offers scientific rigor to the benchmarking effort. The tests stress double precision floating point accuracy: the absolute prerequisite in the scientific world. In addition, the tests include careful verification of the outputs – undoubtedly an important fault-detection feature at extreme computing scales.

Discussion

The HPC Challenge benchmark suite was initially developed for the DARPA HPCS program [6] to provide a set of standardized hardware probes based on commonly occurring computational software kernels. The HPCS program involves a fundamental reassessment of how to define and measure performance, programmability, portability, robustness and, ultimately, productivity across the entire high-end domain. Consequently, the HPCC suite aimed both to give conceptual expression to the underlying computations used in this domain, and to be applicable to a broad spectrum of computational science fields. Clearly, a number of compromises needed to be embodied in the current form of the suite, given such a broad scope of design requirements. HPCC was designed to provide approximate bounds on computations that can be characterized by either high or low spatial and temporal locality (see Fig. 1, which gives the conceptual design space for the HPCC component tests). In addition, because the HPCC tests consist of simple mathematical operations, HPCC provides a unique opportunity to look at language and parallel programming model issues. As such, the benchmark is designed to serve both the system user and designer communities.

Figure 2 shows a generic memory subsystem in the leftmost column and how each level of the hierarchy is tested by the HPCC software (the second column from the left), along with the design goals for the future

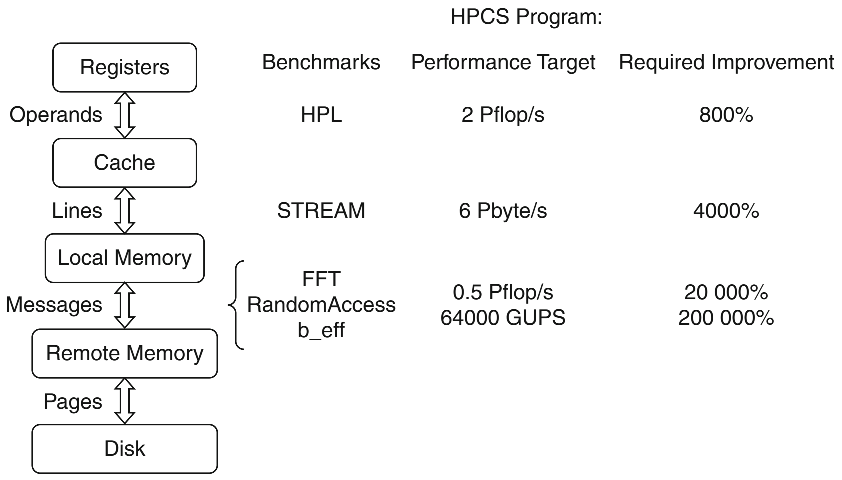


HPC Challenge Benchmark. Fig. 1 The application areas targeted by the HPCS Program are bound by the HPCC tests in the memory access locality space

systems that originated in the HPCS program (the third column from the right). In other words, these are the projected target performance numbers that are to come out of the wining HPCS vendor designs. The last column shows the relative improvement in performance that needs to be achieved in order to meet the goals.

The TOP500 Influence

The most commonly known ranking of supercomputer installations around the world is the TOP500 list [14]. It uses the equally well-known LINPACK benchmark [4] as a single figure of merit to rank 500 of the world's most powerful supercomputers. The often-raised question about the relation between the TOP500 list and HPCC can be addressed by recognizing the positive aspects of the former. In particular, the longevity of the TOP500 list gives an unprecedented view of the high-end arena across the turbulent era of Moore's law [10] rule and the emergence of today's prevalent computing paradigms. The predictive power of the TOP500 list is likely to have a lasting influence in the future, as it has had in the past. HPCC extends the TOP500 list's concept of exploiting a commonly used kernel and, in the context of the HPCS goals, incorporates a larger, growing suite of computational kernels. HPCC has already



HPC Challenge Benchmark. Fig. 2 HPCS program benchmarks and performance targets

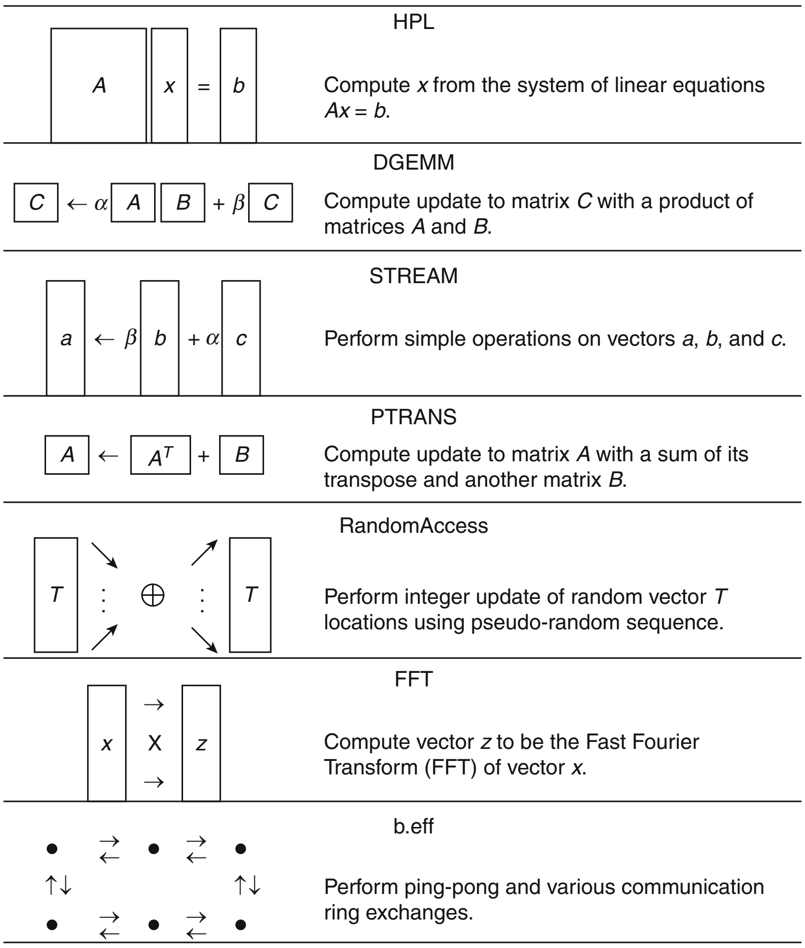
HPC Challenge Benchmark. Table 1 All of the top 10 entries of the 27th TOP500 list that have results in the HPCC database

Rank	Name	Rmax	HPL	PTRANS	STREAM	FFT	RandomAccess	Lat.	B/w
1	BG/L	280.6	259.2	4665.9	160	2, 311	35.47	5.92	0.16
2	BG W	91.3	83.9	171.5	50	1, 235	21.61	4.70	0.16
3	ASC purple	75.8	57.9	553.0	44	842	1.03	5.11	3.22
4	Columbia	51.9	46.8	91.3	21	230	0.25	4.23	1.39
9	Red storm	36.2	33.0	1813.1	44	1, 118	1.02	7.97	1.15

begun to serve as a valuable tool for performance analysis. Table 1 shows an example of how the data from the HPCC database can augment the TOP500 results (for the current version of the table please visit the HPCC website).

Short History of the Benchmark

The first reference implementation of the HPCC suite of codes was released to the public in 2003. The first optimized submission came in April 2004 from Cray, using the then-recent X1 installation at Oak Ridge National Lab. Ever since, Cray has championed the list of optimized HPCC submissions. By the time of the first HPCC birds-of-a-feather session at the Supercomputing conference in 2004 in Pittsburgh, the public database of results already featured major supercomputer makers – a sign that vendors were participating in the new benchmark initiative. At the same time, behind the scenes, the code was also being tried out by government and private institutions for procurement and marketing purposes. A 2005 milestone was the announcement of the HPCC Awards contest. The two complementary categories of the competition emphasized performance and productivity – the same goals as the sponsoring HPCS program. The performance-emphasizing Class 1 award drew the attention of many of the biggest players in the supercomputing industry, which resulted in populating the HPCC database with most of the top 10 entries of the TOP500 list (some exceeding their performances reported on the TOP500 – a tribute to HPCC's continuous results update policy). The contestants competed to achieve the highest raw performance in one of the four


HPL
Compute x from the system of linear equations $Ax = b$.
DGEMM
Compute update to matrix C with a product of matrices A and B.
STREAM
Perform simple operations on vectors a, b, and c.
PTRANS
Compute update to matrix A with a sum of its transpose and another matrix B.
RandomAccess
Perform integer update of random vector T locations using pseudo-random sequence.
FFT
Compute vector z to be the Fast Fourier Transform (FFT) of vector x.
b.eff
Perform ping-pong and various communication ring exchanges.

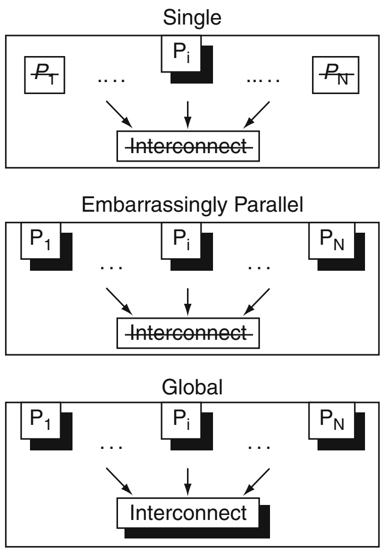
HPC Challenge Benchmark. Fig. 3 Detail description of the HPCC component tests (A, B, C – matrices, a, b, c, x, z – vectors, α, β – scalars, T – array of 64-bit integers)

tests: HPL, STREAM, RANDA, and FFT. At the SC09 conference in Portland, Oregon, HPCC listed its first Pflop/s machine – Cray XT 5 called Jaguar from Oak Ridge National Laboratory. The Class 2 award, by solely focusing on productivity, introduced a subjectivity factor into the judging and also into the submission criteria, regarding what was appropriate for the contest. As a result, a wide range of solutions were submitted, spanning various programming languages (interpreted and compiled) and paradigms (with explicit and implicit parallelism). The Class 2 contest featured openly available as well as proprietary technologies, some of which were arguably confined to niche markets and some were widely used. The financial incentives for entering turned out to be all but needless, as the HPCC seemed to have gained enough recognition within the high-end community to elicit entries even without the monetary assistance. (HPCwire provided both press coverage and cash rewards for the four winning contestants in Class 1 and the single winner in Class 2.) At the HPCC's second birds-of-a-feather session during the SC07 conference in Seattle, the former class was dominated by IBM's BlueGene/L at Lawrence Livermore National Lab, while the latter class was split among MTA pragma-decorated C and UPC codes from Cray and IBM, respectively.

The Benchmark Tests' Details

Extensive discussion and various implementations of the HPCC tests are available elsewhere [5, 8, 15]. However, for the sake of completeness, this section provides the most important facts pertaining to the HPCC tests' definitions.

All calculations use double precision floating-point numbers as described by the IEEE 754 standard [1], and no mixed precision calculations [9] are allowed. All the tests are designed so that they will run on an arbitrary number of processors (usually denoted as p). Figure 3 shows a more detailed definition of each of the seven tests included in HPCC. In addition, it is possible to run the tests in one of three testing scenarios to stress various hardware components of the system. The scenarios are shown in Fig. 4. In the "Single" scenario, only one process is chosen to run the test. Accordingly, the remaining processes remain idle and so does the interconnect (shown with strike-out font in the Figure). In the "Embarrassingly Parallel" scenario, all process run the tests simultaneously but they do not communicate with each other. And finally, in the "Global" scenario, all components of the system work together on all tests.



HPC Challenge Benchmark. Fig. 4 Testing scenarios of the HPCC components

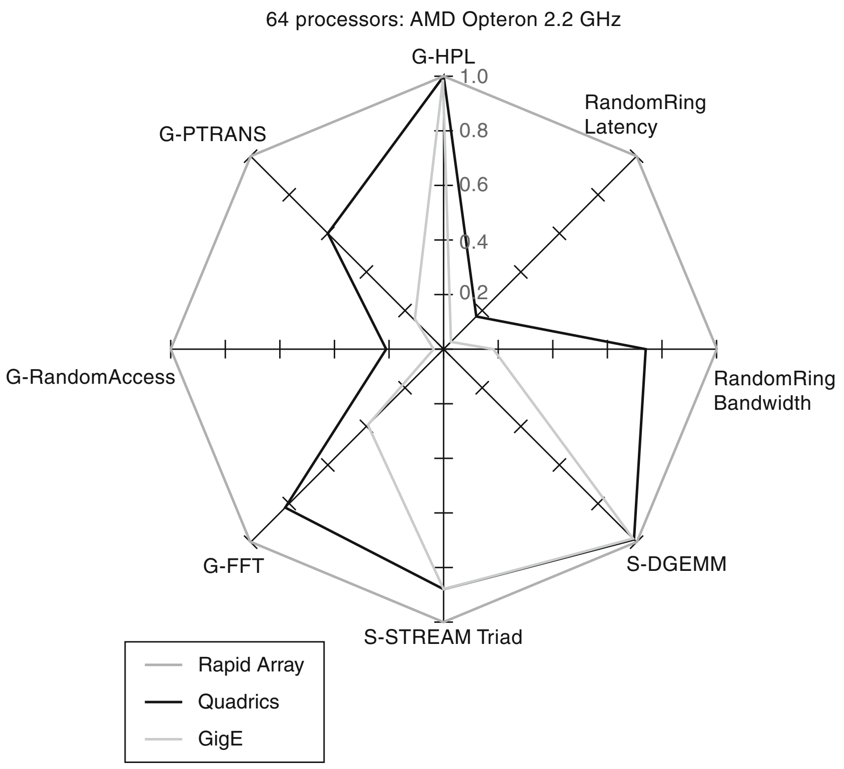
Benchmark Submission Procedures and Results

The reference implementation of the benchmark may be obtained free of charge at the benchmark's web site (http://icl.cs.utk.edu/hpcc/). The reference implementation should be used for the base run: it is written in a portable subset of ANSI C [7] using a hybrid programming model that mixes OpenMP [2, 16] threading with MPI [11–13] messaging. The installation of the software requires creating a script file for Unix's make(1) utility. The distribution archive comes with script files for many common computer architectures. Usually, a few changes to any of these files will produce the script file for a given platform. The HPCC rules allow only standard system compilers and libraries to be used through their supported and documented interface, and the build procedure should be described at submission time. This ensures repeatability of the results and serves as an educational tool for end users who wish to use a similar build process for their applications.

After a successful compilation, the benchmark is ready to run. However, it is recommended that changes be made to the benchmark's input file that describes the sizes of data to use during the run. The sizes should reflect the available memory on the system and the number of processors available for computations.

There must be one baseline run submitted for each computer system entered in the archive. An optimized run for each computer system may also be submitted. The baseline run should use the reference implementation of HPCC, and in a sense it represents the scenario when an application requires use of legacy code – a code that cannot be changed. The optimized run allows the submitter to perform more aggressive optimizations and use system-specific programming techniques (languages, messaging libraries, etc.), but at the same time still includes the verification process enjoyed by the base run.

All of the submitted results are publicly available after they have been confirmed by email. In addition to the various displays of results and exportable raw data, the HPCC website also offers a kiviat chart display to visually compare systems using multiple performance numbers at once. A sample chart that uses actual HPCC results data is shown in Fig. 5.



HPC Challenge Benchmark. Fig. 5 Sample kiviat diagram of results for three different interconnects that connect the same processors

Related Entries

►Benchmarks
►LINPACK Benchmark
►Livermore Loops
►TOP500

Bibliography

1. ANSI/IEEE Standard 754–1985 (1985) Standard for binary floating point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 1985
2. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP. Morgan Kaufmann Publishers, San Francisco, 2001
3. Dongarra JJ (1996) Performance of various computers using standard linear equations software. Computer Science Department. Technical Report, University of Tennessee, Knoxville, TN, April 1996. Up-to-date version available from http://www.netlib.org/benchmark/
4. Dongarra JJ, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. In: Dou Y, Gruber R, Joller JM (2003) Concurrency and computation: practice and experience, vol 15, pp 1–18
5. Dongarra J, Luszczek P (2005) Introduction to the HPC challenge benchmark suite. Technical Report UT-CS-05-544, University of Tennessee, Knoxville
6. Kepner J (2004) HPC productivity: an overarching view. Int J High Perform Comput Appl 18(4):393–397
7. Kernighan BW, Ritchie DM (1978) The C Programming Language. Prentice-Hall, Upper Saddle River, New Jersey
8. Luszczek P, Dongarra J (2006) High performance development for high end computing with Python Language Wrapper (PLW). Int J High Perfoman Comput Appl. Accepted to Special Issue on High Productivity Languages and Models
9. Langou J, Langou J, Luszczek P, Kurzak J, Buttari A, Dongarra J (2006) Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In: Proceedings of SC06, Tampa, Florida, Nomveber 11–17 2006. See http://icl.cs.utk.edu/iter-ref
10. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 8(8):114–117
11. Message Passing Interface Forum (1994) MPI: A Message-Passing Interface Standard. The International Journal of Supercomputer Applications and High Performance Computing 8(3/4):165–414
12. Message Passing Interface Forum (1995) MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: http://www.mpi-forum.org/

13. Message Passing Interface Forum (1997) MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at http://www.mpi-forum.org/docs/mpi-20.ps
14. Meuer HW, Strohmaier E, Dongarra JJ, Simon HD (2006) TOP500 Supercomputer Sites, 28th edn. November 2006. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html)
15. Nadya Travinin and Jeremy Kepner (2007) pMatlab parallel Matlab library. International Journal of High Perfomance Computing Applications 21(3):336–359
16. OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org/

HPF (High Performance Fortran)

High Performance Fortran (HPF) is an extension of Fortran 90 for parallel programming. In HPF programs, parallelism is represented as data parallel operations in a single thread of execution. HPF extensions included statements to specify data distribution, data alignment, and processor topology, which were used for the translation of HPF codes onto an SPMD message-passing form.

Bibliography

1. Kennedy K, Koelbel C, Zima H (2007) The rise and fall of high performance Fortran: an historical object lesson. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III), ACM, New York, pp. 7-1–7-22, doi:10.1145/1238844.1238851, http://doi.acm.org/10.1145/1238844.1238851
2. High Performance Fortran Forum Website. http://hpff.rice.edu

HPS Microarchitecture

Yale N. Patt
The University of Texas at Austin, Austin, TX, USA

Synonyms

The high performance substrate

Definition

The microarchitecture specified by Yale Patt, Wen-mei Hwu, Stephen Melvin, and Michael Shebanow in 1984 for implementing high-performance microprocessors.

It achieves high performance via aggressive branch prediction, speculative execution, wide issue, and out-of-order execution, while retaining the ability to handle precise exceptions via in-order retirement.

Discussion

The High Performance Substrate (HPS) was the name given to the microarchitecture conceived by Professor Yale Patt and his three PhD students, Wen-mei Hwu, Michael Shebanow, and Stephen Melvin, at the University of California, Berkeley in 1984 and first published in Micro 18 in October, 1985 [1, 2]. Its goal was high-performance processing of single-instruction streams by combining aggressive branch prediction, speculative execution, wide issue, dynamic scheduling (out-of-order execution), and retirement of instructions in program order (i.e., in-order).

Out-of-order execution had appeared in previous machines from Control Data [3] and IBM [4], but had pretty much been dismissed as a nonviable mechanism due to its lack of in-order retirement, which prevented the processor from implementing precise exceptions. The checkpoint retirement mechanisms of HPS removed that problem [5, 6]. It should also be noted that solutions to the precise exception problem were also being developed simultaneously and independently by James Smith and Andrew Plezskun [7].

HPS was first targeted for the VAX instruction set architecture and demonstrated that an HPS implementation of the VAX could process instruction streams at a rate of three cycles per instruction (CPI), as compared to the VAX-11/780, which processed at the rate of 10.5 CPI [8].

Instructions are processed as follows: Using an aggressive branch predictor and wide-issue fetch/decode mechanism, multiple instructions are fetched each cycle, decoded into data flow graphs (one per instruction), and merged into a global data flow graph containing all instructions in process. Instructions are scheduled for execution when their flow dependencies (RAW hazards) have been satisfied and executed speculatively and out-of-order with respect to the program order of the program. Results produced by these instructions are stored temporarily in a results buffer (aka re-order buffer) until they can be retired in-order. The essence of the paradigm is that the global data graph consists of nodes corresponding to micro-operations and edges corresponding to linkages between micro-ops that produce operands and micro-ops that source them. The edges of the data flow graph produced as a result of decode correspond to internal linkages within an instruction. Edges created as a result of merging an individual instruction's data flow graph into the global data flow graph correspond to linkages between live-outs of one instruction and live-ins of a subsequent instruction. A Register Alias Table was conceived to maintain correct linkages between live-outs and live-ins. A node, corresponding to a micro-op, is available for execution when all its flow dependencies are resolved.

The HPS research group refers to the paradigm as Restricted Data Flow (RDF) since at no time does the data flow graph for the entire program exist. Rather, the size of the global data flow graph is increased every cycle as a result of new instructions being decoded and merged, and decreased every cycle as a result of old instructions retiring. At every point in time, only those instructions in the active window – the set of instructions that have been fetched but not yet retired – are present in the data flow graph. The set of instructions in the active window are often referred to as "in-flight" instructions. The number of in-flight instructions is orders of magnitude smaller than the size of a data flow graph for the entire program. The result is data flow processing of a program without incurring any of the problems of classical data flow.

Since 1985, the HPS microarchitecture has seen continual development and improvement by many research groups at many universities and industrial labs. The basic paradigm has been adopted for most cutting-edge high-performance microprocessors, starting with Intel on its Pentium Pro microprocessor in the late 1990s [9].

Bibliography

1. Patt YN, Hwu W, Shebanow M (1985) HPS, a new microarchitecture: rationale and introduction. In: Proceedings of the 18th microprogramming workshop, Asilomar, CA
2. Patt YN, Melvin S, Hwu W, Shebanow M (1985) Critical issues regarding HPS, a high performance microarchitecture. In: Proceedings of the 18th microprogramming workshop, Asilomar, CA
3. Thornton JE (1970) Design of a computer – the Control Data 6600. Scott, Foresman and Co. Glenview, IL
4. Anderson DW, Sparacio FJ, Tomasulo RM (1967) The IBM system/360 model 91: machine philosophy and instruction-handling. IBM J Res Development 11(1):8–24
5. Hwu W, Patt Y (1986) HPSm, a high performance restricted data flow architecuture having minimal functionality. In: Proceedings, 13th annual international symposium on computer architecture, Tokyo
6. Hwu W, Patt Y (1987) Checkpoint repair for high performance out-of-order execution machines. IEEE Trans Computers 36(12):1496–1514
7. Smith JE, Pleszkun A (1985) Implementing precise interrupts. In: Proceedings, 12th annual international symposium on computer architecture, Boston, MA
8. Hwu W, Melvin S, Shebanow M, Chen C, Wei J, Patt Y (1986) An HPS implementation of VAX; initial design and analysis. In: Proceedings of the Hawaii international conference on systems sciences, Honolulu, HI
9. Colwell R (2005) The pentium chronicles: the people, passion, and politics behind intel's landmark chips. Wiley-IEEE Computer Society Press, NJ, ISBN: 978-0-471-73617-2

HT

►HyperTransport

HT3.10

►HyperTransport

Hybrid Programming With SIMPLE

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

Definition

Most high performance computing systems are clusters of shared-memory nodes. Hybrid parallel programming handles distributed-memory parallelization across the nodes and shared-memory parallelization within a node.

SIMPLE refers to the joining of the **SMP** and **MPI**-like message passing paradigms [7] and the *simple* programming approach. It provides a methodology of programming cluster of SMP nodes. It advocates a hybrid methodology which maps directly to underlying architectural aspects. SIMPLE combines shared memory programming on shared memory nodes with message passing communication between these nodes. SIMPLE provides (1) a complexity model and set of efficient communication primitives for SMP nodes and clusters; (2) a programming methodology for clusters of SMPs which is both efficient and portable; and (3) high performance algorithms for sorting integers, constraint-satisfied searching, and computing the two-dimensional FFT.

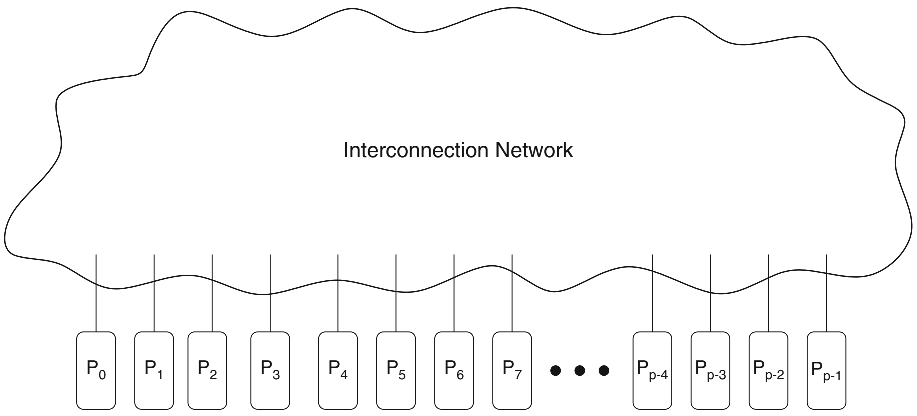
The SIMPLE Computational Model

A simple paradigm is used in SIMPLE for designing efficient and portable parallel algorithms. The architecture consists of a collection of SMP nodes interconnected by a communication network (as shown in Fig. 1) that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. Each SMP node contains several identical processors, each typically with its own on-chip cache and a larger off-chip cache, which have uniform access to a shared memory and other resources such as the network interface.

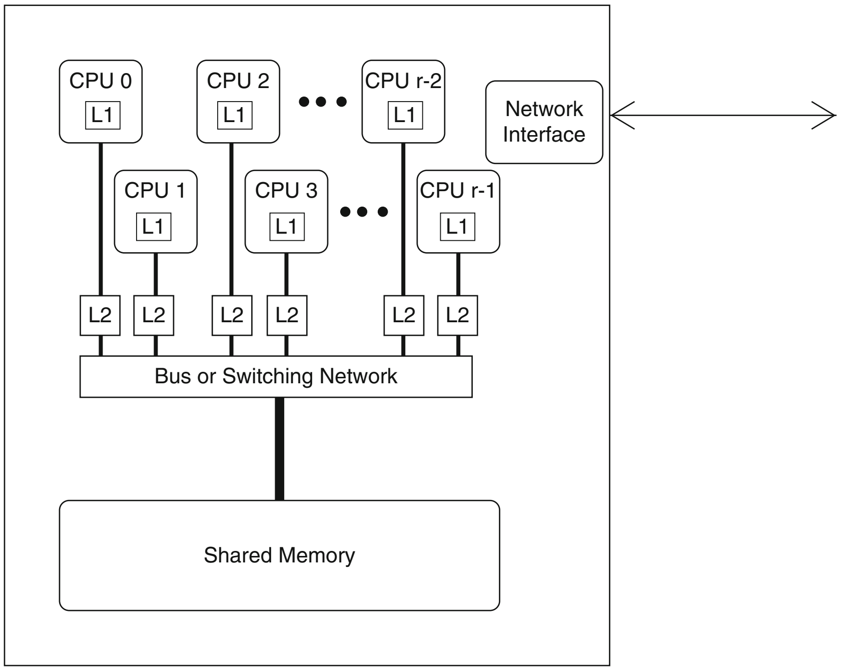
Parameter r is used to represent the number symmetric processors per node (see Fig. 2 for a diagram of a typical node). Notice that each CPU typically has its own on-chip cache (L1) and a larger off-chip level two cache (L2), which can be tightly integrated into the memory system to provide fast memory accesses and cache coherence. The shared memory programming of each SMP node is based on threads which communicate via coordinated accesses to shared memory. SIMPLE provides several primitives that synchronize the threads at a barrier, enable one thread to broadcast data to the other threads, or calculate reductions across the threads. In SIMPLE, only the CPUs from a certain node have access to that node's configuration. In this manner, there is no restriction that all nodes must be identical, and certainly configuration can be constructed from SMP nodes of different sizes. Thus, the number of threads on a specific remote node is not globally available. Because of this, SIMPLE supports only node-oriented communication, meaning that communication is restricted such that, given any source node s and destination node d, with $s \neq d$, only one thread on node s can send (receive) a message to (from) node d at any given time.

Complexity Model

In the SIMPLE complexity model, each SMP is viewed as a two-level hierarchy for which good performance requires both good load distribution and the minimization of secondary memory access. The cluster is viewed as a collection of powerful processors connected by a communication network. Maximizing performance on the cluster requires both efficient load balancing and regular, balanced communication. Hence, our performance model combines two separate but complimentary models.



Hybrid Programming With SIMPLE. Fig. 1 Cluster of processing elements



Hybrid Programming With SIMPLE. Fig. 2 A typical symmetric multiprocessing (SMP) node used in a cluster. L1 is on-chip level-one cache, and L2 is off-chip level-two cache

The SIMPLE model recognizes that efficient algorithm design requires the efficient decomposition of the problem among the available processors, and so, unlike some other models for hierarchical memory, the cost of computation is included in the complexity. The cost model also encourages the exploitations of temporal and spatial locality. Specifically, memory at each SMP is seen as consisting of two levels: cache and main memory. A block of m contiguous words can be read from or written to main memory in $\left(\epsilon + \frac{mr}{\alpha}\right)$ time, where ϵ is the latency of the bus, r is the number of processors competing for access to the bus, and α is the bandwidth. By contrast, the transfer of m noncontiguous words would require $m\left(\epsilon + \frac{r}{\alpha}\right)$ time.

A parallel algorithm is viewed as a sequence of local SMP computations interleaved with communication steps, where computation and communication is allowed to overlap. Assuming no congestion, the transfer of a block consisting of m words between two nodes takes $\left(\tau + \frac{m}{\beta}\right)$ time, where τ is the latency of the network, and β is the bandwidth per node. SIMPLE assumes that the bisection bandwidth is sufficiently high to support block permutation routings among the p nodes at the rate of $\frac{1}{\beta}$. In particular, for any subset of q nodes, a block permutation among the q nodes takes $\left(\tau + \frac{m}{\beta}\right)$ time, where m is the size of the largest block. Using this cost model, the communication time $T_{comm}(n, p)$ of an algorithm can be viewed as a function of the input size n, the number of nodes p, and the parameters τ and β. The overall complexity of algorithm for the cluster $T(n, p)$ is given by the sum of T_{smp} and $T_{comm}(n, p)$.

Communication Primitives

The communication primitives are grouped into three modules: Internode Communication Library (ICL), **SMP Node**, and SIMPLE. ICL communication primitives handle internode communication, **SMP Node** primitives aid shared-memory node algorithms, and SIMPLE primitives combine **SMP Node** with ICL on SMP clusters.

The ICL communication library services internode communication and can use any of the vendor-supplied or freely available thread-safe implementation of MPI. The ICL libraries are based upon a reliable, application-layer `send` and `receive` primitive, as well as a `send-and-receive` primitive which handles the exchanging of messages between sets of nodes where each participating node is the source and destination of one message. The library also provides a barrier operation based upon the send and receive which halts the execution at each node until all nodes check into the barrier, at which time, the nodes may continue execution. In addition, ICL includes collective communication primitives, for example, scan, reduce, broadcast, allreduce, alltoall, alltoallv, gather, and scatter. See [1].

SMP Node

The **SMP Node** Library contains important primitives for an SMP node: barrier, replicate, broadcast, scan, reduce, and allreduce, whereby on a single node, barrier synchronizes the threads, broadcast ensures that each thread has the most recent copy of a shared memory location, scan (reduce) performs a prefix (reduction) operation with a binary associative operator (e.g., addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR) with one datum per thread, and allreduce replicates the result from reduce.

Each of these collective SMP primitives can be implemented using a *fan-out* or *fan-in* tree constructed as follows. A logical k-ary balanced tree is built for an SMP node with r processors, which serves as both a fan-out and fan-in communication pattern. In a k-ary tree, level 0 has one processor, level 1 has k processors, level 2 has k^2 processors, and so forth, with level j containing k^j processors, for $0 \le j \le L-1$), where there are L levels in the tree. If $\log_k r$ is not an integer, then the last level $(L-1)$ will hold less than k^{L-1} processors. Thus, the number of processors r is bounded by

$$\sum_{j=0}^{L-2} k^j < r \le \sum_{j=0}^{L-1} k^j. \tag{1}$$

Solving for the number of levels L, it is easy to see that

$$L = \lceil \log_k (r(k-1)+1) \rceil \tag{2}$$

where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than or equal to x.

An efficient algorithm for replicating a data buffer B such that each processor i, $(0 \le i \le r-1)$, receives a unique copy B_i of B makes use of the fan-out tree, with the source processor situated as the root node of the k-ary tree. During step j of the algorithm, for $(0 \le j \le L-2)$, each of k^j processors writes k unique copies of the data for its k children. The time complexity of this SMP replication algorithm given an m-word buffer is

$$\begin{aligned} T_{smp} &= \sum_{j=0}^{L-2} k\left(\epsilon + \frac{k^j m}{\alpha}\right) \\ &= k(L-1)\epsilon + \frac{m}{\alpha}(r-1) \\ &\le k\left(\log_k(r(k-1)+1)-1\right)\epsilon + \frac{m}{\alpha}(r-1) \\ &\le k\left(\log_k(r+1/k)\right)\epsilon + \frac{m}{\alpha}(r-1) \\ &= O\left((\log r)\epsilon + \tfrac{mr}{\alpha}\right). \end{aligned} \tag{3}$$

The best choice of k, $(2 \le k \le r-1)$, depends on SMP size r and machine parameters ϵ and α, but can be chosen to minimize Eq. 3.

An algorithm which barrier synchronizes r SMP processors can use a fan-in tree followed by a fan-out tree, with a unit message size m, taking twice the replication time, namely, $O((\log r)\epsilon + r/\alpha)$.

For certain SMP algorithms, it may not be necessary to replicate data, but to share a read-only buffer for a given step. A broadcast SMP primitive supplies each processor with the address of the shared buffer by replicating the memory address in $T_{smp} = O((\log r)\epsilon + r/\alpha)$.

A reduce primitive, which performs a reduction with a given binary associate operator, uses a fan-in tree, combining partial sums during each step. For initial data arrays of size m per processor, this takes $O((\log r)\epsilon + mr/\alpha)$. The allreduce primitive performs a reduction followed by replicate so that each processor receives a copy of the result with a cost of $O\left((\log r)\epsilon + \frac{mr}{\alpha}\right)$.

Scans (also called prefix-sums) are defined as follows. Each processor i, $(0 \le i \le r-1)$, initially holds an element a_i, and at the conclusion of this primitive, holds the prefix-sum $b_i = a_0 * a_1 * \ldots * a_i$, where $*$ is any binary associative operator. An SMP algorithm similar to the PRAM algorithm (e.g., [5]) is employed which uses a binary tree for finding the prefix-sums. Given an array of elements A of size $r = 2^d$ where d is a nonnegative integer, the output is array C such that $C(0,i)$ is the ith prefix-sum, for $(0 \le i \le r-1)$.

In fact, arrays A, B, and C in the SMP prefix-sum algorithm (Alg. 1) can be the same array. The analysis is as follows. The first **for** loop takes $\sum_{h=1}^{\log r} 3\left(\epsilon + \frac{r}{2^h}\frac{m}{\alpha}\right)$, and the second **for** loop takes $\sum_{h=0}^{\log r}\left(\frac{3+2}{2}\right)\left(\epsilon + \frac{r}{2^h}\frac{m}{\alpha}\right)$ for a total complexity of $T_{smp} \leq 8\epsilon\log r + 3\frac{mr}{\alpha} = O\left((\log r)\epsilon + \frac{mr}{\alpha}\right)$.

Simple

Finally, the SIMPLE communication library, built on top of ICL and **SMP Node**, includes the primitives for the SIMPLE model: `barrier`, `scan`, `reduce`, `broadcast`, `allreduce`, `alltoall`, `alltoallv`, `gather`, and `scatter`. These hierarchical layers of our communication libraries are pictured in Fig. 3.

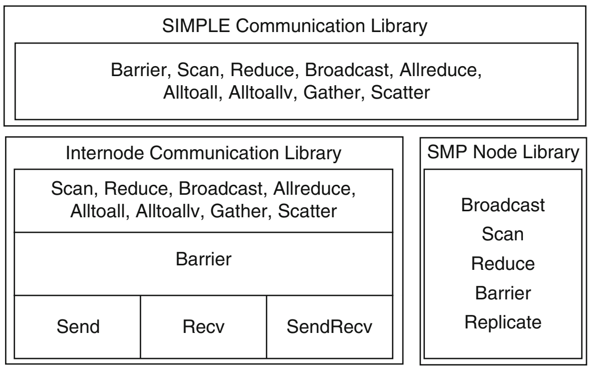
The **SMP Node**, ICL, and SIMPLE libraries are implemented at a high level, completely in user space

Algorithm 1 SMP `scan` algorithm for processor i, $(0 \leq i \leq r-1)$ and binary associative operator $*$

```
set $B(0,i) = A(i)$.

for $h = 1$ to $\log r$ do
  if $1 \leq i \leq r/2^h$ then
    set $B(h,i) = B(h-1, 2i-1) * B(h-1, 2i)$.

for $h = \log r$ downto 0 do
  if $1 \leq i \leq r/2^h$ then
    If $i$ even, Set $C(h,i) = C(h+1, i/2)$;
    If $i = 1$, Set $C(h,1) = B(h,1)$;
    If $i$ odd, Set $C(h,i) = C(h+1, (i-1)/2) * B(h,i)$.
```


SIMPLE Communication Library
Barrier, Scan, Reduce, Broadcast, Allreduce, Alltoall, Alltoallv, Gather, Scatter
Internode Communication Library
Scan, Reduce, Broadcast, Allreduce, Alltoall, Alltoallv, Gather, Scatter
Barrier
Send
Recv
SendRecv
SMP Node Library
Broadcast
Scan
Reduce
Barrier
Replicate

Hybrid Programming With SIMPLE. Fig. 3 Hierarchy of SMP, message passing, and SIMPLE communication libraries

(see Fig. 4). Because no kernel modification is required, these libraries easily port to new platforms.

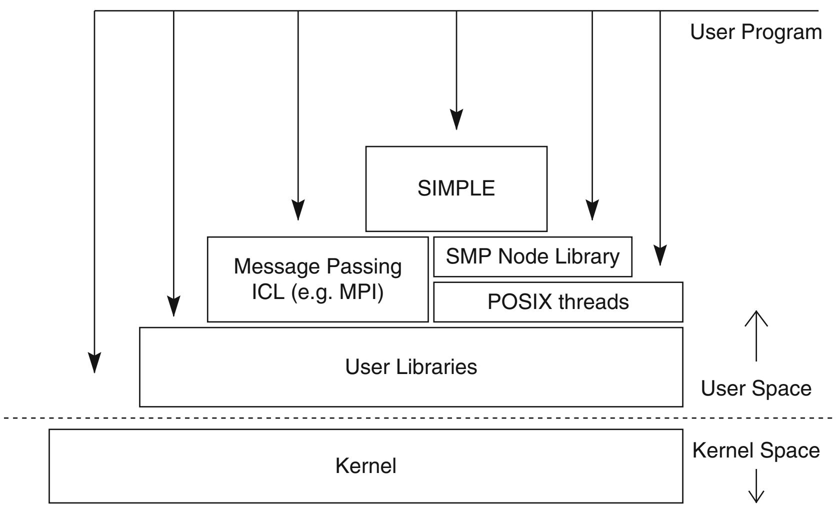
As mentioned previously, the number of threads per node can vary, along with machine size. Thus, each thread has a small set of context information which holds such parameters as the number of threads on the given node, the number of nodes in the machine, the rank of that node in the machine, and the rank of the thread both (1) on the node and (2) across the machine. Table 1 describes these parameters in detail.

Because the design of the communication libraries is modular, it is easy to experiment with different implementations. For example, the ICL module can make use of any of the freely available or vendor-supplied thread-safe implementations of MPI, or a small communication kernel which provides the necessary message passing primitives. Similarly, the **SMP Node** primitives can be replaced by vendor-supplied SMP implementations.

The `Alltoall` Primitive

One of the most important collective communication events is the `alltoall` (or `transpose`) primitive which transmits regular sized blocks of data between each pair of nodes. More formally, given a collection of p nodes each with an m element sending buffer, where p divides m, the `alltoall` operation consists of each node i sending its jth block of $\frac{m}{p}$ data elements to node j, where node j stores the data from i in the ith block of its receiving buffer, for all $(0 \leq i,j \leq p-1)$. An efficient message passing implementation of `alltoall` would be as follows. The notation "var_i" refers to memory location "$var + (\frac{m}{p} * i)$," and *src* and *dst* point to the source and destination arrays, respectively.

To implement this algorithm (Alg. 2), multiple threads $(r \leq p)$ per node are used. The local memory copy trivially can be performed concurrently by one thread while the remaining threads handle the internode communication as follows. The $p-1$ iterations of the loop are partitioned in a straightforward manner to the remaining threads. Each thread has the information necessary to calculate its subset of loop indices, and thus, this loop partitioning step requires no synchronization overheads. The complexity of this primitive is twice $\left(\epsilon + \frac{m}{r}\frac{r}{\alpha}\right)$ for the local memory read



Hybrid Programming With SIMPLE. Fig. 4 User code can access SIMPLE, SMP, message passing, and standard user libraries. Note that SIMPLE operates completely in user space

Hybrid Programming With SIMPLE. Table 1 The local context parameters available to each SIMPLE thread

Parameter	Description
NODES = *p*	Total number of nodes in the cluster
MYNODE	My node rank, from 0 to **NODES** – 1
THREADS = *r*	Total number of threads on my node
MYTHREAD	The rank of my thread on this node, from 0 to **THREADS** – 1
TID	Total number of threads in the cluster
ID	My thread rank, with respect to the cluster, from 0 to **TID** – 1

Algorithm 2 SIMPLE `Alltoall` primitive

```
copy the appropriate $\frac{m}{p}$ elements from $src_{\text{MYNODE}}$ to $dst_{\text{MYNODE}}$.

for $i = 1$ to NODES $- 1$ do
    set $k = \text{MYNODE} \oplus i$;
    send $\frac{m}{p}$ elements from $src_k$ to node $k$, and
        receive $\frac{m}{p}$ elements from node $k$ to $dst_k$.
```

and writes, and $\left(\tau + \frac{m}{\beta}\right)$ for internode communication, for a total cost of $O\left(\tau + \frac{m}{\beta} + \epsilon + \frac{m}{\alpha}\right)$.

Computation Primitives

SIMPLE computation primitives do not communicate data but affect a thread's execution through (1) loop parallelization, (2) restriction, or (3) shared memory management. Basic support for data parallelism, that is, "parallel do" concurrent execution of loops across processors on one or more nodes, is provided.

Data Parallel

The SIMPLE methodology contains several basic "`pardo`" directives for executing loops concurrently on one or more SMP nodes, provided that no dependencies exist in the loop. Typically, this is useful when an independent operation is to be applied to every location in an array, for example, in the element-wise addition of two arrays. `Pardo` implicitly partitions the loop to the threads without the need for coordinating overheads such as synchronization or communication between processors. 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 left-hand side of the assignment being owned by local caches more often than not. However, SIMPLE explicitly provides both block and cyclic partitioning interfaces for the `pardo` directive.

Similar mechanisms exist for parallelizing loops across nodes. The `all_pardo_cyclic` (*i*, *a*, *b*) directive will cyclically assign each iteration of the loop across the entire collection of processors. For example, $i = a$ will be executed on the first processor of the first node, $i = a + 1$ on the second processor of the first node, and so on, with $i = a + r - 1$ on the last processor of the first node. The iteration with $i = a + r$ is executed by the first processor on the second node. After $r \cdot p$ iterations are assigned, the next index will again be assigned to the first processor on the first node. A similar directive called `all_pardo_block`, which accepts the same arguments, assigns the iterations in a block fashion to the processors; thus, the first $\frac{b-a}{rp}$ iterations are assigned to the first processor, the next block of iterations are assigned to the second processor, and so forth. With either of these SIMPLE directives, each processor will execute at most $\left\lceil \frac{n}{rp} \right\rceil$ iterations for a loop of size n.

Control

The second category of SIMPLE computation primitives control which threads can participate in the context by using restrictions.

Table 2 defines each control primitive and gives the largest number of threads able to execute the portion of the algorithm restricted by this statement. For example, if only one thread per node needs to execute a command, it can be preceded with the `on_one_thread` directive. Suppose data has been `gathered` to a single node. Work on this data can be accomplished on that node by preceding the statement with `on_one_node`. The combination of these two primitives restricts execution to exactly one thread, and can be shortcut with the `on_one` directive.

Memory Management

Finally, shared memory allocations are the third category of SIMPLE computation primitives. Two directives are used:

1. `node_malloc` for dynamically allocating a shared structure
2. `node_free` for releasing this memory back to the heap

The `node_malloc` primitive is called by all threads on a given node, and takes as a parameter the number of bytes to be allocated dynamically from the heap. The primitive returns to each thread a valid pointer to the shared memory location. In addition, a thread may allow others to access local data by broadcasting the corresponding memory address. When this shared memory is no longer required, the `node_free` primitive releases it back to the heap.

SIMPLE Algorithmic Design

Programming Model

The user writes an algorithm for an arbitrary cluster size p and SMP size r (where each node can assign possibly different values to r at runtime), using the parameters from Table 1. SIMPLE expects a standard main function (called SIMPLE_main()) that, to the user's view, is immediately up and running on each thread. Thus, the user does not need to make any special calls to initialize the libraries or communication channels. SIMPLE makes available the rank of each thread on its node or across the cluster, and algorithms can use these ranks in a straightforward fashion to break symmetries and

Hybrid Programming With SIMPLE. Table 2 Subset of SIMPLE control primitives

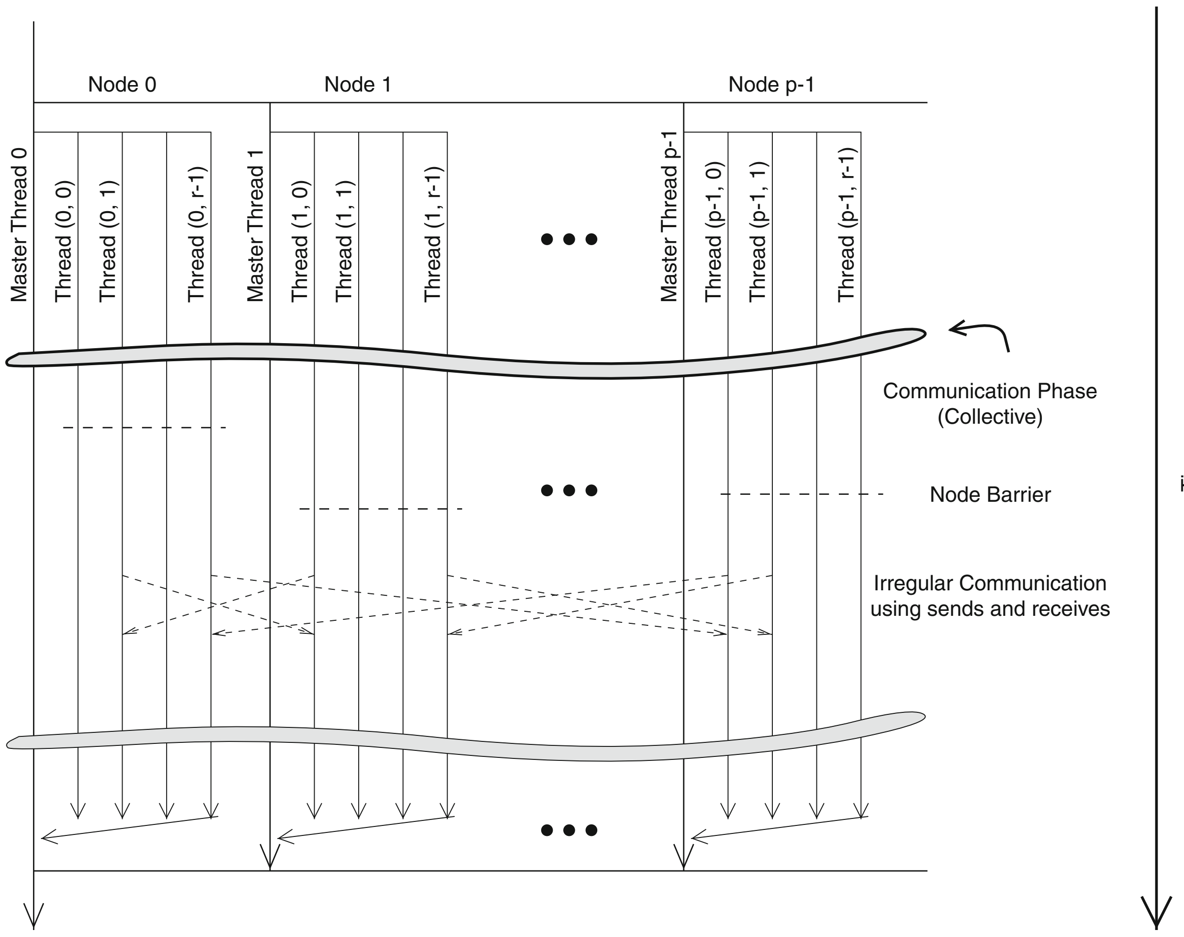
Control Primitives				
Primitive	Definition	Max number of participating threads	MYNODE restriction	MYTHREAD restriction
`on_one_thread`	only one thread per node	p		0
`on_one_node`	all threads on a single node	r	0	
`on_one`	only one thread on a single node	1	0	0
`on_thread(i)`	one thread (i) per node	p		i
`on_node(j)`	all threads on node j	r	j	

partition work. The only argument of SIMPLE_main() is "**THREADED**," a macro pointing to a private data structure which holds local thread information. If the user's main function needs to call subroutines which make use of the SIMPLE library, this information is easily passed via another macro "**TH**" in the calling arguments. After all threads exit from the main function, the SIMPLE code performs a shut down process.

Runtime Support

When a SIMPLE algorithm first begins its execution, the SIMPLE runtime support has already initialized parallel mechanisms such as barriers and established the network-based internode communication channels which remain open for the life of the program. The various libraries described in Sect."The SIMPLE Computational Model" have runtime initializations which take place as follows.

The runtime start-up routines for a SIMPLE algorithm are performed in two steps. First, the ICL initialization expands computation across the nodes in a cluster by launching a master thread on each of the p nodes and establishing communication channels between each pair of nodes. Second, each master thread launches r user threads, where each node is at least an r-way SMP. (A rule of thumb in practice is to use r threads on an $r + 1$-way SMP node, which allows operating system tasks to fully utilize at least one CPU.) It is assumed that the r CPUs concurrently execute the r threads. The thread flow of an example SIMPLE algorithm is shown in Fig. 5. As mentioned previously, our methodology supports



Hybrid Programming With SIMPLE. Fig. 5 Example of a SIMPLE algorithm flow of master and compute-based user threads. Note that the only responsibility of each master thread is only to launch and later join user threads, but never to participate in computation or communication only node-oriented communication, that is, given any source node s and destination node d, with $s \neq d$, only one thread on node s can send (receive) a message to (from) node d during a communication step. Also note that the master thread does not participate in any computation, but sits idle until the completion of the user code, at which time it coordinates the joining of threads and exiting of processes.

The programming model is simply implemented using a portable thread package called POSIX threads (**pthreads**).

A Possible Approach

The latency for message passing is an order of magnitude higher than accessing local memory. Thus, the most costly operation in a SIMPLE algorithm is internode communication, and algorithmic design must attempt to minimize the communication costs between the nodes.

Given an efficient message passing algorithm, an incremental process can be used to design an efficient SIMPLE algorithm. The computational work assigned to each node is mapped into an efficient SMP algorithm. For example, independent operations such as those arising in *functional parallelism* (e.g., independent I/O and computational tasks, or the local memory copy in the SIMPLE `alltoall` primitive presented in the previous section) or *loop parallelism* typically can be *threaded*. For functional parallelism, this means that each thread acts as a functional process for that task, and for loop parallelism, each thread computes its portion of the computation concurrently. Loop transformations may be applied to reduce data dependencies between the threads. Thread synchronization is a costly operation when implemented in software and, when possible, should be avoided.

Example: Radix Sort

Consider the problem of sorting n integers spread evenly across a cluster of p shared-memory r-way SMP nodes, where $n \geq p^2$. Fast integer sorting is crucial for solving problems in many domains, and as such, is used as a kernel in several parallel benchmarks such as NAS (Note that the NAS IS benchmark requires that the integers be ranked and not necessarily placed in sorted order.) [4] and SPLASH [8].

Consider the problem of sorting n integer keys in the range $[0, M-1]$ that are distributed equally in the shared memories of a p-node cluster of r-way SMPs. **Radix Sort** decomposes each key into groups of ρ-bit digits, for a suitably chosen ρ, and sorts the keys by applying a counting sort routine on each of the ρ-bit digits beginning with the digit containing the least significant bit positions [6]. Let $R = 2^\rho \geq p$. Assume (w.l.o.g.) that the number of nodes is a power of two, say $p = 2^k$, and hence $\frac{R}{p}$ is an integer $= 2^{\rho-k} \geq 1$.

SIMPLE Counting Sort Algorithm

Counting Sort algorithm sorts n integers in the range $[0, R-1]$ by using R counters to accumulate the number of keys equal to the value i in bucket B_i, for $0 \leq i \leq R-1$, followed by determining the rank of each key. Once the rank of each key is known, each key can be moved into its correct position using a permutation ($\frac{n}{p}$-relation) routing [2, 3], whereby no node is the source or destination of more than $\frac{n}{p}$ keys. Counting Sort is a **stable** sorting routine, that is, if two keys are identical, their relative order in the final sort remains the same as their initial order.

Algorithm 3 SIMPLE Counting Sort Algorithm

```
Step (1): For each node i, (0 ≤ i ≤ p − 1), count the frequency of its $\frac{n}{p}$ keys; that is, compute $H_{i[k]}$, the number of keys equal to k, for (0 ≤ k ≤ R − 1).

Step (2): Apply the alltoall primitive to the H arrays using the block size $\frac{R}{p}$. Hence, at the end of this step, each node will hold $\frac{R}{p}$ consecutive rows of H.

Step (3): Each node locally computes the prefix-sums of its rows of the array H.

Step (4): Apply the (inverse) alltoall primitive to the R corresponding prefix-sums augmented by the total count for each bin. The block size of the alltoall primitive is $2\frac{R}{p}$.

Step (5): On each node, compute the ranks of the $\frac{n}{p}$ local elements using the arrays generated in Steps (1) and (4).

Step (6): Perform a personalized communication of keys to rank location using an $\frac{n}{p}$-relation algorithm.
```

The pseudocode for our Counting Sort algorithm (Alg. 3) uses six major steps and can be described as follows.

In **Step (1)**, the computation can be divided evenly among the threads. Thus, on each node, each of r threads **(A)** histograms $\frac{1}{r}$ of the input concurrently, and **(B)** merges these r histograms into a single array for node i. For the prefix-sum calculations on each node in **Step (3)**, since the rows are independent, each of r threads can compute the prefix-sum calculations for $\frac{R}{rp}$ rows concurrently. Also, the computation of ranks on each node in **Step (5)** can be handled by r threads, where each thread calculates $\frac{n}{rp}$ ranks of the node's local elements. Communication can also be improved by replacing the message passing `alltoall` primitive used in **Steps (2)** and **(4)** with the appropriate SIMPLE primitive.

The h-relation used in the final step of Counting Sort is a permutation routing since $h = \frac{n}{p}$, and was given in the previous section.

Histogramming in **Step (1A)** costs $O\left(\epsilon + \frac{n}{p}\frac{1}{\alpha}\right)$ to read the input and $O(\epsilon + R\frac{r}{\alpha})$ for each processor to write its histogram into main memory. Merging in **Step (1B)** uses an SMP `reduce` with cost $O\left((\log r)\epsilon + R\frac{r}{\alpha}\right)$. SIMPLE `alltoall` in **Step (2)** and the `inverse alltoall` in **Step (4)** take $O\left(\tau + \frac{R}{\beta} + \epsilon + \frac{R}{\alpha}\right)$ time. Computing local prefix-sums in **Step (3)** costs $O\left(\epsilon + \frac{R}{pr}p\frac{r}{\alpha}\right)$. Ranking each element in **Step (5)** takes $O\left(\epsilon + \frac{n}{p}\frac{1}{\alpha}\right)$ time. Finally, the SIMPLE permutation with $h = \frac{n}{p}$ costs $O\left(\tau + \frac{n}{p}\left(\frac{1}{\beta} + \frac{1}{\alpha} + \frac{\epsilon}{r}\right)\right)$, for $\frac{n}{p} \geq r \cdot \max(p, \log r)$. Thus, the total complexity for Counting Sort, assuming that $n \geq p \cdot \max(R, r \cdot \max(p, \log r))$ is

$$O\left(\tau + \frac{n}{p}\left(\frac{1}{\beta} + \frac{r}{\alpha} + \frac{\epsilon}{r}\right)\right). \qquad (4)$$

SIMPLE Radix Sort Algorithm

The message passing **Radix Sort** algorithm makes several passes of the previous message passing Counting Sort in order to completely sort integer keys. Counting Sort can be used as the intermediate sorting routine because it provides a stable sort. Let the n integer keys fall in the range $[0, M-1]$, and $M = 2^b$. Then $\frac{b}{\rho}$ passes of Counting Sort is needed; each pass works on ρ-bit digits of the input keys, starting from the least significant digit of ρ bits to the most significant digit. Radix Sort easily can be adapted for clusters of SMPs by using the SIMPLE Counting Sort routine. Thus, the total complexity for Radix Sort, assuming that $n \geq p \cdot \max(R, r \cdot \max(p, \log r))$ is

$$O\left(\frac{b}{\rho}\left(\tau + \frac{n}{p}\left(\frac{1}{\beta} + \frac{r}{\alpha} + \frac{\epsilon}{r}\right)\right)\right). \qquad (5)$$

Bibliography

1. Bader DA (1996) On the design and analysis of practical parallel algorithms for combinatorial problems with applications to image processing, PhD thesis, Department of Electrical Engineering, University of Maryland, College Park
2. Bader DA, Helman DR, JáJá J (1995) Practical parallel algorithms for personalized communication and integer sorting. Technical Report CS-TR-3548 and UMIACS-TR-95-101, UMIACS and Electrical Engineering. University of Maryland, College Park
3. Bader DA, Helman DR, JáJá J (1996) Practical parallel algorithms for personalized communication and integer sorting. ACM J Exp Algorithmics 1(3):1–42. www.jea.acm.org/1996/BaderPersonalized/
4. Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Fineberg S, Frederickson P, Lasinski T, Schreiber R, Simon H, Venkatakrishnan V, Weeratunga S (1994) The NAS parallel benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility. NASA Ames Research Center, Moffett Field
5. JáJá J (1992) An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company, New York
6. Knuth DE (1973) The Art of Computer Programming: Sorting and Searching, vol 3. Addison-Wesley Publishing Company, Reading
7. Message Passing Interface Forum. MPI (1995) A message-passing interface standard. Technical report. University of Tennessee, Knoxville. Version 1.1
8. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: Characterization and methodological considerations. In Proceeding of the 22nd Annual Int'l Symposium Computer Architecture, pp 24–36

Hypercube

▸Hypercubes and Meshes
▸Networks, Direct

Hypercubes and Meshes

Thomas M. Stricker
Zürich, Switzerland

Synonyms

Distributed computer; Distributed memory computers; Generalized meshes and tori; Hypercube; Interconnection network; k-ary n-cube; Mesh; Multicomputers; Multiprocessor networks

Definition

In the specific context of computer architecture, a *hypercube* refers to a parallel computer with a common regular interconnect topology that specifies the layout of processing elements and the wiring in between them. The etymology of the term suggests that a hypercube is an unbounded, higher dimensional cube alike geometric structure, that is scaled beyond (greek "hyper") the three dimensions of a platonic cube (greek "kubos"). In its broader meaning, the term is also commonly used to denote a genre of supercomputer-prototypes and supercomputer-products, that were designed and built in the time period of 1980–1995, including the *Cosmic Cube*, the *Intel iPSC* hypercube, the *FPS T-Series,* and a similar machine manufactured by *nCUBE* corporation. A mesh-connected parallel computer is using a regular interconnect topology with an array of multiple processing elements in one, two, or three dimensions. In a generalization from hypercubes and meshes to the broader class of *k-ary n-cubes,* the concept extends to many more distributed-memory multi-computers with highly regular, direct networks.

Discussion

Introduction and Technical Background

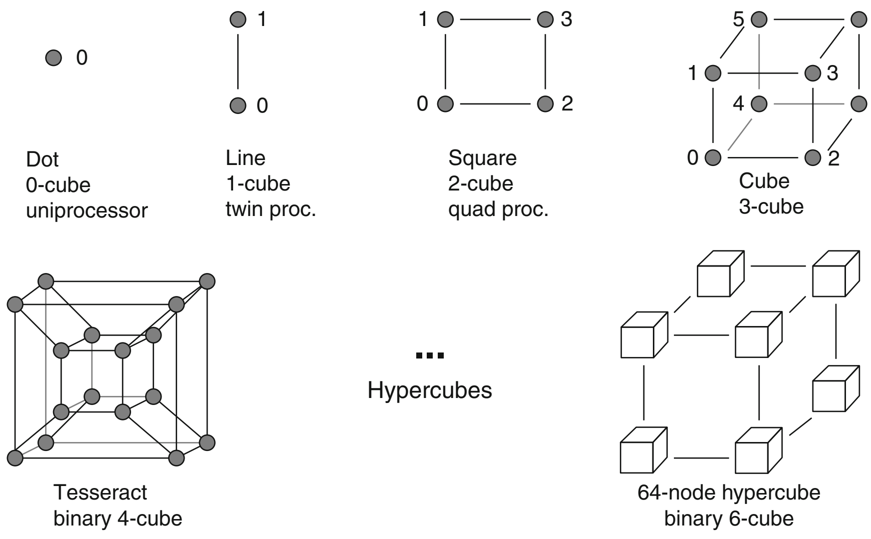
In a *distributed memory multicomputer* with a *direct network,* the processing elements also serve as switching nodes in the network of wires connecting them. For a mathematical abstraction, the arrangement of processors and wires in a parallel machine is commonly expressed as a graph of nodes of processing elements (vertices) and interconnect wires (edges) that are connecting the processors. Since high-speed data transfers normally require an unidirectional point-to-point connection, the resulting interconnection graphs are directed. In addition to the abstract connectivity graph, the optimal layout of processing elements and wires in physical space must be studied. In reference to the branch of mathematics dealing with invariant geometric properties in relation to different spaces, this specification is called the *topology* of a parallel computer system.

In geometry, the classical term *cube* refers to the regular convex hexahedron as one of the five platonic solids in three-dimensional space. Cube alike structures, beneath and beyond the three dimensions of a physical cube can be listed as follows and are drawn in Fig. 1.

- *Zero dimensional*: A dot in geometry or a non-connected uniprocessor in parallel computing.
- *One dimensional*: A geometric line segment or a twin processor system, connected with two unidirectional wires.
- *Two dimensional*: A geometric square or a four processor array, connected with four unidirectional wires in the horizontal and four wires in the vertical direction.
- *Three dimensional*: A geometric cube (hexahedron) or eight processors connected with 24 wires.
- *Four dimensional*: A tesseract in geometry or sixteen processors, arranged as two cubes with the eight corresponding vertices linked by two unidirectional wires each.
- *n-dimensional*: A hypercube as an abstract geometric figure, that becomes increasingly difficult to draw on paper or the arrangement of 2^n processor with each processor having n wires to its n immediate neighbors in all n dimensions.

History

With the evolution of supercomputers from a single vector processor toward a collection of high performance microprocessors during the mid-1980s, a large amount of research work focused on the best possible topology for massively parallel machines. In that time period the majority of parallel systems were built from a large



Hypercubes and Meshes. Fig. 1 Graphical rendering of dot (0 dimensional), line (1D), square (2D), cube (3D), and hypercubes (>3D)

number of identical processing elements, each comprising a standard microprocessor, a dedicated semiconductor random access memory, a network interface and – in some cases – even a dedicated local storage disk in every node. All processing elements also served as switching points in the interconnect fabric. This evolution of technology in highly parallel computers resulted in the class of *distributed memory parallel computers* or *multi-computers.*

The extensive research activity goes back to a special focus on geometry in the theory of parallel algorithms pre-dating the development of the first practical parallel systems. This led to a widespread belief, that the physical topology of parallel computing architectures had to reflect the communication pattern of the most common parallel algorithms known at the time. The obvious mathematical treatment of the question through graph theory resulted in countless theoretical papers that established many new topologies, mappings, emulations, and entire equivalence classes of topologies including meshes, hypercubes, and fat trees. The suggestions for a physical layout to connect the nodes of a parallel multicomputer range from simple one-dimensional arrays of processors to some fully interconnected graphs with direct wires from every node to every other node in the system. Hierarchical rings and busses were also considered. The many results of the effort are compiled into a comprehensive text book [1].

During the golden age of the practical hypercubes (roughly during the years of 1985–1995), it was assumed that the topology of high-dimensional binary hypercubes would result in the best network for the direct mapping of many well-known parallel algorithms. In a binary hypercube each dimension is populated with just two processing nodes as shown in Fig. 1. The wiring of $\boldsymbol{P * \log_2 P}$ unidirectional wires to connect $\boldsymbol{P}$ processors in hypercube topology seemed to be an optimal compromise between the minimal interconnect of just $\boldsymbol{P}$ wires for $\boldsymbol{P}$ processors in a ring and the complete interconnect with $\boldsymbol{P^{*}(P\text{-}1)}$ wires in a fully connected graph (clique).

In a hypercube or a mesh layout for a parallel system, the processors can only communicate data directly to a subset of neighboring processors and therefore require a mechanism for indirect communication through data forwarding at the intermediate processing nodes. Indirect communication can take place along pre-configured virtual channels or through the use of forwarded messages. Accordingly the practical hypercube and mesh systems are primarily designed for *message passing* as their programming model. However, most stream-based programming models can be supported efficiently with channel virtualization. Shared memory programming models are often supported on top of a low level message passing software layer or by hardware mechanisms like directory-based caches that rely on messages for communication.

Binary hypercube parallel computers like the *Cosmic Cube* [7], the *iPSC* [8, 9] the *FPS T Series* [10] or *NCube* [11] were succeeded around 1992 by a series of distributed memory multi-computers with a two- or three-dimensional mesh/torus topology, such as the *Ametek, Warp/iWarp*, *MassPar*, *J-Machine*, the Intel *Paragon*, Intel *ASCI Red*, *Cray T3D*, *T3E*, *Red Storm*, and *XT4/Seastar*. Scaling to larger machines, the early binary hypercubes were at a severe disadvantage, because they required a high number of dimensions resulting in an unbounded number of ports for the switches at every node (due to the *unbounded in/out degree* of the hypercube topology). Something all the hypercubes and the more advanced mesh/torus architectures have in common is that they rely heavily on the message routing technologies developed for the early hypercubes. They are classified as *generalized hypercubes* (*k-ary n-cubes)* in the literature and therefore included in this discussion. By the turn of the century the concept of hypercubes was definitely superseded by the *network of workstations* (NOWs) and by the *cluster of personal computers* (COPs) that used a more or less irregular interconnect fabric built with dedicated network switches that are readily available from the global working technology of the Internet.

Significance of Parallel Algorithms and Their Communication Patterns

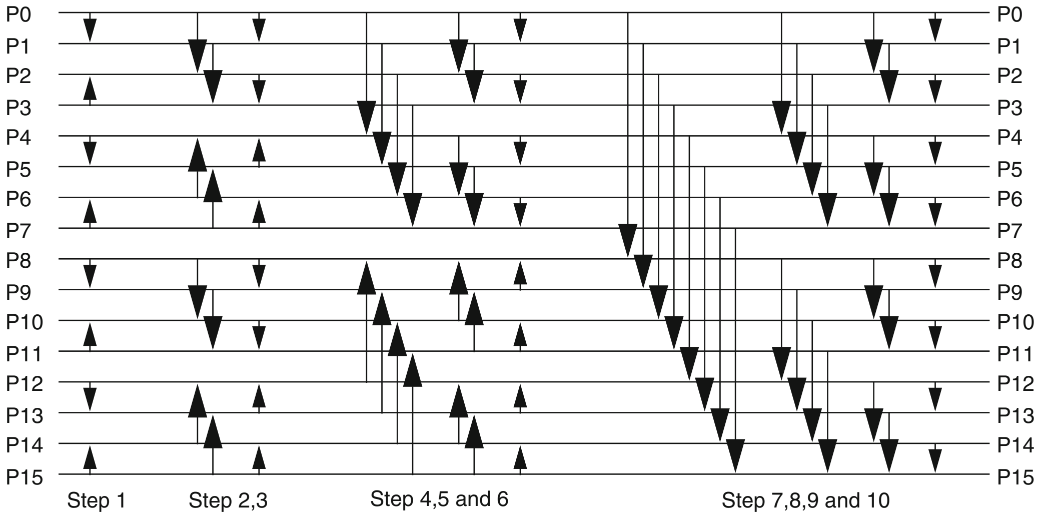
A valid statement about the optimal interconnect layout in a parallel system can only be made under the assumption that there is a best suitable mapping between the $\boldsymbol{M}$ data elements of a large simulation problem and the $\boldsymbol{P}$ processing elements in a parallel computer. For simple, linear mappings (e.g., $\boldsymbol{M/P}$ elements on every processor in block or block/cyclic distribution) the communication patterns of operations between data elements translate into roughly the same communication pattern of messages between the processors in the parallel machine. For simplicity, we also assume $\boldsymbol{M} >> \boldsymbol{P}$ and that $\boldsymbol{M}$ and $\boldsymbol{P}$ are powers of two. In an algorithm with a hypercube communication pattern all communication activities take place between nodes that are direct neighbors in the layout of a hypercube. For the common number scheme of nodes this is between node pairs that differ by exactly one bit in their number. Every communication in a hypercube pattern therefore crosses just one wire in the hypercube machine.

Algorithms with Pure Hypercube Communication Patterns

Several well-known and important algorithms do require a sequence of inter-node communication steps that follow a pure hypercube communication pattern. In most of these algorithms, the communication happens in steps along increasing or decreasing dimensions, e.g., the first communication step between neighbors whose number differs in the least significant bit followed by a next step until a last step with communication to neighbors differing in the most significant bit. Most parallel computers cannot use more than one or two links for communication at the same time due to bandwidth constraints within the processor node. Therefore, a true pipeline with more than one or two overlapping hypercube communication steps is rarely encountered in practical programs.

One of the earliest and best-known algorithms with a pure hypercube communication pattern is the *bitonic sorting algorithm* that goes back to a generalized merge sort algorithm for parallel computers published in 1968 [3]. The communication graph in Fig. 2 shows all the hypercube communication steps that are required to merge locally sorted subsequences into a globally sorted sequence on 16 processors.

The most significant and most popular algorithm communicating between nodes in a pure hypercube pattern is the classic calculation of a *Fast Fourier Transform (FFT)* over an one-dimensional array, with its data distributed among P processors in a simple block partitioning. During the Fourier transformation of a large distributed array a certain number of butterfly calculations can be carried out in local memory up to the point when the distance between the elements in the butterfly becomes large enough that processor boundaries are crossed and some inter-processor communication over



Hypercubes and Meshes. Fig. 2 Batcher bitonic parallel sorting algorithm. A series of hypercube communication steps will merge 16 locally sorted subsequences into one globally sorted array. The *arrows* indicate the direction of the merge, i.e., one processor gets the upper and the other processor gets the lower half of the merged sequence

the hypercube links is required. The distance of butterfly operations is always a power of two and therefore it maps directly to a hypercube.

In the more advanced and multi-dimensional FFT library routines the communication operations between elements at the larger distances might eventually be reduced by dynamically rearranging the distribution of array elements during the calculation. Such a redistribution is typically performed by global data transpose operation that requires an *all-to-all personalized communication* (AAPC), i.e., individual and distinct data blocks are exchanged among all ***P*** processors. In general, fully efficient and scalable AAPC does require a scalable bandwidth interconnect, like the binary hypercube. In practice a talented programmer can optimize such collective communication primitives to run at maximal link speed for sufficiently large mesh- and torus-connected multi-computers (i.e., Cray T3D/T3E tori with up to 512 nodes) [5].

Algorithms with Next Neighbor Communication Pattern

Most simulations in computational physics, chemistry, and biology are calculating some physical interactions (forces) between a large number of model nodes (i.e., particles or grid points) in three-dimensional physical space. Therefore, they only require a limited amount of communication between the model nodes in at most three dimensions because the longer range forces can be summarized or omitted. Consequently the communication in the parallel code of an equation solver, based on relaxation techniques is also limited to nearby processor in two or three dimensions.

Many calculations in natural science do not require communication in dimensions that are higher than three and there is no need to arrange processing elements of a parallel computer in hypercube topologies. Users in the field of scientific computing have recently introduced some optimizations that require increased connectivity. Modern physical simulations are carried out on *irregular meshes*, that are modeling the physical space more accurately by adapting the density of meshing points to some physical key parameter like the density of the material, the electric field, or a current flow. Irregular meshes grid points and sparse matrices are more difficult to partition into regular parallel machines. Advanced simulation methods also involve the summation of certain particle interactions and forces in Fourier and Laplace spaces to achieve a better accuracy in fewer simulation steps (e.g., the particle-mesh Ewald summation in molecular dynamics). Both improvements result in denser communication patterns that require higher connectivity. In the end it remains fairly hard to judge for the systems architect, whether these codes alone can justify the more complex wiring of a high-dimensional hypercube within the range of the machine sizes, that are actually purchased by the customers.

Meshes as a Generalization of Binary Hypercubes into k-ary n-Cubes

In the original form of a binary hypercube, the wires connect only two processing nodes in every dimension. In linear arrays and rings a larger number of processors can be connected in every dimension. The combination of multiple dimensions and multiple processors connected along a line in one dimension leads to a natural generalization of the hypercube concept that includes the popular 1D, 2D, and 3D arrays of processing elements. In computer architecture, the term "mesh" is used for a two- or higher-dimensional processor array. A linear array of processing nodes can be wired into a ring by a wrap-around link. A two or higher dimensional mesh, that is equipped with wrap-around links is called a torus.

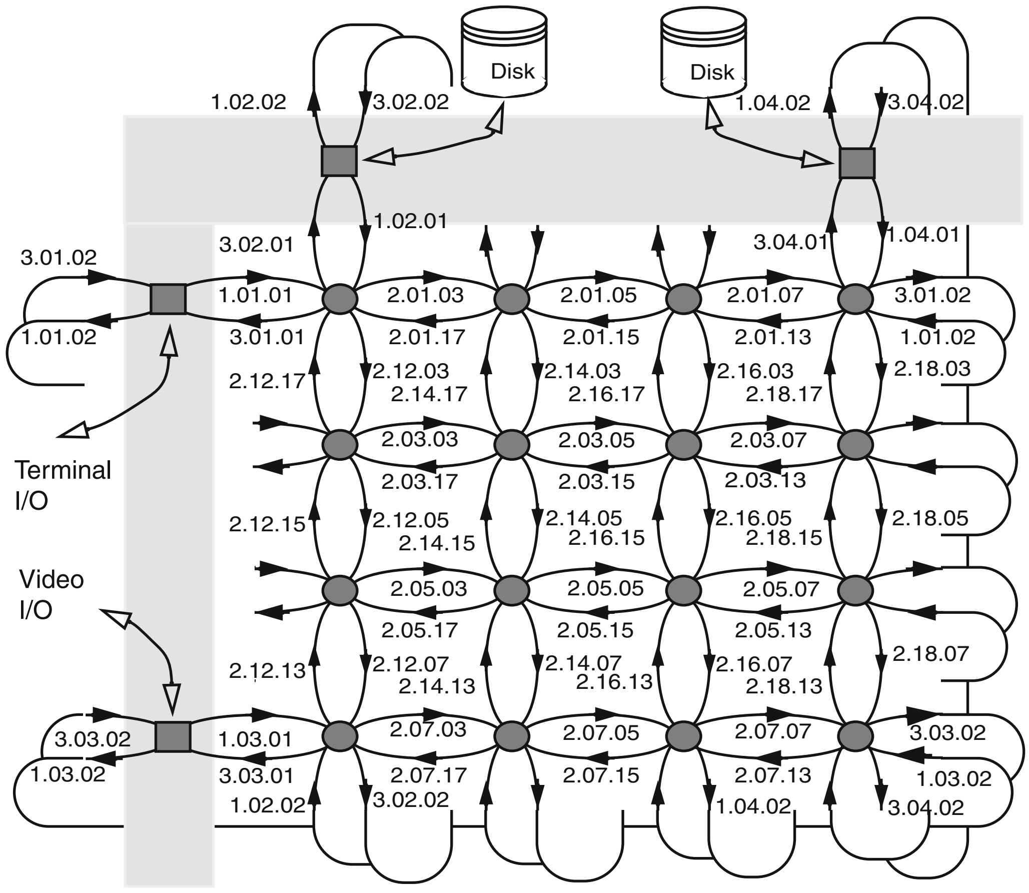
The generalized form of hypercubes and meshes is characterized by two parameters, the maximal number of elements found along one dimension, ***k***, and the total number of dimensions used, ***n***. The binary hypercubes described in the introduction are classified as a ***2-ary n-cubes***. A ***k*** × ***k***, two-dimensional torus can be classified as a ***k-ary 2-cube.***

With the popularity of the hypercube topology in the beginning of distributed memory computing, many key technologies for message forwarding and routing were developed in the early hypercube machines. These ideas extend quite easily to the generalized k-ary n-cubes and were incorporated quickly into meshes and tori. In hypercubes, meshes and tori alike, the messages traveling between arbitrary processing nodes have to traverse multiple links to reach their destination. They have to be forwarded in a number of intermediate nodes, raising the important issues of *message forwarding policies* and *deadlock avoidance*.

In a general setting it would be very hard to establish a global guarantee that every message can travel right through without any delay along a given route. Therefore, some low level flow control mechanisms are used to stop and hold the messages, if another message is already occupying a channel along the path. The most common strategies for forwarding the messages in a hypercube network are *wormhole* or *cut-through routing*. If not done carefully, blocking and delaying messages along the path can lead to tricky deadlock situations, in particular when the messages are indefinitely blocking each other within an interconnect fabric. During the era of the first hypercube multicomputers two key technologies for deadlock avoidance were described [7] and applied to the construction of prototypes and products:

- *Dimension order routing:* In binary hypercubes routes between two nodes must be chosen in a way that the distance in the highest dimension is traveled first, before any distance of a lesser dimension is traveled. Therefore messages can only be stopped due to a busy channel in a dimension lower than the one that they are traveling. The channels in the lowest dimension always lead to a final destination. The messages occupying these channels can make progress immediately and free the resource for other messages blocked at the same and higher dimensions. This is sufficient to prevent a routing deadlock.
- *Dateline-based torus routing:* With the wrap-around link of rings and tori, the case of messages blocking each other in a single dimension around the back-loops has to be addressed. This is done by replicating the physical wires into some parallel virtual channels forming a higher and a lower virtual ring. A dateline is established at a particular position in the ring. The dateline can only be crossed if the message switches from the higher to the lower ring at that position. With this simple trick, all connections along the ring remain possible, but the messages in the ring can no longer block each other in a circular deadlock.

Both *deadlock avoiding* techniques are normally combined to permit deadlock-free routing in the generalized k-ary n-cube topologies. Extending the well-known dimension-order and dateline routing strategies to slightly irregular network topologies is an additional challenge. The two abstract rules for legal message routes can be translated into a simple channel numbering scheme providing a total order of all channels within an interconnect, including all irregular nodes. A simple



Hypercubes and Meshes. Fig. 3 Generalized hypercubes. Channel numbering based on the original hyperube routing rules for the validation of a deadlock-free router in a slightly irregular two-dimensional torus with several IO nodes (an iWarp system)

rule that channels must be followed in a strictly increasing/decreasing order to form a legal route is sufficient to prevent deadlock. With the technique, illustrated in Fig. 3, it becomes possible to code and validate the router code for any k-ary 2-cube configuration that was offered as part of the Intel/CMU *iWarp* product line, including the service nodes that are arbitrarily inserted into the otherwise regular torus network [6].

Mapping Hypercube Algorithms into Meshes and Tori

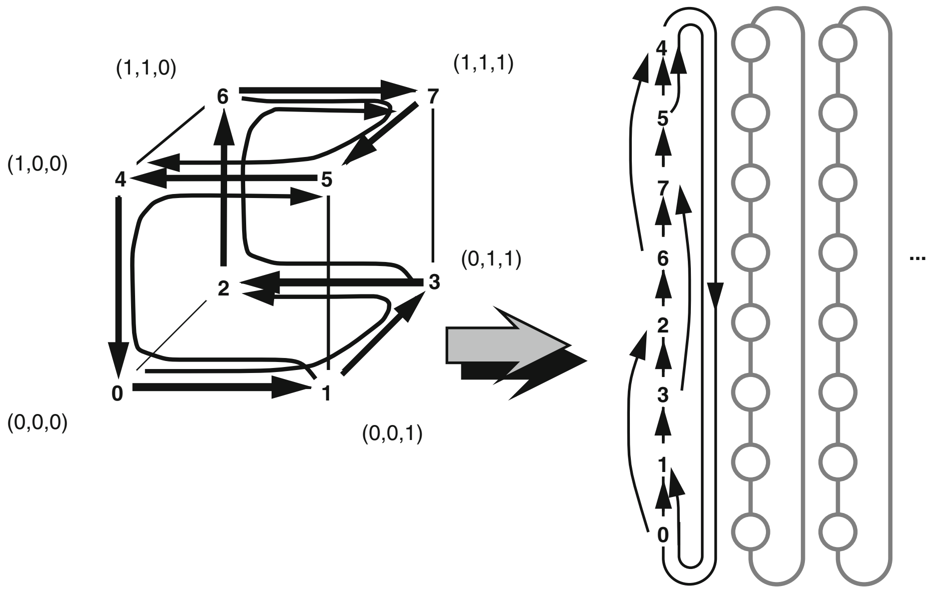
By design the binary hypercube topology provides a fully scalable *worst case bisection bandwidth*. Regardless of how the machine is scaled or partitioned, the algorithm can count on the same amount of bandwidth per processor for the communication between the two halves. In meshes and tori the total bisection bandwidth *does not scale* up linearly with the number of nodes. The global operations, that are usually communication bound, become increasingly costly in larger machines. The concept of the scalable bisection bandwidth was thought to be a key advantage of the hypercube designs that is not present in meshes for a long time.

In the theory of parallel algorithms the asymptotic lower bounds on communication in binary hypercubes usually differ from the bounds holding for communication in an one-, two- or three-dimensional mesh/torus. The lower bound on the number of steps (i.e., the time complexity) is determined by the amount of data that has to be moved between the processing elements and by the number of wires that are available for this task (i.e., the bisection bandwidth). The upper bound is usually determined by the sophistication of the algorithm. One of the most studied algorithms is *parallel sorting* in a 1–1 comparison based model [2, 3]. Similar considerations can be made for *matrix transposes* or *redistributions of block/cyclic arrays* distributed across the nodes of a parallel machine.

In the practice of parallel computing, a mesh-connected machine must be fairly large until an algorithmic limitation due to bisection bandwidth is encountered and a slowdown over a properly hypercube connected machine can be observed [5]. As an example, the bandwidth of an 8-node ring, a 64-node two - dimensional torus or a 512-node three-dimensional torus is sufficient to permit arbitrary array transpose operations without any slowdown over a hypercube of the same size. The comparison assumes that the processing nodes of both machines can transfer data from and to the network at about the speed of a bidirectional link – as this is the case for most parallel machines. Therefore, a few simple mapping techniques are used in practice to implement higher dimensional binary hypercubes on k-ary meshes or tori. A three-dimensional binary hypercube can be mapped into an eight node ring with a small performance penalty using a *Gray code mapping* as illustrated in Fig. 4.

In this simple mapping some links have to be shared or multiplexed by a factor of two and next-neighbor communication is extended to a distance of up to three hops. The mapping of larger virtual hypercube networks comes at the cost of an increasing dilation (increase in link distance) and congestion (a multiplexing factor of each link) for higher dimensions. The congestion factor can be derived using the calculations of bisection bandwidth for both topologies. A binary 6-cube can be mapped into a 2D torus and the 9-cube into a 3D torus by extending the linear scheme to multiple dimensions.

In their VLSI implementation, a two- or three-dimensional mesh/torus-machine is much easier to lay out in an integrated circuit than an eight- or ten-dimensional hypercube with its complicated wiring structure. Therefore its simple next-neighbor links can be built with faster wires and wider channels for multiple wires. The resulting advantage for the implementation is known to offset the congestion and dilation factors in practice. It is therefore highly interesting to study these technology trade-offs in practical machines. For 1–1 sorting, the algorithmic complexity and the measured execution times on a virtual hypercube mapped to mesh match the known complexities and



Hypercubes and Meshes. Fig. 4 Gray code mapping of a binary three-cube into an eight node ring. Virtual network channels were used to implement virtual hypercube network in machines that are actually wired as meshes or tori running times of the algorithm for direct execution on a mesh fairly closely [2, 4].

Limitations of the Hypercube Machines

The ideal topology of a binary hypercube comes at the high cost of significant disadvantages and restrictions regarding their practical implementation in the VLSI circuits of a parallel computer. Therefore, a number of alternative options to build multicomputers were developed beyond hypercubes, meshes, and their superclass of the *k-ary n-cubes*.

Among the alternatives studied are: *hierarchical rings*, that differ slightly from multi-dimensional tori, *cube-connected cycles*, that successfully combine the scalable bisection bandwidth of hypercubes with the bounded in/out degrees of lower - dimensional meshes and finally *fat trees* that are provably optimal in their layout of wiring for VLSI. Those interconnect structures have in common, that they are still fairly regular and can leverage of the same practical technical solutions for message forwarding, routing and link level flow control, the way the traditional hypercubes do. None of these regular interconnects requires a spanning routing protocol or TCP/IP connections to function properly.

Towards the end of the golden age of hypercubes, the regular, direct networks were facing an increasing competition by new types of networks built either from physical links with different strength (i.e., different link bandwidths implemented by a multiplication of wires or in a different technology). In 1991, the Thinking Machines Corporation announced the Connection Machine CM5 using a fat tree topology and this announcement contributed to end of the hypercube age. After the year 2000, the physical network topologies became less and less important in supercomputer architecture. The detailed structure of the networks became hidden and de-emphasized due to many higher-level abstractions of parallelizing compilers and message passing libraries available to the programmers.

The availability of high performance commodity networking switches for use in the highly irregular topology of the global Internet accelerated this trend. In most newer designs, a fairly large multistage network of switches (e.g., a Clos network) has replaced the older direct networks of hypercubes and meshes. At this time only a small fraction of PC Clusters is still including the luxury of a low latency, high performance interconnect. Despite all trends to abstraction and de-emphasis of network structure, the key figures of *message forwarding latency* and the *bisection bandwidth* remain the most important measure of scalability in a parallel computer architecture.

Hypercube Machine Prototypes and Products

One of the earliest practical computer systems using a hypercube topology is the *Cosmic Cube* prototype designed by the Group of C. Seitz at the California Institute of Technology [7]. The system was developed in the first half of the 1980s using the VLSI integration technologies newly available at the time. The first version of the system comprised 64 nodes with an Intel 8086/8087 microprocessor/floating point coprocessors combination running at 5 MHz and using 128 kB of memory in each node. The nodes were connected with point-to-point links running at 2 Mbit/s nominal speed. The original system of 64 nodes was allegedly planned as a 4 × 4 × 4 three-dimensional torus with bidirectional links – but this particular topology is fully equivalent to a binary six-dimensional hypercube under a gray code mapping and became therefore known as a first *hypercube multicomputer*, rather than as a first three-dimensional torus. Subsequently a small number of commercial machines were built in a collaboration with a division of *AMETEK Corporation* [15].

The Caltech prototypes lead to the development of *iPSC* at the Intel Supercomputer Systems Division, a commercial product series based on hypercube topology [8, 9]. An example of the typical technical specifications is the iPSC/2 system as released in 1987: 64 nodes, Intel 80386/387 processor/floating point coprocessor running at 16 MHz, 4–16 MB of main memory in each node, the nodes connected with links running 22 Mbit/s. iPSC systems were programmed using the NX message passing library, similar in the functionality, but pre-dating the popular portable message passing systems like *PVM* or *MPI*. The development of the product line was strongly supported by (D)ARPA, the Advanced Research Project Agency of the US Dept. of Defense.

During roughly the same time period a hypercube machine was also offered by NCube Corporation, a super-computing vendor that was fully dedicated to the development of hypercube systems at the time. The

NCube 6,400 model, available to Nasa AMES in 1991 was a 64-node system with a 20 MHz full custom 64 bit CPU/FPU with up to 64 MB main memory in every node and 20 Mbit/s interconnects. The largest configuration commercially offered was a binary 10-cube with 1,024 processing nodes [11, 12].

The *FPS (Floating Point Systems)* T Series Parallel Vector Supercomputer was also made of multiple processing nodes interconnected as a binary n-cube. Each node is implemented on a single printed circuit board, contains 1 MB of data storage memory, and has a peak vector computational speed of 12 MFLOPS. Eight vector nodes are grouped around a system support node and a disk storage system to form a module. The module is the basic building block for larger system configurations in hypercube topology. The T series was named after the tesseract, a geometric cube in four-dimensional space [10].

The *Thinking Machine* CM2 also included a hypercube communication facility for data exchanges between the nodes as well as a mesh (called NEWS grid) [13]. As an SIMD machine with a large number of bit slice processors, its architecture differed significantly from the classical hypercube multicomputer. In its successor, the *Thinking Machine* CM5, the direct networks were given up in favor of a fat tree using a variable number of (typically N-1) of additional switches to form a network in fat tree topology. A similar trend was followed in the machines of *Meiko/Quadrics*. The delay of the T9000 transputer chips with processing and communication capabilities on a single chip forced the designers of the *CS2* system to build their communication system with a multi-stage fabric of switches instead of a hypercube interconnect. The multicomputer line by *IBM*, the IBM SP1 and its follow-on products also used a multistage switch fabric. Therefore, these architectures do no longer qualify as generalized hypercubes with direct networks.

It is worth mentioning that the first *Beowulf* clusters of commodity PCs were equipped with three to five network interfaces that could be wired directly in hypercube topology. Routing of messages between the links was achieved by system software in the ethernet drivers of the LINUX operating system. The communication benchmark presented in the first beowulf paper were measured on an 8-node system, wired as 2ˆ3 cube [14]. The author also remembers encountering a Beowulf system on display at the Supercomputing trade show wired as a 32-node binary 5-cube using multiple 100BaseT network interface cards and simple CAT5 crossover cables.

But as mentioned before, the designs of processors with their own communication hardware on board and direct networks were quickly replaced by commodity switches manufactured by a large number of network equipment vendors in the Internet world. So the topology of the parallel system suddenly became a secondary issue. A large variety of different network configurations rapidly succeeded the regular Beowulf systems in the growing market for clusters of commodity PCs.

Research Conferences on Hypercubes

The popularity of hypercube architectures in the second half of the 1980s led to several instances of a computer architecture conference dedicated solely to distributed memory multicomputers with hypercube and mesh topologies. The locations and dates of the conferences are as remembered or collected from citations in subsequent papers:

- First Hypercube Conference in Knoxville, August 26–27, 1985, with proceedings published by SIAM after the conference.
- Second Conference on Hypercube Multiprocessors, Knoxville, TN, September 29–October 1, 1986, with proceedings published by SIAM after the conference.
- Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, January 19–20, 1988 with proceedings published by the ACM, New York.
- Fourth Conference on Hypercube Concurrent Computers and Applications, Monterrey, CA, March 6–9, 1989, proceedings publisher unknown.

and later giving in to the trend that the architecture as distributed memory computer is more important than the hypercube layout:

- Fifth Distributed Memory Computing Conference, Charleston, SC, April 8–12, 1990 with proceedings published by IEEE Computer Society Press.
- Sixth Distributed Memory Computing Conference, Portland, OR, April 29–May 1, 1991 with proceedings published by IEEE Computer Society Press.

- First European Distributed Memory Computing, EDMCC, held in Rennes, France.
- Second European Distributed Memory Computing, EDMCC2, Munich, FRG, April 22–24, 1991 with Proceedings by Springer, Lecture Notes in Computer Science.

After these rather successful meetings of computer architects and application programmers, the conference series on hypercubes and distributed memory parallel systems lost its momentum and in 1992, the specialized venues were unfortunately discontinued.

Related Entries

►Beowulf clusters
►Bitonic Sort
►Clusters
►Connection Machine
►Cray T3E
►Cray Vector Computers
►Cray XT4 and Seastar 3-D Torus Interconnect
►Distributed-Memory Multiprocessor
►Fast Fourier Transform (FFT)
►IBM RS/6000 SP
►Interconnection Networks
►MasPar
►MPI (Message Passing Interface)
►MPP
►nCUBE
►Networks, Direct
►Routing (including Deadlock Avoidance)
►Sorting
►Warp and IWarp

Bibliographic Notes and Further Reading

A most detailed description of the algorithms optimally suited for hypercubes, the classes of hypercube equivalent networks, and the emulation of different topologies with respect to algorithmic models is given in an 835 page textbook by F.T. Leighton that appeared in 1991 [1]. A detailed description of the network topologies and the technical data of all practical hypercube prototypes built and machines commercially sold between 1985 and 1995 can be found through Google Scholar in the numerous papers written by the architects working for the computer manufacturers or by researches at the different US national labs evaluating and benchmarking these distributed memory multicomputers. The most interesting study dedicated entirely to binary hypercubes appeared in 1991 in Parallel Computing [12]. The visit to the trade show of "Supercomputing" during the "hypercube" years was a memorable experience, because countless new distributed memory system vendors surprised the audience with a new parallel computer architecture every year. Most of them contained some important innovation and can still be admired in the permanent collections of the computer museums in Boston (MA), Mountain View (CA), and Paderborn (Germany).

Bibliography

1. Leighton FT (1991) Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers, San Francisco 837p, ISBN:1-55860-117-1
2. Stricker T (1992) Supporting the hypercube programming model on meshes (a fast parallel sorter for iwarp). In: Proceedings of the symposium for parallel algorithms and architectures, pp 148–157, San Diego, June 1992
3. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the american federation of information processing societies spring joint computer conference, vol 32. AFIPS Press, Montvale, pp 307–314
4. Nassimi D, Sahni S (1979) Bitonic sort on a mesh-connected parallel computer. In: IEEE TransComput 28(1):2–7
5. Hinrichs S, Kosak C, O'Hallaron D, Stricker T, Take R (1994) Optimal all-to-all personalized communication in meshes and tori. In: Proceedings of the symposium of parallel algorithms and architectures, ACM SPAA'94, Cape May, Jan 1994
6. Stricker T (1991) Message routing in irregular meshes and tori. In: Proceedings of the 6th IEEE distributed memory computing conference, DMCC, Portland, May 1991
7. Seitz CL (1985) The cosmic cube. Commun ACM 28(1):22–33
8. Close P (1988) The iPSC/2 node architecture. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 43–50
9. Nugent SF (1988) The iPSC/2 direct-connect communications technology. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 51–60
10. Hawkinson S (1988) The FPS T series supercomputer, system modelling and optimization. In: Lecture notes in control and information sciences, vol 113/1988. Springer, Berlin/Heidelberg, pp 766–785
11. The nCUBE Handbook, Beaverton OR, 1986 and the nCUBE 6400 processor, user manual, Beaverton
12. Dunigan TH (1991) Performance of the Intel iPSC 1/2/860 and Ncube 3200/6400 hypercubes, parallel computing 17. North Holland, Elsevier, pp 1285–1302

13. Tucker LW, Robertson GG (1988) Architecture and applications of the connection machines. IEEE Comput 21(8):26–38
14. Becker J, Sterling D, Savarese T, Dorband JE, Ranawake UA, Packer CV (1995) Beowulf: a parallel workstation for scientific computation. In: Proceedings of ICPP workshop on challenges for parallel processing, CRC Press, Oconomowc, August 1995
15. Seitz CL, Athas W, Flaig C, Martin A, Seieovic J, Steele CS, Su WK (1988) The architecture and programming of the ametek series 2010 multicomputer. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 31–36

Hypergraph Partitioning

Ümit V. Çatalyürek[1], Bora Uçar[2], Cevdet Aykanat[3]
[1]The Ohio State University, Columbus, OH, USA
[2]ENS Lyon, Lyon, France
[3]Bilkent University, Ankara, Turkey

Definition

Hypergraphs are generalization of graphs where each edge (hyperedge) can connect more than two vertices. In simple terms, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal-sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized.

Discussion

Introduction

During the last decade, hypergraph-based models gained wide acceptance in the parallel computing community for modeling various problems. By providing natural way to represent multiway interactions and unsymmetric dependencies, hypergraph can be used to elegantly model complex computational structures in parallel computing. Here, some concrete applications will be presented to show how hypergraph models can be used to cast a suitable scientific problem as an hypergraph partitioning problem. Some insights and general guidelines for using hypergraph partitioning methods in some general classes of problems are also given.

Formal Definition of Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices (cells) $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n \in \mathcal{N}$ is a subset of vertices, that is, $n \subseteq \mathcal{V}$. The vertices in a net n are called its *pins*. The *size* of a net is equal to the number of its pins. The *degree* of a vertex is equal to the number of nets it is connected to. Graph is a special instance of hypergraph such that each net has exactly two pins. Vertices can be associated with weights, denoted with $w[\cdot]$, and nets can be associated with costs, denoted with $c[\cdot]$.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_K\}$ is a *K-way partition* of $\mathcal{H}$ if the following conditions hold:

- Each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, that is, $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$
- Parts are pairwise disjoint, that is, $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$
- Union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$

In a partition Π of $\mathcal{H}$, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity* λ_n of a net n denotes the number of parts connected by n. A net n is said to be *cut* (*external*) if it connects more than one part (i.e., $\lambda_n > 1$), and *uncut* (*internal*) otherwise (i.e., $\lambda_n = 1$). A partition is said to be balanced if each part $\mathcal{V}_k$ satisfies the *balance criterion*:

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \dots, K. \tag{1}$$

In (1), weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v \in \mathcal{V}_k} w[v]$), W_{avg} denotes the weight of each part under the perfect load balance condition (i.e., $W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K$), and ε represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. There are various [20] *cutsize* definitions for representing the cost $\chi(\Pi)$ of a partition Π. Two relevant definitions are:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \tag{2}$$

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \tag{3}$$

In (2), the cutsize is equal to the sum of the costs of the cut nets. In (3), each cut net n contributes $c[n](\lambda_n - 1)$ to the cutsize. The cutsize metrics given in (2) and (3) will be referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (1) among part weights is maintained.

A recent variant of the above problem is the *multi-constraint hypergraph* partitioning [9, 18] in which each vertex has a vector of weights associated with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. Here, $w[v,i]$ denotes the C weights of a vertex v for $i = 1,\dots,C$. Hence, the balance criterion (1) can be rewritten as

$$W_{k,i} \le W_{avg,i}\,(1+\varepsilon) \text{ for } k = 1,\dots,K \text{ and } i = 1,\dots,C\,, \tag{4}$$

where the ith weight $W_{k,i}$ of a part $\mathcal{V}_k$ is defined as the sum of the ith weights of the vertices in that part (i.e., $W_{k,i} = \sum_{v\in\mathcal{V}_k} w[v,i]$), and $W_{avg,i}$ is the average part weight for the ith weight (i.e., $W_{avg,i} = (\sum_{v\in\mathcal{V}} w[v,i])/K$), and ε again represents the allowed imbalance ratio.

Another variant is the *hypergraph partitioning with fixed vertices*, in which some of the vertices are fixed in some parts before partitioning. In other words, in this problem, a *fixed-part* function is provided as an input to the problem. A vertex is said to be *free* if it is allowed to be in any part in the final partition, and it is said to be fixed in part k if it is required to be in $\mathcal{V}_k$ in the final partition Π.

Yet another variant is *multi-objective hypergraph partitioning* in which there are several objectives to be minimized [1, 21]. Specifically, a given net contributes different costs to different objectives.

Sparse Matrix Partitioning

One of the most elaborated applications of hypergraph partitioning (HP) method in the parallel scientific computing domain is the parallelization of sparse matrix-vector multiply (SpMxV) operation. Repeated matrix-vector and matrix-transpose-vector multiplies that involve the same large, sparse matrix are the kernel operations in various iterative algorithms involving sparse linear systems. Such iterative algorithms include solvers for linear systems, eigenvalues, and linear programs. The pervasive use of such solvers motivates the development of HP models and methods for efficient parallelization of SpMxV operations.

Before discussing the HP models and methods for parallelizing SpMxV operations, it is favorable to discuss parallel algorithms for SpMxV. Consider the matrix-vector multiply of the form $y \leftarrow \mathbf{A}x$, where the nonzeros of the sparse matrix $\mathbf{A}$ as well as the entries of the input and output vectors x and y are partitioned arbitrarily among the processors. Let $map(\cdot)$ denote the nonzero-to-processor and vector-entry-to-processor assignments induced by this partitioning. A parallel algorithm would execute the following steps at each processor P_k.

1. Send the local input-vector entries x_j, for all j with $map(x_j) = P_k$, to those processors that have at least one nonzero in column j.
2. Compute the scalar products $a_{ij}x_j$ for the local nonzeros, that is, the nonzeros for which $map(a_{ij}) = P_k$ and accumulate the results y_i^k for the same row index i.
3. Send local nonzero partial results y_i^k to the processor $map(y_i) \neq P_k$, for all nonzero y_i^k.
4. Add the partial y_i^ℓ results received to compute the final results $y_i = \sum y_i^\ell$ for each i with $map(y_i) = P_k$.

As seen in the algorithm, it is necessary to have partitions on the matrix $\mathbf{A}$ and the input- and output-vectors x and y of the matrix-vector multiply operation. Finding a partition on the vectors x and y is referred to as the vector partitioning operation, and it can be performed in three different ways: by decoding the partition given on $\mathbf{A}$; in a post-processing step using the partition on the matrix; or explicitly partitioning the vectors during partitioning the matrix. In any of these cases, the vector partitioning for matrix-vector operations is called *symmetric* if x and y have the same partition, and *non-symmetric* otherwise. A vector partitioning is said to be *consistent*, if each vector entry is assigned to a processor that has at least one nonzero in the respective row or column of the matrix. The consistency is easy to achieve for the nonsymmetric vector partitioning; x_j can be assigned to any of the processors that has a nonzero in the column j, and y_i can be assigned to any of the processors that has a nonzero in the row i. If a symmetric vector partitioning is sought, then special care must be taken to assign a pair of matching input- and output-vector entries, e.g., x_i and

y_i, to a processor having nonzeros in both row and column i. In order to have such a processor for all vector entry pairs, the sparsity pattern of the matrix $\mathbf{A}$ can be modified to have a zero-free diagonal. In such cases, a consistent vector partition is guaranteed to exist, because the processors that own the diagonal entries can also own the corresponding input- and output-vector entries; x_i and y_i can be assigned to the processor that holds the diagonal entry a_{ii}.

In order to achieve an efficient parallelism, the processors should have balanced computational load and the inter-processor communication cost should have been minimized. In order to have balanced computational load, it suffices to have almost equal number of nonzeros per processor so that each processor will perform almost equal number of scalar products, for example, $a_{ij}x_j$, in any given parallel system. The communication cost, however, has many components (the total volume of messages, the total number of messages, maximum volume/number of messages in a single processor, either in terms of sends or receives or both) each of which can be of utmost importance for a given matrix in a given parallel system. Although there are alternatives and more elaborate proposals, the most common communication cost metric addressed in hypergraph partitioning-based methods is the total volume of communication.

Loosely speaking, hypergraph partitioning-based methods for efficient parallelization of SpMxV model the data of the SpMxV (i.e., matrix and vector entries) with the vertices of a hypergraph. A partition on the vertices of the hypergraph is then interpreted in such a way that the data corresponding to a set of vertices in a part are assigned to a single processor. More accurately, there are two classes of hypergraph partitioning-based methods to parallelizing SpMxV. The methods in the first class build a hypergraph model representing the data and invoke a partitioning heuristic on the so-built hypergraph. The methods in this class can be said to be models rather than being algorithms. There are currently three main hypergraph models for representing sparse matrices, and hence there are three methods in this first class. These three main models are described below in the next section. Essential property of these models is that the cutsize (3) of any given partition is equal to the total communication volume to be incurred under a consistent vector partitioning when the matrix elements are distributed according to the vertex partition. The methods in the second class follow a mix-and-match approach and use the three main models, perhaps, along with multi-constraint and fixed-vertex variations in an algorithmic form. There are a number of methods in this second class, and one can develop many others according to application needs and matrix characteristics. Three common methods belonging to this class are described later, after the three main models. The main property of these algorithms is that the sum of the cutsizes of each application of hypergraph partitioning amounts to the total communication volume to be incurred under a consistent vector partitioning (currently these methods compute a vector partitioning after having found a matrix partitioning) when the matrix elements are distributed according to the vertex partitions found at the end.

Three Main Models for Matrix Partitioning

In the *column-net hypergraph model* [11] used for 1D rowwise partitioning, an $M \times N$ matrix $\mathbf{A}$ with Z nonzeros is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ with $|\mathcal{V}_R| = M$ vertices, $|\mathcal{N}_C| = N$ nets, and Z pins. In $\mathcal{H}_{\mathcal{R}}$, there exists one vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ for each row i of matrix $\mathbf{A}$. Weight $w[v_i]$ of a vertex v_i is equal to the number of nonzeros in row i. The name of the model comes from the fact that columns are represented as nets. That is, there exists one unit-cost net $n_j \in \mathcal{N}_{\mathcal{C}}$ for each column j of matrix $\mathbf{A}$. Net n_j connects the vertices corresponding to the rows that have a nonzero in column j. That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$.

In the *row-net hypergraph model* [11] used for 1D columnwise partitioning, an $M \times N$ matrix $\mathbf{A}$ with Z nonzeros is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{C}} = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{R}})$ with $|\mathcal{V}_C| = N$ vertices, $|\mathcal{N}_R| = M$ nets, and Z pins. In $\mathcal{H}_{\mathcal{C}}$, there exists one vertex $v_j \in \mathcal{V}_{\mathcal{C}}$ for each column j of matrix $\mathbf{A}$. Weight $w[v_j]$ of a vertex $v_j \in \mathcal{V}_{\mathcal{R}}$ is equal to the number of nonzeros in column j. The name of the model comes from the fact that rows are represented as nets. That is, there exists one unit-cost net $n_i \in \mathcal{N}_{\mathcal{R}}$ for each row i of matrix $\mathbf{A}$. Net $n_i \subseteq \mathcal{V}_{\mathcal{C}}$ connects the vertices corresponding to the columns that have a nonzero in row i. That is, $v_j \in n_i$ if and only if $a_{ij} \neq 0$.

In the *column-row-net hypergraph model*, otherwise known as the *fine-grain model* [13], used for 2D nonzero-based fine-grain partitioning, an $M \times N$ matrix $\mathbf{A}$ with Z nonzeros is represented as a unit-weight and unit-cost hypergraph $\mathcal{H}_{\mathcal{Z}} = (\mathcal{V}_{\mathcal{Z}}, \mathcal{N}_{\mathcal{RC}})$ with $|\mathcal{V}_{\mathcal{Z}}| = Z$ vertices, $|\mathcal{N}_{\mathcal{RC}}| = M+N$ nets and $2Z$ pins. In $\mathcal{V}_Z$, there exists one unit-weight vertex v_{ij} for each nonzero a_{ij} of matrix $\mathbf{A}$. The name of the model comes from the fact that both rows and columns are represented as nets. That is, in $\mathcal{N}_{\mathcal{RC}}$, there exist one unit-cost row-net r_i for each row i of matrix $\mathbf{A}$ and one unit-cost column-net c_j for each column j of matrix A. The row-net r_i connects the vertices corresponding to the nonzeros in row i of matrix $\mathbf{A}$, and the column-net c_j connects the vertices corresponding to the nonzeros in column j of matrix $\mathbf{A}$. That is, $v_{ij} \in r_i$ and $v_{ij} \in c_j$ if and only if $a_{ij} \neq 0$. Note that each vertex v_{ij} is in exactly two nets.

Some Other Methods for Matrix Partitioning

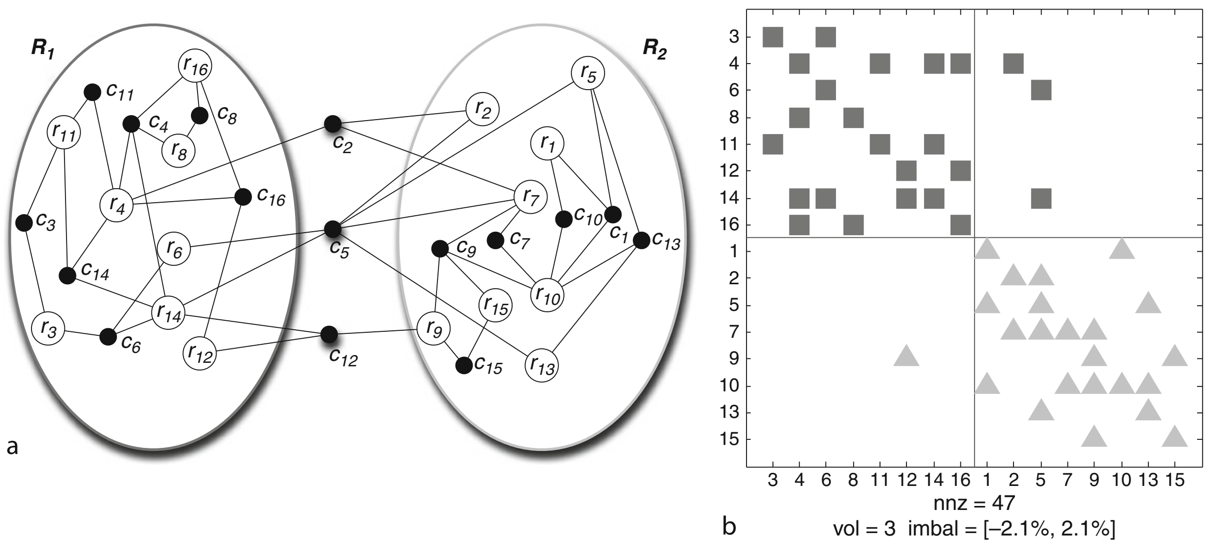
The *jagged-like partitioning method* [16] uses the row-net and column-net hypergraph models. It is an algorithm with two steps, in which each step models either the *expand phase* (the 1st line) or the *fold phase* (the 3rd line) of the parallel SpMxV algorithm given above. Therefore, there are two alternative schemes for this partitioning method. The one which models the expands in the first step and the folds in the second step is described below.

Given an $M \times N$ matrix $\mathbf{A}$ and the number K of processors organized as a $P \times Q$ mesh, the jagged-like partitioning model proceeds as shown in Fig. 1. The algorithm has two main steps. First, $\mathbf{A}$ is partitioned rowwise into P parts using the column-net hypergraph model $\mathcal{H}_{\mathcal{R}}$ (lines 1 and 2 of Fig. 1). Consider a P-way partition $\Pi_{\mathcal{R}}$ of $\mathcal{H}_{\mathcal{R}}$. From the partition $\Pi_{\mathcal{R}}$, one obtains P submatrices $\mathbf{A}_p$, for $p = 1, \ldots, P$ each having roughly equal number of nonzeros. For each p, the rows of the submatrix $\mathbf{A}_p$ correspond to the vertices in $\mathcal{R}_p$ (lines 6 and 7 of Fig. 1). The submatrix $\mathbf{A}_p$ is assigned to the pth row of the processor mesh. Second, each submatrix $\mathbf{A}_p$ for $1 \le p \le P$ is independently partitioned columnwise into Q parts using the row-net hypergraph $\mathcal{H}_p$ (lines 8 and 9 of Fig. 1). The nonzeros in the ith row of $\mathbf{A}$ are partitioned among the Q processors in a row of the processor mesh. In particular, if $v_i \in \mathcal{R}_p$ at the end of line 2 of the algorithm, then the nonzeros in the ith row of $\mathbf{A}$ are partitioned among the processors in the pth row of the processor mesh. After partitioning the submatrix $\mathbf{A}_p$ columnwise, the *map* array contains the partition information for the nonzeros residing in $\mathbf{A}_p$.

For each i, the volume of communication required to fold the vector entry y_i is accurately represented as a part of "foldVolume" in the algorithm. For each j, the volume of communication regarding the vector entry x_j

```
JAGGED-LIKE-PARTITIONING ($\mathbf{A}$, $K = P \times Q$, $\varepsilon_1$, $\varepsilon_2$)
Input : a matrix $\mathbf{A}$, the number of processors $K = P \times Q$, and the imbalance ratios $\varepsilon_1$, $\varepsilon_2$.
Output: $map(a_{ij})$ for all $a_{ij} \neq 0$ and total Volume.
1:  $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}}) \leftarrow$ columnNet($\mathbf{A}$)
2:  $\Pi_{\mathcal{R}} = \{\mathcal{R}_1,\ldots,\mathcal{R}_P\} \leftarrow$ partition($\mathcal{H}_{\mathcal{R}}$, $P$, $\varepsilon_1$)      ▷ rowwise partitioning of $\mathbf{A}$
3:  expand Volume $\leftarrow$ cutsize($\Pi_{\mathcal{R}}$)
4:  foldVolume $\leftarrow$ 0
5:  for $p = 1$ to $P$ do
6:      $R_p = \{r_i : v_i \in \mathcal{R}_p\}$
7:      $\mathbf{A}_p \leftarrow \mathbf{A}(R_p,:)$      ▷ submatrix indexed by rows $R_p$
8:      $\mathcal{H}_p = (\mathcal{V}_p, \mathcal{N}_p) \leftarrow$ rowNet($\mathbf{A}_p$)
9:      $\Pi_p^{\mathcal{C}} = \{\mathcal{C}_p^1,\ldots,\mathcal{C}_p^Q\} \leftarrow$ partition($\mathcal{H}_p$, $Q$, $\varepsilon_2$)      ▷ columnwise partitioning of $\mathbf{A}_p$
10:     foldVolume $\leftarrow$ foldVolume + cutsize($\Pi_p^{\mathcal{C}}$)
11:     for all $a_{ij} \neq 0$ of $\mathbf{A}_p$ do
12:         $map(a_{ij}) = P_{p,q} \Leftrightarrow c_j \in \mathcal{C}_p^q$
13: return totalVolume $\leftarrow$ expandVolume+foldVolume
```

Hypergraph Partitioning. Fig. 1 Jagged-like partitioning



Hypergraph Partitioning. Fig. 2 First step of four-way jagged-like partitioning of a matrix; (**a**) two-way partitioning $\Pi_{\mathcal{R}}$ of column-net hypergraph representation $\mathcal{H}_{\mathcal{R}}$ of **A**, (**b**) two-way rowwise partitioning of matrix $\mathbf{A}^{\Pi}$ obtained by permuting **A** according to the partitioning induced by Π; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval `imbal`; `vol` denotes the number of nonzeros and the total communication volume

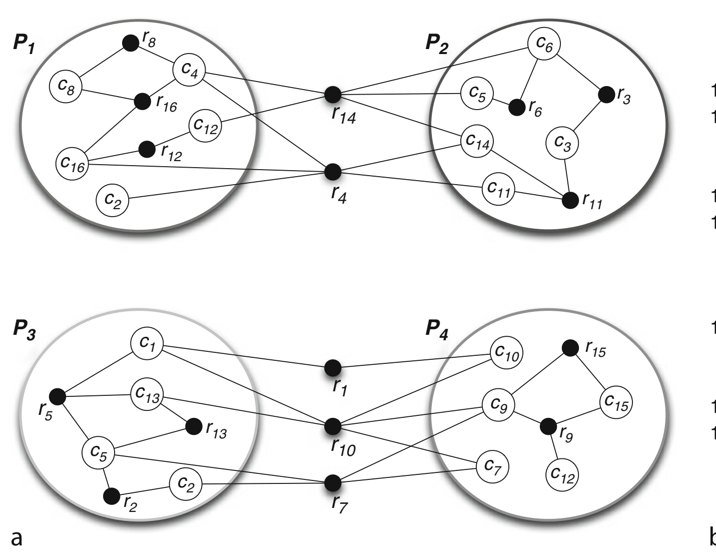
is accurately represented as a part of "expandVolume" in the algorithm.

Figure 2a illustrates the column-net representation of a sample matrix to be partitioned among the processors of a 2×2 mesh. For simplicity of the presentation, the vertices and the nets of the hypergraphs are labeled with letters "r" and "c" to denote the rows and columns of the matrix. The matrix is first partitioned rowwise into two parts, and each part is assigned to a row of the processor mesh, namely to processors $\{P_1, P_2\}$ and $\{P_3, P_4\}$. The resulting permuted matrix is displayed in Fig. 2b. Figure 3a displays the two row-net hypergraphs corresponding to each submatrix $\mathbf{A}_p$ for $p = 1, 2$. Each hypergraph is partitioned independently; sample partitions of these hypergraphs are also presented in this figure. As seen in the final symmetric permutation in Fig. 3b, the nonzeros of columns 2 and 5 are assigned to different parts, resulting P_3 to communicate with both P_1 and P_2 in the expand phase.

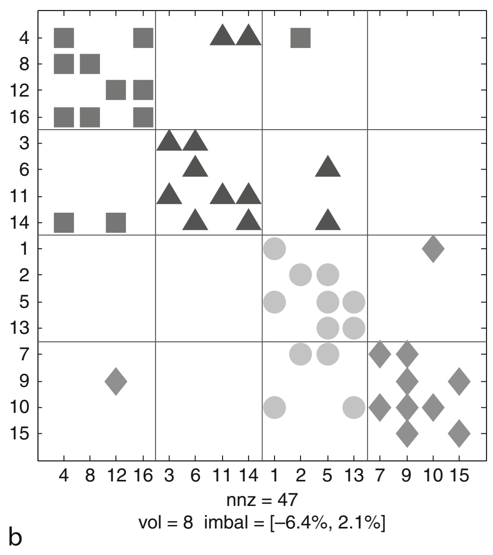
The *checkerboard partitioning method* [14] is also a two-step method, in which each step models either the expand phase or the fold phase of the parallel SpMxV. Similar to jagged-like partitioning, there are two alternative schemes for this partitioning method. The one which models the expands in the first step and the folds in the second step is presented below.

Given an $M \times N$ matrix **A** and the number K of processors organized as a $P \times Q$ mesh, the checkerboard partitioning method proceeds as shown in Fig. 4. First, **A** is partitioned rowwise into P parts using the column-net model (lines 1 and 2 of Fig. 4), producing $\Pi_{\mathcal{R}} = \{\mathcal{R}_1, \ldots, \mathcal{R}_P\}$. Note that this first step is exactly the same as that of the jagged-like partitioning. In the second step, the matrix **A** is partitioned columnwise into Q parts by using the multi-constraint partitioning to obtain $\Pi_{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_Q\}$. In comparison to the jagged-like method, in this second step the whole matrix **A** is partitioned (lines 4 and 8 of Fig. 4), not the submatrices defined by $\Pi_{\mathcal{R}}$. The rowwise and columnwise partitions $\Pi_{\mathcal{R}}$ and $\Pi_{\mathcal{C}}$ together define a 2D partition on the matrix **A**, where $map(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$ and $c_j \in \mathcal{C}_q$.

In order to achieve a load balance among processors, a multi-constraint partitioning formulation is used (line 8 of the algorithm). Each vertex v_i of $\mathcal{H}_{\mathcal{C}}$ is assigned P weights: $w[i, p]$, for $p = 1, \ldots, P$. Here, $w[i, p]$ is equal to the number of nonzeros of column c_i in rows $\mathcal{R}_p$ (line 7 of Fig. 4). Consider a Q-way partitioning of $\mathcal{H}_{\mathcal{C}}$ with P constraints using the vertex weight definition



Hypergraph Partitioning. Fig. 3 Second step of four-way jagged-like partitioning: (**a**) Row-net representations of submatrices of **A** and two-way partitionings, (**b**) Final permuted matrix; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval `imbal`; `nnz` and `vol` denote, respectively, the number of nonzeros and the total communication volume

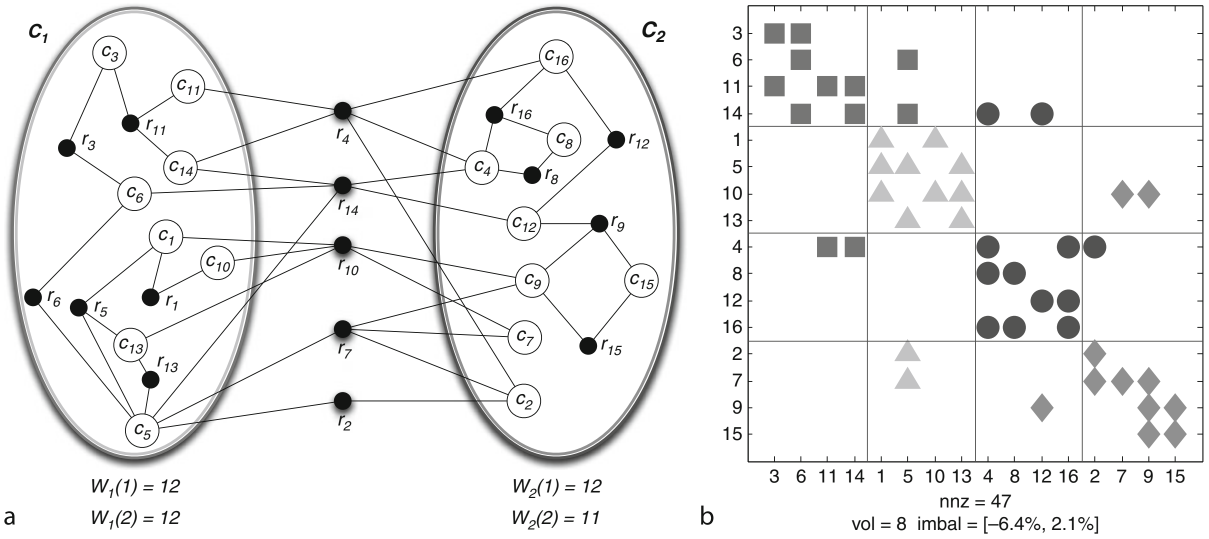


```
CHECKERBOARD-PARTITIONING(A, $K = P \times Q$, $\varepsilon_1$, $\varepsilon_2$)
Input: a matrix A, the number of processors $K = P \times Q$, and the imbalance ratios $\varepsilon_1$, $\varepsilon_2$.
Output: $map(a_{ij})$ for all $a_{ij} \neq 0$ and totalVolume.
 1: $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, \mathcal{N}_\mathcal{C}) \leftarrow$ columnNet(A)
 2: $\Pi_\mathcal{R} = \{\mathcal{R}_1,...,\mathcal{R}_P\} \leftarrow$ partition($\mathcal{H}_\mathcal{R}$, P, $\varepsilon_1$)      ▷ rowwise partitioning of A
 3: expand Volume $\leftarrow$ cutsize($\Pi_\mathcal{R}$)
 4: $\mathcal{H}_\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{N}_\mathcal{R}) \leftarrow$ rowNet(A)
 5: for $j = 1$ to $|\mathcal{V}_\mathcal{C}|$ do
 6:     for $p = 1$ to $P$ do
 7:         $w_{j,p} = |\{n_j \cap \mathcal{R}_p\}|$
 8: $\Pi_\mathcal{C} = \{\mathcal{C}_1,...,\mathcal{C}_Q\} \leftarrow$ MCPartition($\mathcal{H}_\mathcal{C}$, Q, $\varepsilon_2$)      ▷ columnwise partitioning of A
 9: foldVolume $\leftarrow$ $cutsize(\Pi_\mathcal{C})$
10: for all $a_{ij} \neq 0$ of A do
11:     $map(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$ and $c_j \in \mathcal{C}_q$
12: totalVolume $\leftarrow$ expandVolume+foldVolume
```

Hypergraph Partitioning. Fig. 4 Checkerboard partitioning

above. Maintaining the *P* balance constraints (4) corresponds to maintaining computational load balance on the processors of each row of the processor mesh.

Establishing the equivalence between the total communication volume and the sum of the cutsizes of the two partitions is fairly straightforward. The volume of communication for the fold operations corresponds exactly to the $cutsize(\Pi_\mathcal{C})$. The volume of communication for the expand operations corresponds exactly to the $cutsize(\Pi_\mathcal{R})$.



Hypergraph Partitioning. Fig. 5 Second step of four-way checkerboard partitioning: (**a**) two-way multi-constraint partitioning $\Pi_{\mathcal{C}}$ of row-net hypergraph representation $\mathcal{H}_{\mathcal{C}}$ of **A**, (**b**) Final checkerboard partitioning of **A** induced by $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval `imbal`; `nnz` and `vol` denote, respectively, the number of nonzeros and the total communication volume

Figure 5b displays the 2×2 checkerboard partition induced by $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$. Here, $\Pi_{\mathcal{R}}$ is a rowwise two-way partition giving the same figure as shown in Fig. 2, and $\Pi_{\mathcal{C}}$ is a two-way multi-constraint partition $\Pi_{\mathcal{C}}$ of the row-net hypergraph model $\mathcal{H}_{\mathcal{C}}$ of **A** shown in Fig. 5a. In Fig. 5a, $w[9,1] = 0$ and $w[9,2] = 4$ for internal column c_9 of row stripe $\mathcal{R}_2$, whereas $w[5,1] = 2$ and $w[5,2] = 4$ for external column c_5.

Another common method of matrix partitioning is the *orthogonal recursive bisection* (ORB) [27]. In this approach, the matrix is first partitioned rowwise into two submatrices using the column-net hypergraph model, and then each part is further partitioned columnwise into two parts using the row-net hypergraph model. The process is continued recursively until the desired number of parts is obtained. The algorithm is shown in Fig. 6. In this algorithm, *dim* represents either rowwise or columnwise partitioning, where $-dim$ switches the partitioning dimension.

In the ORB method shown above, the step *bisect* $(\mathbf{A}, dim, \varepsilon)$ corresponds to partitioning the given matrix either along the rows or columns with, respectively, the column-net or the row-net hypergraph models into two. The total sum of the cutsizes (3) of each each bisection step corresponds to the total communication volume. It is possible to dynamically adjust the ε at each recursive call by allowing larger imbalance ratio for the recursive call on the submatrix $\mathbf{A}_1$ or $\mathbf{A}_2$.

Some Other Applications of Hypergraph Partitioning

As said before, the initial motivations for hypergraph models were accurate modeling of the nonzero structure of unsymmetric and rectangular sparse matrices to minimize the communication volume for iterative solvers. There are other applications that can make use of hypergraph partitioning formulation. Here, a brief overview of general classes of applications is given along with the names of some specific problems. Further application classes are given in bibliographic notes.

Parallel reduction or aggregation operations form a significant class of such applications, including the MapReduce model. The reduction operation consists of computing M output elements from N input elements. An output element may depend on multiple input elements, and an input element may contribute to multiple output elements. Assume that the operation on which

```
ORB-PARTITIONING($\mathbf{A}$, $dim$, $K\,min$, $K\,max$, $\varepsilon$)
Input: a matrix $\mathbf{A}$, the part numbers $K\,min$ (at initial call, it is equal to 1) and $K\,max$ (at initial call it is
equal to $K$, the desired number of parts), and the imbalance ratio $\varepsilon$.
Output: $map(a_{ij})$ for all $a_{ij} \neq 0$.
1: if $K\,max - K\,min > 0$ then
2:     $mid \leftarrow (K\,max - K\,min + 1)/2$
3:     $\Pi = \langle \mathbf{A}_1, \mathbf{A}_2 \rangle \leftarrow$ bisect($\mathbf{A}$, $dim$, $\varepsilon$)     ▷Partition A along dim into two, producing two submatrices
4:     totalVolume$\leftarrow$totalVolume+$cutsize(\Pi)$
           ▷ Recursively partition each submatrix along the orthogonal direction
5:     $map1(\mathbf{A}_1) \leftarrow$ORB-PARTITIONING($\mathbf{A}_1$, $-dim$, $Kmin$, $Kmin + mid - 1$, $\varepsilon$)
6:     $map2(\mathbf{A}_2) \leftarrow$ORB-PARTITIONING($\mathbf{A}_2 - dim$, $Kmin + mid$, $Kmax$, $\varepsilon$)
7:     $map(\mathbf{A}) \leftarrow map1(\mathbf{A}_1) \cup map2(\mathbf{A}_2)$
8: else
9:     $map(\mathbf{A}) \leftarrow Kmin$
```

Hypergraph Partitioning. Fig. 6 Orthogonal recursive bisection (ORB)

reduction is performed is commutative and associative. Then, the inherent computational structure can be represented with an $M \times N$ dependency matrix, where each row and column of the matrix represents an output element and an input element, respectively. For an input element x_j and an output element y_i, if y_i depends on x_j, a_{ij} is set to 1 (otherwise zero). Using this representation, the problem of partitioning the workload for the reduction operation is equivalent to the problem of partitioning the dependency matrix for efficient SpMxV.

In some other reduction problems, the input and output elements may be preassigned to parts. The proposed hypergraph model can be accommodated to those problems by adding K *part* vertices and connecting those vertices to the nets which correspond to the preassigned input and output elements. Obviously, those part vertices must be fixed to the corresponding parts during the partitioning. Since the required property is already included in the existing hypergraph partitioners [6, 12, 19], this does not add extra complexity to the partitioning methods.

Iterative methods for solving linear systems usually employ *preconditioning* techniques. Roughly speaking, preconditioning techniques modify the given linear system to accelerate convergence. Applications of explicit preconditioners in the form of approximate inverses or factored approximate inverses are amenable to parallelization. Because, these techniques require SpMxV operations with the approximate inverse or factors of the approximate inverse at each step. In other words, preconditioned iterative methods perform SpMxV operations with both coefficient and preconditioner matrices in a step. Therefore, parallelizing a full step of these methods requires the coefficient and preconditioner matrices to be well partitioned, for example, processors' loads are balanced and communication costs are low in both multiply operations. To meet this requirement, the coefficient and preconditioner matrices should be partitioned simultaneously. One can accomplish such a simultaneous partitioning by building a single hypergraph and then partitioning that hypergraph. Roughly speaking, one follows a four-step approach: (i) build a hypergraph for each matrix, (ii) determine which vertices of the two hypergraphs need to be in the same part (according to the computations forming the iterative method), (iii) amalgamate those vertices coming from different hypergraphs, (iv) if the computations represented by the two hypergraphs of the first step are separated by synchronization points then assign multiple weights to vertices (the weights of the vertices of the hypergraphs of the first step are kept), otherwise assign a single weight to vertices (the weights of the vertices of the hypergraphs of the first step are summed up for each amalgamation).

The computational structure of the preconditioned iterative methods is similar to that of a more general class of scientific computations including multiphase, multiphysics, and multi-mesh simulations.

In *multiphase simulations*, there are a number of computational phases separated by global synchronization points. The existence of the global synchronizations necessitates each phase to be load balanced individually. Multi-constraint formulation of hypergraph partitioning can be used to achieve this goal.

In *multi-physics simulations*, a variety of materials and processes are analyzed using different physics procedures. In these types of simulations, computational as well as the memory requirements are not uniform across the mesh. For scalability issues, processor loads should be balanced in terms of these two components. The multi-constraint partitioning framework also addresses these problems.

In *multi-mesh simulations*, a number of grids with different discretization schemes and with arbitrary overlaps are used. The existence of overlapping grid points necessitates a simultaneous partitioning of the grids. Such a simultaneous partitioning scheme should balance the computational loads of the processors and minimize the communication cost due to interactions within a grid as well as the interactions among different grids. With a particular transformation (the vertex amalgamation operation, also mentioned above), hypergraphs can be used to model the interactions between different grids. With the use of multi-constraint formulation, the partitioning problem in the multi-mesh computations can also be formulated as a hypergraph partitioning problem.

In obtaining partitions for two or more computation phases interleaved with synchronization points, the hypergraph models lead to the minimization of the overall sum of the total volume of communication in all phases (assuming that a single hypergraph is built as suggested in the previous paragraphs). In some sophisticated simulations, the magnitude of the interactions in one phase may be different than that of the interactions in another one. In such settings, minimizing the total volume of communication in each phase separately may be advantageous. This problem can be formulated as a multi-objective hypergraph partitioning problem on the so-built hypergraphs.

There are certain limitations in applying hypergraph partitioning to the multiphase, multiphysics, and multi-mesh-like computations. The dependencies must remain the same throughout the computations, otherwise the cutsize may not represent the communication volume requirements as precisely as before. The weights assigned to the vertices, for load balancing issues, should be static and available prior to the partitioning; the hypergraph models cannot be used as naturally for applications whose computational requirements vary drastically in time. If, however, the computational requirements change gradually in time, then the models can be used to re-partition the load at certain time intervals (while also minimizing the redistribution or migration costs associated with the new partition).

Ordering methods are quite common techniques to permute matrices in special forms in order to reduce the memory and running time requirements, as well as to achieve increased parallelism in direct methods (such as *LU* and Cholesky decompositions) used for solving systems of linear equations. Nested-dissection is a well-known ordering method that has been used quite efficiently and successfully. In the current state-of-the-art variations of the nested-dissection approach, a matrix is symmetrically permuted with a permutation matrix $\mathbf{P}$ into doubly bordered block diagonal form

$$\mathbf{A}_{DB} = \mathbf{PAP}^T \begin{bmatrix} A_{11} & & & & A_{1S} \\ & A_{22} & & & A_{2S} \\ & & \ddots & & \vdots \\ & & & A_{KK} & A_{KS} \\ A_{S1} & A_{S2} & \cdots & A_{SK} & A_{SS} \end{bmatrix},$$

where the nonzeros are only in the marked blocks (the blocks on the diagonal and the row and column borders). The aim in such a permutation is to have reduced numbers of rows/columns in the borders and to have equal-sized square blocks in the diagonal. One way to achieve such a permutation when $\mathbf{A}$ has symmetric pattern is as follows. Suppose a matrix $\mathbf{B}$ is given (if not, it is possible to find one) where the sparsity pattern of $\mathbf{B}^T\mathbf{B}$ equals to that of $\mathbf{A}$ (here arithmetic cancellations are ignored). Then, one can permute $\mathbf{B}$ nonsymmetrically into the singly bordered form

$$\mathbf{B}_{SB} = \mathbf{QBP}^T \begin{bmatrix} B_{11} & & & & B_{1S} \\ & B_{22} & & & B_{2S} \\ & & \ddots & & \vdots \\ & & & B_{KK} & B_{KS} \end{bmatrix},$$

so that $\mathbf{B}_{SB}^T\mathbf{B}_{SB} = \mathbf{PAP}^T$; that is, one can use the column permutation of $\mathbf{B}$ resulting in $\mathbf{B}_{SB}$ to obtain a symmetric permutation for $\mathbf{A}$ which results in $\mathbf{A}_{DB}$. Clearly, the column dimension of B_{kk} will be the size of the square matrix A_{kk} and the number of rows and columns in the border will be equal to the number of columns in the column border of $\mathbf{B}_{SB}$. One can achieve such a permutation of $\mathbf{B}$ by partitioning the column-net model of $\mathbf{B}$ while reducing the cutsize according to the cut-net metric (2), with unit net costs, to obtain the permutation $\mathbf{P}$ as follows. First, the permutation $\mathbf{Q}$ is defined to be able to define $\mathbf{P}$. Permute all rows corresponding to the vertices in part k before those in a part ℓ, for $1 \leq k < \ell \leq K$. Then, permute all columns corresponding to the nets that are internal to a part k before those that are internal to a part ℓ, for $1 \leq k < \ell \leq K$, yielding the diagonal blocks, and then permute all columns corresponding to the cut nets to the end, yielding the column border (the order of column defining a diagonal block). Clearly the correspondence between the size of the column border of $\mathbf{B}_{SB}$ and the doubly border of $\mathbf{A}_{DB}$ is exact, and hence the cutsize according to the cut-net metric is an exact measure. The requirement to have almost equal sized square blocks A_{kk} decoded as the requirement that each part should have an almost equal number of internal nets in the partition of the column-net model of $\mathbf{B}$. Although such a requirement is neither the objective nor the constraint of the hypergraph partitioning problem, the common hypergraph-partitioning heuristics easily accommodate such requirements.

Related Entries

►Data Distribution
►Graph Algorithms
►Graph Partitioning
►Linear Algebra, Numerical
►PaToH (Partitioning Tool for Hypergraphs)
►Preconditioners for Sparse Iterative Methods
►Sparse Direct Methods

Bibliographic Notes and Further Reading

The first use of the hypergraph partitioning methods for efficient parallel sparse matrix-vector multiply operations is seen in [10]. A more comprehensive study [11] describes the use of the row-net and column-net hypergraph models in 1D sparse matrix partitioning. For different views and alternatives on vector partitioning see [5, 22, 24].

A fair treatment of parallel sparse matrix-vector multiplication, analysis, and investigations on certain matrix types along with the use of hypergraph partitioning is given in [4, Chapter 4]. Further analysis of hypergraph partitioning on some model problems is given in [25].

Hypergraph partitioning schemes for preconditioned iterative methods are given in [23], where vertex amalgamation and multi-constraint weighting to represent different phases of computations are given. Discussions on application of such methodology for multiphase, multiphysics, and multi-mesh simulations are also discussed in the same paper.

Some different methods for sparse matrix partitioning using hypergraphs can be found in [26], including jagged-like and checkerboard partitioning methods, and in [27], the orthogonal recursive bisection approach. A recipe to choose a partitioning method for a given matrix is given in [16].

The use of hypergraph models for permuting matrices into special forms such as singly bordered block-diagonal form can be found in [3]. This permutation can be leveraged to develop hypergraph partitioning-based symmetric [9, 15] and nonsymmetric [17] nested-dissection orderings.

The standard hypergraph partitioning and the hypergraph partitioning with fixed vertices formulation, respectively, is used for static and dynamic load balancing for some scientific applications in [7, 8].

Some other applications of hypergraph partitioning are briefly summarized in [2]. These include image-space parallel direct volume rendering, parallel mixed integer linear programming, data declustering for multi-disk databases, scheduling file-sharing tasks in heterogeneous master-slave computing environments, and work-stealing scheduling, road network clustering methods for efficient query processing, pattern-based data clustering, reducing software development and maintenance costs, processing spatial join operations, and improving locality in memory or cache performance.

Bibliography

1. Ababei C, Selvakkumaran N, Bazargan K, Karypis G (2002) Multi-objective circuit partitioning for cutsize and path-based delay minimization. In: Proceedings of ICCAD 2002, San Jose, CA, November 2002
2. Aykanat C, Cambazoglu BB, Uçar B (2008) Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. J Parallel Distr Comput 68(5): 609–625
3. Aykanat C, Pınar A, Çatalyürek UV (2004) Permuting sparse rectangular matrices into block-diagonal form. SIAM J Sci Comput 26(6):1860–1879
4. Bisseling RH (2004) Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, Oxford, UK
5. Bisseling RH, Meesen W (2005) Communication balancing in parallel sparse matrix-vector multiplication. Electron Trans Numer Anal 21:47–65
6. Boman E, Devine K, Heaphy R, Hendrickson B, Leung V, Riesen LA, Vaughan C, Catalyurek U, Bozdag D, Mitchell W, Teresco J (2007) Zoltan 3.0: parallel partitioning, load balancing, and data-management services; user's guide. Sandia National Laboratories, Albuquerque, NM, 2007. Technical Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html
7. Cambazoglu BB, Aykanat C (2007) Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. IEEE Trans Parallel Distr Syst 18(1):3–16
8. Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (2009) A repartitioning hypergraph model for dynamic load balancing. J Parallel Distr Comput 69(8):711–724
9. Çatalyürek UV (1999) Hypergraph models for sparse matrix partitioning and reordering. Ph.D. thesis, Computer Engineering and Information Science, Bilkent University. Available at http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html
10. Çatalyürek UV, Aykanat C (1995) A hypergraph model for mapping repeated sparse matrixvector product computations onto multicomputers. In: Proceedings of International Conference on High Performance Computing (HiPC'95), Goa, India, December 1995
11. Çatalyürek UV, Aykanat C (1999) Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans Parallel Distr Syst 10(7):673–693
12. Çatalyürek UV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, version 3.0. Department of Computer Engineering, Bilkent University, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm
13. Çatalyürek UV, Aykanat C (2001) A fine-grain hypergraph model for 2D decomposition of sparse matrices. In: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, April 2001
14. Çatalyürek UV, Aykanat C (2001) A hypergraph-partitioning approach for coarse-grain decomposition. In: ACM/IEEE SC2001, Denver, CO, November 2001
15. Çatalyürek UV, Aykanat C, Kayaaslan E (2009) Hypergraph partitioning-based fill-reducing ordering. Technical Report OSUBMI-TR-2009-n02 and BU-CE-0904, Department of Biomedical Informatics, The Ohio State University and Computer Engineering Department, Bilkent University (Submitted)
16. Çatalyürek UV, Aykanat C, Uçar B (2010) On two-dimensional sparse matrix partitioning: models, methods, and a recipe. SIAM J Sci Comput 32(2):656–683
17. Grigori L, Boman E, Donfack S, Davis T (2008) Hypergraph unsymmetric nested dissection ordering for sparse LU factorization. Technical Report 2008-1290J, Sandia National Labs, Submitted to SIAM J Sci Comp
18. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, Department of Computer Science, University of Minnesota/Army HPC Research Center, Minneapolis, MN 55455
19. Karypis G, Kumar V, Aggarwal R, Shekhar S (1998) hMeTiS a hypergraph partitioning package, version 1.0.1. Department of Computer Science, University of Minnesota/Army HPC Research Center, Minneapolis
20. Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Wiley–Teubner, Chichester
21. Selvakkumaran N, Karypis G (2003) Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In: Proceedings of ICCAD 2003, San Jose, CA, November 2003
22. Uçar B, Aykanat C (2004) Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. SIAM J Sci Comput 25(6):1837–1859
23. Uçar B, Aykanat C (2007) Partitioning sparse matrices for parallel preconditioned iterative methods. SIAM J Sci Comput 29(4):1683–1709
24. Uçar B, Aykanat C (2007) Revisiting hypergraph models for sparse matrix partitioning. SIAM Review 49(4):595–603
25. Uçar B, Çatalyürek UV (2010) On the scalability of hypergraph models for sparse matrix partitioning. In: Danelutto M, Bourgeois J, Gross T (eds), Proceedings of the 18th Euromicro Conference on Parallel, Distributed, and Network-based Processing, IEEE Computer Society, Conference Publishing Services, pp 593–600
26. Uçar B, Çatalyürek UV, Aykanat C (2010) A matrix partitioning interface to PaToH in MATLAB. Parallel Computing 36(5–6):254–272
27. Vastenhouw B, Bisseling RH (2005) A two-dimensional data distribution method for parallel sparse-matrix-vector multiplication. SIAM Review 47(1):67–95

Hyperplane Partitioning

►Tiling

HyperTransport

Federico Silla
Universidad Politécnica de Valencia, Valencia, Spain

Synonyms

HT; HT3.10

Definition

HyperTransport is a scalable packet-based, high-bandwidth, and low-latency point-to-point interconnect technology intended to interconnect processors and also link them to I/O peripheral devices. HyperTransport was initially devised as an efficient replacement for traditional system buses for on-board communications. Nevertheless, the last extension to the standard, referred to as High Node Count HyperTransport, as well as the recent standardization of new cables and connectors, allow HyperTransport to efficiently extend its interconnection capabilities beyond a single motherboard and become a very efficient technology to interconnect processors and I/O devices in a cluster. HyperTransport is an open standard managed by the HyperTransport Consortium.

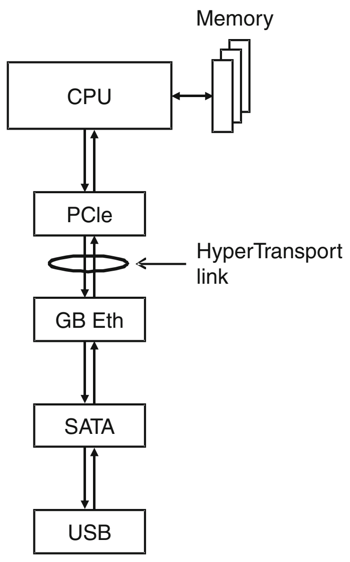
Discussion

A complete description of the HyperTransport technology should include both an introduction to the protocol used by communicating devices to exchange data and also a description of the electrical interface that this technology makes use of in order to achieve its tremendous performance. However, describing the electrical interface seems to be less interesting than the protocol itself and, additionally, requires the reader to have a large electrical background. Therefore, the following description will be focused on the protocol used by the HyperTransport technology, leaving aside the electrical details. On the other hand, the protocol used by HyperTransport is a quite complex piece of technology, thus requiring an extensive explanation, which may be out of the scope of this encyclopedia. For this reason, the following description just tries to be a brief introduction to HyperTransport. Finally, the reader should note that AMD uses an extended version of the HyperTransport protocol in order to provide cache coherency among the processors in a system. Such extended protocol, usually referred to as coherent HyperTransport (cHT), is proprietary to AMD, and therefore it is not described in this document. Nevertheless, the main difference between both protocols is that the coherent one includes some additional types of packets.

HyperTransport Links

The HyperTransport technology is a point-to-point communication standard, meaning that each of the HyperTransport links in the system connects exactly two devices. Figure 1 shows a simplified diagram of a system that deploys HyperTransport in order to interconnect the devices it is composed of. As can be seen in that figure, the main processor is connected to a PCIe device by means of a HyperTransport link. That PCIe device is, at the same time, connected to a GigaByte Ethernet device, which is, additionally, connected to a SATA device. This device connects to a USB device. All of these devices make up a HyperTransport daisy chain. Nevertheless, devices can implement multiple HyperTransport links in order to build larger HyperTransport fabrics.

Each of the links depicted in Fig. 1 consists of two unidirectional and independent sets of signals. Each of these sets includes its own CAD signal, as well as CTL and CLK signals.



HyperTransport. Fig. 1 System deploying HyperTransport

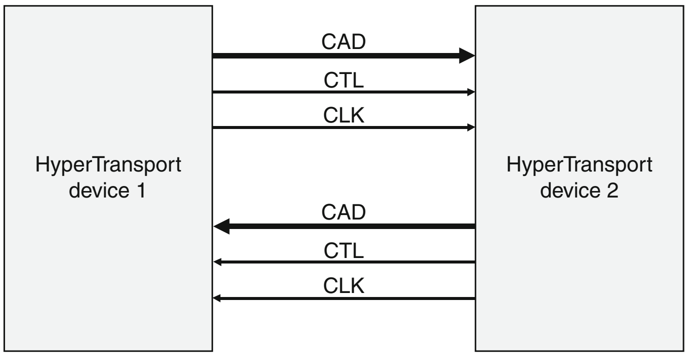
The CAD signal (named after Command, Address, and Data) carries control and data packets. The CTL signal (named after ConTroL) is intended to differentiate control from data packets in the CAD signal. Finally, the CLK signal (named after CLocK) carries the clock for the CAD and CTL signals. Figure 2 shows a diagram presenting all these signals.

The width of the CAD signal is variable from 2 to 32 bits. Actually, not all values are possible. Only widths of 2-, 4-, 8-, 16-, or 32-bits are allowed. Nevertheless, note that the HyperTransport protocol remains the same independently of the exact link width. More precisely, the format of the packets exchanged among HyperTransport devices does not depend on link width. However, a given packet will require more time to be transmitted in a 2-bit link than in a 32-bit one. The reason for having several widths for the CAD signal is allowing system developers to tune their system for a given performance/cost design point. Narrower links will be cheaper to implement, but they will also provide lower performance.

On the other hand, the width of the CAD signal in each of the unidirectional portions of the link may be different, that is, HyperTransport allows asymmetrical link widths. Therefore, a given device may implement a 32-bit wide link in one direction while deploying a 4-bit link in the other, for example. The usefulness of such asymmetry is based on the fact that, if such a device sends most of its data in one direction and receives a limited amount of data in the other direction, then the system designer can reduce manufacturing cost by providing a wide link in the direction that requires higher bandwidth and a narrow link for the opposite direction, which requires much less bandwidth.

In addition to the variable link width, HyperTransport also supports variable clock speeds, thus increasing even more the possibilities the system designer has for tuning the bandwidth of the links. The clock speeds currently supported by the HyperTransport specification are 200 MHz, 300 MHz, 400 MHz, 500 MHz, 600 MHz, 800 MHz, 1 GHz, 1.2 GHz, 1.6 GHz, 2 GHz, 2.4 GHz, 2.6 GHz, 2.8 GHz, 3 GHz, and 3.2 GHz. Moreover, clock frequency in both directions of a link does not need to be the same, thus introducing an additional degree of asymmetry. On the other hand, the clock mechanism used in HyperTransport is referred to as double data rate (DDR), which means that rising and falling edges of the clock signal are used to latch data, thus achieving an effective clock frequency that doubles the actual clock rate.

In summary, when variable link width is combined with variable clock frequency, HyperTransport links present an extraordinary scalability in terms of performance. For example, when both directions of a link are 2-bit wide working at 200 MHz, link bandwidth is 200 MB/s. This would be the lowest performance/lowest cost configuration. On the opposite, when 32-bit wide links are used at 3.2 GHz, overall link performance rises up to 51.2 GB/s. This implementation represents the highest bandwidth configuration, also presenting the highest manufacturing cost. This link configuration could be interesting for extreme performance systems, for example, which usually require as much bandwidth as possible. On the other hand, slow devices not



HyperTransport. Fig. 2 Signals in a HyperTransport link requiring high bandwidth could reduce cost by using narrow links. If these links are additionally clocked at low frequencies, then power consumption is further reduced.

In the same way that the CAD signal can be implemented with a variable width, the CTL and CLK signals can also be implemented with several widths. However, their width does not depend on the implementer's criterion to choose a performance/cost design point, but on the width of the CAD signal. Additionally, as the CAD signal can present a different width in each direction of the link, then the width of the CTL and CLK signals can also be different in each direction, depending on the corresponding CAD signal in that direction.

In the case of the CTL signal, there is an individual CTL bit for each set of 8, or fewer, CAD bits. Therefore, 1, 2, or 4 CTL bits can be found in a HyperTransport link. Moreover, the CTL bits are encoded in such a way that four CTL bits are transferred every 32 CAD bits. These four bits are intended to denote different flavors of the information being transmitted in the CAD signal. These different flavors may include, for example, that a command is being transmitted, that a CRC for a command without data is in the CAD signal, that the CAD signal is being used by a data packet, etc.

In the case for the CLK signal, a HyperTransport link has an individual CLK bit for every set of 8, or fewer, CAD bits. Thus, the number of CTL and CLK bits in a given link is the same. The reason for having a CLK bit for every 8 bits of the CAD signal is because link implementation is made easier. Effectively, the HyperTransport clocking scheme requires that the skew between clock and data signals must be minimized in order to achieve high transmission rates. Therefore, having a CLK bit for every 8 CAD bits allows that differences in trace lengths in the board layout are much lower than just having a CLK bit for the entire set of CAD bits.

In addition to the signals mentioned above, all HyperTransport devices share one PWROK and one RESET# signals for initialization and reset purposes. Moreover, if devices require power management, they should additionally include LDTSTOP# and LDTREQ# signals.

HyperTransport Packets

Once described the links that connect devices in a HyperTransport fabric, this section presents the packets that are forwarded along those links. Packets in HyperTransport are multiples of 4-bytes long and carry the command, address, and data associated with each transaction among devices. Packets can be classified into control and data packets.

Control packets are used to initiate and finalize transactions as well as to manage several HyperTransport features, and consist of 4 or 8 bytes. Control packets can be classified into information, request, and response packets.

Information packets are used for several link management purposes, like link synchronization, error condition signaling, and updating flow control information. Information packets are always 4 bytes long and can only be exchanged among adjacent devices directly interconnected by a link.

On the other hand, request and response control packets are used to build HyperTransport transactions. Request packets, which are 4- or 8-bytes long, are used to initiate HyperTransport transactions. On the other hand, response packets, which are always 4-bytes long, are used in the response phase of transactions to reply to a previous request. Table 1 shows the different request and response types of packets. As can be seen, there are two different types of sized writes: posted and non-posted. Although both types write data to the target device of the request packet, their semantics are different. Non-posted writes require a response packet to be sent back to the requesting device in order to confirm that the operation has completed. On the other hand, posted writes do not require such confirmation.

HyperTransport. Table 1 Types of request and response control packets

	Packet type
Request packet	Sized read
	Sized write (non-posted)
	Sized write (posted)
	Atomic read-modify-write
	Broadcast
	Flush
	Fence
	Address extension
	Source identifier extension
Response packet	Read response
	Target done

All the packet types will be further described in next section, except the extension packets. These packets are 4-bytes long extensions that prepend some of the other packets, when required. Their purpose is to allow 64-bit addressing instead of the 40-bit one used by default, in the case of the Address Extension, or 16-bit source identifiers in bus-device-function format in the case of the Source Identifier Extension.

With respect to data packets, they carry the actual data transferred among devices. Data packets only include data, with no associated control field. Therefore, data packets immediately follow the associated control packet. For example, a read response control packet, which includes no data, will be followed by the associated data packet carrying the read data. In the same way, a write request control packet will be followed by the data to be written.

Data packet length depends on the command that generated that data packet. Nevertheless, the maximum length is 64 bytes.

HyperTransport Transactions

HyperTransport devices transfer data by means of transactions. For example, every time a device requests some data from another device, the former initiates a read transaction targeted to the latter. Write transactions happen in a similar way. For example, when a program writes some data to disk, a write transaction is initiated among the processor, which reads the data from main memory, and the SATA device depicted in Fig. 1. Other transactions include broadcasting a message or explicitly taking control of the order of pending transactions.

Every transaction has a request phase. Request control packets are used in this phase. The exact request control packet to be used depends on the particular transaction taking place. On the other hand, many transactions require a response stage. Response control packets are used in this case, for example, to return read data from the target of the transaction, or to confirm its completion.

There are six basic transaction types:

- Sized read transaction
- Sized write transaction
- Atomic read-modify-write transaction
- Broadcast transaction
- Flush transaction
- Fence transaction

Sized Read Transaction

Sized read transactions are used by devices when they request data located in the address space of another device. For example, when a device wants to read data from main memory, it starts a read transaction targeted to the main processor. Also, when the processor requires some data from the USB device in Fig. 1, it will issue a read transaction destined to that device.

Read transactions begin with a sized read request control packet being issued by the device requesting the data. Once this packet reaches the destination device, it accesses the requested data and generates a response. This response will be composed of a read response control packet followed by the read data, included in a data packet. Once these two packets arrive at the device that initiated the process, the transaction is completed.

It is noteworthy mentioning that during the time elapsed since the requestor delivered the read request on the link until it receives the corresponding response, the HyperTransport chain is not idle. On the opposite, as HyperTransport is a split transaction protocol, other transactions can be issued (even finalized) before our requestor receives the required data.

Sized Write Transaction

Sized write transactions are similar to sized read ones, with the difference that the requestor device writes data to the target device instead of requesting data from it. A write transaction may happen when a device sends data to memory, or when the processor writes back data from memory to disk, for example.

There are two different sized write transactions: posted and non-posted. Posted write transactions start when the requestor sends to the target a posted write request control packet followed by a data packet containing the data to be written. In this case, because of the posted nature of the transaction, no response packet is sent back to the requestor. On the other hand, non-posted write transactions begin with a non-posted write control packet being issued by the requestor (followed by a data packet). In this case, when both packets reach the destination and the data are written, the target issues back a target-done response control packet to the requestor. When the requestor receives this response, the transaction is finished.

Atomic Read-Modify-Write Transaction

Atomic read-modify-write transactions are intended to atomically access a memory location and modify it. This means that no other device in the system may access the same memory location during the time required to read and modify it. This is useful to avoid race conditions among devices while performing the mutual exclusion required to access a critical section, for example.

Two different types of atomic operations are allowed:

- Fetch and Add
- Compare and Swap

The Fetch and Add operation is:

```
Fetch_and_Add(Out, Addr, In) {
Out = Mem[Addr];
Mem[Addr] = Mem[Addr] + In;
}
```

The Compare and Swap operation is:

```
Compare_and_Swap(Out, Addr, Compare,
 In) {
Out = Mem[Addr];
If (Mem[Addr] == Compare) Mem[Addr]
= In;
}
```

The atomic transaction begins when the requestor issues an atomic read-modify-write request control packet on the link followed by a data packet containing the argument of the atomic operation. Once both packets are received at the target and it performs the requested atomic operation, it will send back a read response control packet followed by a data packet containing the original data read from the memory location.

Broadcast Transaction

This transaction is used by the processor to communicate information to all HyperTransport devices in the system. This transaction is started with a broadcast request control packet, which can only be issued by the processor. All the other devices accept that packet and forward it to the rest of devices in the system. Broadcast requests include halt, shutdown, and End-Of-Interrupt commands.

Flush Transaction

Posted writes do not generate any response once completed. Therefore, when a device issues one or more posted writes targeted to the main processor in order to write some data to main memory, the issuing device has no way to know that those writes have effectively completed their way to memory. Thus, if some of the posted writes have not been completely written to memory, that data will not be visible to other devices in the system. In this scenario, flush transactions allow the device that issued the posted writes to make sure that data has reached main memory by flushing all pending posted writes to memory.

Flush transactions begin when a device issues a flush control packet targeted to the processor. Once this control packet reaches the destination, all pending transactions will be flushed to memory and then the processor will generate a target-done response control packet back to the requestor. Once this packet is received, the transaction is completed.

Fence Transaction

The fence transaction is intended to provide a barrier among posted writes. When a processor (the only possible target of a fence transaction) receives a fence command, it will make sure that no posted write received after the fence command is written to memory before any of the posted writes received earlier than the fence command.

The fence transaction begins when a device issues a fence control packet. No response is generated for this transaction.

Virtual Channels and Flow Control in HyperTransport

All the different types of control and data packets are multiplexed on the same link and stored in input buffers when they reach the receiving end of the link. Then, they are either accepted by that device in case they are targeted to it or forwarded to the next link in the chain.

If packets are not carefully managed, a protocol deadlock may occur. For example, if many devices in the system issue a large number of non-posted requests, those requests may fill up all the available buffers and hinder responses to make forward progress back to the initial requestors. In this case, requestors would stop forever because they are waiting for responses that will never arrive because the interconnect is full of requests that avoid responses to advance. Additionally, requests stored in the intermediate buffers will not be able to advance toward their destination because output buffers at the targets will be filled with responses that cannot enter the link, thus hindering targets to accept new requests from the link because they have no free space neither to store them nor to store the responses they would generate. As can be seen, in this situation, no packet can advance because of lack of available free buffers. The result is that the system freezes.

In order to avoid such deadlocks, HyperTransport splits traffic into virtual channels and stores different types of packets in buffers belonging to different virtual channels. Additionally, HyperTransport does not allow that packets traveling in one virtual channel move to another virtual channel. In this way, if non-posted requests use a virtual channel different from the one used by responses, the deadlock described above can be avoided.

HyperTransport defines a base set of virtual channels that must be supported by all HyperTransport devices. Moreover, some additional virtual channel sets are also defined, although support for them is optional. Regarding the base set, it includes three different virtual channels:

- The posted request virtual channel, which carries posted write transactions
- The non-posted request virtual channel, which includes reads, non-posted writes, and flush packets
- The response virtual channel, which is responsible for read responses and target-done control packets

In addition to separate traffic into the three virtual channels mentioned above, each device must implement separate control and data buffers for each of the virtual channels. Therefore, there are six types of buffers:

- Non-posted request control buffer
- Posted request control buffer
- Response control buffer
- Non-posted request data buffer
- Posted request data buffer
- Response data buffer

Figure 3 shows the basic buffer configuration for a HyperTransport link. The exact number of packets that can be stored in each buffer depends on the implementation. Nevertheless, request and response buffers must contain, at least, enough space to store the largest control packet of that type. Also, all data buffers can hold 64 bytes. Moreover, in order to improve performance, a HyperTransport device may have larger buffers, able to store multiple packets of each type.

The HyperTransport protocol states that a transmitter should not issue a packet that cannot be stored by the receiver. Thus, the transmitter must know how many buffers of each type the receiver has available. To achieve this, a credit-based scheme is used between transmitters and receivers. With such scheme, the transmitter has a counter for each type of buffer implemented at the receiver. When the transmitter sends a packet, it decrements the associated counter. When one of the counters reaches zero, the transmitter stops sending packets of that type. On the other hand, when the receiver frees a buffer, it sends back a NOP control packet (No Operation Packet) to the transmitter in order to update it about space availability.

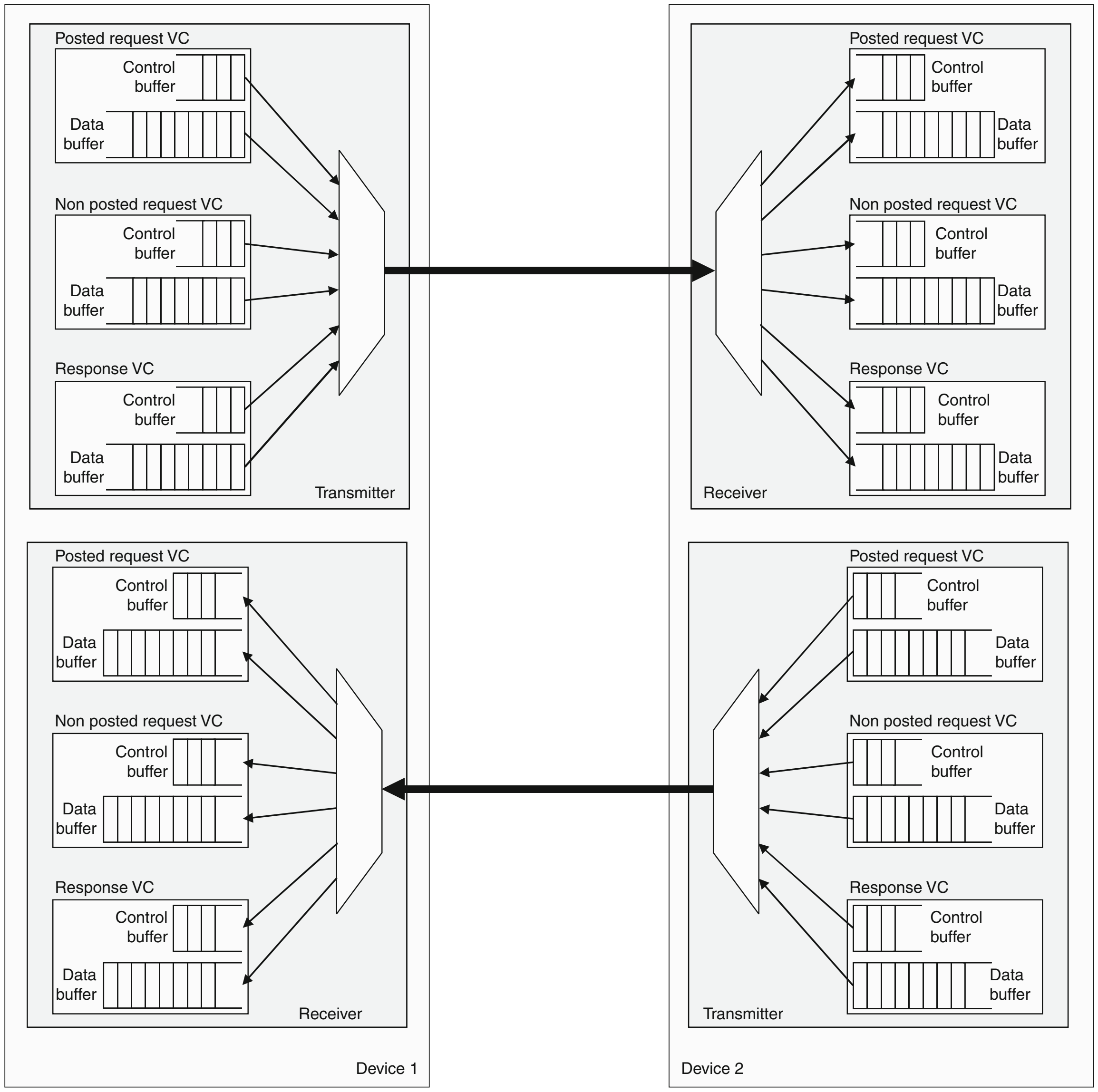
Extending the HyperTransport Fabric

The system depicted in Fig. 1 consists of several HyperTransport devices interconnected in a daisy chain. However, more complex topologies can also be implemented by using HyperTransport bridges. Bridges in HyperTransport are devices having a primary link connecting toward the processor and one or more secondary links that allow extending the topology in the opposite direction. In this way, HyperTransport trees, like the one shown in Fig. 4, can be implemented.

In addition to the use of bridges, HyperTransport defines two more features able to expand a HyperTransport system. These features are the AC mode and the HTX connector. The AC mode allows devices to be connected over longer distances than allowed by regular links, which make use of the DC mode. On the other hand, the HTX connector allows external expansion cards to be plugged to the HyperTransport link, and be presented to the rest of the system as any other HyperTransport device.

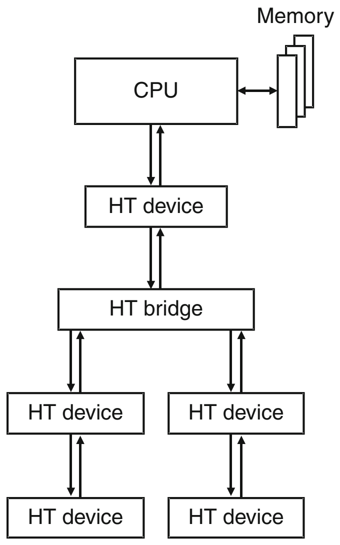
Improving the Scalability of HyperTransport

As shown above, HyperTransport offers some degree of scalability that enables the implementation of efficient



HyperTransport. Fig. 3 Buffer configuration for a HyperTransport link

topologies. Nevertheless, as HyperTransport was initially designed to replace traditional system buses, its benefits are mainly confined to interconnects within a single motherboard. Therefore, when HyperTransport topologies are to be scaled to larger sizes, in order to interconnect the processors and I/O subsystems in several motherboards (e.g., an entire cluster), HyperTransport is not able to do so because such large system sizes require routing capabilities that exceed current HyperTransport ones. More specifically, HyperTransport is not able to provide device addressability beyond 32 devices. Additionally, it does not support efficient routing in scalable network topologies. As a result, high-performance computing vendors have no choice but to complement HyperTransport with other interconnect technologies, link Infiniband in the case for general purpose clusters, or proprietary interconnects, like in the case for Cray's XT4 and XT5 supercomputers [1].



HyperTransport. Fig. 4 HyperTransport tree topology

In order to overcome the limitations of HyperTransport 3.10, an extension to it named High Node Count HyperTransport Specification was recently released. This extension supports very large system sizes, like the ones found in large data centers, at the same time that it is fully compatible with the HyperTransport specification. Additionally, the new extension adds a few bytes to current HyperTransport packets, but only in the cases where strictly required, thus minimizing the protocol overhead.

Briefly, the new extension provides an improved addressing scheme and a new control packet that allow HyperTransport devices to address any other device in large clusters. Additionally, new HyperTransport connectors and cables have been recently standardized in order to efficiently allow the deployment of the High Node Count HyperTransport Specification.

Related Entries

►Busses and Crossbars
►Data Centers
►Interconnection Networks
►PCI-Express
►PGAS (Partitioned Global Address Space) Languages

Bibliographic Notes and Further Reading

The complete description of HyperTransport 3.10 can be found in the HyperTransport I/O link specification 3.10 [5]. Additionally, readers are also encouraged to look up more information on HyperTransport in [3]. This book nicely describes the HyperTransport technology. On the other hand, a complete description of the High Node Count HyperTransport Specification can be found in [2]. Finally, many white papers and additional information are publicly available in the HyperTransport Consortium web site [4].

Bibliography

1. Cray Inc. (2009) Cray XT5 specifications. Available online at http://www.cray.com
2. Duato J, Silla F, Holden B, Miranda P, Underhill J, Cavalli M, Yalamanchili S, Brüning U (2009) Extending HyperTransport protocol for improved Scalability. In: Proceedings of the first international workshop on hypertransport research and applications, Mannheim, Germany, pp 46–53
3. Holden B, Trodden J, Anderson D (2008) HyperTransport 3.1 interconnect technology: a comprehensive guide to the 1st, 2nd, and 3rd generations. MindShare Inc, Colorado Springs, CO
4. HyperTransport Technology Consortium web site. http://www.hypertransport.org. Accessed 2010
5. HyperTransport Technology Consortium. HyperTransport I/O link specification revision 3.10. Available online at http://www.hypertransport.org. Accessed 2010