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

David A. Bader* Joseph JáJá†

Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{dbader, joseph}@umiacs.umd.edu

May 16, 1997

Abstract

We describe a methodology for developing high performance programs running on clusters of SMP nodes. Our methodology is based on a small 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.

Keywords: Cluster Computing, Symmetric Multiprocessors (SMP), ATM Networks, Parallel Algorithms, Shared Memory, message passing (MPI), Experimental Parallel Algorithms, Parallel Performance.

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 [23]. Indeed, this trend highly leverages both workstation-focused technologies including systems software and networking infrastructure, for example, COTS networks (e.g. Ethernet, Myrinet, FDDI, or ATM). In recent years, we have seen the maturing of Symmetric Multiprocessors (SMPs) technology (for example, hardware

*The support by NSF CISE Postdoctoral Research Associate in Experimental Computer Science No. 96-25668 is gratefully acknowledged.

†Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

support for hierarchical memory management, multithreaded 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 [14], SGI Challenge/PowerChallenge [11], or Sun Ultra HPC machines, and the IBM SP system with SMP "High" nodes [16, 13]. With the acceptance of message passing standards such as MPI [19], 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 processors, and although MPI programs run on clusters of SMPs, this is not necessarily the optimal methodology for these platforms.

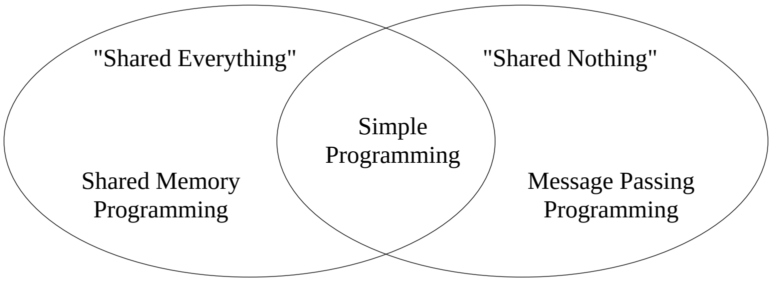


Figure 1: The SIMPLE methodology efficiently combines shared memory programming on a node with message passing between nodes.

This paper describes a methodology for programming clusters of SMP nodes (herein referred to as COSMOS [1]) which aids in the design and implementation of efficient high performance parallel algorithms. We call this model SIMPLE , referring to the joining of the **SMP** and **MPI**-like message passing paradigms and the *simple* programming approach (see Figures 1 and 2).

Programming methodologies for COSMOS fall into two categories. The first, distributed shared memory (DSM) systems (for example, TreadMarks [2] from Rice University, Multigrain Shared Memory (MGS) [30] from MIT and Coherent Virtual Machine (CVM) [17] 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 [19]), 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 model assumed in [26] is that each processor in the cluster will be assigned a message passing (MPI-level)

[1]cosmos ('käz-mōs) *noun* Greek *kosmos* c. 1650
1: an orderly harmonious systematic universe
2: a complex orderly self-inclusive system
3: **C**luster **O**f **S**hared **M**emory **N**odes

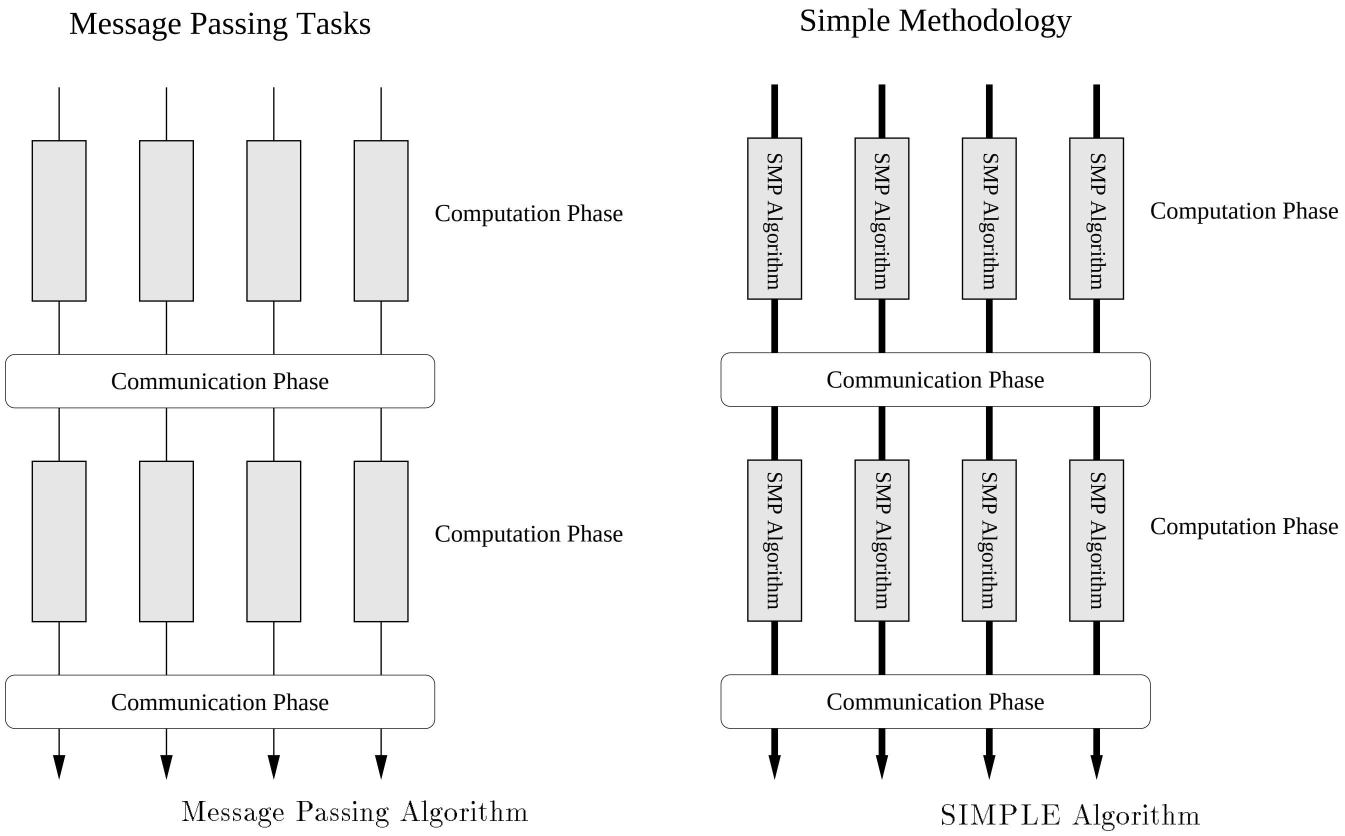


Figure 2: On the left, we show a message passing algorithm where each task uses sequential code during computation phases. On the right, the SIMPLE approach replaces each computation step with an optimal SMP algorithm.

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.

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 message passing kernel for clusters connected by ATM switches which is superior in performance when compared with the known MPI implementations.

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

The organization of this paper is as follows. Section 2 addresses our computation methodology and target parallel machine architectures. The design of algorithms for COSMOS is described in Section 3.

Our SIMPLE communication primitives are described in Sections 2.1 and 2.2, which include collective communication and computation operations as well as functions for spreading work among processors on a node, or across an entire cluster of machines. We present several examples of efficient algorithms using the SIMPLE model for design, analysis, and empirical testing. The first algorithm, given in Section 5, sorts integers using a radix-based approach. The performance of this algorithm is compared with that of an efficient MPI radix sort, highlighting the significant improvement introduced by our methodology. Section 6 presents the second algorithm, two-dimensional FFT, which is the cornerstone computation in many applications. The third algorithm, an example of constraint-satisfied searching, finds all solutions to the n-queens problem and can be found in Section 7. 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 the MPI (e.g., LAM 6.1 [22], MPICH 1.0.13 [12], or CHIMP 2.1.1c [1]) and POSIX threads (DECthreads [9] or freely available pthreads implementations [25, 20]) packages. Finally, Section 9 presents a direction for future work.

2 The SIMPLE Parallel Computation Methodology

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

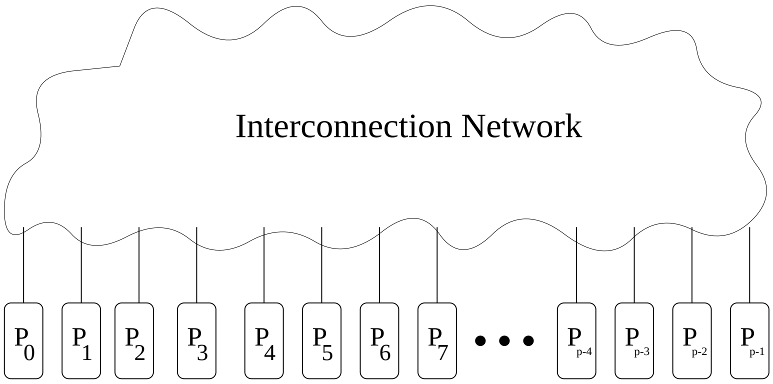


Figure 3: Cluster of processing elements

Our architecture consists of a collection of SMP nodes interconnected by a communication network (as shown in Figure 3) 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. We view a parallel algorithm as a sequence of local computation interleaved with communication steps.

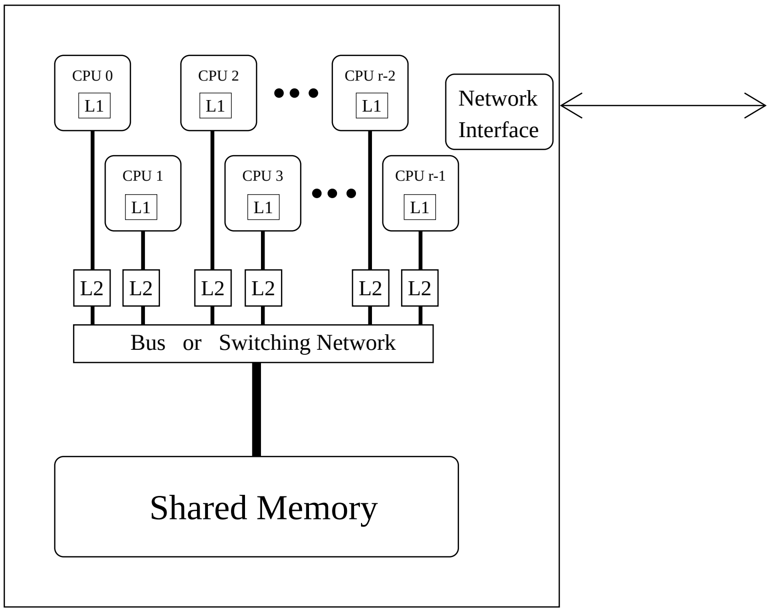


Figure 4: A typical symmetric multiprocessing (SMP) node used in a cluster. L1 is on-chip level-one cache, and L2 is off-chip level-two cache.

We use the parameter r to represent the number symmetric processors per node (see Figure 4 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. In practice, SMP configurations range between 2 and 36 CPU modules attached to a shared bus and main memory. The shared memory programming of each SMP node is based on threads which communicate via coordinated accesses to shared memory. Several primitives will be discussed in the following section which, for example, synchronize the threads at a barrier, enable one thread to broadcast data to the other threads, or calculate reductions across the threads. 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.

Next we describe the SIMPLE primitives which aid in the design of efficient and portable parallel algorithms. For ease of discussion, we separate the primitives into two categories, communication (in Section 2.1) and computation (in Section 2.2), where communication directs the flow of information between threads and computation refers to the control mechanisms among threads.

2.1 Communication Primitives

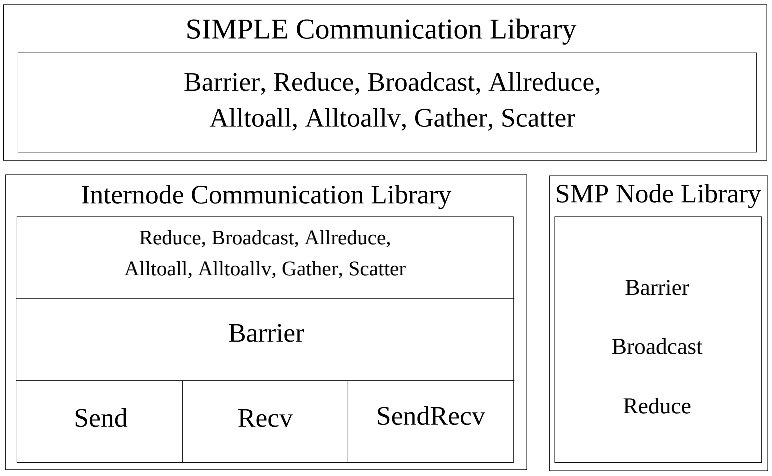


Figure 5: Hierarchy of SMP, message passing, and SIMPLE communication libraries

The communication primitives are grouped into three modules: Internode Communication Library (ICL), **SMP Node**, and SIMPLE . The ICL communication library provides a small kernel for internode communication, similar to MPI, but with less overhead than several of the freely available implementations of MPI (for example, MPICH, LAM, or CHIMP), and 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, **reduce**, **broadcast**, **allreduce**, **alltoall**, **alltoallv**, **gather,** and **scatter**. The SMP Node Library contains three important primitives for an SMP node: **barrier**, **broadcast**, and **reduce**, whereby on a single node, **barrier** synchronizes the threads, **broadcast** ensures that each thread has the most recent copy of a shared memory location, and **reduce** performs a reduction operation with a binary associative operator (for example, addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR) with one datum per thread. Finally, the SIMPLE communication library, built on top of ICL and SMP Node, includes the primitives for the SIMPLE model: **barrier**, **reduce**, **broadcast**, **allreduce**, **alltoall**, **alltoallv**, **gather**, and **scatter**. These hierarchical layers of our communication libraries are pictured in Figure 5.

The SMP Node, ICL, and SIMPLE libraries are implemented at a high-level, completely in user space (see Figure 6). Because no kernel modification is required, these libraries easily port to new platforms.

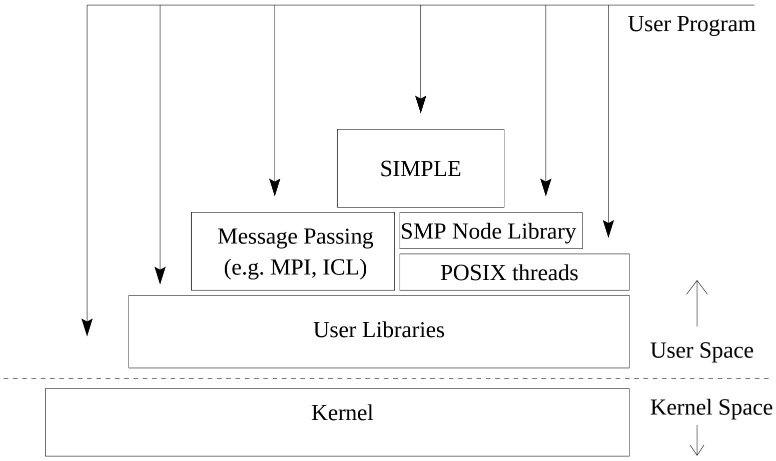


Figure 6: User code can access SIMPLE , SMP, message passing, and standard user libraries. Note that SIMPLE operates completely in user space.

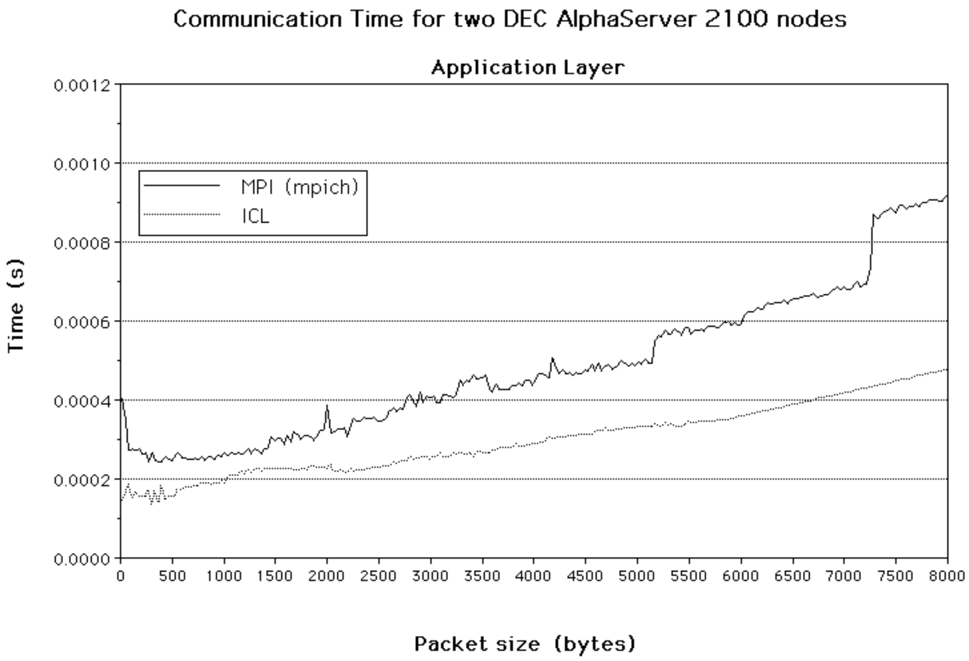
Parameter	Description
NODES $= p$	Total number of nodes in the cluster.
MYNODE	My node rank, from 0 to **NODES** $- 1$.
THREADS $= r$	Total number of threads on my node.
MYTHREAD	The rank of my thread on this node, from 0 to **THREADS** $- 1$.
TID	Total number of threads in the cluster.
ID	My thread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

Table I: The local context parameters available to each SIMPLE thread.

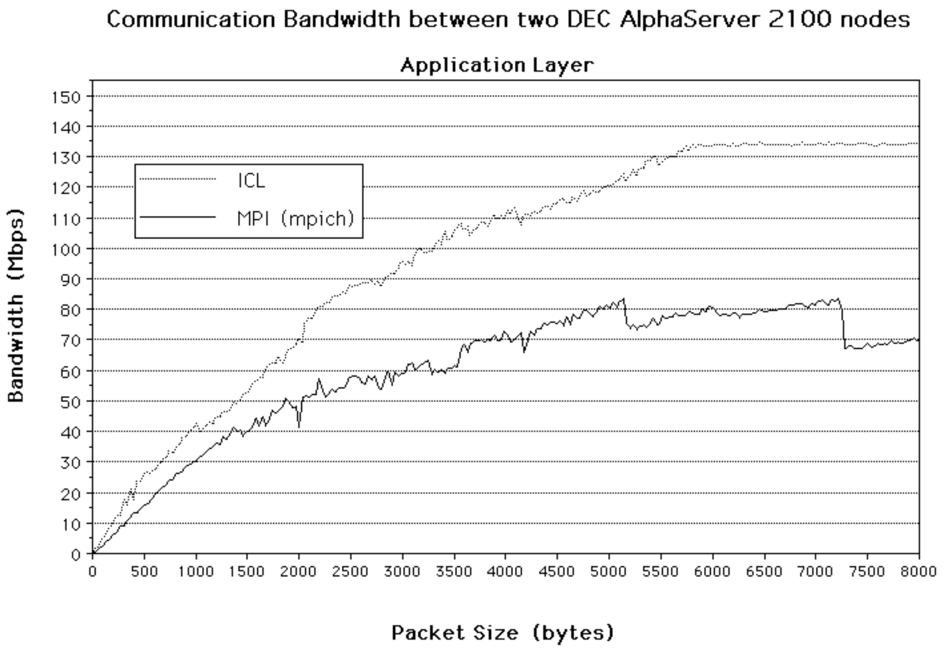
As mentioned previously, the number of threads per node can vary, along with machine size. Thus, each thread has a small set of context information which holds such parameters as the number of threads on the given node, the number of nodes in the machine, the rank of that node in the machine, and the rank of the thread both 1) on the node and 2) across the machine. Table I 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 MPI libraries offer a more robust communication suite than our ICL Library, at a significant cost. However, the lower-level ICL and **SMP Node** primitives can be replaced by vendor-supplied MPI and SMP primitives. We ran a simple experiment whereby a message is sent between two nodes and plotted the results. Figure 7 shows the communication time and respective bandwidth for sending a message between two DEC AlphaServer 2100 nodes, using the Digital Gigaswitch/ATM and OC-3c adapter cards, which have a peak bandwidth rating of 155.52 Mbps.

The results, summarized in Table II, reflect the latency and bandwidth characteristics of point-to-point messages between a pair of DEC AlphaServer 2100 nodes, using the Internode Communication Library (ICL) and the best MPI implementation (MPICH). The theoretical raw peak bandwidth is



Time



Bandwidth

Figure 7: Internode Communication Performance

Communication Library	Latency	Bandwidth
ICL	$170\mu s$	132 Mbps
MPI (MPICH)	$400\mu s$	75 Mbps

Table II: The latency and bandwidth characteristics of point-to-point messages between a pair of DEC AlphaServer 2100 nodes, with ATM OC-3c adapters, using both the Internode Communication Library (ICL) and MPI (MPICH).

155.52 Mbps, and our application level measurement finds the ICL library achieving 132 Mbps, while MPI/MPICH only reaches about half of that. In addition, our latency measurements are less than half of that incurred using MPI/MPICH. The SIMPLE library can use either MPI or ICL for passing messages between the nodes, taking into account the following two important considerations. First, ICL is not a replacement for MPI. The ICL library offers only a small subset of the functionality available in MPI, for example, ICL uses only a single communication group, specializes the implementation for an ATM network instead of implementing communication on an abstract channel device, restricts the number of outstanding communication events, and provides less status information and no additional debugging hooks. Second, ICL provides both weak support for multithreading where the user is responsible for maintaining mutually exclusive use of communication channels via implicit algorithmic design or explicit locks, and strong support where internal locking mechanisms automatically protect the user from corrupting the communication layer. However, the MPI implementation must be thread-safe. Thus, the ICL communication library achieves the higher performance for two main reasons, first latency is reduced because, by purpose, ICL is not as generalized as MPI, and second, bandwidth is increased in ICL by optimizing the network parameters for an ATM switch.

2.1.1 Implementation of the SMP Node Library

As Figure 6 shows, the **SMP Node** library can be implemented on top of a portable threads layer, such as POSIX threads (pthreads), or if available, via possibly faster native primitives. Our **SMP Node** library is based upon pthreads, and thus, is portable to POSIX standard platforms. The three SMP Node primitives which we require for SIMPLE are **reduce**, **barrier**, and **broadcast**. For example, if the number of threads is small, each thread entering a **reduce** primitive first acquires a lock, stores the reduction of its element with the shared element, and increases the counter of waiting threads. If it is not the last to enter, the thread releases the lock and blocks waiting for a condition. If in fact the thread is the last to enter, it resets the operation and uses a condition broadcast to wake up the other threads. Finally, all threads return the result. For a larger number of threads, the **reduce** primitive can be implemented with an efficient parallel k-ary tree for a suitable value of k.

The pthreads standard requires primitives for synchronization with condition variables and mutual exclusion locks, but does not include a primitive for barrier synchronization. The **barrier** primitive can be implemented similarly to **reduce**, since all threads must enter before each thread can continue. Since a side effect of the pthreads locking mechanism is an SMP memory coherence barrier, the thread with data to **broadcast** writes this information in a shared memory location, and then all threads enter a **barrier**. Afterwards, each thread reads this shared memory location which is guaranteed to be consistent.

Now that the basics of the communication system and node library have been presented, we are ready to describe some of the SIMPLE communication primitives.

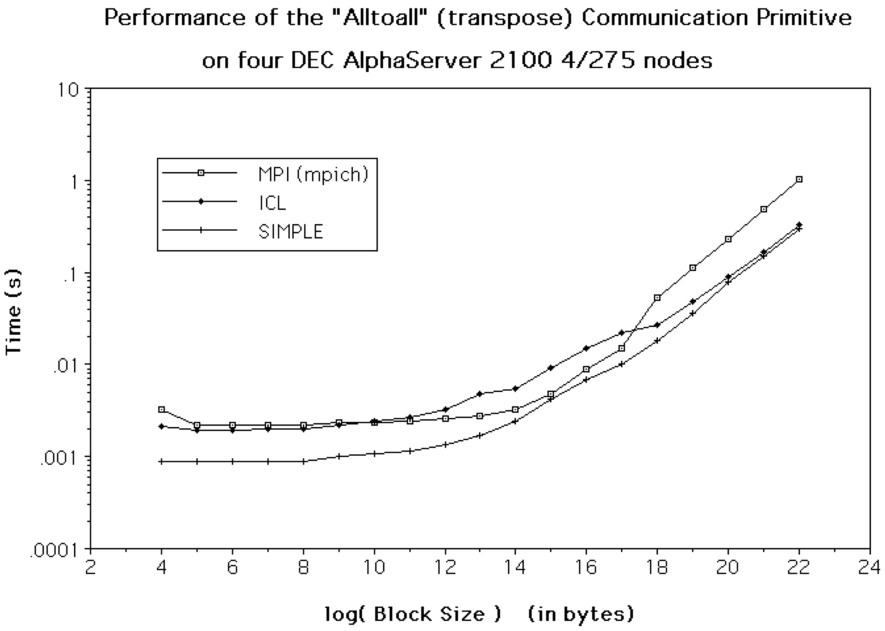
2.1.2 The Alltoall primitive

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 n element sending buffer, where p divides n, the **Alltoall** operation consists of each node i sending its j^{th} block of $\frac{n}{p}$ data elements to node j, where node j stores the data from i in the i^{th} 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 "$var + (\frac{n}{p} * i)$", and src and dst point to the source and destination arrays, respectively.

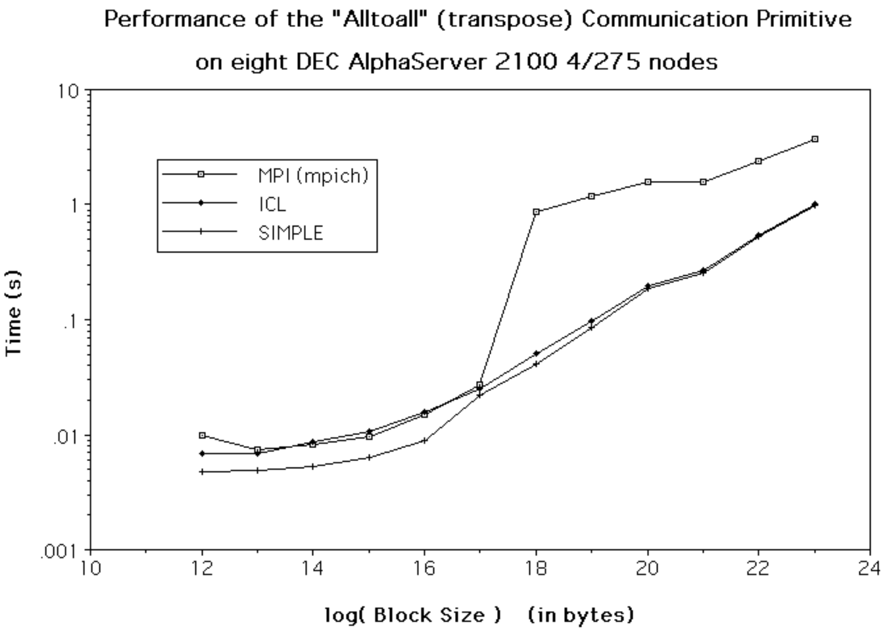
- **Step (1): Copy** the appropriate $\frac{n}{p}$ elements from src_{MYNODE} to dst_{MYNODE}.
- **Step (2): For** $i = 1$ to **NODES** $- 1$ do
  - **A) Set** $k = \textbf{MYNODE} \oplus i$;
  - **B) Send** $\frac{n}{p}$ elements from src_k to node k, and
    Receive $\frac{n}{p}$ elements from node k to dst_k.

To implement this algorithm, we use multiple threads per node. The local memory copy in **Step (1)** 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 in **Step (2)** 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 Figure 8, we compare the performance of three **Alltoall** primitives, using the MPI, ICL , and SIMPLE communication libraries on four and eight DEC AlphaServer 2100 4/275 nodes. In all cases, the SIMPLE primitive is the fastest, typically by a factor or two or three over MPI. 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 concurrently 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.



Four Nodes



Eight Nodes

Figure 8: Comparison of Alltoall (Transpose) Primitives

Unlike clusters of workstations where each network interface is closed coupled to a single processor's communication stream, on an SMP node, the operating system is itself capable of internal parallelism (via multi-threaded kernel routines) and can more efficiently pipeline requests between the processors and the network interface.

2.2 Computation Primitives

In the previous section, we provided an overview of our communication library. Next we will explore the set of user level directives, called SIMPLE computation primitives, which 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 discussed first. Next we describe the control primitives which restrict (or contextualize) thread execution, for example, to some subset of threads or nodes. Lastly, 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.

2.2.1 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 a COSMOS . 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.

2.2.2 Control

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

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	

Table III: Subset of SIMPLE Control Primitives.

Table III 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.

2.2.3 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, and
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.

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

3 SIMPLE **Algorithmic Design**

In this section we describe the SIMPLE programming model as seen by the user and the runtime support for executing SIMPLE code.

3.1 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 I. 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 . 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 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.

3.2 Runtime Support

When a SIMPLE algorithm first begins its execution on a COSMOS , 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 Section 2 have runtime initializations which take place as follows.

The runtime startup 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[2]. It is assumed that the r CPUs concurrently execute the r threads. The thread flow of an example SIMPLE algorithm is shown in Figure 9. As mentioned previously, our methodology supports 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.

[2] 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

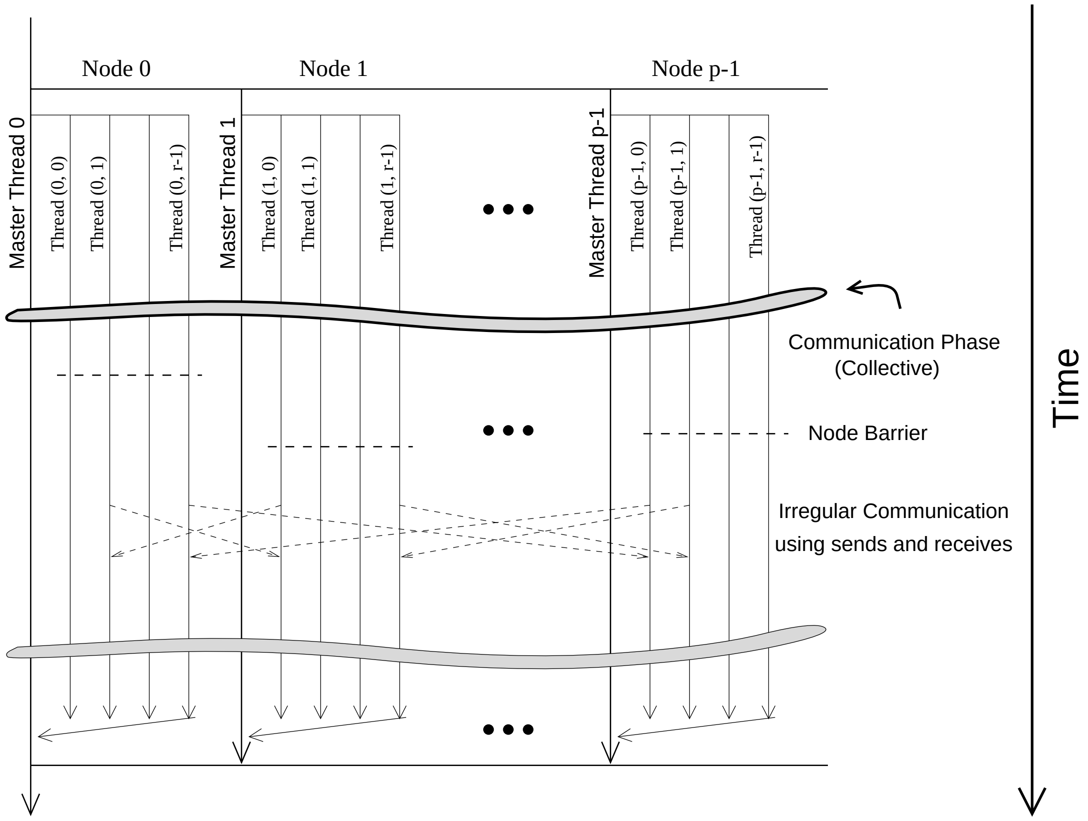


Figure 9: Example of a SIMPLE algorithm flow of master and compute-based user threads. Note that the only responsibility of each master thread is only to launch and later join user threads, but never to participate in computation or communication.

Our model is simply implemented using a portable thread package called POSIX threads (**pthreads**), which is a standard (IEEE Std. 1003.1c), supplied with POSIX 1.c ([24, 27]). Note that **pthreads** are also available in the "standard" Distributed Computing Environment (DCE) used in operating systems such as OSF [10] and AIX [15].

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 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 Example: SIMPLE 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 [28]. The resulting h-**relation** personalized communication [5] 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. A description of the algorithm is as follows.

- **Step (1):** For each node i, count the number of keys labeled with destination node j, for $(0 \le j \le p-1)$. 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 (**sendCount**) for the node.
- **Step (2):** Using **sendCount** and the arrays generated in **Step (1A)**, rearrange the input elements into a single **send** buffer such that all keys with destination node j are in contiguous memory and appear before keys with destination $j+1$. On each node, each of r threads place $\frac{1}{r}$ of the elements concurrently.
- **Step (3):** Apply the SIMPLE **Alltoall** primitive to the **sendCount** array using the block size 1. Hence, at the end of this step, each node will know the number of keys it will receive from every other node (**recvCount**).

- **Step (4):** Route the all-to-all communication event (with the SIMPLE **Alltoallv** communication primitive) using the **send**, **sendCount**, and **recvCount** arrays.
- **Step (5):** Each node unpacks its received elements and places each in the correct array position. Since this is a permutation routing, no collisions will occur in the final array, and r threads can each unpack $\frac{1}{r}$ of the array concurrently into shared memory.

This algorithm relies on efficient implementations of the **Alltoall** and **Alltoallv** primitives and assumes that the number of messages exchanged between each pair of nodes is fairly balanced. However, if significant imbalance exists, an alternative algorithm might replace the one-phase data routing in **Step (4)** with a two-phase routing approach using balanced **Alltoall** primitives in each phase (see [5]). Similarly, other complex communication algorithms can be developed using the SIMPLE methodology. The above permutation algorithm minimizes the number of communication steps, which is optimal on our COSMOS testbed where communication is expensive compared with local computation. Next, we show an example of an algorithm for sorting which makes use of a special case of the h-relation personalized communication, where the number of messages to be sent and received are the same.

5 Radixsort

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[3] [6] and SPLASH [29]. 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 [5], except when applicable, shared memory computation replaces sequential node work, and communication uses the improved SIMPLE communication library.

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 [18]. 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 $= 2^{\rho-k} \geq 1$.

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

5.1 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, we can move each key into its correct position using a permutation ($\frac{n}{p}$-relation) routing [4, 5], 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.

We present an overview of the original message passing Counting Sort algorithm and follow this with the adaptations to the algorithm using our SIMPLE methodology. In a practical integer sorting problem, we expect $R \approx \frac{n}{r^2 p}$. The pseudocode for our Counting Sort algorithm uses six major steps and can be described as follows.

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

We can adapt this message passing algorithm to our SIMPLE methodology with the following changes. 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.

5.2 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 we need $\frac{b}{\rho}$ 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. Radix Sort easily can be adapted for clusters of SMPs by using the SIMPLE Counting Sort routine.

5.3 Performance

We now provide empirical performance results for the Radix Sort algorithm on various platforms. We first graph the performance of the SIMPLE Radix Sort on a cluster of SMPs and show that indeed, our implementation is efficient. Next, we show results of a good MPI Radix Sort on an IBM SP-2, and compare this code with that of a shared memory sort on a single SMP node. Finally, we compare the SIMPLE Radix Sort with that of DSM and MPI Radix Sorts on a cluster of SMPs.

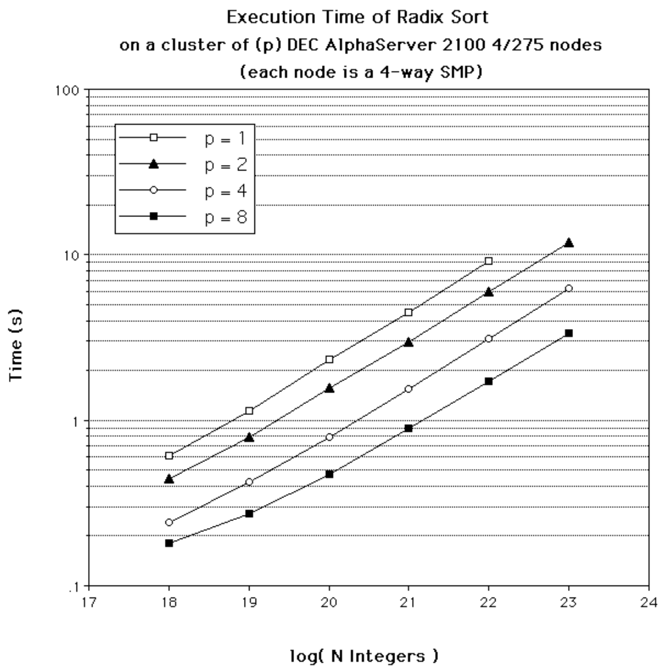
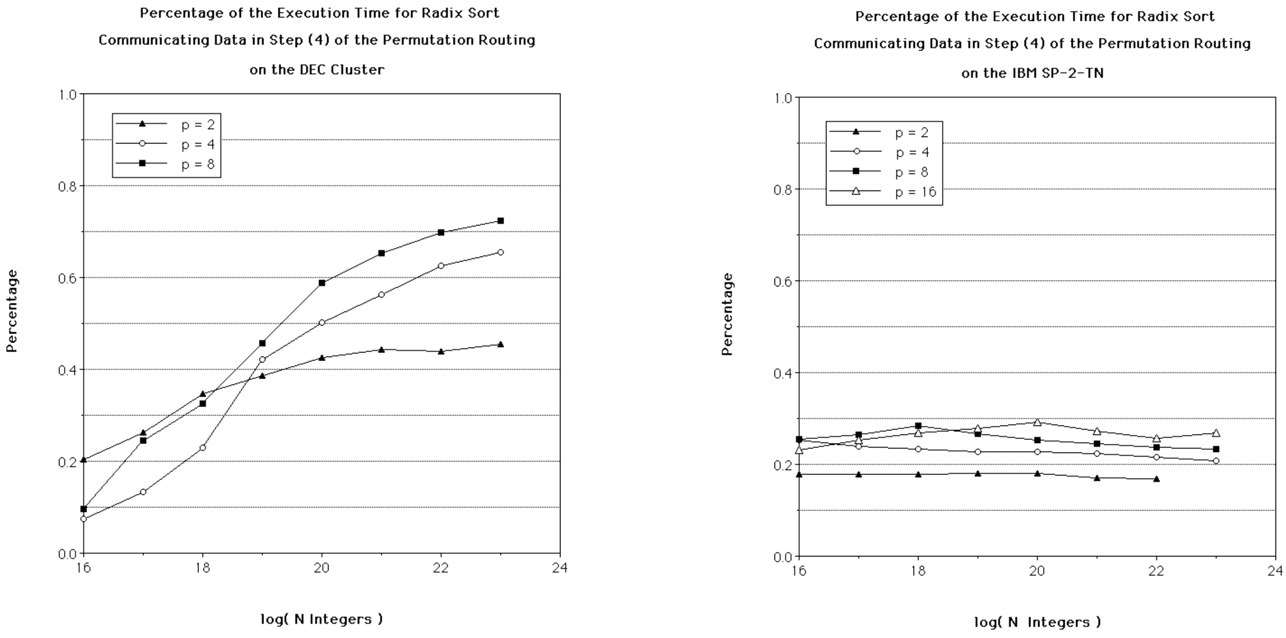


Figure 10: Execution Time of SIMPLE Radix Sort with $r = 4$ and $p = 1, 2, 4$, and 8 nodes.

The performance of the SIMPLE Radix Sort algorithm on a COSMOS of DEC AlphaServer nodes is given in Figure 10. 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.



DEC AlphaServer 2100 4/275 IBM SP-2-TN

Figure 11: Percentage of execution time of radix sort spent in the Alltoallv communication primitive used in **Step (4)** of the permutation routing on clusters of DEC and IBM nodes.

In Figure 11 we have plotted the percentage of the running time of radix sort spent performing the **Alltoallv** communication primitive used in **Step (4)** of the permutation algorithm for various IBM and DEC cluster sizes. After each key is ranked during the Counting Sort, this step sends each key to its destination. For moderately sized inputs on the DEC cluster, roughly a third of the execution time is spent in this communication step, and for larger problems, more than half the time is spent in this step. In comparison, for most inputs, the IBM SP-2-TN spends less than 30 percent of its execution time for in the corresponding step. These performance graphs indicate that radix sort is largely communication bound on the DEC Cluster, while computation bound on the IBM SP-2-TN. These results are expected, as the IBM SP-2 has a faster network but less processing power on each node than the DEC cluster.

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) [17] which is an extension of the commercial TreadMarks [2] DSM implementation. The performance of this DSM radix sort is given in Figure 14. In addition, we took an efficient message passing code for radix sort (the reader is referred to [5] for a complete analysis of the algorithm and its performance) whose performance on an IBM SP-2 is shown in Figure 12. The IBM SP-2 contains uniprocessor nodes interconnected by a fast switch. On this platform, the message passing algorithm

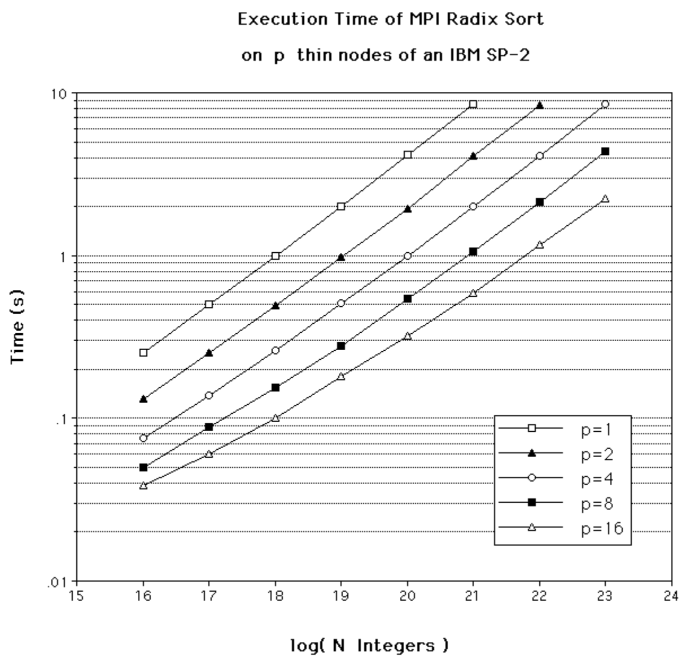


Figure 12: Performance of MPI Radix Sort on an IBM SP-2-TN with $p = 1, 2, 4, 8$, and 16 thin nodes.

performs very well. That is, for a fixed machine size, when the problem size is halved, the performance roughly is cut in half as well. In addition, for a fixed problem size, when twice as many processors are used to solve a given sorting problem, as expected the time is again halved.

An analysis of the difference in raw performance between the IBM SP-2 and the DEC cluster shows the following. When computation dominates, the DEC platform is faster in raw execution time, however, as communication increases, the imbalance of communication bandwidth to computation speed on the DEC cluster becomes more pronounced, and the IBM SP-2 is the faster platform. For example, consider the problem of sorting one million keys. A single node of the DEC AlphaServer cluster sorts these keys in approximately 2.3 seconds, whereas one node of an IBM SP-2-TN requires more than four seconds. However, when $p = 8$ nodes, both the DEC cluster and the SP-2 require roughly a half a second, even though the DEC cluster is using four times as many processors.

In Figure 13 we plot the execution of the MPI radix sort code on a single DEC AlphaServer 2100 4/275 (4-way SMP) node using one, two, and four threads of execution. For large inputs, notice that the performance improves slightly when more threads are used, but still there is no great difference when using multiple threads on a single node. In this same figure, following the SIMPLE methodology, we plot the performance of a shared memory radix sort of the same input on this 4-way SMP node. In addition to being almost an order of magnitude faster, unlike the message passing code, the SIMPLE algorithm shows significant speedups when using multiple threads.

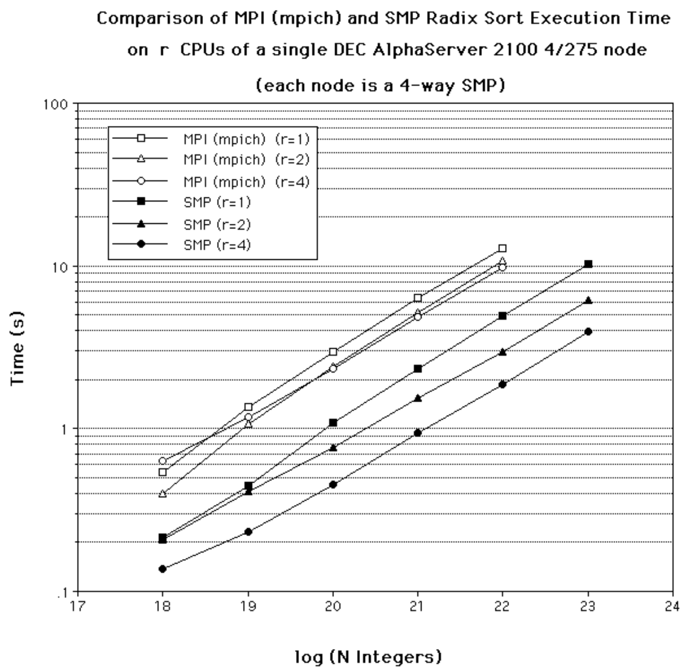
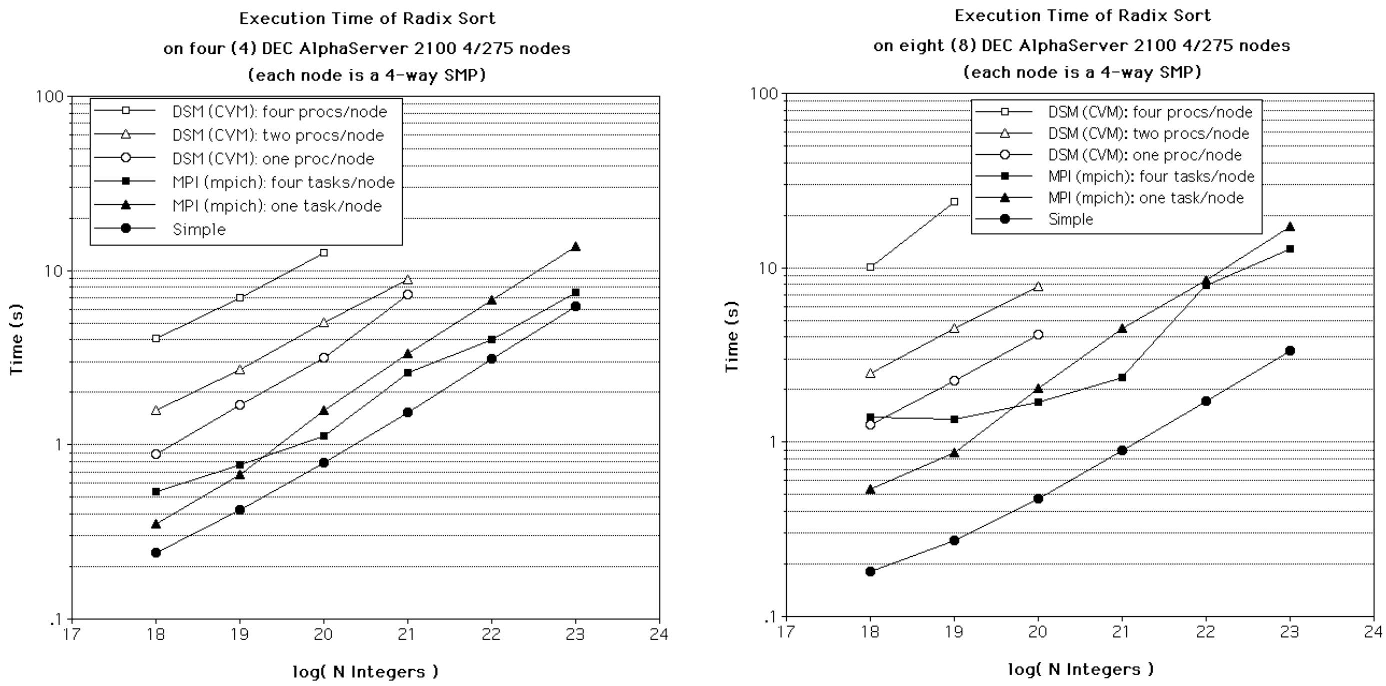


Figure 13: Comparison of MPI (MPICH) and SIMPLE Radix Sort with $r = 1, 2$, and 4 with $p = 1$ node.



Four Nodes

Eight Nodes

Figure 14: Comparison of DSM, MPI, and SIMPLE Radix Sort on a cluster of DEC AlphaServer 2100 4/275 nodes. Note that we tested the DSM/CVM radix sort implementation using one to four processes per node, and the MPI/MPICH implementation using both one and four MPI tasks per node. The SIMPLE implementation uses $r = 4$ threads per node, and $p = 4$ and $p = 8$ nodes on the left and right, respectively.

Figure 14 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 both four and 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 cluster of SMPs testbed, the SIMPLE algorithm substantially outperforms that of both the distributed shared memory and the message passing implementations.

6 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 [7, 8] which performs 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.

In Figure 15, we illustrate the major steps of the two-dimensional FFT algorithm. Assume that an $n \times n$ image is originally partitioned in strips among the p nodes such that each node originally holds $\frac{n}{p}$ rows of the image.

- **Step (1):** Each node performs $\frac{n}{p}$ n-point 1-D FFTs across the rows of its local image strip.
- **Step (2):** Locally rearrange the image such that each $\frac{n}{p} \times \frac{n}{p}$ 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 $\frac{n}{p}$ columns of the image in contiguous memory.
- **Step (5):** Each node performs $\frac{n}{p}$ n-point 1-D FFTs down the columns[4] of its local image strip.

Note that the 2-D FFT algorithm above is valid for both the message passing and SIMPLE paradigms. The SIMPLE optimization assigns $\frac{n}{rp}$ rows and columns in **Steps (1)** and **(5)**, respectively, to each thread, and substitutes the SIMPLE **Alltoall** primitive in **Step (3)**. (Note that the

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

local rearrangements in **Steps (2)** and **(4)** similarly can be optimized for shared memory threads on each node.)

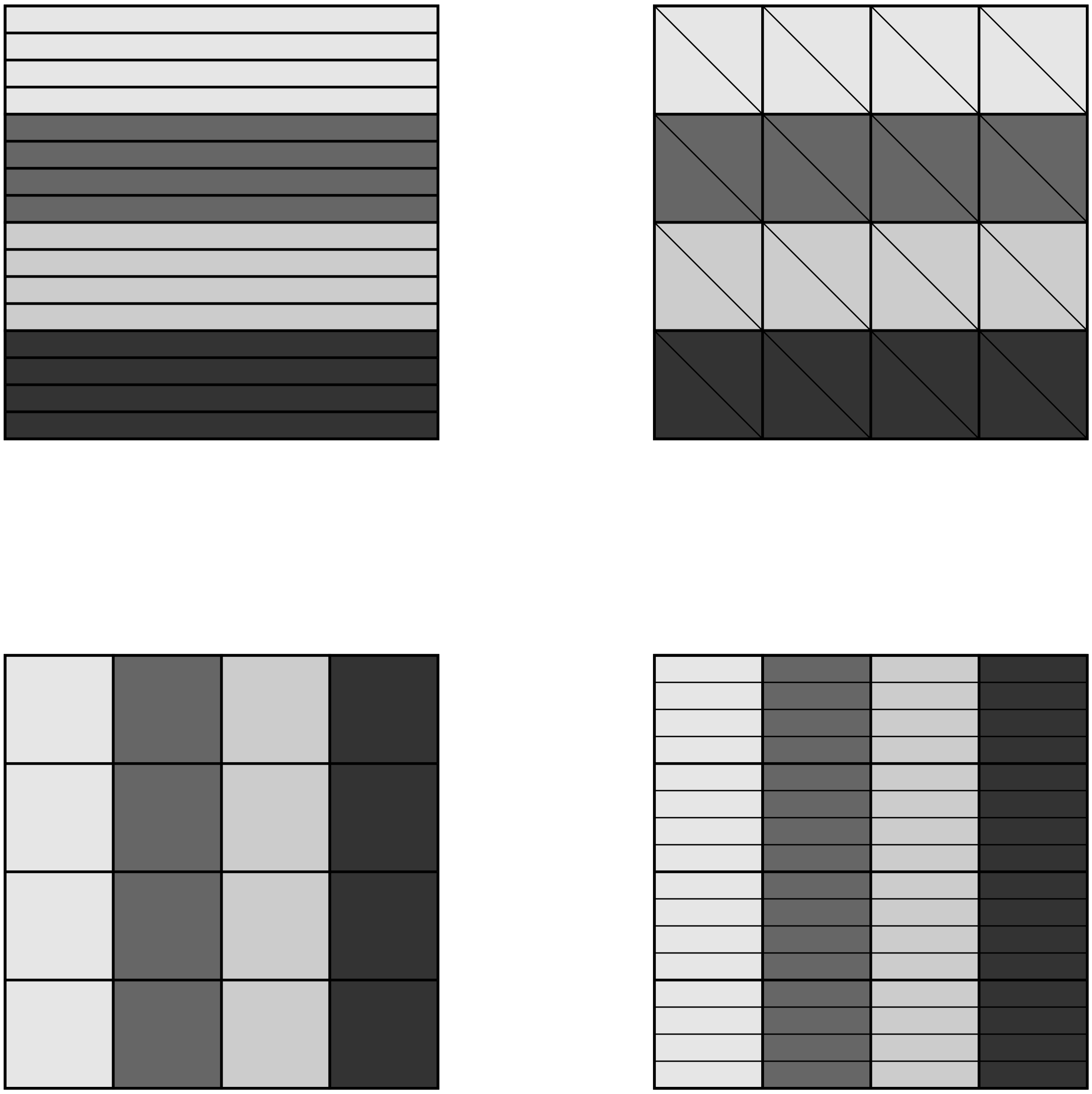


Figure 15: The Two-dimensional FFT Algorithm with blocks of rows initially distributed across the nodes: (top left) performs local one-dimensional FFTs across the rows, (top right) locally rearranges data, (bottom left) transposes the image such that each node holds a block of columns, and (bottom right) performs local one-dimensional FFTs across the columns.

We begin with an efficient message passing algorithm for the FFT. The one-dimensional FFT used in the first and last steps is a benchmark kernel from netlib [21]. As shown in Figure 16, the message passing implementation performs very well on the IBM SP-2. When we fix a problem size and double the number of processors, the execution time scales appropriately. Also, when the image size is increased four-fold (say, from 512×512 to 1024×1024 pixels), on a given number of processors, again as expected, the execution time follows the predicted complexity of the FFT algorithm.

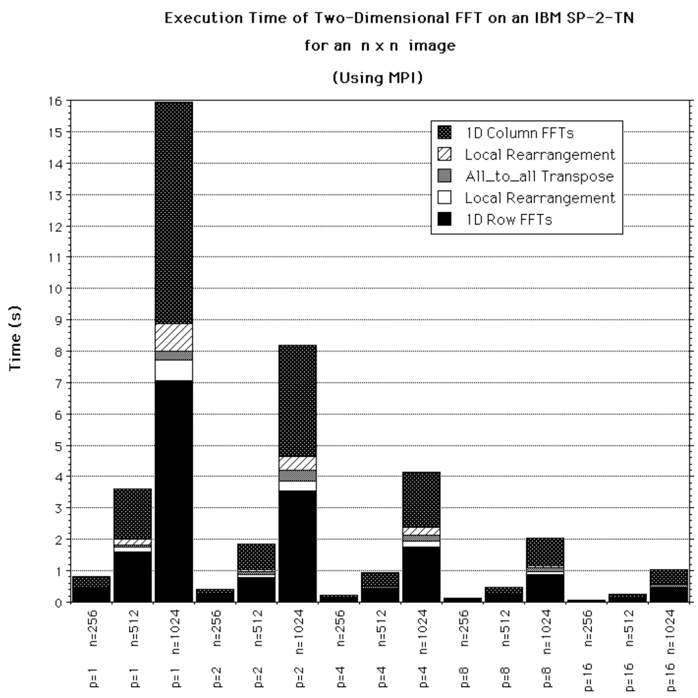
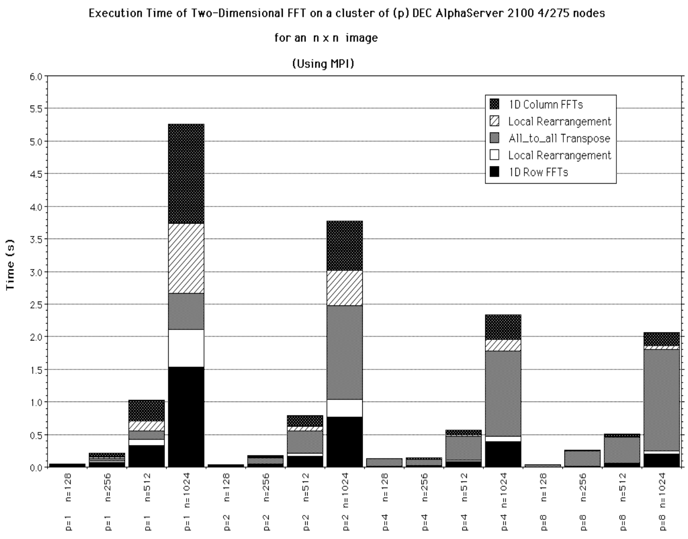
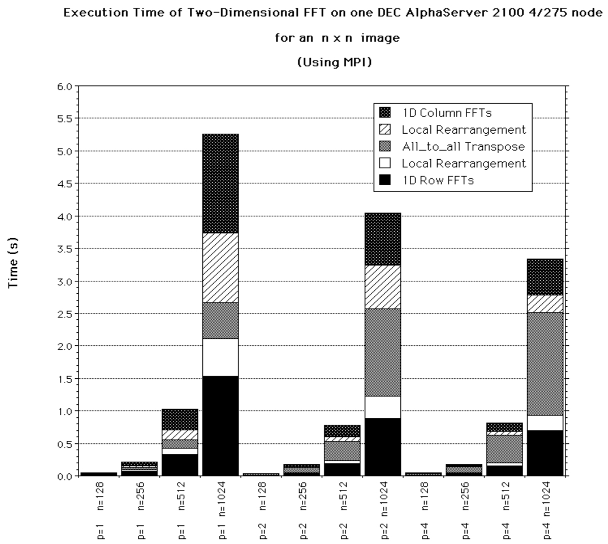


Figure 16: MPI Code for Two-dimensional FFT on an IBM SP-2-TN

Without any modifications, we ran the message passing code on both a cluster of DEC AlphaServer 2100 4/275 nodes (with only one task per node) and using message passing solely on a single node (see the left and right plates of Figure 17, respectively). For a fixed image size, the performance does not scale well with four more more nodes. In addition, the code running on one, two, and four, processors of a single node shows very little gain by using more than a single CPU per node. Compare these results with the SIMPLE execution times presented in Figure 18 for a variety of configurations (from one to eight nodes and from one to four CPUs per node) and image sizes (128×128 to 1024×1024 pixels). For instance, on a 1024×1024 pixel image, using just a single node and four tasks, the message passing implementation takes approximately 3.3 seconds, while the SIMPLE approach is about a second faster, or equivalently, two-thirds the execution time. We see an improvement for using multiple CPUs on a node, even at our largest available machine configuration of eight nodes.



cluster



node

Figure 17: MPI Code for Two-Dimensional FFT. On the left, we show the performance on a cluster of DEC AlphaServer nodes. On the right, multiple processors on a single DEC AlphaServer 2100 4/275 are used.

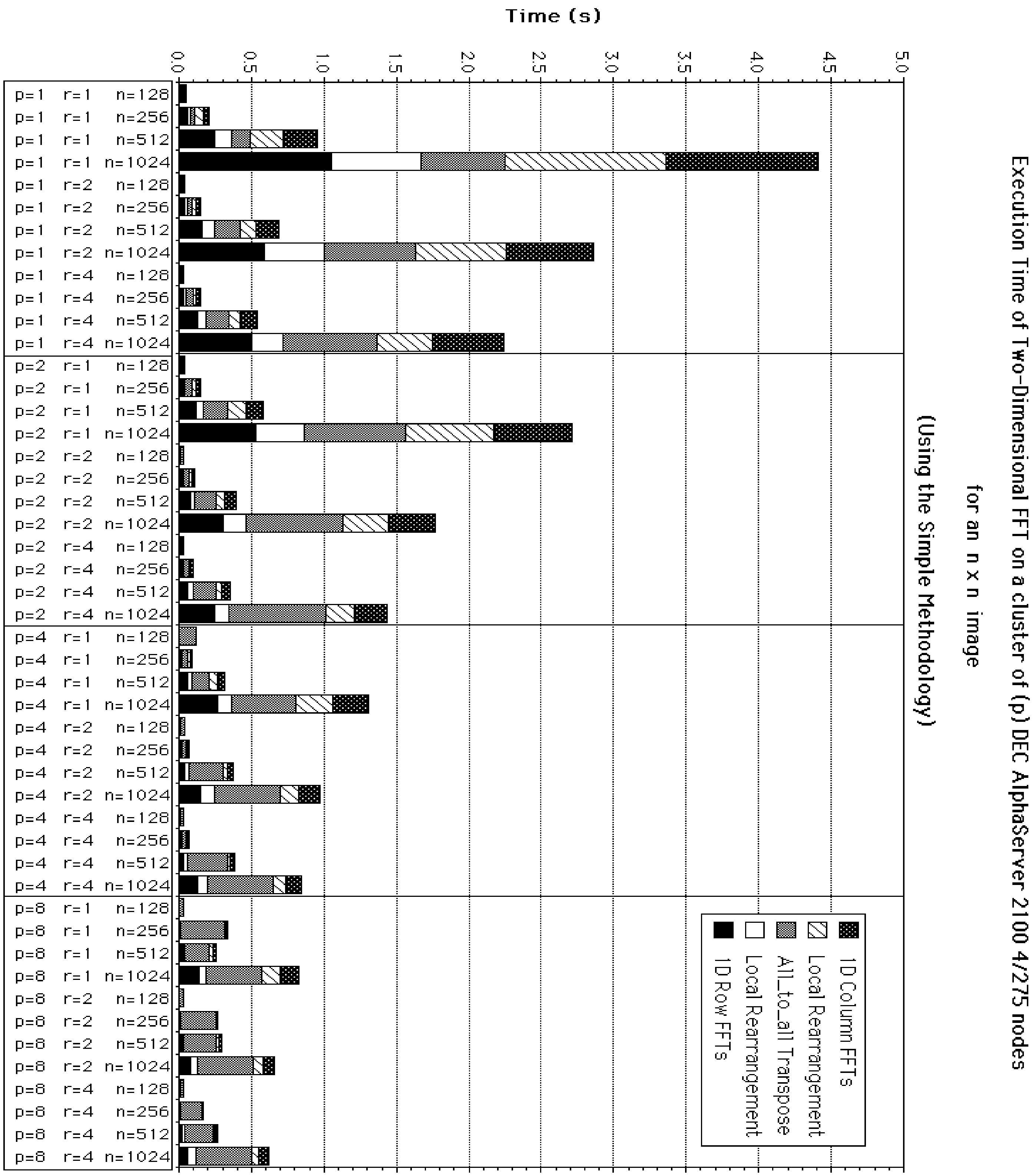


Figure 18: Two-dimensional FFT on a cluster of DEC AlphaServer 2100 nodes using the SIMPLE methodology

7 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 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 queens share the same rank (or row), column, or diagonal. Since there are ${}_{n^2}C_n = \frac{n^2!}{n!(n^2-n)!}$ 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 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.

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.

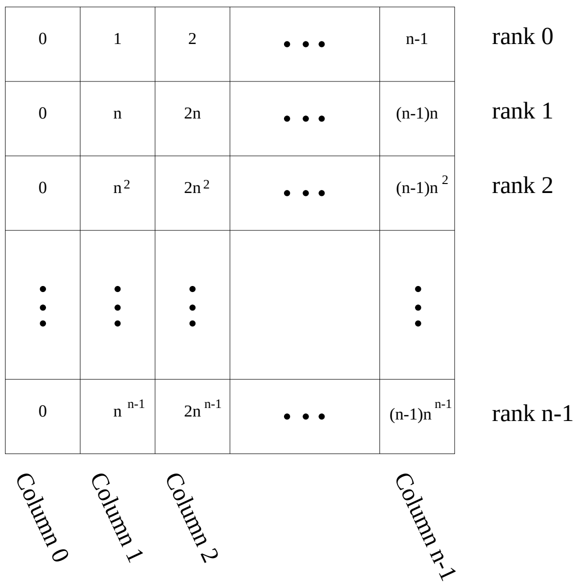


Figure 19: Encoding of the chessboard

A parallel n-queens constraint-satisfaction search algorithm with p processors uses distributed search tree approach as follows. First, the algorithm enumerates a set of independent search-tree seed

Figure 20: Search Tree for a constrained search, e.g. the nqueens problem.

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 $n^k - 1$ by summing a term from each queen placed on rank i, $(0 \leq i < k)$, and column j, $(0 \leq j < n)$, equal to jn^i. For clarity, Figure 19 shows the value of each position on the chessboard. Note that this is equivalent to converting to decimal a base n number with digit i, $(0 \leq i < k)$, representing the column position of queen i. These n^k partial placements can then be partitioned evenly among the processors and 1) checked for validity, and 2) used as a root node for a sequential depth-first search of the remaining $n - k$ queen positions from that starting point. Figure 20 contains an example of this search tree for $k = 2$. The algorithm which decodes the array of k column positions from a partial solution σ, with $(0 \leq \sigma < n^k)$, is as follows.

- For $i = 0$ to $k - 1$ do

  $column_i = \left\lfloor \frac{\sigma \bmod n^{i+1}}{n^i} \right\rfloor$;

We have looked at three approaches, and in each, running time is directly proportional to the maximum of the number of solutions found on each of the threads. The first uses a block partitioning of the n^k search nodes to the p processors (using the *all_pardo_block()* SIMPLE computation primitive), such that processor i searches nodes $\frac{n^k}{p} i$ through $\frac{n^k}{p}(i+1) - 1$, inclusive. The second approach cyclicly assigns the n^k integers to p processors (using the *all_pardo_cyclic()* SIMPLE computation primitive).

Thread	Block Partitioning				Cyclic Partitioning				Random Partitioning			
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0	69516	98156	118964	149691	69516	0	178944	191446	69516	106100	140601	135096
1	98156	114216	135201	144279	98156	0	156055	161157	98156	114545	137452	139266
2	122763	145221	136293	143107	122763	183946	135110	149503	206294	126311	143181	143473
3	157034	156914	143439	137485	157034	186905	132685	126341	98156	214920	144096	141031
4	175296	173634	146298	140338	175296	174349	120073	145543	218738	164550	136209	143113
5	201164	173914	148253	138769	201164	180358	130586	118268	69516	149524	136999	139012
6	206294	185820	143268	133407	206294	161023	123294	142711	157034	194000	162541	145240
7	218738	183434	167876	152516	218738	171347	130816	127433	175296	156660	147198	154468
8	206294	185820	167876	152516	206294	163328	123294	145826	206294	136476	140263	144914
9	201164	173914	143268	133407	201164	171347	130586	127433	175296	159105	146169	139555
10	175296	173634	148253	138769	175296	161023	120073	142711	201164	131564	147724	150637
11	157034	156914	146298	140338	157034	180358	132685	118268	122763	107366	147410	135424
12	122763	145221	143439	137485	122763	174349	135110	145543	157034	138129	136104	138323
13	98156	114216	136293	143107	98156	186905	156055	126341	201164	113832	142321	139530
14	69516	98156	135201	144279	69516	183946	178944	149503	122763	133500	143905	143784
15	0	0	118964	149691	0	0	194874	161157	0	132602	127011	146318
Time	16.9	15.5	16.1	17.4	16.9	18.6	17.2	19.5	17.1	18.2	15.5	15.5
Minimum	0	0	118964	133407	0	0	120073	118268	0	106100	127011	135096
Maximum	218738	185820	167876	152516	218738	186905	194874	191446	218738	214920	162541	154468
Mean	142449	142449	142449	142449	142449	142449	142449	142449	142449	142449	142449	142449
s.d.	60766	47177	129365	5937	60766	68910	22684	18320	60766	29248	7409	5030

Table IV: Number of solutions found by each thread ($p = 4$, $r = 4$) with $n = 15$. The total number of solutions is $2,279,184$. Execution time is directly proportional to the maximum of the number of solutions found on each of the threads.

Thread	Block Partitioning				Cyclic Partitioning				Random Partitioning			
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0	69516	48064	48367	64565	69516	0	86650	92328	69516	30660	60804	63533
1	98156	50092	70668	85126	98156	0	82838	83235	98156	75440	78318	71563
2	122763	65617	42749	51616	122763	92491	61969	78043	0	54679	72288	71952
3	157034	51709	92705	92663	157034	96691	69895	65095	0	59866	66643	67537
4	175296	94422	34981	45751	175296	87165	59263	87157	206294	55658	86526	72918
5	201164	57701	104058	97394	201164	92174	64779	59152	98156	92309	56655	70332
6	206294	108539	43044	59387	206294	80770	64782	86489	0	99647	69981	71445
7	218738	62155	104974	78264	218738	86558	62832	64557	0	93617	74115	69586
8	206294	121031	68014	80932	206294	81664	64782	72913	218738	71139	66679	68520
9	201164	70033	78725	59660	201164	84789	64779	62876	69516	93411	68194	74593
10	175296	115750	76729	77287	175296	80253	59263	56222	0	76422	66288	67051
11	157034	84305	67782	61761	157034	88184	69895	59116	0	73102	72047	71961
12	122763	110912	82248	75105	122763	87184	61969	58386	157034	97628	86697	76463
13	98156	99262	64747	57565	98156	90214	82838	61246	175296	96372	75844	68777
14	69516	91717	87375	84361	69516	91455	86650	71460	0	60505	71980	80473
15	0	118457	72426	68155	0	0	97437	77922	0	96155	75218	73995
16	0	74368	81102	69324	0	0	92294	99118	206294	87107	66234	73484
17	0	125687	96401	84266	0	0	73217	77922	175296	49369	74029	71430
18	0	57608	58643	58214	0	91455	73141	71460	0	94060	67937	73491
19	0	133456	90875	75735	0	90214	62790	61246	0	65045	78232	66064
20	0	48125	63805	60803	0	87184	60810	58386	201164	53108	69590	74624
21	0	122569	77657	77572	0	88184	65807	59116	122763	78456	78134	76013
22	0	43567	72994	57980	0	80253	58512	56222	0	63597	74931	69079
23	0	108556	77330	82511	0	84789	67984	62876	0	43769	72479	66345
24	0	38927	84213	76685	0	81664	58512	72913	157034	48804	76942	75764
25	0	78399	57598	60312	0	86558	65807	64557	201164	89325	59162	62559
26	0	39281	104906	97457	0	80770	60810	86489	0	70117	64149	69380
27	0	58875	31883	45055	0	92174	62790	59152	0	43715	79542	70150
28	0	0	94351	93558	0	87165	73141	87157	122763	50358	76532	73331
29	0	0	34744	50429	0	96691	73217	65095	0	83142	67298	70453
30	0	0	76056	86123	0	92491	92294	78043	0	57216	68441	72910
31	0	0	37034	63568	0	0	97437	83235	0	75386	57275	73408
Time	16.9	10.0	9.49	9.63	16.9	9.32	8.64	10.7	17.1	9.39	8.59	8.05
Minimum	0	0	31883	45055	0	0	58512	56222	0	30660	56655	62559
Maximum	218738	133456	104974	97457	218738	96691	97437	99118	218738	99647	86697	80473
Mean	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5	71224.5
s.d.	83182	38795	21037	14675	83182	34483	11822	11913	83182	18967	7227	3774

Table V: Number of solutions found by each thread ($p = 8$, $r = 4$) with $n = 15$. The total number of solutions is $2,279,184$. Execution time is directly proportional to the maximum of the number of solutions found on each of the threads.

Thread	Block Partitioning				Cyclic Partitioning				Random Partitioning			
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0	1005759	569531	736363	892999	436228	436228	436228	436228	436228	593592	757035	859987
1	1629362	705788	840988	936350	569531	569531	569531	569531	436228	826866	806923	890821
2	2211042	876866	904770	947433	736363	736363	736363	736363	569531	728812	915866	898635
3	2540093	1006695	948304	906130	892999	892999	892999	892999	569531	850266	922600	922046
4	2540093	1124437	977349	925816	1050762	1050762	1050762	1050762	736363	885655	969374	886783
5	2211042	1201779	986811	935033	1160280	1160280	1160280	1160280	736363	894789	903152	904786
6	1629362	1263315	1002574	919284	1249262	1249262	1249262	1249262	892999	972658	908905	906218
7	1005759	1275690	1010651	923211	1290831	1290831	1290831	1290831	892999	861916	933833	930362
8	0	1263315	1010843	923211	1290831	1290831	1290831	1290831	1050762	787536	976046	949414
9	0	1201779	1003088	919284	1249262	1249262	1249262	1249262	1050762	833993	957054	928577
10	0	1124437	986725	935033	1160280	1160280	1160280	1160280	1160280	963988	943920	898572
11	0	1006695	975143	925816	1050762	1050762	1050762	1050762	1160280	1011439	957437	946126
12	0	876866	946296	906130	892999	892999	892999	892999	1249262	1063841	967246	933564
13	0	705788	895153	947433	736363	736363	736363	736363	1249262	1133237	929616	967328
14	0	569531	836553	936350	569531	569531	569531	569531	1290831	1177960	927971	981314
15	0	0	710901	892999	436228	436228	436228	436228	1290831	1185964	995534	967979
Time	224	111	108	122	112	112	112	112	114	118	107	107
Minimum	0	0	710901	892999	436228	436228	436228	436228	436228	593592	757035	859987
Maximum	2540093	1275690	1010843	947433	1290831	1290831	1290831	1290831	1290831	1185964	995534	981314
Mean	923282	923282	923282	923282	923282	923282	923282	923282	923282	923282	923282	923282
s.d.	1011654	336066	92958	16301	298327	298327	298327	298327	298327	158726	59589	32538

Table VI: Number of solutions found by each thread ($p = 4$, $r = 4$) with $n = 16$. The total number of solutions is $14,772,512$. Execution time is directly proportional to the maximum of the number of solutions found on each of the threads.

Thread	Block Partitioning				Cyclic Partitioning				Random Partitioning			
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0	436228	303358	323941	402890	436228	217768	217768	217768	436228	552009	500912	419489
1	569531	266173	412422	490109	569531	277797	277797	277797	736363	581228	494622	440498
2	736363	496440	349598	389175	736363	376922	376922	376922	0	676677	480094	443452
3	892999	209348	491891	547175	892999	419521	419521	419521	0	509287	435772	478594
4	1050762	713628	370444	383931	1050762	557733	557733	557733	1290831	272181	445361	475642
5	1160280	247671	535567	563502	1160280	531864	531864	531864	569531	613474	498559	411141
6	1249262	768939	394648	363676	1249262	678243	678243	678243	0	434395	429612	492134
7	1290831	381293	561055	542454	1290831	584709	584709	584709	0	529593	500004	475845
8	1290831	708831	392371	379652	1290831	706122	706122	706122	569531	412643	426575	477536
9	1249262	563088	595454	546164	1249262	571019	571019	571019	436228	414223	496025	468590
10	1160280	633290	398378	375148	1160280	628416	628416	628416	0	267860	441639	456077
11	1050762	694077	594582	559885	1050762	493029	493029	493029	0	325732	534407	434744
12	892999	640607	408346	360188	892999	473478	473478	473478	892999	295767	463286	441837
13	736363	657162	596332	559096	736363	359441	359441	359441	1290831	491769	494151	462949
14	569531	740196	412638	426046	569531	291734	291734	291734	0	532633	455468	462773
15	436228	495278	595000	497165	436228	218460	218460	218460	0	478806	453437	467589
16	0	768037	502178	497165	0	218460	218460	218460	1249262	506323	454545	463827
17	0	450243	501376	426046	0	291734	291734	291734	1160280	466335	352378	464750
18	0	751536	590947	562698	0	359441	359441	359441	0	427939	464503	491112
19	0	455587	409638	356586	0	473478	473478	473478	0	300873	502743	458302
20	0	668850	672143	568540	0	493029	493029	493029	1160280	388994	427591	501676
21	0	508096	321274	366493	0	628416	628416	628416	1050762	461272	541783	465652
22	0	529472	718747	554693	0	571019	571019	571019	0	436813	476970	490010
23	0	531973	267438	371123	0	706122	706122	706122	0	457976	480084	443554
24	0	348402	685339	551356	0	584709	584709	584709	736363	437003	365216	457218
25	0	531095	268064	354774	0	678243	678243	678243	1249262	424913	391819	441354
26	0	202094	623428	572286	0	531864	531864	531864	0	541040	476390	478188
27	0	465160	275840	375147	0	557733	557733	557733	0	522801	451581	428030
28	0	42588	566593	554186	0	419521	419521	419521	892999	753166	527933	488249
29	0	0	256061	382164	0	376922	376922	376922	1050762	424794	375219	493065
30	0	0	485257	496786	0	277797	277797	277797	0	349677	448865	440311
31	0	0	195522	396213	0	217768	217768	217768	0	484316	484968	458324
Time	112	76.4	65.6	68.5	112	59.5	59.5	59.5	114	65.0	55.5	54.2
Minimum	0	0	195522	354774	0	217768	217768	217768	0	267860	352378	411141
Maximum	1290831	768939	718747	572286	1290831	706122	706122	706122	1290831	753166	541783	501676
Mean	461641	461641	461641	461641	461641	461641	461641	461641	461641	461641	461641	461641
s.d.	507555	235508	138282	83552	507555	153618	153618	153618	507555	108400	45395	22253

Table VII: Number of solutions found by each thread ($p = 8$, $r = 4$) with $n = 16$. The total number of solutions is $14,772,512$. Execution time is directly proportional to the maximum of the number of solutions found on each of the threads.

Both the block and cyclic partitioning schemes can be performed implicitly without the need for any explicit inter-processor communication. The third method, however, will require communication, but because it more evenly distributes the computational load (see the standard deviation of the number of solutions found by each thread in Tables IV–VII), we find that it is superior in performance to the first two methods.

Algorithm	n	CPUs		Time (s)
		p	r	
Netlib	14		1	36.336
SIMPLE	14	1	1	38.8
SIMPLE	14	1	4	10.0
SIMPLE	14	4	4	2.73
SIMPLE	14	8	4	1.32
Netlib	15		1	237.080
SIMPLE	15	1	1	255.
SIMPLE	15	1	4	66.4
SIMPLE	15	4	4	15.5
SIMPLE	15	8	4	8.05
Netlib	16		1	1646.131
SIMPLE	16	1	1	1785.
SIMPLE	16	1	4	455.
SIMPLE	16	4	4	107
SIMPLE	16	8	4	54.2

Table VIII: n-Queens Performance Summary.

The third approach randomizes the integers from 0 to $n^k - 1$, and assigns $\frac{1}{p}^{\text{th}}$ of these to each processor. The overhead for randomization and communication is minimal compared with the faster completion time due to improved load balance. See Tables IV and V for a comparison of these three algorithms when $n = 15$, on $p = 4$ and $p = 8$ nodes, each an $r = 4$-way SMP, varying k from 1 to 4. Similar results for $n = 16$ are given in Tables VI and VII. Because of the special topology inherent in this search problem, the block and cyclic partitioning schemes are inferior to a randomized approach. Table VIII gives the performance of our SIMPLE algorithm compared to the standard netlib "queens" benchmark results for $n = 14, 15$, and 16. Because our algorithm is generalized for COSMOS , it takes slightly longer to compute on a single processor, but scales linearly with the total number of processor used.

8 Experimental Platform

Our experimental platform consists of 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 the

MPI (e.g., LAM 6.1, MPICH 1.0.13, or CHIMP 2.1.1c) and pthreads (DECthreads) packages. 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. [14] 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.

9 Future Work

The future research directions of the SIMPLE project can be categorized into two areas: methodology and algorithmics. In methodology, we plan an extension of the SIMPLE kernel to handle more communication events. Also, in a cluster of SMPs, it is not always the case that nodes are homogeneous in size, memory, speed, load, or even architecture. We are currently researching load sharing inside SIMPLE algorithms such that a problem initially is distributed across the cluster such that each node no longer has $\frac{1}{p}^{\mathrm{th}}$ of the input but a portion of the input directly proportional to each node's current ability to solve the task. In addition, tasks may migrate across nodes during runtime to reflect changing conditions in the cluster, or to redistribute work when the current pool of nodes shrinks or grows. For the second area, algorithmics, we are examining various experimental data sets for benchmarking algorithms on clusters of SMPs, and are implementing high performance application codes using the SIMPLE methodology.

10 Release Notes

Please see `http://www.umiacs.umd.edu/research/EXPAR` for additional performance information. In addition, all the code used in this paper is freely available for interested parties from our anonymous ftp site, `ftp://ftp.umiacs.umd.edu/pub/EXPAR`.

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):18–28, 1996.

[3] D.A. Bader. *On the Design and Analysis of Practical Parallel Algorithms for Combinatorial Problems with Applications to Image Processing.* PhD 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. CS-TR-3548 and UMIACS-TR-95-101 Technical Report, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, November 1995.

[5] D.A. Bader, D.R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1(3):1–42, 1996. `http://www.jea.acm.org/1996/BaderPersonalized/`.

[6] 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.

[7] W.P. Brown. Parallel Computation of Atmospheric Propagation. Technical report, Maui High Performance Computing Center and Phillips Laboratory, Kihei, Maui, HI, 1995.

[8] 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.

[9] Digital Equipment Corp. *Guide to DECthreads.* Maynard, MA, July 1994.

[10] Digital Equipment Corporation, Maynard, MA. *Digital UNIX (formerly OSF/1)*, v3.2c edition, July 1995.

[11] 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. 10 pp. Available from `ftp://ftp.sgi.com/sgi/whitepaper/challenge_paper.ps.Z`.

[12] 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/`.

[13] C. Harris. Node Selection for the IBM RS/6000 SP System. Version 2.1. IBM RS/6000 Division, November 1996.

[14] F.M. Hayes. Design of the AlphaServer Multiprocessor Server Systems. *Digital Technical Journal*, 6(3):8–19, Summer 1994.

[15] IBM Corporation. *AIX DCE Base Services/6000 Version 1.2*, 7 edition, October 1993.

[16] IBM Corporation. RS/6000 SP System. RS/6000 Division, 1997.

[17] P. Keleher. *CVM: The Coherent Virtual Machine.* University of Maryland, 0.1 edition, November 1996.

[18] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.

[19] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.

[20] F. Müller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the 1993 Winter USENIX Conference*, pages 29–41, San Diego, CA, January 1993. `http://www.informatik.hu-berlin.de/~mueller/projects.html`.

[21] Netlib Repository for mathematical software, papers, and databases. University of Tennessee and Oak Ridge National Laboratory. `http://www.netlib.org/`.

[22] Ohio Supercomputer Center. *LAM / MPI Parallel Computing.* The Ohio State University, Columbus, OH, 1995. `http://www.osc.edu/lam.html`.

[23] G.F. Pfister. *In Search of Clusters.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[24] Portable Applications Standards Committee of the IEEE. *Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API)*, 1996-07-12 edition, 1996. ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1.

[25] C. Provenzano. Proven Pthreads. WWW page., 1995. `http://www.mit.edu/people/proven/pthreads.html`.

[26] 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.

[27] Sun Microsystems, Inc. POSIX Threads. WWW page., 1995. `http://www.sun.com/developer-products/sig/threads/posix.html`.

[28] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[29] 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*, pages 24–36, June 1995.

[30] 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.