The Design and Evaluation of a Shared Object System for Distributed Memory Machines

Daniel J. Scales and Monica S. Lam

Computer Systems Laboratory
Stanford University, CA 94305
{scales,lam}@cs.stanford.edu

Abstract

This paper describes the design and evaluation of SAM, a shared object system for distributed memory machines. SAM is a portable run-time system that provides a global name space and automatic caching of shared data. SAM incorporates mechanisms to address the problem of high communication overheads on distributed memory machines; these mechanisms include tying synchronization to data access, chaotic access to data, prefetching of data, and pushing of data to remote processors. SAM has been implemented on the CM-5, Intel iPSC/860 and Paragon, IBM SP1, and networks of workstations running PVM. SAM applications run on all these platforms without modification.

This paper provides an extensive analysis on several complex scientific algorithms written in SAM on a variety of hardware platforms. We find that the performance of these SAM applications depends fundamentally on the scalability of the underlying parallel algorithm, and whether the algorithm's communication requirements can be satisfied by the hardware. Our experience suggests that SAM is successful in allowing programmers to use distributed memory machines effectively with much less programming effort than required today.

1 Introduction

Distributed memory systems, especially in the form of networks of workstations, are an important computational resource. However, programming distributed memory machines using commonly available message-passing libraries is a difficult process. The necessary communication and synchronization in an application must be reduced to low-level message-passing calls that require coordination between the senders and receivers. The difficulties become even greater for applications that have highly irregular parallelism and communication. The single address space provided by shared memory machines significantly eases the process of programming these kinds of applications. Shared data are easily accessed via the shared address space without regard to where data are currently located in the system. In addition, cache-coherent shared memory multiprocessors reduce the cost of accesses to shared data by automatically caching recently accessed data locally.

Our thesis is that it is possible to build a portable software layer that provides the shared memory programming model in a distributed memory environment, while providing mechanisms to help tolerate the relatively high cost of communication on these machines. In this paper we describe the design and evaluation of a shared object system called SAM which we have implemented that supports this thesis. Shared data items in SAM are accessed via a global name space and are automatically cached in the main memory of each processor to exploit the locality of reference in applications. For efficiency reasons, shared data are accessed and managed only at the level of user-defined data types. By requiring that programmers signify how data will be used when it is accessed, SAM can combine synchronization and data access and reduce communication. In addition, SAM provides operations that allow the programmer to explicitly optimize communication when desired. SAM is intended to be used in exploiting task-level parallelism; we believe that the parallelism inherent in operations on regular data structures, such as arrays, is most effectively exploited by parallelizing compilers.

SAM has been implemented as a C library on a variety of platforms: on the CM-5, Intel iPSC/860, Intel Paragon, and IBM SP1, and on heterogeneous networks of workstations using PVM [26]. To evaluate the effectiveness of SAM, we have chosen a number of complex scientific applications that have previously been developed by other researchers either for machines with hardware shared memory support or for distributed memory machines using the message-passing style. We have implemented these applications on distributed memory machines using SAM; each application runs unchanged across the various platforms. We then attempt to evaluate the usefulness of the SAM system by analyzing the performance, programming difficulty,

This research was supported in part by DARPA contract DABT63-91-K-0003.

and SAM overheads of these applications, and comparing with the original implementations on shared memory multiprocessors or using message passing.

In the following section, we describe the SAM design rationale and discuss some related work. Next, we describe the basic SAM primitives. Then we describe the applications, explain some of the aspects of programming these applications using SAM, and give performance numbers on all the machines. We also give figures on the various costs of parallelizing these applications, including the SAM software overhead. We then describe the effects of various aspects of SAM's design on the performance of the applications, and conclude.

2 Design Rationale

In this section, we give background on some existing software distributed shared memory systems and describe the basic SAM design principles.

2.1 Background

There have been a number of software systems for distributed memory machines that give the user the illusion of a shared memory and provides automatic caching of shared data. Shared virtual memory (SVM) systems such as Ivy [18] apply the concepts of hardware caching to virtual memory pages. Such systems transparently support the view of a single address space across processors. However, the use of pages as the unit of coherence can result in extensive false sharing; unnecessary communication occurs because of the placement of unrelated shared data on the same page, which may be difficult to avoid in applications with complex data structures. In addition, the protocols for maintaining cache consistency may produce excess communication, since the system treats the entire memory uniformly. Processes use lock and unlock operations to synchronize with each others' data accesses. The system must guarantee that *all* memory updates preceding a synchronization operation are observable by all processors when the synchronization completes, though most updates may be relevant to only a few processors.

Munin [15] and Treadmarks [16] address the problem of false sharing by allowing multiple processors to write to a page and merging changes at the next synchronization point. In addition, Treadmarks attempts to alleviate the problems of excess consistency messages by implementing lazy release consistency, in which the modifications or invalidations to data are not sent until a processor acquires a lock that requires that it see those modifications. Cox et al. [12] compare the performance of several applications on a network of eight workstations running Treadmarks and on a hardware shared-memory multiprocessor. Some of the applications perform comparably under the two systems, but several do not scale well for certain inputs and one application must be restructured to get speedups under Treadmarks. It remains undemonstrated as to how these approaches perform for applications with more demanding communication patterns.

To provide the user with control over the granularity of communication and avoid false sharing, systems such as Amber [11], Prelude [28], Orca [1], Midway [3], and Linda [5, 7] communicate data at the level of user-defined data types (or *objects*), rather than virtual-memory pages. Amber, Prelude, and Orca provide access to shared data in the context of object-oriented languages. Amber and Prelude primarily provide access to shared objects by moving tasks to the processor containing the data. Orca and Midway allow for replication of shared objects across processors. One Orca implementation replicates objects on all the processors and uses an update protocol to maintain the consistency of shared data. Such a protocol works well for some applications, but can cause performance problems in many other applications. Midway associates arbitrary regions of shared data with locks, and ensures that shared data are consistent when the lock associated with the data is acquired. In this way, it successfully hides synchronization messages by combining them with data access messages. The reader-writer locks protecting the shared data are managed via an invalidation protocol. The Linda system provides operations that insert, read, and remove "tuples" from a shared tuple space. Sophisticated compiler analysis is typically required to analyze and optimize the communication in a Linda program. We do not know of a distributed-memory Linda implementation that provides dynamic caching of tuples (except for one which broadcasts all tuples to all processors [7]).

2.2 Design of SAM

SAM takes one step further than Orca and Midway in providing user control over communication on a distributed memory machine. SAM also communicates data at the level of user-defined data types. While most shared object systems differ primarily in the consistency schemes and implementations used, SAM is based on a significantly different set of primitives that are motivated by optimizations commonly used on distributed memory machines. Message-passing programs seldom incur extra communication just for the sake of synchronization, since synchronization is tied with the arrival of messages that also contain data. Furthermore, it is possible to relax synchronization by taking advantage of the explicit copies made on distributed address space machines. For example, a write to a local copy of shared data need not wait for all reads of the older version of the data to complete as long as a copy of the original version exists. Finally, message-passing programs can hide the latency of communication by sending data to remote processors before they require the data. SAM preserves these advantages of message-passing systems while providing the ease of programming associated with shared memory machines.

The basic principle of SAM is to require the programmer to designate the way in which data will be accessed. With this access information, SAM can combine synchronization and communication of data. There are two kinds of data in SAM, which correspond to the two kinds of data relationships (hence synchronization) in parallel programs:

values with a *single-assignment semantics*. Values make it simple to express producer-consumer relationships or precedence constraints; any read of a value must wait for the creation of the value.

accumulators [21], whose data accesses are *mutually exclusive*. Data are migrated automatically in turn to processors which request mutually exclusive access.

The explicit naming of all values makes it possible to eliminate anti-dependence constraints between processors; a processor can continue to read an older version of a data item while another processor generates another version (using a different name). The concept of single-assignment values have been used in a variety of parallel languages [10, 9, 13, 14, 20] as a way of exposing parallelism and synchronizing independent tasks. Such systems often must deal with problems of excessive memory usage and data copying as shared data items are created and modified. SAM avoids these problems by requiring that the programmer explicitly supply information on when a piece of data is no longer needed, and allowing the programmer to specify that one value should reuse the storage of another value. We have found that managing the names and memory of these values to be straightforward using the primitives provided by SAM.

SAM also provides access to stale, local copies of accumulator data, which can be used in *chaotic* algorithms which do not always need to use the most up-to-date data. Using a local, possibly older copy is sometimes sufficient and can reduce the total execution time of these algorithms. A number of systems have supported the notion of chaotic access. Agora [4] supports a memory model in which all accesses are chaotic, since all modifications to shared data structures are allowed to complete before holders of cached copies have been notified. Mether [19] and Clouds [22] support operations for accessing a read-only copy of a page (or segment) that will not be kept coherent even if the contents of the page (or segment) are changed by another processor.

Finally, besides minimizing communication, SAM also provides primitives for optimizing communication. SAM provides mechanisms for producers to *push* data to consumers, and for processes to *prefetch* a data item. All these optimizations are well integrated into SAM's shared memory model.

3 SAM Overview

In this section, we present a brief overview of SAM mainly via several examples. A more detailed description is provided in [24].

3.1 Basic Primitives

In SAM, all shared data are represented by either a *value* or an *accumulator*. (SAM deals only with the management and communication of shared data; data that are completely local to a processor can be managed by any appropriate method.) In Figure 1, we show several common idioms, as they would be expressed using semaphores on a shared address space machine and using SAM primitives.

In the first example, mutual exclusion is required to protect updates to shared data. In SAM, an accumulator is used to represent a piece of data that is to be updated a number of times, and whose final value is independent of the order in which the updates occurs. SAM automatically migrates the accumulator between processors as necessary and ensures that a process does not access the accumulator until mutual exclusion is obtained. Updates to an accumulator must be encapsulated by the SAM primitives `begin_update_accum` and `end_update_accum`. The call to `begin_update_accum` returns a pointer by which the accumulator can be accessed. SAM supports the idiom of chaotic computation via primitives which provide read access to a "recent" value of the accumulator, which is not guaranteed to be the most current value of the accumulator. SAM maintains a cache on each processor of versions of accumulators that have been recently accessed and therefore may be able to satisfy the chaotic request locally without communication.

In the second example, a consumer (right column) accesses data created by a producer (left column). In SAM, a value provides producer/consumer synchronization. Values have a single-assignment semantics: a value is atomically created once its initial contents are set and is henceforth immutable. The code to create a value, which may include arbitrary updates to different components of the value, is encapsulated by a pair of primitives `begin_create_value` and `end_create_value`. Similarly, code accessing a value is encapsulated by the primitives `begin_use_value` and `end_use_value`. A process that attempts to access a value will automatically be suspended until the value is created and has been brought to the local processor. Conversely, an access will succeed immediately if the value is already cached on the local processor, returning a pointer to the local copy.

In the third example, a consumer accesses a sequence of values created by a producer through a limited-sized buffer. To avoid memory usage problems that are associated with single-assignment values, SAM allows one value to reuse the storage of another value via the `begin_rename_value` primitive. This primitive pro-

Shared Address Space | **Distributed Address Space + SAM**

1) Mutual Exclusion

Shared Address Space:

```
wait(lock)
a[1] = ...;
a[34] = ...;
signal(lock)
```

Distributed Address Space + SAM:

```
begin_update_accum(a)
a[1] = ...;
a[34] = ...;
end_update_accum(a)
```

2) Producer/consumer

Shared Address Space:

```
i=i+1              wait(flag_j)
allocate a_i       j=j+1
a_i = ...          ... = a_j
signal(flag_i)
```

Distributed Address Space + SAM:

```
i=i+1                     j=j+1
begin_create_value(a_i)   begin_use_value(a_j)
a_i = ...                 ... = a_j
end_create_value(a_i)     end_use_value(a_j)
```

3) Finite buffer (size 4)

Shared Address Space:

```
i=i+1 mod 4       j=j+1 mod 4
wait(buf_i)       wait(flag_j)
a[i] = ...        ... = a[j]
signal(flag_i)    signal(buf_j)
```

Distributed Address Space + SAM:

```
i=i+1                              j=j+1
begin_rename_value(a_{i-4}→a_i)    begin_use_value(a_j)
a_i = ...                          ... = a_j
end_rename_value(a_i)              end_use_value(a_j)
                                   free(a_j)
```

Figure 1: Example of Creating and Accessing Values

vides the necessary synchronization to ensure that the producer does not reuse the storage of a value before the consumer has accessed it. In a similar fashion, imperative data objects are easily represented in SAM via a sequence of values which can all share the same storage. SAM also allows a value to be converted to an accumulator and vice versa.

3.2 Memory Management

The creator of a value or an accumulator must specify the type of the new data. With the help of a preprocessor, SAM uses this type information to allocate space for, pack (for sending in a message), unpack, and free the storage of the data. The preprocessor can handle complex C data types, including types that contain pointers and therefore are not necessarily stored in one contiguous block in memory.[1] In heterogeneous environments, SAM also handles any necessary data conversion between dissimilar machines.

SAM maintains local copies of values fetched from remote processors in the form of a cache. Because all values have distinct names and are immutable, there is no consistency problem associated with maintaining this cache. SAM automatically frees up local copies that are not in use when the cache memory becomes filled. SAM must ensure that at least one copy of a value is maintained in the system, until it can determine that there will not be any other processes that will need to access the value. The SAM programmer provides this information by specifying the number of accesses to the value that will occur or explicitly indicating when all accesses to the value have occurred.

[1]The preprocessor only handles simple hierarchical data types; it does not handle general data structures that contain multiple pointers to the same data.

3.3 Communication Optimizations

An important mechanism for tolerating the communication latency is support for asynchronous access. SAM provides the capability to fetch values and accumulators asynchronously. An asynchronous fetch succeeds immediately if a copy of the value is available on the local processor. However, if the value is not immediately available, the fetch operation returns an indication that the value is not available. The requesting process can proceed with other accesses or computation. When the value becomes available on the local processor, the requesting process is notified by calling a function specified when the request was made. For asynchronous access to an accumulator, the process is notified when the accumulator has been fetched to the local processor and mutual exclusion has been obtained.

Another method for optimizing communication is to send data directly from one processor to another processor which will need it. A copy of any specified value available on a processor can be explicitly sent ("pushed") to a remote processor via the `push_value` primitive. SAM's basic mechanisms combine smoothly to provide the buffering necessary to support a message-passing style. If a process attempts to access a piece of data before it has arrived, then the process suspends until the named value arrives. Conversely, if a value arrives at a processor before it is needed, it is automatically buffered by caching it as a local copy. Note, however, that the push operation is only an optimization and does not change the correctness of a SAM program.

4 Applications

In this section, we describe the applications we have used in evaluating SAM, assess the ease of programming these applications using SAM, and give performance results on a variety of hardware platforms. For comparison purposes, we have chosen several complex applications that have been implemented previously either on shared-memory multiprocessors or distributed memory machines using message passing. The applications are:

- Block Cholesky - application that does a parallel Cholesky factorization of a sparse matrix by doing block updates in parallel.
- Barnes-Hut - application that simulates the evolution of an n-body system using a tree data structure to speed up the force calculations.
- Grobner Basis - application that computes the Grobner basis for a polynomial set by repeatedly processing pairs of polynomials and potentially adding new polynomials to the basis until no further polynomials need to be added.

	serial code	SAM code	DASH code	msg-pass. code
Block Cholesky	NA	6713	6813	NA
Barnes-Hut	1959	2896	2232	3973
Grobner Basis	3757	4082	NA	5747

Figure 2: Application Line Counts

Figure 2 gives approximate line counts for different versions of each application. The SAM version of the block Cholesky application is derived from the original code written for the DASH shared-memory multiprocessor [17]. Both the SAM and DASH versions of the Barnes-Hut application are derived from the original serial code. Warren and Salmon's message-passing Barnes-Hut code [27] is a completely separate code not based at all on the serial Barnes-Hut code. Chakrabarti's message-passing version of the Grobner basis algorithm for the CM-5 [8] is based on the original serial code, as is the SAM version. The line counts indicate the size of the applications and give a rough comparison of the difficulty in programming the different versions, which we will address further below.

Each of the SAM applications runs without modification on the CM-5, iPSC/860, Paragon, SP1, and networks of workstations. Below, we present performance results for a 64-processor CM-5 (running CMOST 7.3 and CMMD 3.2), a 32-processor iPSC/860, a 56-processor Paragon (running OSF 1.0.4 and NX 1.2.1), a 16-processor SP1, and a 48-processor DASH.[2] In Figure 3, we give a summary of the important characteristics of each machine. The first five columns describe the processor at each node of the machine. (The third column reports peak *double-precision* megaflops.) The last three columns give the values that we have measured for bandwidth, one-way message send time, and round-trip message time between two nodes. By analyzing the performance of these applications on a variety of machines with different characteristics, we get a better understanding of the hardware and software factors that affect their performance.

4.1 Block Sparse Cholesky Factorization

The block Cholesky application [23] performs a Cholesky factorization of a sparse, symmetric matrix in parallel. It decomposes the sparse matrix into blocks and assigns work to processors at the granularity of updates to blocks. Such updates typically involve using two source blocks to update one destination block. The parallel algorithm involves executing updates to blocks in parallel while respecting the necessary data dependences. The block Cholesky algorithm benefits from dynamic caching, since each sparse block may be used many times by a processor to update other blocks.

The SAM implementation of the block Cholesky application is derived directly from a version for the DASH multiprocessor. Each individual block of the matrix is a SAM data item, and the matrix data structures remained largely unchanged. SAM's ability to deal with complex, non-contiguous data types as a single item is important, since each block actually contains a number of dynamically allocated index and data arrays. Each block in the matrix goes through three phases, which are readily apparent from the basic factorization algorithm. The first phase consists of a series of commutative updates (updates that can occur in any order). In the second phase (after the last update is done), the contents of the block are finalized by a matrix division. In the third and final phase, the block is not modified any further and is only used to update other blocks. Thus each block is represented as an accumulator in the first phase, and the second phase creates a value (using the same storage as the accumulator) that is used in the third phase.

Each processor is responsible for all the updates to a statically assigned set of blocks in the matrix. A task is created when one of the source blocks becomes available and is assigned to the processor that "owns" the destination block. The processor then accesses the second source block asynchronously. If the block is not immediately available, the processor continues computing with other data while the system fetches the block in the background.

[2] We were not able to get full numbers on the SP1 because of limited access to the machine. We were also not able to get 48-processor numbers for DASH for the block Cholesky application because of changes in the machine configuration.

Machine	processor	clock rate	peak MFLOPS[3]	I-Cache size	D-Cache size	network topology	measured bandwidth	send time	round-trip time
CM-5	Sparc	33 MHz	8	64KB	64KB	fat tree	8MB/s	$11\mu s$	$57\mu s$
iPSC/860	i860	40 MHz	60	4KB	8KB	hypercube	2.8MB/s	$47\mu s$	$154\mu s$
Paragon	i860	50 MHz	75	16KB	16KB	mesh	61MB/s	$50\mu s$	$125\mu s$
SP1	RS6000	62.5 MHz	125	32KB	64KB	multistage	7MB/s	$240\mu s$	$415\mu s$
DASH	R3000	33 MHz	10	64KB	64KB	bus/mesh	NA	NA	NA

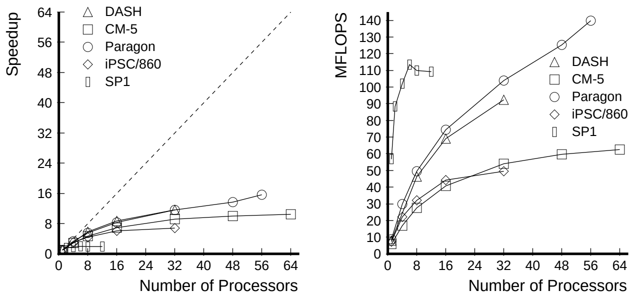
Figure 3: Machine Characteristics

Figure 4 gives performance results for a sparse matrix BCSSTK15 (from the Harwell Boeing sparse matrix test set) and a dense matrix D1000, respectively.[3] The left graph gives parallel speedups with reference to an efficient left-looking, column-based serial factorization algorithm on the same machine; the right graph gives the corresponding absolute performance in double-precision megaflops. For these results, we use blocks of 32 by 32 double-precision numbers. For the sparse matrix, the average size of data transfer over a 32-processor run is 4233 bytes; for the dense matrix, it is 8384 bytes. (Because of the static task and data placement, these figures depend only on the input matrix and number of processors used.) Figure 5 gives information on the average time between shared data references in block Cholesky factorizations of the sparse matrix. To get these numbers, we have divided the serial run times (which represent useful work) by the total number of shared data accesses and the number of these accesses that must request their data remotely (i.e. that incur a cache miss), respectively.

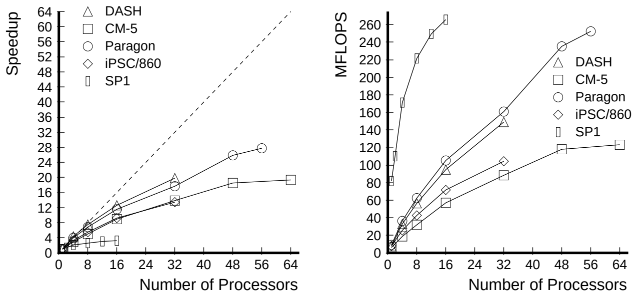
The speedup trends are similar on all machines, indicating the success of SAM in providing portability across a range of distributed memory machines. As we discuss in Section 4.4, the low speedups for the sparse matrix are largely due to limited parallelism and poor load balancing in the parallel algorithm rather than SAM overheads; continued improvement in the parallel algorithm will result in better speedups on all machines.

The differences in the speedup curves reflect the different characteristics of the various machines. In particular, parallel performance can be limited by the available bandwidth between nodes on each machine. The Paragon and the DASH multiprocessor have the best speedups because of their high network bandwidth. The similar speedups of the Paragon and DASH suggest that performance is not significantly affected by the software shared memory implementation, because of the coarse granularity of shared data access. The SP1 reaches bandwidth limits for larger number of nodes, but has very high uniprocessor performance and achieves impressive parallel performance for a small number of nodes. The CM-5 and iPSC/860 also reach bandwidth limits, but at much larger numbers of processors.

[3] On the CM-5, we do not use the four vector units at each node to enhance floating-point performance, because significant additional programming is required to utilize them.



(a)



(b)

Figure 4: Block Cholesky Speedup (left) and Performance (right) for (a) BCSSTK15 and (b) D1000

4.2 Barnes-Hut Algorithm

The Barnes-Hut algorithm [2] is a fast algorithm for simulating the evolution of a system of astronomical bodies as they interact with each other via the gravitational force (the "n-body problem"). At each time step, the algorithm computes the gravitational forces between the n bodies, and determines the new position and velocity of each of the bodies. The Barnes-Hut application builds a tree data structure, called an *oct-tree*, at each time step to summarize the gravitational effects of nearby groups of bodies, so that the force calculation can be done more quickly. The structure of the oct-tree is complex and dependent on the input data, so the memory to hold the oct-tree structure cannot be statically allocated and partitioned across processors. However, the work of the force calculation phase can be partitioned so that there is extensive locality in each processor's access to the tree nodes [25]. The programmer only needs to worry about doing this partitioning correctly; SAM auto-

	number of processors	avg. useful work between accesses to shared data	avg. useful work between accesses to remote data
CM-5	32	438μs	1910μs
iPSC/860	32	364μs	1588μs
Paragon	32	292μs	1274μs
SP1	12	76μs	409μs

Figure 5: Frequency of Shared Data Access in Block Cholesky Factorization of BCSSTK15

matically exploits this locality by caching recently accessed tree nodes on each processor.

To reduce address translation and message-passing overhead for this application, we have experimented with blocking the nodes of the tree together. That is, as the oct-tree is built in each time step, we combine several nodes of the tree into one SAM data item. We have hidden the complexity of the blocking in an oct-tree library, and the user can specify the option of blocking or not blocking. When blocking, the tree library automatically brings over a whole block when the "top" node in the block is accessed. Such blocking increases the granularity and reduces the frequency of communication. It also does a form of prefetching, since it fetches a whole collection of nodes that are likely to be accessed in the near future when one of them is accessed. The disadvantage is that extra bandwidth and memory are used in bringing over nodes that are never accessed. In addition, for large blocks, the parallelism in some of the tree traversal phases is decreased, because only one processor can modify a block at a time.

Figure 6 shows speedup (measured against the efficient serial algorithm) and absolute performance (bodies processed per second) of our parallel version running on all machines for a simulation of a highly irregular distribution of 25000 bodies. We have also included performance numbers (labeled in the graph as MP-iPSC) for the message-passing Barnes-Hut code by Warren and Salmon [27]. We have ported this message-passing code to the iPSC/860 and run it on an identical problem.[4]

For these performance figures, tree blocking is used for the runs on the iPSC/860, Paragon, and SP1. Because of the low cost of sending and receiving messages on the CM-5, tree blocking is unnecessary and not used. The average granularity at which shared data are accessed depends on whether blocking is used or not. Each tree node has a size of 152 bytes. For typical 32-processor runs, the average size of messages that communicate data objects is about 220 bytes when blocking is not used, and 1340 bytes when blocking is used. Figure 7 gives information on the frequency of shared data references for the 25000-body simulation. These figures show that the granularity of access to shared data is much more fine-grain than in the block Cholesky application, but the locality of reference is also much higher.

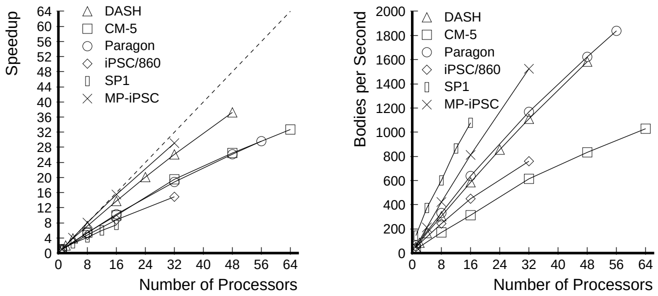


Figure 6: Barnes-Hut Speedup (left) and Absolute Performance (right) for 25000-body Simulation

	number of processors	avg. useful work between accesses to shared data	avg. useful work between accesses to remote data
CM-5	32	27μs	3170μs
iPSC/860	32	39μs	8603μs
Paragon	32	32μs	7069μs
SP1	16	13μs	8848μs

Figure 7: Frequency of Shared Data Access in 25000-body Barnes-Hut Simulation

The speedup curves scale for all versions of the application up to the maximum number of processors, though with different slopes. The message-passing version achieves the best speedups overall. However, as illustrated by the line counts in Figure 2, this version is much more complex and difficult to program than the original serial algorithm. It minimizes communication overhead by distributing tree nodes to all of the processors that will reference them in a single communication phase, and is highly dependent on the way that the Barnes-Hut algorithm uses the oct-tree. The SAM version is somewhat more complex than the serial version, but a large fraction of the increased number of lines is due to the tree blocking implementation, which is isolated in the tree library and reusable for other applications. By using SAM to program the Barnes-Hut algorithm, we have chosen to trade off some overall performance in return for considerably less programming effort and greater portability to newer machines.

The DASH shared-memory multiprocessor also achieves better speedups than the distributed memory machines using SAM. The performance of DASH benefits from its additional hardware that does address translation, caching, and communication without any software overheads. As we will see in Section 4.4, because of the finer granularity of shared data access in this application, software address translation and cache management take up a significant portion of each processor's time in the SAM versions. Nevertheless, the SAM versions scale well and achieve high ab-

[4]Because the error criterion in the message-passing version is different from the original serial version (and the SAM version), it does not do exactly the same computations as these versions, even for identical initial conditions. However, we have set the error parameters so that the work done in all runs is comparable.

solute performance. In addition, the use of machines with higher uniprocessor performance can compensate for the overheads of providing a shared name space in software. As with the block Cholesky algorithm, the SP1 achieves very good absolute performance with only a small number of processors. The low overhead of sending messages on the CM-5 allows us to get good speedups without blocking (and a much finer granularity of access to shared data). The SAM versions on the iPSC/860 and SP1 have the lowest speedups, because of the high cost of sending and receiving messages on these machines.

4.3 Grobner Basis Algorithm

We have used SAM to parallelize an important algorithm from symbolic algebra systems. The algorithm computes the Grobner basis [6] of a set of polynomials, which is used to solve systems of non-linear equations and determine implicit forms for parametric equations. This algorithm has been previously implemented on the CM-5 by Chakrabarti [8]. The basic structure of the algorithm is to start with the initial basis equal to the input set of polynomials. Then, each possible pair of polynomials from the basis is examined; potentially a new polynomial is produced that is added to the basis, and a new set of pairs between the new polynomial and the current members of the basis is generated. The algorithm continues until there are no more pairs left to be examined. All calculations of polynomial coefficients are done using an arbitrary-precision arithmetic package.

In the Grobner basis algorithm, each polynomial remains unchanged once added to the basis. In our parallel implementation of the Grobner basis algorithm using SAM, each polynomial is represented by a SAM value. The dynamic caching of the polynomials in the basis set that is automatically provided by SAM is crucial to good performance, since each processor repeatedly accesses these polynomials. The basis set is a monotonically growing set of polynomials. We represent the set by a linked list of polynomials and an accumulator which points to the polynomials at the head and tail of the list. Using SAM, we have built a distributed set abstraction, which allows polynomials to be added to the set and provides an operation to iterate over the current elements in the set. The set abstraction uses chaotic access to the head and tail pointers of the list when appropriate to reduce contention for these pointers.

Figure 8 shows the speedups and absolute performance of our Grobner basis program for a representative sample of input polynomial sets. The absolute performance is calculated as the number of polynomials tested in the serial execution divided by the parallel run times. The parallel algorithm is inherently non-deterministic due to different task orderings and can give superlinear speedup. In addition, the run times for comparable serial implementations can vary widely, depending on the heuristic used for ordering tasks. Speedups are determined by running the algorithm five times for a particular input set and number of processors, and dividing the average run time by the run time for an efficient serial algorithm with the same task ordering heuristic. For 32-processor runs, the average size of messages communicating data (mainly transmitting polynomials) varies from 200 to 1000 bytes depending on the input set. Figure 9 gives statistics on the frequency of shared data access in 32-processor runs using the Lazard input set. Because the parallel execution can do much more work than the serial execution, as we describe next, we give these figures in terms of the amount of time between accesses in the parallel execution.

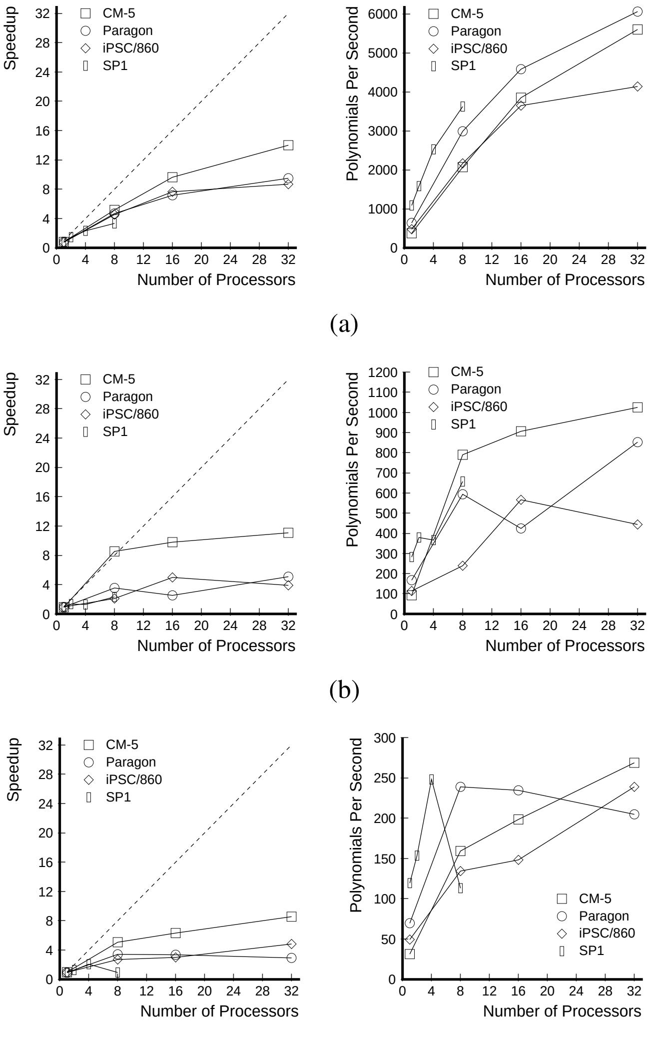


Figure 8: Grobner Speedups (left) and Performance (right) for Inputs (a) Lazard, (b) katsura4, and (c) trinks1

The actual speedups are limited for increasing numbers of processors in large part because the parallel algorithm almost always performs more work than the sequential counterpart. In the serial algorithm, as each polynomial is added to the basis set, it increases the fraction of the remain-

	number of processors	avg. parallel work between accesses to shared data	avg. parallel work between accesses to remote data
CM-5	32	55μs	3188μs
iPSC/860	32	75μs	4315μs
Paragon	32	51μs	2947μs
SP1	8	30μs	7100μs

Figure 9: Frequency of Shared Data Access in Grobner Basis Runs on Lazard

ing candidate polynomials that can be eliminated quickly. However, when the algorithm runs in parallel, each processor executes independently without the knowledge of the polynomials that are about to be added to the basis set by other processors. Therefore, the processors typically do more work and the basis set grows larger than it would in the serial execution. This effect increases with larger numbers of processors and with longer communication latencies.

Our implementation of the algorithm appears to have better heuristics for setting the priority of tasks than the Chakrabarti's message-passing implementation for the CM-5 [8]. In consequence, we have better absolute serial and parallel run times on the majority of the available polynomial benchmarks. Our uniprocessor and 10-processor times are better than those of the message-passing implementation for eight of the nine input sets whose timings are reported in [8]. For one input set, our 10-processor time is substantially slower than our 1-processor time, because our task-ordering heuristic happens not to work well for that input set. Our speedups are comparable to those in [8] for several of the benchmarks, but smaller for the other benchmarks because of our faster uniprocessor times.

As shown in Figure 2, the line count for the SAM version is not much larger than the line count of the serial version. The implementation of the distributed set abstraction accounts for most of the extra lines. On the other hand, the message-passing implementation has many more lines than the serial version, mainly because it implements an application-specific form of caching and consistency based on invalidation.

4.4 Parallelization and Communication Costs

In Figures 10 and 11, we give statistics on the parallelization and communication costs for 32-processor runs of our applications. Figure 10 displays the average values of the overheads, while each entry in Figure 11 has both an average value and a range over all the processors. In Figure 10, we have also included a segment (labeled *Application time*) that indicates the amount of work which would be done by each processor if there was perfect speedup (i.e. if each processor did 1/32 of the work done by the serial algorithm). The overheads divide naturally into those associated with the parallel algorithm (idle time), the communication hardware (message and stall time), and the software shared memory layer (address translation and pack/unpack time):

- *Idle time* indicates the percentage of time that a processor was idle because of lack of work.
- *Message time* is the total time spent sending messages and responding to incoming messages (e.g. requests for a data item).
- *Stall time* is the time spent waiting for data from a remote processor, excluding time that is spent serving incoming messages.
- *Address translation time* is the percentage of time spent in the SAM system ensuring that a copy of a data item is available and determining the address of the local copy. This time includes the cost of a hash table lookup and LRU management of the cache of data items.
- *Pack/unpack time* is the total time spent packing and unpacking messages and has been separated out from the "message time". Packing and unpacking is necessary because SAM allows each data item to be an arbitrary non-contiguous data structure (connected by pointers).

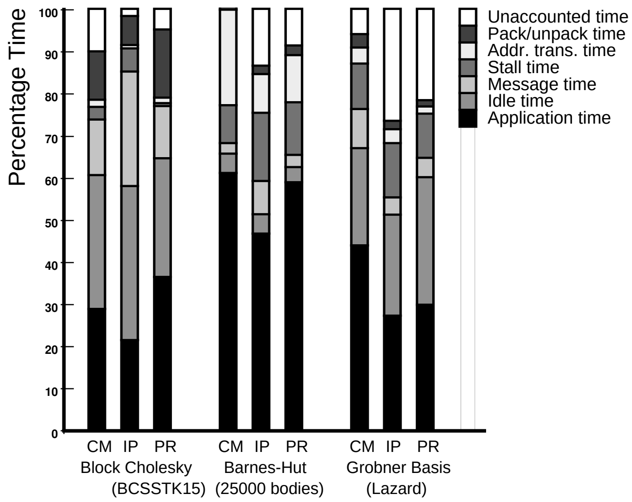


Figure 10: Parallelization and Communication Costs for 32-processor Runs on the CM-5 (CM), iPSC/860 (IP) and Paragon (PR)

In Figure 10, we have also shown the time that is not accounted for in any of the above categories. The biggest component of this unaccounted time is the extra computing done by the parallel application that is not done by the serial application. Some examples are the extra work done in the parallel Barnes-Hut algorithm to determine the appropriate partitioning of bodies across processors, and the extra work done in the parallel Grobner basis algorithm because the basis grows larger in parallel runs. The unaccounted time also covers some uncertainties in measuring the overheads.

Block Cholesky (BCSSTK15)	idle time (%)	message time (%)	stall time (%)	addr. trans. time (%)	pack/unpack time (%)
CM-5	31.8 (1.9-57.0)	13.2 (9.1-22.3)	3.0 (0.0-6.8)	1.7 (1.1-2.1)	11.5 (8.6-14.4)
iPSC/860	36.6 (15.4-55.4)	27.2 (17.0-38.9)	5.5 (0.0-12.6)	0.8 (0.6-1.0)	6.9 (5.1-8.5)
Paragon	28.2 (4.8 - 53.5)	12.4 (7.4-19.2)	0.7 (0.0-4.1)	1.3 (1.0-1.9)	16.2 (8.5-21.4)
Barnes-Hut (25000 part.)					
CM-5	4.6 (0.6 - 8.8)	2.5 (1.9-5.8)	9.0 (5.2-11.0)	22.7 (21.8-26.2)	2.0 (1.5-6.8)
iPSC/860	4.6 (0.2-10.6)	7.9 (6.1-13.8)	16.2 (4.7-18.6)	9.2 (7.9-10.2)	2.0 (1.3-2.4)
Paragon	3.6 (0.0-10.8)	2.9 (2.4-3.4)	12.5 (5.5-14.9)	11.2 (9.7-12.4)	2.3 (1.5-2.8)
Grobner Basis (Lazard)					
CM-5	23.1 (4.6 - 40.4)	9.3 (7.1 - 16.1)	10.8 (5.7 - 17.9)	3.8 (1.7 - 5.8)	3.2 (2.5 - 5.1)
iPSC/860	24.0 (0.0 - 54.9)	4.1 (2.0 - 13.0)	12.9 (6.6 - 19.4)	3.3 (2.8 - 4.4)	2.0 (1.6 - 4.6)
Paragon	30.3 (0 - 54.9)	4.6 (2.5 - 24.3)	10.5 (6.6 - 17.6)	1.7 (1.3 - 2.2)	1.5 (1.0 - 2.6)

Figure 11: Parallelization and Communication Costs for 32-processor Runs

All of the statistics are only approximate, since the timer calls used to measure the overheads may perturb the execution of the applications.[5] In addition, the message times given on the iPSC/860 and Paragon are only a lower bound, since much of the process of transmitting and receiving messages is handled via interrupts in system routines and are not instrumented.

We can make a number of general observations from the data. First, for applications with complex data structures and irregular parallelism like the ones we have used, the limited parallelism inherent in the underlying parallel algorithms can lead to large amounts of idle time that dominate other hardware and software overheads. In the block Cholesky application, the limited parallelism results from long critical paths and the static placement of tasks; in the Grobner basis application, the limited parallelism is due to difficulties in achieving good load balance as variable-sized tasks are dynamically created and distributed across processors.

For applications that must communicate a lot of data between processors, message sending and receiving can constitute a large fraction of the run time, as illustrated by the block Cholesky application. These overheads are largely governed by machine characteristics such as network bandwidth and can also limit performance regardless of other overheads. Note the high percentage of message time in the block Cholesky application for the iPSC/860, which has the least bandwidth of all the machines. The time for packing and unpacking complex objects (the blocks) is also significant in the block Cholesky application, because of the large amount of data communication. Much of this message handling time may be eliminated (or potentially offloaded to a message co-processor, as in the Paragon) with more efficient support for handling messages on newer machines.

[5]The instrumentation adds at most 11% to the execution time of all runs.

The stall time is a function of both the parallel algorithm and machine characteristics. There are significant stall times in the Barnes-Hut algorithm, because of contention for nodes as the tree is built and frequent accesses to remote nodes. Similarly, the Grobner basis algorithm has significant stall times because of contention for the shared basis set. The percentage of stall time is larger in the iPSC/860 than in the other machines likely because of the higher cost of sending messages and the lower network bandwidth of the iPSC/860.

Finally, for applications with finer-grain access to shared data, the software address translation overhead can be significant. The SAM software overhead in the Barnes-Hut algorithm is substantial, especially for the CM-5, where we have not used tree blocking. However, because it depends only on the number of shared accesses and not on the amount of communication, software address translation is a fixed percentage overhead on each processor and does not affect the scalability of the parallel application.

In summary, the scalability of parallel applications is principally determined by the parallel algorithm used. The speedups of the block Cholesky and Grobner basis applications are limited by the characteristics of their underlying parallel algorithms. The Barnes-Hut application has good scalability because of its inherent parallelism and extensive locality of reference. SAM applications with fine-grain access to shared data, such as the Barnes-Hut application, may have significant software overhead in the form of address translation. The speedups of parallel applications can also be limited by machine characteristics such as network bandwidth.

5 Evaluation of Communication Optimizations

In this section, we evaluate the usefulness of several aspects of SAM that allow for optimizing communication by measuring their effects on some of the applications we have described.

5.1 Caching

We are able to evaluate the usefulness of caching in the SAM system by measuring the performance of our applications when there is no caching (i.e. each object must be fetched from the processor on which it was created each time it is accessed). Figure 12 shows the improvements in 32-processor run times when caching is added. For reference, it also includes the serial times for the same applications. These figures do not include the pushing and chaotic optimizations of sections 5.3 and 5.4.

		32-processor time		
Block Cholesky (BCSSTK15)	serial time	without caching	with caching	factor improve.
CM-5	27.6s	4.13s	3.39s	1.22
iPSC/860	23.0s	5.14s	3.94s	1.30
Paragon	18.8s	2.62s	2.18s	1.20
Barnes-Hut (25000 bodies)				
CM-5	794.8s	618.5s	42.3s	14.6
iPSC/860	490.2s	2643.5s	42.4s	62.3
Paragon	402.8s	743.9s	27.2s	27.3
Grobner Basis (Lazard)				
CM-5	29.7s	50.60s	3.36s	15.1
iPSC/860	23.3s	87.25s	3.94s	22.1
Paragon	17.4s	45.56s	3.08s	14.8

Figure 12: Caching Performance for 32-processor Runs

The figures show that caching is important for all three applications, especially the Barnes-Hut and Grobner basis applications, which show a large amount of locality between the references of tasks executing on the same processor. The block Cholesky application does not show as much inter-task locality and benefits less from the caching provided by SAM, since any particular remote block may be used only a small number of times to update local blocks. However, the block Cholesky application has significant intra-task locality in block update operations themselves, which is already exploited well by the hardware caches of the nodes.

Many recent scientific algorithms use complex data structures and dynamic task placement and can benefit substantially from dynamic caching of remote data. Often, such caching functionality is implemented in an application-specific way in message-passing programs (for example, in [8]), at the cost of much programmer effort. SAM provides caching as a common functionality that is reusable in many applications and automatically exploits the locality of reference in these programs.

5.2 Synchronization

In a typical shared object system based on an imperative shared memory model, programmers express synchronization separately from data access. These explicit synchronization operations result in extra messages in addition to the messages used to transmit actual data. In contrast, SAM data access primitives directly express the mutual exclusion and producer/consumer data relationships in parallel program. No synchronization other than the synchronization implicit in shared data access is necessary to ensure correctness. For instance, during the parallel, post-order modification of the oct-tree in the Barnes-Hut algorithm, all synchronization occurs appropriately as nodes are accessed. In an imperative shared memory system, additional locks and flags would be necessary at each node to ensure that nodes are accessed in the proper order.

Figure 13 attempts to quantify the synchronizations that would be necessary for our applications in an imperative shared object system. In the first column, we show the number of barriers used in shared-memory implementations of our applications, and in the second column we show the total number of accesses to shared data objects. In the last two columns, we give the number of accesses (and percent of total shared accesses) that, in an imperative shared object system, would require extra synchronization not handled by the barriers. These figures are determined by classifying the types of accesses in runs of the SAM versions of the applications. The numbers show that there are a large number of synchronizations in our applications that would incur extra communication in imperative shared object systems.

Our applications include several common synchronization patterns which cannot be handled via barriers. For example, the producer/consumer synchronization necessary for the post-order tree modification in the Barnes-Hut algorithm is similar to the synchronization of a tree-based reduction, which occurs in many parallel programs. By combining synchronization with data access, SAM makes these important types of communication patterns more efficient.

5.3 Pushing Data

SAM provides a mechanism for sending values to remote processors that may require them. This "pushing" mechanism has the advantage of eliminating the latency for access to data if the pushed data item arrives at a processor before the processor requires the data. We were able to use this pushing mechanism in both the Barnes-Hut algorithm and in the block Cholesky application to improve performance. In the Barnes-Hut algorithm, we push the first few levels of the oct-tree to all processors after these levels have been

	barriers	total shared data accesses	est. non-barrier synchronizations	
			prod/cons	mutual excl
Barnes-Hut (25000 bodies)	7	14649035	11210 (.08%)	27463 (.19%)
Block Cholesky (BCSSTK15)	2	93093	13197 (14%)	0
Grobner (Lazard)	2	1147680	0	17301 (1.5%)

Figure 13: Number of Synchronizations in 32-processor Runs

modified, since all processors will likely access the top part of the tree. In the block Cholesky algorithm, we push completed blocks of the matrix to (exactly) the processors that will access them, because a major limiting factor in the run time is the critical path of block dependencies. Anything that reduces this critical path by reducing latencies for accessing blocks improves the run time. Figure 14 shows the improvements in 32-processor run times when the push and chaotic (see Section 5.4) optimizations are used (with caching). In both applications, the push optimizations are simple additions to the code, and the pushes produce a substantial improvement in the run time of the computations where they are used. However, since the push optimizations are only used in certain phases of each application, the effects on the overall run times are lower.

5.4 Chaotic Access

In a variety of scientific applications, it is possible to use chaotic accesses to shared data to improve parallel performance, while still producing correct results. In the Barnes-Hut algorithm, we use chaotic accesses when traversing the tree to determine where an item should be inserted; the properties of the oct-tree allow this optimization as long as an exclusive access is used when a potential insertion point has been reached. In the Grobner basis algorithm, we use chaotic accesses in some references to the head and tail pointers of the shared polynomial basis. In Figure 14, we show the improvement in run times for 32-processors runs when chaotic accesses are used. (When not using chaotic access, we allow copies of data items to be cached, but force these copies to be invalidated when the items are modified.) The Barnes-Hut improvements are small but significant, because the chaotic optimization applies only to the tree building phase of the algorithm, not the main force calculation phase. The results show that significant performance improvement is possible by using application-specific knowledge to determine when accesses with relaxed consistency can be employed.

6 Conclusion

In this paper we have presented the design and evaluation of a shared object system for distributed memory machines called SAM. SAM is a portable run-time system that provides a global name space and automatic caching of shared data, but also provides the programmer with the ability to optimize communication when necessary. To evaluate SAM, we have used it to parallelize several complex scientific algorithms, each of which consists of thousands of lines of code. Programming these applications in SAM is significantly easier than writing message-passing code and provides portability across machines.

Our results show that the various communication optimizations provided by SAM are important for obtaining good performance results on these complex applications. SAM provides generic caching functionality that can be reused across applications and therefore simplifies programming effort. This caching is essential in a variety of complex applications for getting good performance by exploiting the dynamic locality of reference. By tying synchronization to data access, SAM improves communication efficiency in applications with complex synchronization patterns that cannot be managed via barriers. Our results also show that the use of “pushes” and chaotic accesses, when appropriate to the application, can provide significant performance improvements with minimal effort. We believe that software distributed shared memory system must address the problem of high communication overhead by providing flexible mechanisms for optimizing communication.

We find that the performance results achieved using the SAM system depend on a variety of factors. For the complex scientific algorithms we used, the largest factor can be the limited parallelism available in the parallel algorithms. Performance is also significantly affected by the limited communication capabilities of current distributed memory machines, which will improve in future machines. The amount of software overhead is a function of the granularity of access to shared data and is acceptable for moderately-grained applications. Overall, we found that SAM significantly eased programming of our applications and allowed us to achieve good performance for these applications on a variety of platforms.

Availability

The SAM system is available via anonymous FTP at `suif.stanford.edu` in `/pub/sam` and via the World Wide Web at `http://suif.stanford.edu`.

		with caching only	with pushes	percent speedup	with chaotic	percent speedup	with pushes and chaotic	percent speedup
Barnes-Hut (25000 bodies)	CM-5	42.3s	42.0s	1%	41.5s	2%	40.7	4%
	iPSC/860	42.4s	36.2s	17%	38.1s	11%	32.9	29%
	Paragon	27.2s	25.6s	6%	24.5s	11%	21.4	27%
Block Cholesky (BCSSTK15)	CM-5	3.39s	3.20s	6%	NA	NA	NA	NA
	iPSC/860	3.94s	3.52s	12%				
	Paragon	2.18s	1.66s	31%				
Grobner (Lazard)	CM-5	3.36s	NA	NA	1.98s	70%	NA	NA
	iPSC/860	3.99s			2.68s	49%		
	Paragon	2.54s			1.83s	39%		

Figure 14: Effects of Optimizations on 32-Processor Runs

Acknowledgments

We thank Ed Rothberg for his block Cholesky code, J.P. Singh for his DASH Barnes-Hut code, and Soumen Chakrabarti for his Grobner basis code. Evan Torrie ported the Barnes-Hut message-passing code to run on the iPSC/860. Some of the performance numbers were obtained using the CM-5 at Berkeley, which is supported by National Science Foundation Infrastructure Grant number CDA-8722788. The IBM SP1 was donated by IBM Corporation. We also thank the referees for comments that helped improve the paper.

References

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), Mar. 1992.

[2] J. E. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON 1993*, Mar. 1993.

[4] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8):930–945, Aug. 1988.

[5] R. D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, Nov. 1992.

[6] B. Buchberger. Grobner Basis: an Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Company, 1985.

[7] N. Carriero. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, Department of Computer Science, 1987.

[8] S. Chakrabarti and K. Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Proceedings of the Fourth ACM/SIGPLAN Symposium on Principles and Practices and Parallel Programming*, pages 169–179, May 1993.

[9] K. M. Chandy and C. Kesselman. Composition C++: Compositional Parallel Programming. In *Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 124–144, Aug. 1992.

[10] K. M. Chandy and S. Taylor. The Composition of Concurrent Programs. In *Proceedings of Supercomputing '89*, Reno, Nevada, Nov. 1989. ACM.

[11] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems*, pages 147–158, Dec. 1989.

[12] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[13] M. J. Feeley and H. M. Levy. Distributed Shared Memory with Versioned Objects. In *1992 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1992.

[14] J. T. Feo, D. C. Cann, and R. R. Oldeheoft. A Report on the SISAL Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, Dec. 1990.

[15] J.B. Carter and J.K. Bennett and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[16] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared

Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[17] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *Computer*, 25(3):63–79, Mar. 1992.

[18] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages II 94–101, Aug. 1988.

[19] R. G. Minnich and D. J. Farber. Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory. In *Tenth International Conference on Distributed Computing Systems*, pages 468–475, June 1990.

[20] R. S. Nikhil. The Parallel Programming Language Id and Its Compilation for Parallel Machines. *International Journal of High Speed Computing*, 5(2):171–223, June 1993.

[21] K. Pingali and K. Ekanadham. Accumulators: New Logic Variable Abstractions for Functional Languages. *Theoretical Computer Science*, 81(2):201–221, Apr. 1991.

[22] U. Ramachandran, M. Yousef, and A. Khalidi. An Implementation of Distributed Shared Memory. *Software - Practice and Experience*, 21(5):443–464, May 1991.

[23] E. Rothberg and A. Gupta. An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization. In *Proceedings of Supercomputing '93*, pages 503–512, Nov. 1993.

[24] D. J. Scales and M. S. Lam. An Efficient Shared Memory System for Distributed Memory Machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[25] J. P. Singh. Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors. Technical Report CSL-TR-93-565, Stanford University, Mar. 1993.

[26] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[27] M. Warren and J. Salmon. An O(NlogN) Hypercube N-body Integrator. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages II 971–975, Jan. 1988.

[28] W. Weihl et al. Prelude: A System for Portable Parallel Software. Technical Report MIT/LCS/TR-519, MIT, Oct. 1991.