Journal of Parallel and Distributed Computing **58**, 92–108 (1999)
Article ID jpdc.1999.1541, available online at http://www.idealibrary.com on IDEAL®

SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs)

David A. Bader*, [1] and Joseph JáJá†, [2]

*Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, New Mexico 87131;
†Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742

Received July 23, 1997; accepted February 19, 1999

We describe a methodology for developing high performance programs running on clusters of SMP nodes. The SMP cluster programming methodology is based on a small prototype kernel (SIMPLE) of collective communication primitives that make efficient use of the hybrid shared and message-passing environment. We illustrate the power of our methodology by presenting experimental results for sorting integers, two-dimensional fast Fourier transforms (FFT), and constraint-satisfied searching. Our testbed is a cluster of DEC AlphaServer 2100 4/275 nodes interconnected by an ATM switch.

Key Words: cluster computing; symmetric multiprocessors (SMP); parallel algorithms; shared memory; message passing (MPI); communication primitives; experimental parallel algorithms; parallel performance. Please see http://www.umiacs.umd.edu/research/EXPAR for additional information.

1. PROBLEM OVERVIEW

With the cost of commercial off-the-shelf (COTS) high performance interconnects falling and the respective performance of microprocessors increasing, workstation clusters have become an attractive computing platform offering potentially a superior cost-effective performance [27]. Indeed, this trend highly leverages both workstation-focused technologies, including system software and networking infrastructure, for example, COTS networks (e.g., Ethernet, Myrinet, FDDI, or ATM). In recent

[1] The support by NSF CISE Postdoctoral Research Associate in Experimental Computer Science 96-25668 is gratefully acknowledged. This work was performed in part at the Institute for Advanced Computer Studies, University of Maryland, College Park. Corresponding author. E-mail: dbader@eece.unm.edu.

[2] Supported in part by NSF Grant CCR-9627210 and NSF HPCC/GCAG Grant BIR-9318183. E-mail: joseph@umiacs.umd.edu.

0743-7315/99 $30.00

years, we have seen the maturing of symmetric multiprocessor (SMP) technology (for example, hardware support for hierarchical memory management, multi-threaded operating system kernels, and optimizing compilers) and the heavy reliance upon SMPs as the work-intensive servers for client/server applications. It can be argued that (1) many future workstations will be SMPs with more than one processor and (2) SMP nodes will be the basis of workstation clusters. There are already several examples of clusters of SMPs, such as clusters of DEC AlphaServer [17], SGI Challenge/PowerChallenge [13], or Sun Ultra HPC machines, and the IBM SP system with SMP "High" nodes [18, 16]; moreover, the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) program relies on the success of computational clusters such as Option White, a 512-node IBM SP-2 with 16-way SMP nodes. With the acceptance of message-passing standards such as MPI [22], it has become easier to design portable parallel algorithms making use of these primitives. However, the focus of MPI is a standard for communicating between shared-nothing processes, and although MPI programs run on clusters of SMPs, this is not necessarily the optimal methodology for these platforms.

This paper describes a methodology for programming clusters of SMP nodes (herein referred to as Cosmos[3]) which aids in the design and implementation of efficient high performance parallel algorithms. We call this approach Simple, referring to the joining of the **SMP** and **MPI**-like message-passing paradigms and the *simple* programming approach (see Fig. 1). Note that our overall algorithmic style is similar in spirit to the one advocated by the bulk synchronous parallel (BSP) model [32].

Most popular programming methodologies for Cosmos fall into two categories [15]. The first, distributed shared memory (DSM) systems (for example, TreadMarks [2] from Rice University, Multigrain Shared Memory (MGS) [34] from MIT, and Coherent Virtual Machine (CVM) [19] from University of Maryland) provides a software layer which simulates coherent shared memory between nodes by internally using messaging to move around specific data or referenced memory pages. The second, based on message-passing primitives (for example, MPI [22]), enforces a shared-nothing paradigm between tasks, and all communication and coordination between tasks are performed through the exchange of explicit messages, even between tasks on a node with physically shared memory. For example, the models assumed in [21, 30] are that each processor in the cluster will be assigned a message-passing (MPI-level) process, with lower latency communication between processes on the same SMP node than with internode messages. However, our work differs from both of these approaches, in that we advocate a hybrid methodology which maps directly to underlying architectural aspects. As such, we combine shared memory programming on shared memory nodes with message-passing communication between these nodes.

Other recent research which utilizes SMP clusters includes KeLP and Globus. The KeLP library [11] improves data parallel language performance by providing the user with a high-level programming abstraction for block-structured scientific

[3] Cosmos ('käz-mōs) *noun* Greek *kosmos*, c. 1650: 1. an orderly harmonious systematic universe; 2. a complex orderly self-inclusive system; 3. cluster of shared memory nodes.

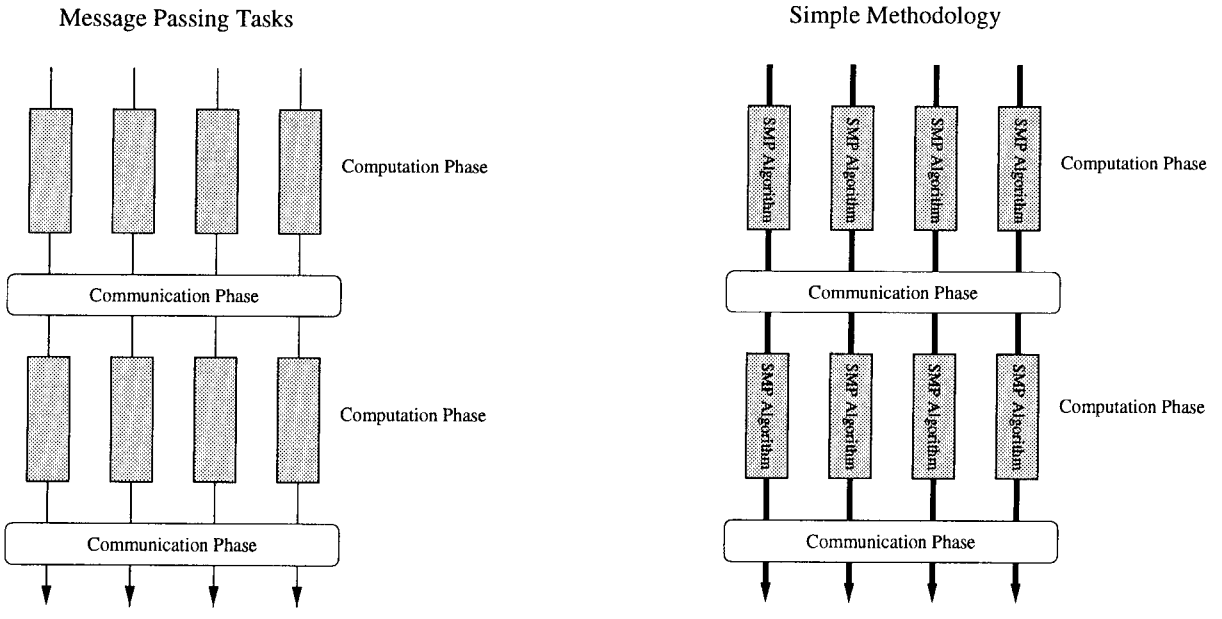


FIG. 1. On the left, we show a message-passing algorithm where each task uses sequential code during computation phases. On the right, the SIMPLE approach replaces each computation step with an optimal SMP algorithm.

calculations. KeLP contains runtime support for nonuniform domain decomposition partitioning, taking into consideration the two levels (intra- and inter-node) of memory hierarchy. The Globus toolkit [12] contains the tools necessary to interconnect heterogeneous systems (including SMP nodes) in a wide-area network, allowing message-passing and shared memory programs to take advantage of these distributed resources.

The main results of this paper are

1. A programming methodology for COSMOS which is both efficient and portable. This methodology provides a path for optimizing message-passing algorithms to clusters of SMPs.

2. A small communication kernel for clusters of SMPs which has superior performance, compared to known MPI implementations.

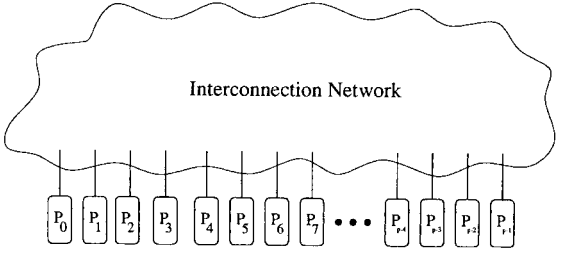
3. High performance algorithms based on our methodology for sorting integers, constraint-satisfied searching, and computing the two-dimensional FFT.

Experimental results are provided from implementations on a cluster of DEC AlphaServer 2100 4/275 nodes, each with a DEC (OC-3c) 155.52 Mbps PCI card connected to a DEC Gigaswitch/ATM switch, and using MPI (e.g., LAM 6.1 [25], MPICH 1.1.0 [14], or CHIMP 2.1.1c [1]) and POSIX threads (**pthreads**), a standard (IEEE Std. 1003.1c [28, 31]) portable threads library (e.g., DECthreads [10] or freely available pthreads implementations [29, 23]). Alternatively, the OpenMP shared memory programming API [26] may be used to provide shared memory support. Each DEC AlphaServer 2100 4/275 node is a symmetric multiprocessor with four 64-bit, dual-issue, DEC 21064A (EV4) Alpha RISC processors clocked at 275 MHz. Each Alpha chip has two separate data and instruction on-chip caches. Both on-chip caches are 16 KB, but the instruction cache is direct mapped, while the data cache is two-way set-associative. In addition, each CPU has a 4 MB backup (L2) cache [17]. All CPUs communicate via a 128-bit system bus which connects the four CPU modules to a shared memory up to 2 GB in size [17].

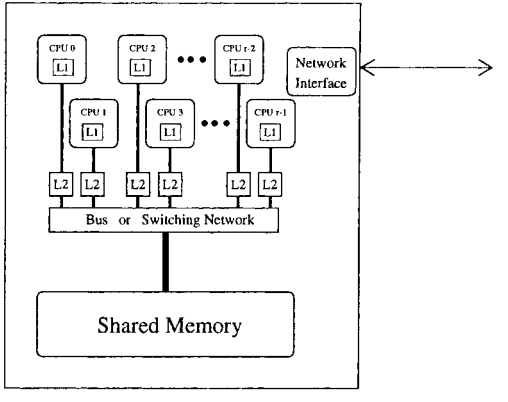
2. THE SIMPLE COMPUTATIONAL MODEL

We use a simple paradigm for designing efficient and portable parallel algorithms. First we will describe characteristics of our target parallel machine architecture. Second, we describe a set of efficient SIMPLE communication and computation primitives which are intended as user level directives.

Our architecture (shown in Fig. 2) consists of a collection of SMP nodes interconnected by a communication network 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 (L1) and a larger off-chip cache (L2), which can be tightly integrated into the memory system to provide fast memory accesses and cache coherence. Each of the r symmetric processors on an SMP node has uniform access to a shared memory and other resources such as the network interface. In practice, SMP configurations range between 2 and 36 CPUs



Cluster of processing elements where each leaf contains an SMP node.



A typical symmetric multiprocessing (SMP) node used in a cluster.

FIG. 2. SMP cluster architecture.

attached to a shared bus and main memory. In our methodology, 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 COSMOS 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, our methodology supports only node-oriented communication, meaning we restrict communication 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. We will show later that no performance loss will be incurred by this restriction.

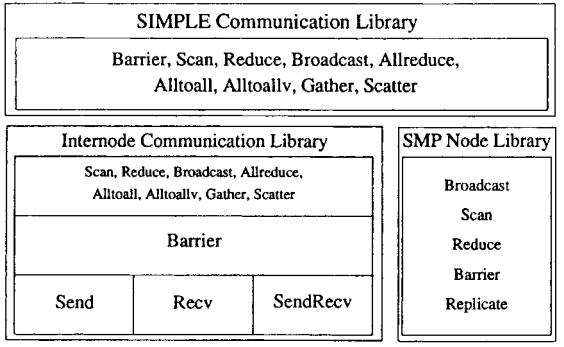
2.1. Communication Primitives

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 a SMP NODE with ICL on SMP clusters.

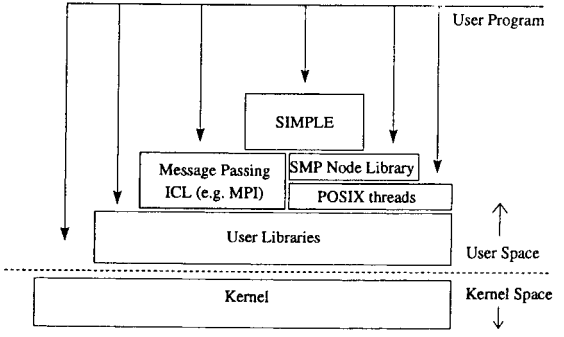
Internode communication (ICL) uses message passing across the network and can use any of the vendor-supplied or freely available thread-safe implementations of MPI. Our ICL library is based upon a reliable, application-layer `send` and `receive` primitive, as well as a `send-and-receive` primitive which 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`.

Processors on an SMP node communicate via coordinated accesses to shared memory. 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, `replicate` uniquely copies a data buffer for each processor, `scan` (`reduce`) performs a prefix (reduction) operation with a binary associative operator (for example, addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR) with one datum per thread, and `allreduce` replicates the result from `reduce`. 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.

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. Because no kernel modification is required, these libraries easily port to new platforms.



Hierarchy of SMP, message passing, and SIMPLE communication libraries



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

FIG. 3. Library design.

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 (Table 1) 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 on the node and across the machine.

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. We ran a simple experiment whereby a message is sent between two DEC AlphaServer 2100 nodes, using the Digital Gigaswitch/ATM and OC-3c adapter cards, which have a theoretical peak bandwidth rating of 155.52 Mbps. Using the ICL, we find a point-to-point communication latency of 150 μs and achieve an application-level bandwidth of 132 Mbps between a pair of nodes. For more details, see [5].

Now that the basics of the communication system and node library have been presented, we are ready to describe an example of a SIMPLE communication primitive.

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 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 $(0 \leqslant i, j \leqslant p-1)$.

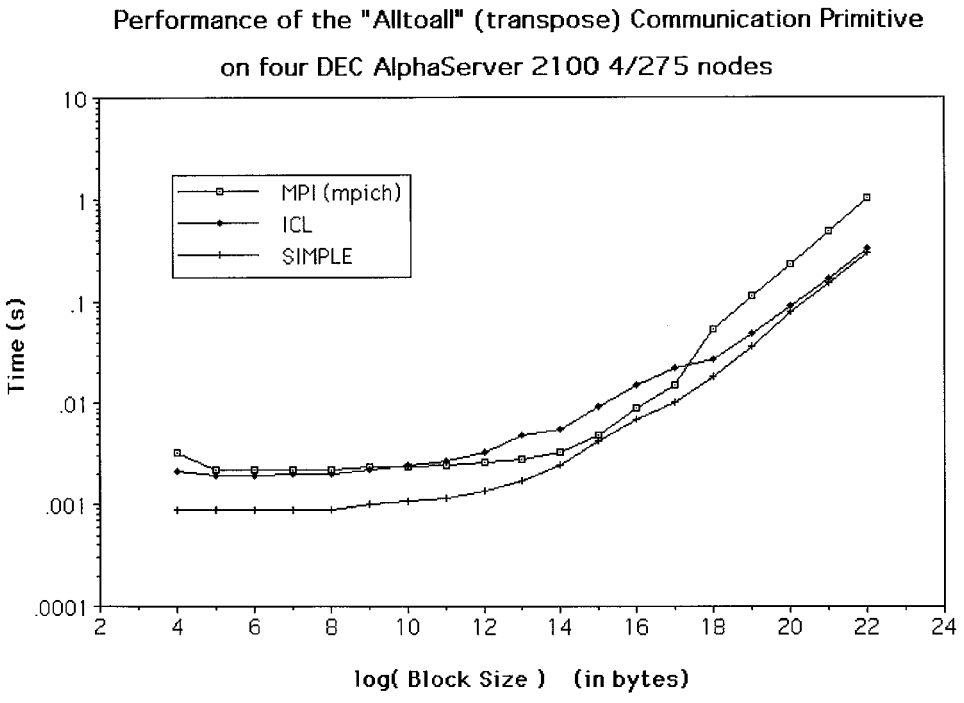
To implement this algorithm efficiently on a COSMOS [5, 6], we use multiple threads ($r \leqslant p$) per node. Trivially, one thread on node i concurrently can perform a local memory copy of the data block i, while the remaining $p-1$ internode communications are partitioned in a straightforward manner 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.

In Fig. 4, we compare the performance of three `alltoall` primitives, using the MPI, ICL, and SIMPLE communication libraries on four and eight DEC

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** $- 1$.



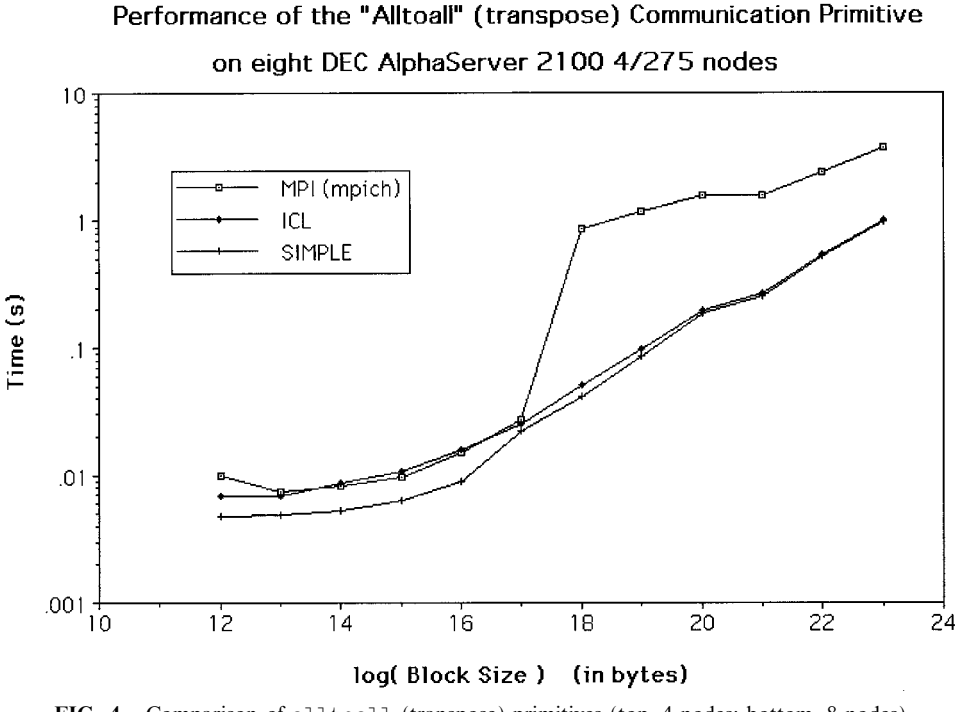


FIG. 4. Comparison of `alltoall` (transpose) primitives (top, 4 nodes; bottom, 8 nodes).

AlphaServer 2100 4/275 nodes. In all cases, the SIMPLE primitive is faster than an implementation using only message passing, such as ICL or MPI.[4] Now, with only a single network interface per node, why would one expect a performance improvement by using multiple threads? Our algorithm exploits two main sources of parallelism. The first is task level concurrency exhibited by one thread performing the local memory copy while other threads utilizing the network. The second form of parallelism is less obvious, but nonetheless an important observation. Unlike clusters of workstations where each network interface is closely coupled to a single processor's communication stream, on an SMP node, the operating system is itself capable of internal parallelism (via multithreaded kernel routines) and can more efficiently pipeline requests between the processors and the network interface.

2.2. Computation Primitives

We first discuss basic support for data parallelism, that is, "parallel do" concurrent execution of loops across processors on one or more nodes. Next we describe the control primitives which restrict (or contextualize) thread execution, for example, to some subset of threads or nodes. Last, we cover a few shared memory management directives which make it easier for the user to develop portable shared memory code by standardizing the interface for allocating and deallocating shared memory locations.

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 the 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.

Control. SIMPLE control primitives restrict which threads can participate in the context. For instance, control may be given to a single thread on each node in a cluster, all threads on one node, or a particular thread on a particular node.

Memory management. Finally, shared memory management is the third category of SIMPLE computation primitives. Two directives are provided (`node_malloc` and `node_free`) that, respectively, dynamically allocate a shared structure and release this memory back to the heap.

Thus, we have described the fundamental elements of the SIMPLE methodology and can now present a high-level approach for designing algorithms on COSMOS.

[4] The MPI `alltoall` implementation switches from a small-sized input algorithm to one for larger inputs during this experiment. Thus, the performance graph reflects a discontinuity in execution time with respect to the critical input size.

3. 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 in the COSMOS. SIMPLE also supplies the program's command line arguments.

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. Since this is a similar optimization criterion used when designing efficient message-passing algorithms [3], it is beneficial to first design an efficient message-passing algorithm on a COSMOS, and then to adapt the algorithm for the SIMPLE paradigm.

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* (for example, 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. Note that we may need to apply loop transformations to reduce data dependencies between the threads. Thread synchronization is a costly operation when implemented in software and, when possible, should be avoided.

4. SIMPLE ALGORITHM EXAMPLES

The following section demonstrates examples of SIMPLE algorithms for a variety of problems, including complex communication routines, integer sorting, scientific computing with the fast Fourier transform, and constrained searching. The reader is referred to [5, 6] for detailed algorithmic descriptions and analyses.

Permutation. As mentioned briefly in the previous section, more complex communication algorithms can be developed from the primitives described in Section 2. For example, the SIMPLE `alltoallv` communication primitive handles the case where the messages for each destination are already collected into a contiguous block of an array holding all of the messages, and the messages to be received from the other nodes likewise will appear in contiguous blocks in another array. Suppose, instead, that each node contains a set of messages, each message holding a destination tag, such that no node sends or receives more than h messages [32]. The resulting **h-relation** personalized communication [4] is a useful communication routine used in a variety of parallel algorithms. Each node determines the number of its keys to be sent to every other node, announces these counts to the destination

nodes, rearranges the input elements into a single send buffer such that all keys for the destination node j are in contiguous memory and appear before the keys for node $j+1$, routes the all-to-all communication event, and finally, unpacks each received element into the correct destination position. The permutation algorithm minimizes the number of communication steps, which is optimal on our Cosmos testbed where communication is expensive, compared with local computation.

Radix sort. Consider the problem of sorting n integers spread evenly across a cluster of p shared-memory r-way SMP nodes, where $n \geqslant p^2$. 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[5] [7] and SPLASH [33]. We present an efficient sorting algorithm based on our Simple methodology. We chose the technique of radix sort since it is well known for sequential programming, but efficient methods for solving this problem on clusters of SMPs are not. The Simple approach for radix sort is similar to our efficient message-passing algorithm [4], except when applicable, shared memory computation replaces sequential node work, and communication uses the improved Simple communication library [5, 6].

Consider the problem of sorting n integer keys in the range $[0, M-1]$ (and $M = 2^b$) 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 [20]. Let $R = 2^\rho \geqslant p$. Assume (w.l.o.g.) that the number of nodes is a power of two, say $p = 2^k$, and hence R/p is an integer $= 2^{\rho-k} \geqslant 1$. We need b/ρ passes of Counting Sort; 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.

The performance of the Simple radix sort algorithm on a Cosmos of DEC AlphaServer nodes is given in the left plate of Fig. 5. In this experiment, we use four user threads per node and vary both the problem size and the number of nodes used. Here, the Simple code shows linear speedups when using multiple nodes of a Cosmos platform.

As we claim in the introduction, software-distributed shared memory and message-passing algorithms are not optimal for Cosmos platforms. For instance, we ported an efficient SMP radix sort code into a software-distributed shared memory package called Coherent Virtual Machine (CVM, version 0.1) [19] which is an extension of the commercial TreadMarks [2] DSM implementation. The performance of this DSM radix sort is given in Fig. 5. In addition, we took an efficient message-passing code for radix sort (the reader is referred to [4, 5] for a complete analysis of the algorithm and its performance) which performs very well on an IBM SP-2.

The right plate of Fig. 5 provides a summary of the performance of the Simple methodology with DSM/CVM or MPI/MPICH on our testbed. In this experiment, we compare the performance of a Simple radix sort code using eight 4-way SMP nodes with that of both DSM/CVM and MPI/MPICH code for various cases, such as using one or multiple threads of execution per node. In all situations on the

[5] Note that the NAS IS benchmark requires that the integers be ranked and not necessarily placed in sorted order.

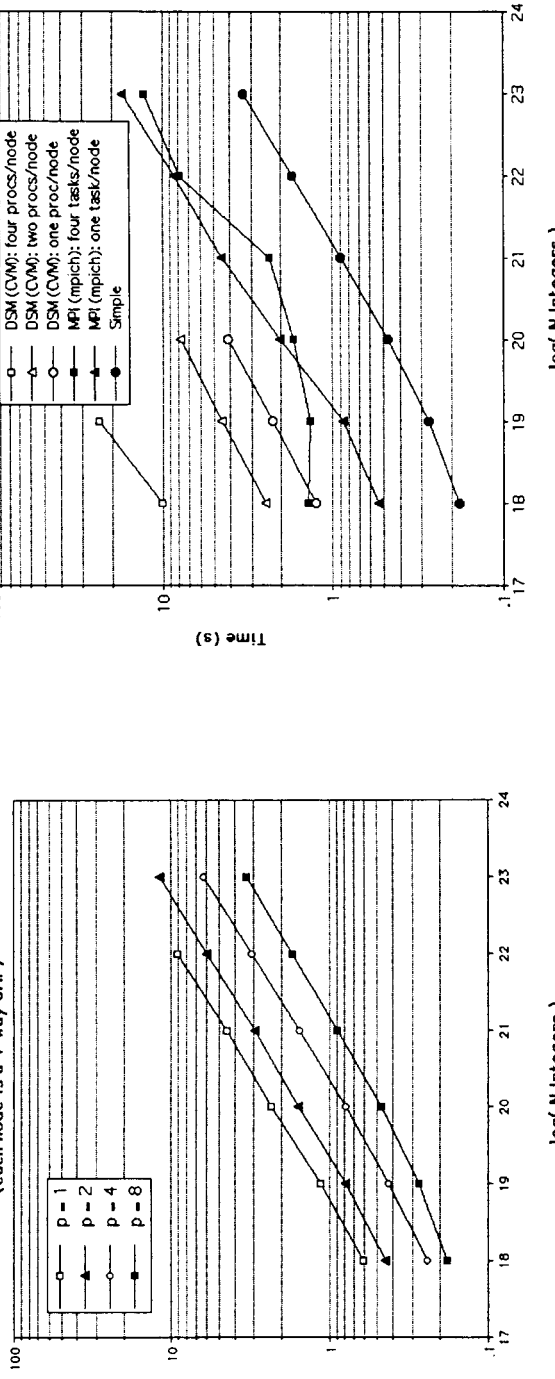


FIG. 5. Performance of SIMPLE Radix Sort on a COSMOS. Note that we tested the DSM/CVM and MPI/MPICH radix sort implementations using one to four processes per node, and the SIMPLE implementation uses $r = 4$ threads per node. Left: Execution time of SIMPLE radix sort with $r = 4$ and $p = 1, 2, 4$, and 8 nodes. Right: Comparison of DSM, MPI, and SIMPLE radix sort on a cluster of eight DEC AlphaServer 2100 4/275 nodes.

cluster of SMPs testbed, the SIMPLE algorithm substantially outperforms that of both the distributed shared memory and the message-passing implementations.

Two-dimensional fast Fourier transform. Fourier transforms are at the heart of many computations in medical image analysis, computational fluid dynamics, speech recognition, seismic analysis, image and signal processing, and detecting surface defects in manufacturing. The straightforward and well-known FFT takes a one-dimensional signal and transforms it into a one-dimensional vector of frequency components. However, when the input is a two-dimensional digital image, a corresponding two-dimensional FFT (2D-FFT) can be used similarly to transform the image into its two-dimensional frequency image. A 2D-FFT computation can be reduced to 1D-FFT's by first performing 1D-FFT's across the rows, followed by 1D-FFT's down the columns, similar to the FFT algorithms in [8, 9] which perform an all-to-all transpose of the data between two phases of local computation. In fact, a k-dimensional transform can be formed by performing k $(k-1)$-dimensional FFTs along each axis.

Assume that an $n \times n$ image is originally partitioned in strips among the p nodes such that each node originally holds n/p rows of the image.

ALGORITHM 1 (SIMPLE two-dimensional FFT algorithm).

Step 1. Each node performs n/p n-point 1D FFTs across the rows of its local image strip.

Step 2. Locally rearrange the image such that each $n/p \times n/p$ block of the image is transposed. Thus, for each block, each column of data is gathered into contiguous memory in preparation for the following step.

Step 3. Apply the `alltoall` primitive to transpose the blocks.

Step 4. Locally rearrange the data such that each node holds n/p columns of the image in contiguous memory.

Step 5. Each node performs n/p n-point 1D FFTs down the columns[6] of its local image strip.

Note that the 2D FFT algorithm above (Algorithm 1) is valid for both the message-passing and SIMPLE paradigms. The SIMPLE optimization assigns n/rp rows and columns in Steps 1 and 5, respectively, to each thread, and substitutes the SIMPLE `alltoall` primitive in Step 3. (Note that the local rearrangements in Steps 2 and 4 similarly can be optimized for shared memory threads on each node.) The SIMPLE implementation resulted in superior performance in all test cases as reported in [5].

CONSTRAINED SEARCH ALGORITHM (The n-queens problem). A classic puzzle used in benchmarking and performance analysis is the n-queens problem. Here, the objective is to report the number of ways to place n queens on an $n \times n$ chessboard such that none of the queens can attack each other. For those readers unfamiliar with the game of chess, this restricts the placement of the queens such that no two

[6] In fact, the image strip is transposed, so the 1D FFTs are performed physically across rows of memory.

queens share the same rank (or row), column, or diagonal. Since there are ${}_{n^2}C_n = n^2!/n!(n^2-n)!$ ways to place n queens on an $n \times n$ board, a brute force algorithm which checks each of these candidate solutions is infeasible. If we limit the search space to include just those candidates which have exactly one queen per rank, then we reduce the search space to n^n (or $n!$ for one queen per rank and column) possible candidates, which is still too large. Therefore, the most desirable search method aggressively eliminates sets of candidate solutions which do not satisfy the constraints. Even with the best search method, solving the n-queens problem has exponential complexity in the problem size.

Our algorithm uses a tree-based backtracking approach where queens are placed one by one on each rank until all n queens are placed. If a constraint is not met, or a solution is found, the last queen placed on the board is removed and re-placed in the next column position. This is equivalent to a depth-first search with pruning of branches where the constraints are not met. Note that we are not taking into consideration the special topological properties and symmetries of the chessboard, for example rotating known solutions by 90°, 180°, and 270°, to discover similar solutions, or reflecting solutions about the horizontal, vertical, or diagonal axes.

A parallel n-queens constraint-satisfaction search algorithm with p processors uses a distributed search tree approach as follows. First, the algorithm enumerates a set of independent search-tree seed nodes and partitions these to the processors. Suppose we generate all possible queen placements on the first k ranks of an $n \times n$ chessboard. There will be n^k of these placements, uniquely encoded into the integers from 0 to $n^k - 1$. These n^k partial placements then can be partitioned randomly and evenly among the processors, checked for validity, and used as a root node for a sequential depth-first search of the remaining $n-k$ queen positions from that starting point. This SIMPLE algorithm scales linearly with the total number of processors used and compares favorably with the standard netlib [24] "queens" sequential benchmark results for $n = 14$, 15, and 16.

REFERENCES

1. R. Alasdair, A. Bruce, J. G. Mills, and A. G. Smith, "CHIMP/MPI User Guide," Edinburgh Parallel Computing Centre, The University of Edinburgh, 1.2 edition, June 1994. [http://www.epcc.ed.ac.uk/epcc-projects/CHIMP/]
2. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer* **29**, 2 (1996), 18–28.
3. D. A. Bader, "On the Design and Analysis of Practical Parallel Algorithms for Combinatorial Problems with Applications to Image Processing," Ph.D. thesis, University of Maryland, College Park, Department of Electrical Engineering, April 1996.
4. D. A. Bader, D. R. Helman, and J. JáJá, Practical parallel algorithms for personalized communication and integer sorting, *ACM J. Exp. Algorithmics* **1**, 3 (1996), 1–42. [http://www.jea.acm.org/1996/BaderPersonalized/]
5. D. A. Bader and J. JáJá, "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs)," Technical Report CS-TR-3798 and UMIACS-TR-97-48, Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD, May 1997. [http://www.umiacs.umd.edu/research/EXPAR]
6. D. A. Bader and J. JáJá, SIMPLE: Efficient communication algorithms for clusters of symmetric multiprocesors (SMPs) with applications, in preparation, 1999.

7. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, March 1994.
8. W. P. Brown, "Parallel Computation of Atmospheric Propagation," Technical report, Maui High Performance Computing Center and Phillips Laboratory, Kihei, Maui, HI, 1995.
9. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, LogP: Towards a realistic model of parallel computation, *in* "Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993."
10. Digital Equipment Corp., "Guide to DECthreads," Maynard, MA, July 1994.
11. S. J. Fink and S. B. Baden, Runtime support for multi-tier programming of block-structured applications on SMP clusters, *in* "Lecture Notes in Computer Science: Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97), Marina del Rey, CA, Dec. 1997" (Y. Ishikawa *et al.*, Ed.), Vol. 1343, pp. 1–8, Springer-Verlag, Berlin.
12. I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit, *Internat. J. Supercomput. Appl.* **11**, 2 (1997), 115–128.
13. M. Galles and E. Williams, "Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor," Technical report, Silicon Graphics Computer Systems, Mountain View, CA, May 1994. [ftp://ftp.sgi.com/sgi/whitepaper/challenge_paper.ps.Z]
14. W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," Technical report, Argonne National Laboratory, Argonne, IL, 1996. [http://www.mcs.anl.gov/mpi/mpich/]
15. W. W. Gropp and E. L. Lusk, A taxonomy of programming models for symmetric multiprocessors and SMP clusters, *in* "Proceedings of 1995 Programming Models for Massively Parallel Computers, Berlin, Germany, October 1995," pp. 2–7.
16. C. Harris, "Node Selection for the IBM RS/6000 SP System," Version 2.1, IBM RS/6000 Division, November 1996.
17. F. M. Hayes, Design of the AlphaServer Multiprocessor Server Systems, *Digital Technical J.* **6**, 3 (Summer 1994), 8–19.
18. IBM Corporation, RS/6000 SP System, RS/6000 Division, 1997.
19. P. Keleher, "CVM: The Coherent Virtual Machine," 0.1 Ed., University of Maryland, November 1996.
20. D. E. Knuth, "The Art of Computer Programming: Sorting and Searching," Vol. 3, Addison–Wesley, Reading, MA, 1973.
21. S. S. Lumetta, A. M. Mainwaring, and D. E. Culler, Multi-Protocol Active Messages on a Cluster of SMP's, *in* "Proceedings of Supercomputing '97," San Jose, CA, November 1997.
22. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Technical report, University of Tennessee, Knoxville, TN, June 1995, Version 1.1.
23. F. Müller, A Library Implementation of POSIX Threads under UNIX, *in* "Proceedings of the 1993 Winter USENIX Conference," pp. 29–41, San Diego, CA, January 1993. [http://www.informatik.hu-berlin.de/~mueller/projects.html]
24. Netlib Repository for mathematical software, papers, and databases, University of Tennessee and Oak Ridge National Laboratory. [http://www.netlib.org/]
25. Ohio Supercomputer Center, "LAM/MPI Parallel Computing," Ohio State University, Columbus, OH, 1995. [http://www.mpi.nd.edu/lam/]
26. OpenMP Architecture Review Board, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming, October 1997." [http://www.openmp.org/]
27. G. F. Pfister, "In Search of Clusters," Prentice–Hall, Englewood Cliffs, NJ, 1995.
28. Portable Applications Standards Committee of the IEEE, "Information Technology—Portable Operating System Interface (POSIX)—Part 1. System Application Program Interface (API)," 1996-07-12 ed., 1996. [ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1]
29. C. Provenzano, "Proven Pthreads," WWW page, 1995. [http://www.mit.edu/people/proven/pthreads.html]
30. W. Saphir, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.1 Results," Report NAS-96-010, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, August 1996.

31. Sun Microsystems, Inc., "POSIX Threads," WWW page, 1995.
32. L. G. Valiant, A bridging model for parallel computation, *Comm. ACM* **33**, 8 (1990), 103–111.
33. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, *in* "Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995," pp. 24–36.
34. D. Yeung, J. Kubiatowicz, and A. Agarwal, MGS: A multigrain shared memory system, *in* "Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May 1996."