An Efficient Shared Memory Layer for Distributed Memory Machines

Daniel J. Scales and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Computer Systems Laboratory Technical Report CSL-TR-94-627

Abstract

This report describes a system called SAM that simplifies the task of programming machines with distributed address spaces by providing a shared name space and dynamic caching of remotely accessed data. SAM makes it possible to utilize the computational power available in networks of workstations and distributed memory machines, while getting the ease of programming associated with a single address space model. The global name space and caching are especially important for complex scientific applications with irregular communication and parallelism.

SAM is based on the principle of tying synchronization with data accesses. Precedence constraints are expressed by accesses to single-assignment values, and mutual exclusion constraints are represented by access to data items called accumulators. Programmers easily express the communication and synchronization between processes using these operations; they can also use alternate paradigms that are built with the SAM primitives. Operations for prefetching data and explicitly sending data to another processor integrate cleanly with SAM's shared memory model and allow the user to obtain the efficiency of message passing when necessary.

We have built implementations of SAM for the CM-5, the Intel iPSC/860, the Intel Paragon, the IBM SP1, and heterogeneous networks of Sun, SGI, and DEC workstations (using PVM). In this report, we describe the basic functionality provided by SAM, discuss our experience in using it to program a variety of scientific applications and distributed data structures, and provide performance results for these complex applications on a range of machines. Performance results for applications of such complexity have not typically been available for most other distributed shared memory systems. Our experience indicates that SAM significantly simplifies the programming of these parallel systems, supports the necessary functionality for developing efficient implementations of sophisticated applications, and provides portability across a range of distributed memory environments.

1 Introduction

Software reuse and portability are two prerequisites to the success of developing a substantial software base for parallel machines. To achieve reuse and portability, it is important to draw upon our experiences in parallel software development to identify useful abstractions that can be generalized to work for many different applications. Once these abstractions are encapsulated either as compiler optimizations or run-time support, these ideas can then be reused by new applications in a portable manner across different machine architectures. We envision that there will be a layered software architecture for parallel machines, consisting of a set of well-established interfaces at different levels of abstraction. Programmers can get access to the machines at different levels according to the need of their applications.

Run-time libraries such as PVM [21], which hide the variations in the message-passing protocols of different hardware, have been shown to be very useful. While its level of abstraction is low, PVM has been extensively used by application builders and language implementors to achieve portability. This message passing interface serves as an excellent foundation to this layered software architecture design. This paper builds upon the message-passing layer and proposes the next higher level of abstraction. This level is especially useful to support applications where the parallel structures are more dynamic and the data accesses are irregular. Programming such applications directly using message passing is difficult. Our run-time system, called SAM, enables the programmer to use a common name space to refer to all shared data in distributed memory systems. SAM manages the details of locating and communicating shared data for the user. Moreover, it automatically exploits the locality of reference in programs by caching and replicating objects. Thus, the programmer need only concentrate on creating programs with high data locality; the system will automatically handle the rest.

SAM is designed to meet the goals of both programmability and efficiency. In this design, a program explicitly specifies how it accesses data, and the system automatically enforces the necessary synchronization and performs the necessary communication. For example, a reader of a value must wait until the value is produced. Commutative updates to an accumulator need not be ordered but must be mutually exclusive. Shared data are accessed and managed at the level of user-defined data types. Allowing global accesses to individual locations would render the implementation too inefficient. The SAM design avoids some of the efficiency problems associated with other distributed shared memory systems and provides the flexibility to optimize communication as necessary.

We demonstrate that SAM supports portability and code reuse by implementing SAM on a variety of parallel machine architectures and implementing a range of applications using this system. Figure 1 illustrates the various abstraction layers we used in developing our applications. SAM has been implemented on the CM-5, the Intel iPSC/860 and Paragon, the IBM SP1, and on heterogeneous networks of workstations. To minimize the implementation effort of SAM, PVM was used in our workstation implementation. Native message operations are used for all the other parallel machines for efficiency reasons. All programs using SAM primitives can run without modification on all these machines.

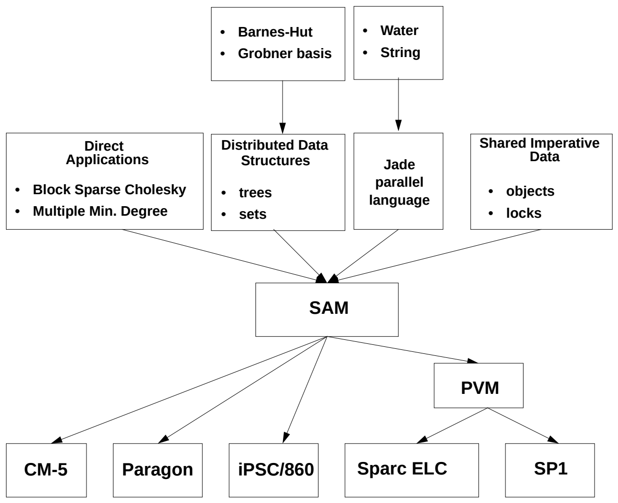


Figure 1: SAM Applications

We have implemented several applications directly in SAM such as the block Sparse Cholesky and the Multiple Minimum Degree algorithm. We have implemented a set of distributed data structures in SAM, which in turn were used in developing the Barnes-Hut n-body algorithm and Grobner Basis, a symbolic algebra application. We have also used SAM to implement Jade, a coarse-grain parallel programming language. Jade is a high-level programming language in that a programmer simply augments a sequential program with data usage specifications, and the system automatically parallelizes the code. By implementing Jade in SAM, all applications written in Jade can run on all machines that SAM runs on. Programmers may use Jade in the high-level coordination of parallelism, and drop down to use SAM on those performance-critical sections to gain control over the parallelization. The combination of Jade and SAM is an example of a multi-layered system that allows programmers to get access to different layers of abstraction, all within the same program. Finally, we have also implemented the shared imperative data model for programmers who would rather stay within the familiar imperative programming paradigm. SAM is the first distributed shared memory system that is portable across all these machines and is practical for such a variety of challenging applications.

In this paper, we describe the design of SAM, provide justification for our choices, and describe the results, in terms of both programmability and performance, of using SAM to build several types of parallel applications. (We include a brief discussion of its implementation in an appendix.)

2 SAM

Systems that provide a shared address space (in hardware or software) typically provide synchronization operations such as locks and barriers. Such operations are used by the applications programmer to establish orderings on events in his parallel program. Such orderings are important to ensure that locations in the shared address space are read and written in the intended sequence so that the proper data is communicated between processes. There are two kinds of data relationships that are enforced via synchronization operations. First, there are producer/consumer relationships where data is produced by one process and then used by one or more other processes. Second, there are mutual exclusion relationships that are necessary for computations (such as data reductions) that produce a number of updates to a memory location which require exclusive access but can occur in any order.

For synchronization operations to be effective, shared memory systems must guarantee that a modification to a storage location will be observable by all processors by the completion of the next synchronization operation associated with that storage location. However, most shared memory systems do not have any knowledge of the connection between memory locations and synchronization operations. Therefore, these systems instead guarantee that *all* modifications to memory are observable by all processors by the completion of the next synchronization operation. In particular, for shared memory systems with caching, this guarantee requires that all caches be made consistent (via invalidates or updates) at every synchronization operation. Were the system more informed about the association between synchronizations and data accesses, it could avoid much of this coherence traffic. Minimizing unnecessary communication is especially important for machines without direct coherence support in hardware.

SAM avoids the need for coherence communication by coupling synchronization with the data accesses directly. SAM provides shared memory operations that directly model the data relationships we described above, producer/consumer and mutual exclusion. SAM has two kinds of shared data, *values* and *accumulators*. Only the names of values and accumulators are globally visible; internal locations are specified as offsets into the shared objects. A process explicitly specifies how it accesses the shared data and the system performs the necessary synchronization and communication. (SAM deals only with the management and communication of shared data; data that is completely local to a processor can be managed by any appropriate method.)

Every value has a unique name and is immutable once created. A process specifies if it wants to create a value or to read a value; the writer of a value shares a producer/consumer relationship with all the readers of that value. Unlike values, an accumulator may be modified multiple times. SAM only guarantees that all modifications to the accumulator are mutually exclusive; processes are allowed to modify the accumulator in any order. A slight extension to this data relationship that has been shown to be valuable in speeding up parallel applications is the concept of a *chaotic* access. Chaotic algorithms are parallel computations in which processes do not need to use the most up-to-date data all the time. A computation that reads data that is slightly out-of-date but on the local processor may outperform computations that always read the most current data that resides on a remote processor. SAM allows the programmer to explicitly make this tradeoff by providing operations that read a recent (but not necessarily current) version of an accumulator.

Besides data access specifications, SAM also provides a set of memory management primitives so that the user can control the number of values in a system. Without such control, the use of single-assignment values can create an excessive number of data copies. This combination of single-assignment values and memory management primitives provides a better alternative to using an imperative memory model and paying for the coherence overhead on machines with nontrivial communication costs. Furthermore, a SAM programmer can use the memory management primitives to experiment with trading off greater memory use to increase parallelism.

The three basic ways of accessing shared data supported by SAM (producer/consumer, mutual exclusion, and chaotic) provide the flexibility required to build a variety of more complex paradigms for accessing data. In particular, it is easy to use SAM to implement locks and barriers and the familiar imperative data model. However, programmers can typically use knowledge of the high-level data relationships in a particular computation or data structure to communicate more efficiently using the basic SAM primitives.

2.1 Values

Every value generated in the system is given a distinct name when it is created and can be of any user-defined type. Values have a single-assignment semantics: a value is atomically created once its initial contents are set and is henceforth immutable. The code to initialize 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`. The system allocates the storage before the initialization, and makes the value immutable and available to other processes after the initialization code. The creator of the value must specify the type of the new value. 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 value. User-defined types may contain pointers and therefore need not be stored contiguously in memory. In heterogeneous environments, SAM also handles any necessary data conversion between dissimilar machines.

Code accessing a value is encapsulated by the primitives `begin_use_value` and `end_use_value`. If the value named is not present on the local processor, then SAM will determine where the value is located and fetch the value from the appropriate remote processor. It is also possible that the value has not yet been created. In either case, SAM will suspend the local process until the value has been created and can be brought to the local processor. SAM maintains local copies of values fetched from remote processors in the form of a cache. Any references to values that are cached locally will proceed immediately without any interprocessor communication by using the copy existing in the cache. Because all values have distinct names and are immutable, there is no consistency problem associated with maintaining this cache. Figure 2 shows some example code for creating and using a new value in two different processes.

task t1

```
Node *p;

p = begin_create_value(n, 1, Node, ...);
... initialize node ...
end_create_value(n, 1);
```

task t2

```
Node *p;

p = begin_use_value(n, 1);
... use node ...
end_use_value(n, 1);
```

Figure 2: Example (in C) of Creating and Accessing Values

SAM adopts a simple naming scheme that facilitates the representation of imperative data as a sequence of values. All data values are named by an ordered pair of integers, which are typically written as (object id, version id). In the common case of modeling imperative data, the "object id" can be used to specify a particular object, and the "version id" specifies a particular version of the object. Another value representing a new version of the object is created each time the contents of the object are modified. Values that are not related to any other value can be assigned a unique object id and a default version id. SAM provides primitives for creating globally unique object ids and sequences of version ids, as an aid to the process of choosing names.

Traditionally, synchronization is the trickiest part in writing a parallel program. SAM programs generally do not have any explicit synchronization operations. SAM programmers need only to focus on how to name the values. It is relatively straightforward for a programmer to derive a naming scheme from his mental model of the program's dynamic control structure. For example, in an iterative algorithm in which a new value is computed on each iteration, the names of the values can be based on the iteration number. A program with several phases can name the results of each phase according to the phase number. (The phase number can be dynamically assigned and need not be coded into the program). In Section 3, we will describe some of the details of naming data items in our example applications.

2.2 Accumulators

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, as long as mutual exclusion is ensured. Because of the mutual exclusion requirement, only one valid copy of an accumulator exists in the system at any given time. SAM automatically migrates the accumulator between processors as necessary and ensures that a process does not access the accumulator until mutual exclusion is obtained. Accumulators are named in the same way as values and can be of any user-defined type. Updates to an accumulator must be encapsulated by the SAM primitives `begin_update_accum` and `end_update_accum`. There are also primitives for creating accumulators directly or converting a value into an accumulator or an accumulator to a value.

SAM supports the idiom of chaotic computation via the `begin_read_recent_accum` and `end_read_recent_accum` operations, which provide 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. The recent value can only be read, not updated. However, a "recent" value may be all that is necessary for some kinds of computation, and the use of chaotic accesses can increase available parallelism by reducing synchronization and communication. Chaotic accesses provide a way for the SAM programmer to build a "relaxed" consistency model into his application based on his knowledge of the higher-level semantics of the program. Such chaotic accesses are especially useful when encapsulated in operations on distributed data structures; we will discuss some examples in Section 3.

2.3 Memory Management

In addition to allocating the memory used for creating the initial copy of a value, SAM automatically manages the memory used for local copies of values that have been fetched from a remote processor. Because each of a process' accesses to a value is delimited by the `begin_use_value` and `end_use_value` primitives, SAM can determine when all processes have finished accessing a local copy of a value. If necessary, the storage for a local copy can be immediately freed up when the local copy is no longer being accessed. However, SAM maintains a main-memory cache of local copies that have been recently accessed, in order to reduce non-local communication in the future. SAM automatically frees up local copies as necessary when the cache memory becomes filled. However, 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 user of SAM provides this information by explicitly indicating when all accesses to the value have occurred or specifying the number of users of the value. In the latter case, SAM maintains at least one copy of the value in the system until the indicated number of accesses have occurred.

SAM also provides a mechanism to reuse the local storage of values and eliminate unnecessary copying of data. Values are frequently created as modifications of other values. In such a situation, it is highly desirable, if possible, to use the storage of the first value in creating the second value. In this way, copying and memory management overhead is avoided. SAM supports this optimization via the `rename_value` primitive, which essentially allows an old value to be renamed and reinitialized to create a new value.

Imperative data objects are easily represented in SAM via a sequence of values that represent the changing contents of each object. A SAM implementation of imperative memory can potentially eliminate all anti-dependences between processes, because all versions of an imperative object are individually named and can be accessed independently. Even if the programmer chooses to keep only one version of an imperative object on each processor, a processor can create a new version of an object while another processor is still reading an older version. In contrast, traditional shared memory systems require that data be consistent across all processors.

2.4 Low-level Control of Communication

An important mechanism for improving the efficiency of systems is support for asynchronous communication. SAM provides the capability to fetch values and accumulators asynchronously; that is, the process does not stall if the fetch cannot complete immediately. 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. For asynchronous access to an accumulator, the process is not notified until the accumulator has been fetched to the local processor and mutual exclusion has been obtained.

Another method for minimizing the communication latency 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 receive a piece of data (via `begin_use_value`) 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. There is no requirement to receive messages (values) in the order in which they are sent.

3 Experience

We have implemented SAM as a run-time library that runs on the CM-5 using Active Messages [22], on the Intel iPSC/860 and Paragon using the native messaging primitives, and on the IBM SP1 and heterogeneous networks of workstations using PVM [21]. Below we describe in more detail several systems we have built using SAM and give some performance numbers. Each of these different applications is programmable in a straightforward way using SAM, and all the applications run without modification across all platforms.

3.1 Block Cholesky

The block Cholesky application [19] does 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 block Cholesky algorithm potentially exposes more parallelism and asymptotically requires less (though still large) communication bandwidth than other column-oriented methods. The block Cholesky algorithm benefits substantially from dynamic caching, since each block may be used many times by a processor to update other blocks.

In the preprocessing phase of the algorithm, blocks are formed by partitioning the rows and the columns of the matrix in a way appropriate to the sparsity structure of the matrix. While the partitioning step is nontrivial, it takes very little time compared to the main computation. The following pseudo-code describes the basic algorithm for factoring matrix L (where the I, J, and K subscripts iterate over the non-zero blocks of the matrix):

```
1. for $K$ = 0 to $N-1$ do
2.     $L_{KK}$ = Factor($L_{KK}$)
3.     for $I$ = $K+1$ to $N-1$ with $L_{IK} <> 0$ do
4.         $L_{IK}$ = $L_{IK} L_{KK}^{-1}$
5.     for $J$ = $K+1$ to $N-1$ with $L_{JK} <> 0$ do
6.         for $I$ = $J$ to $N-1$ with $L_{IK} <> 0$ do
7.             $L_{IJ}$ = $L_{IJ}$ - $L_{IK} L_{JK}^T$
```

The parallel algorithm involves executing the tasks represented by the computations on line 2, 4, and 7 above while respecting the necessary data dependences. The parallel algorithm includes an enhancement in which some parts of the matrix, called *domains*, are processed in a column-oriented fashion on a single processor, before the remaining blocks are processed.

Our implementation of block Cholesky is derived directly from a version for the DASH multiprocessor [14], which supports a shared address space with hardware caching support. Each individual block and domain 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. The first phase consists of a series of commutative updates (updates that can occur in any order) in line 7. When the last update is done, the contents of the block are finalized in lines 2 and 4 above. In the third and final phase, the data are only read to update other blocks in lines 5-7. 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 operands becomes available and is assigned to the processor that "owns" the block. The processor then accesses the second operand asynchronously. The system will supply the processor with the local copy of the requested value if one exists. If the value has not been created, or if the value needs to be fetched remotely, the system will handle the transfer in the background while the processor continues computing with other data.

In Figure 3, we give the parallel speedups of the block Cholesky algorithm for all machines for both a sparse matrix tk15.O and dense matrix d1000, all with reference to the corresponding 1-processor run, in which the matrix is factored efficiently as a single domain (with essentially no SAM overhead). In Figure 4, we give the corresponding absolute performances in megaflops. Because we simply recompiled the block Cholesky DASH code, we did not take full advantage of the floating-point capabilities of the individual node processors of the iPSC/860, Paragon, and CM-5. In the case of the iPSC/860 and Paragon, the compiler does not make use of the special dual-operation instructions that allow a floating-point add and floating-point multiply to be executed in parallel. Also, on the CM-5, we do not explicitly use the four vector units at each node to enhance floating-point performance. The SP1 achieves very good single-node performance without any modifications to the code. For comparison, we have also included the performance of the block Cholesky algorithm on the DASH shared-memory multiprocessor. The parallel speedups for all machines are fairly low for the sparse matrix, because of the relatively small size of the matrix, the limited inherent parallelism available in the algorithm, and the difficulty in load balancing. The speedup curves have very similar shapes on all machines, except that the performance of the iPSC/860 does not scale as well to 32 processors, because of bandwidth limitations. Similarly, the SP1 does not speed up as well even for small numbers of processors, because of the high single-node performance and limited bandwidth; however, these low speedups still yield impressive absolute performance. The results indicate that, with the software support provided by SAM, complex algorithms written for shared-memory machines can be readily adapted to run well on distributed-memory machines.

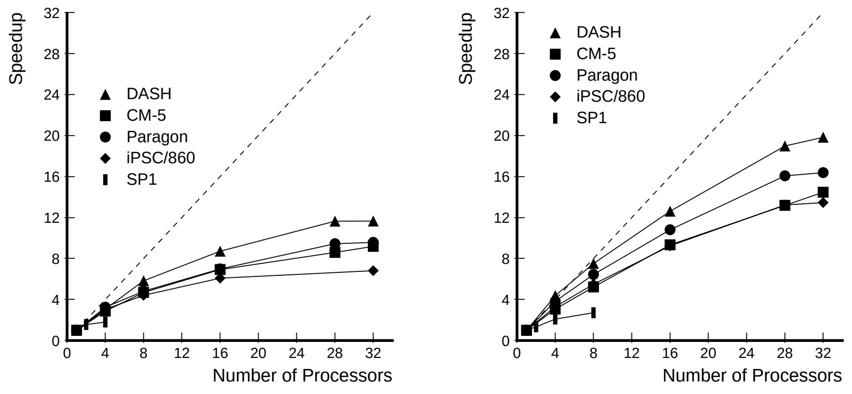


Figure 3: Block Cholesky Speedup for tk15.O (left) and d1000 (right)

3.2 An Oct-tree Library

We have built a simple, easy-to-use system in SAM for running applications that manipulate tree data structures on distributed memory machines. An example application is the Barnes-Hut algorithm [2], which is a fast algorithm for simulating the evolution of a system of astronomical bodies as they interact with each other via the gravitational force

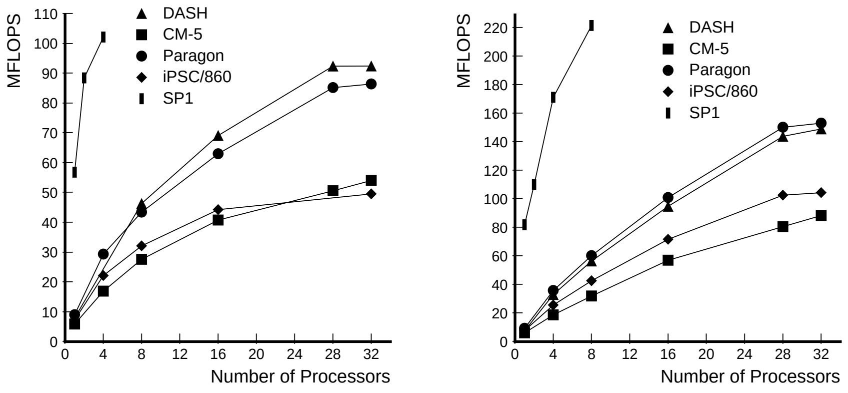


Figure 4: Block Cholesky Performance for tk15.O (left) and d1000 (right)

(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 algorithm uses a tree data structure called an oct-tree to summarize the gravitational effects of nearby groups of bodies, so that the force calculation for each body can be done more quickly. The location of a body in the tree is based on the spatial coordinates of the body. Figure 5 shows a particular configuration of bodies in space and the oct-tree that represents them.

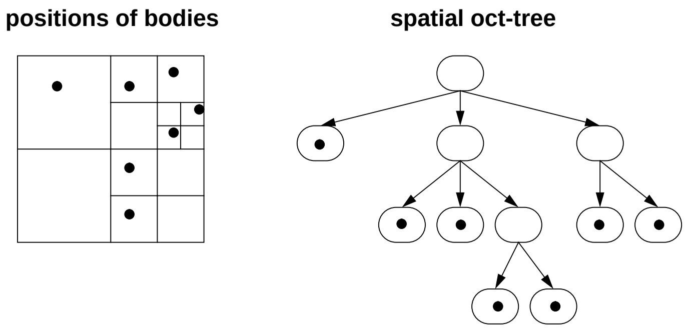


Figure 5: Oct-tree Representation of Bodies in Space

The oct-tree in the Barnes-Hut algorithm may be quite large and therefore cannot be replicated in full on each processor. The memory to hold the oct-tree structure cannot be statically allocated and partitioned across processors, since the structure of the oct-tree is complex and dependent on the input data. In addition, the parts of the tree that various tasks will access cannot be determined statically. 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. The programmer only needs to worry about doing this partitioning correctly; SAM automatically exploits this locality by caching recently accessed tree nodes on each processor.

We have built a library of functions for building and manipulating oct-trees on distributed memory machines. This library includes primitives for building an oct-tree, modifying the contents of the nodes in a pre-order or post-order traversal, reading the nodes of the tree in an arbitrary traversal, and destroying the tree. Tasks can access and modify nodes in the tree using these functions without regard to where the nodes were originally created or currently reside.

The library hides the naming and synchronization issues associated with building, traversing, and modifying the tree. Because the final structure of the oct-tree depends only on the spatial positions of the bodies, the library allows data to be inserted into the tree in parallel in any order. Nodes are therefore initially designated as accumulators while the tree is being loaded. To reduce communication while building the tree, the library uses 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 a typical shared memory system, it is impossible to express such chaotic accesses, and extra coherence communication is unavoidable. When all the nodes of the tree have been inserted, the nodes are converted to values. The library handles the naming of the values as the tree is accessed and modified. The library executes pre-order and post-order modifications of the tree in parallel; all synchronization occurs appropriately as nodes are accessed. In an imperative shared memory system, additional locks and flags at each node would be necessary to ensure that nodes are accessed in the proper order.

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. These blocks are determined dynamically by grouping together nearby nodes into a single block up to a certain maximum number. Figure 6 illustrates the blocks that might be chosen for a particular oct-tree. We have hidden the complexity of the blocking in the oct-tree library and allow 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 is sometimes used in bringing over nodes that are never accessed. In addition, the parallelism in some of the pre-order and post-order tree traversal phases is significantly decreased, because only one processor can modify a block at a time.

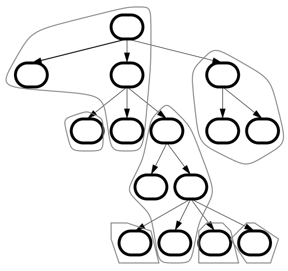


Figure 6: Blocking of Nodes in an Oct-tree

We have written a parallel version of the Barnes-Hut algorithm using the tree library that is adapted directly from the original serial algorithm, with the addition of a phase for partitioning the force calculation for good locality and load balancing [20]. Figure 7 shows speedup and absolute run-time curves of our parallel version running on the CM-5, iPSC/860, Paragon, and SP1 for an n-body simulation of a highly irregular distribution of 25000 bodies. Speedup is measured relative to an efficient serial algorithm. Tree blocking is used for the runs on the iPSC/860, Paragon, and SP1, while tree blocking is not used for the runs on the CM-5. The communication costs on the CM-5 are small enough that tree blocking does not significantly improve performance for small numbers of processors and makes performance worse for more than 32 processors. For comparison, we have also included the performance of an implementation of the Barnes-Hut algorithm on the DASH shared-memory multiprocessor. Though the curves show less than linear speedup for the distributed memory machines, the speedup is scaling with increasing numbers of processors. We expect the overall parallel performance to improve substantially with more efficient support for handling messages on newer machines. Though the SP1 shows lower speedup than the other machines (in part because of its high uniprocessor computation rate), it provides impressive performance for small numbers of processors.

A very finely tuned message-passing version of the Barnes-Hut algorithm has been implemented [23] and can achieve nearly linear speedup for simulations with a large enough number of bodies. This message-passing algorithm is much more complex and difficult to program than the original serial algorithm and is highly dependent on the way that the Barnes-Hut algorithm uses the oct-tree. By using SAM to program the Barnes-Hut algorithm, we have explored the tradeoff between the reusability and ease of programming of the tree library versus achievable performance.

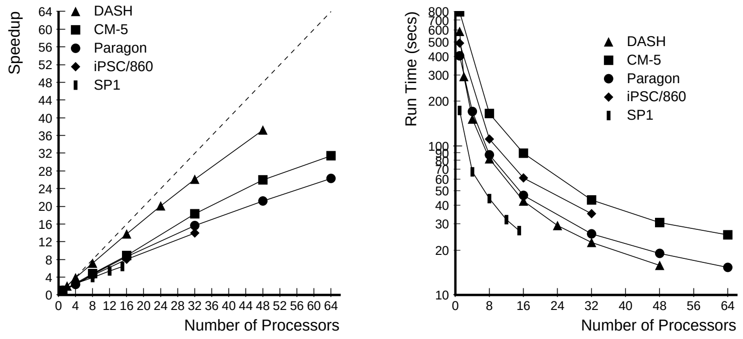


Figure 7: Barnes-Hut Speedup (left) and Times (right) for 25000-body Simulation

3.3 A Symbolic Algebra Application

One application area that can involve large amounts of highly irregular computation is symbolic algebra. We have used SAM to parallelize an important algorithm from symbolic algebra systems on distributed memory machines. The algorithm computes the Grobner basis [6] of a set of polynomials, which is used in solving systems of non-linear equations and determining 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. The two important data structures in the algorithm are `Basis`, the set of polynomials in the basis and `Pairs`, the list of pairs still to be examined. An overview of the algorithm is as follows:

```
Pairs = InitPairs(Basis);       /* Initialize Pairs to all possible pairs of the initial basis. */
while (size(Pairs) != 0) {
    pair = choose(Pairs);       /* Heuristically choose a pair */
    S = Spoly(pair);            /* Compute the S polynomial */
    R = Reduce(S, Basis);       /* Reduce the S polynomial wrt the basis */
    if (R != 0) {
        /* Add the new polynomial to the basis, and all possible pairs
         * between it and current members of the basis to Pairs. */
        Pairs = UpdatePairs(Pairs, Basis, R);
        Basis = union(Basis, R);
    }
}
```

In this code, `Spoly` computes a function of two polynomials called the S-polynomial, and `Reduce` reduces as much as possible one polynomial by a set of other polynomials.

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 in attempting to reduce its current S-polynomial.

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 also provides a “chaotic iterate” which iterates over the recent, but not necessarily latest contents of the set. (That is, the chaotic iterate may not see some of the most recent additions to the set.) In our parallel implementation, we create a separate task to handle each new pair. Therefore, the pair list is implicitly represented by tasks that have been created but not yet run.

Parallel Grobner basis implementations are non-deterministic, since pairs (tasks) distributed across processors may execute in differing orders and cause different polynomials to be added to the basis set. They use a priority scheme to order the executions in such way to maximize the effectiveness of the non-deterministic operations. Priority functions also have the effect of minimizing non-determinism and thus make it possible to compare the speeds of different computations on the same data set.

Figure 8 shows the speedups of our Grobner basis implementation on the CM-5 for a representative sample of input polynomial sets. Speedups are determined by running the algorithm three times for a particular input set and number of processors, and dividing the average run time by the run time for the parallel algorithm on one processor. The smaller input sets (gupta, trinks1) have limited parallelism and therefore do not scale well to many processors. Speedups for some of the larger input sets (Lazard, katsura4) scale fairly well, while the speedup is more limited for other sets.

The speedup is not limited by the system overhead but is inherent in the parallel algorithm. The parallel execution performs more work than the sequential counterpart; the basis set may become larger in a parallel execution since computation on each processor is performed without the knowledge of the polynomials that are about to be added to the basis set by other processors. To quantify the inefficiency of the algorithm, we compare the parallel runs with serial runs that perform exactly the same computation. The normalized speedup curves in Figure 8 measure the speedup of each parallel run with respect to a run on one processor doing identical work. (These figures are not exact because of the instrumentation overheads, but we estimate the error to be within 3%.) The normalized speedup curves of the larger polynomial sets tend to scale more consistently.

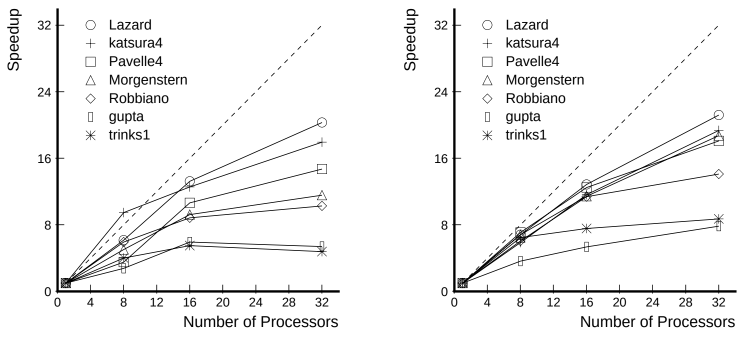


Figure 8: Grobner Speedups on CM-5, Real (left) and Normalized(right)

The parallelization of the Grobner Basis algorithm by Chakrabarti [8] on the CM-5 was written directly in message-passing primitives. The usefulness of SAM can be illustrated by approximate counts for the lines of code in different implementations of the algorithm. All versions link in a multi-precision arithmetic and a polynomial arithmetic library. The sequential algorithm from which the parallel implementations are derived consists of about 3700 lines of code. The SAM parallel implementation is similar to the sequential version and is 300 lines more; the extra code mainly implements the set abstraction and defines necessary packing functions for sending polynomials between machines. The message-passing CM-5 implementation is 2000 lines longer than the sequential version. The extra lines of code implement an application-specific form of caching and consistency based on invalidation. In contrast, SAM provides generic caching functionality that is reusable for a variety of applications.

Our implementation appears to have better heuristics for setting the priority of tasks than the CM-5 implementation. In consequence, we have better serial and parallel run times on the majority of the available polynomial benchmarks;

in addition, our implementation has less variation in parallel run times due to non-determinism. Some runs have superlinear speedup, but this happens less often than with [8], because our serial times are already quite good. Finally, our implementation runs immediately on the iPSC/860, Paragon, and SP1, as well as the CM-5.

3.4 Implementation of the Jade Language Using SAM

Jade [18] is a parallel programming language for exploiting coarse-grain concurrency in sequential, imperative programs. Jade provides the convenience of a shared memory model by allowing any task to access *shared objects* transparently. Jade programmers augment their sequential programs with constructs that decompose the computation into tasks and declare how tasks access shared objects. The Jade implementation dynamically interprets this information to execute the program in parallel while preserving the sequential semantics – if there is a data dependence between tasks, tasks run in the same order as in the sequential execution. The structure of parallelism produced by the Jade program is a directed acyclic graph of tasks, where the edges between tasks are the data dependence constraints. Because the constructs for declaring data accesses are executed dynamically, this task graph can be dynamic and can therefore express data-dependent concurrency available only at run-time.

In our implementation of Jade using SAM, a Jade object is represented by a sequence of values; a value representing a new version of the object is created whenever a task writes the object. The implementation minimizes copying and memory usage by reusing the storage of an old version whenever creating a new version. The important point is that the data usage information provided by the Jade constructs is sufficient to determine exactly which versions of each object a task will access and when a task creates a new version of an object. Thus, conceptually, each task can fetch its required versions using SAM primitives and execute; the synchronization implicit in a use of a value ensures that the ordering constraints between tasks are respected and the task does not run until all the versions it requires have been created. Our implementation optimizes this process by fetching the required versions asynchronously and scheduling the task only when all the versions have become available. The latency of the fetches are hidden via concurrency if there are other tasks on the processor which can run while the fetches are occurring. Because different versions of Jade shared objects are explicitly represented, it is possible for a task on one processor to be reading one version of an object, while a task on another processor is creating a new version of the object from the older version.

We have built an implementation of Jade using SAM, and have also implemented several applications in Jade. These include a program from geophysics that uses seismic travel-time inversion to construct a velocity model of the geology between two oil wells, the Perfect Club benchmark program MDG that evaluates forces and potentials in a system of water molecules in the liquid state, and an application that simulates the interaction of electron beams with various solids using Monte Carlo techniques. Figure 9 gives the speedup curve for a parallelization of the MDG benchmark using Jade. The speedups shown are for a run involving 2197 particles on a collection of Sparc ELCs connected by Ethernet, a CM-5, a Paragon, and an iPSC/860. Our implementation of Jade performs essentially identically to a previous Jade implementation based directly on message-passing primitives, but is much simpler to implement and understand.

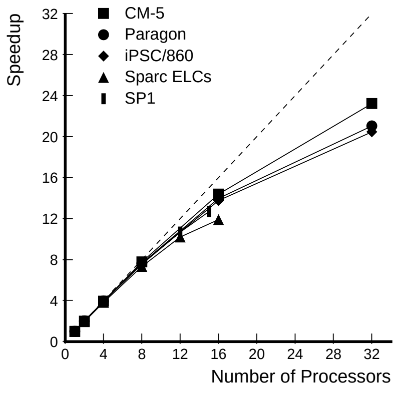
In addition, because our implementation is layered on top of SAM, the Jade programmer can program directly in SAM for critical sections of their application, while using Jade to manage the remaining communication and parallelism. The programmer may choose to use SAM primitives directly to optimize communication, manipulate complex distributed data structures more efficiently, or to exploit parallelism that cannot be expressed via the sequential semantics of Jade.

4 Discussion and Comparison

In this section, we justify some of our design choices in SAM and compare with related work.

4.1 Shared Memory Functionality

One fundamental choice in the design of SAM is to provide shared memory functionality on distributed memory machines. Applications written in a shared memory style often lose some of the efficiencies of pure message-passing



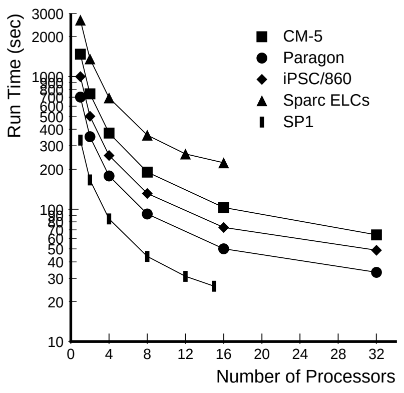


Figure 9: MDG Speedup (left) and Times (right) for Simulation of 2197 Particles

programs. Message-passing programs minimize communication by sending data via point-to-point messages between nodes. The latency of message sends is usually partially hidden by attempting to send messages before they are needed remotely. The design of SAM allows the efficiency of message passing to be achieved when necessary within the framework of a shared memory model. Tasks can send (“push”) a value to a remote processor to reduce latency and eliminate request overhead if that remote processor is likely to access the value. Also, data can be fetched asynchronously, thus allowing for the possibility of prefetching, pipelining reads of data, and/or hiding latency with concurrency. Therefore, SAM’s design not only provides the ease of use of a shared memory model, but it has the flexibility to achieve the efficiency of message passing when necessary.

Another important consideration is whether to provide dynamic caching of shared data. Some imperative distributed shared memory (DSM) systems, such as Amber [9], Prelude [24], and Split-C [10] implement a shared address space without automatic caching. The user must manage any local copies of remote data and make sure that they do not get out-of-date with respect to the current contents of the data. Amber and Prelude also provide mechanisms to access to shared objects by moving tasks to the processor containing the data. Imperative DSM systems that do implement replication of data [1, 3, 4, 15] in local storage require a consistency protocol to ensure that processors do not access the wrong version of a particular data object. The consistency protocol must send information to all processors that currently have a copy of an object when that object is modified. Some of these invalidate/update messages are unnecessary since a cached copy may never be referenced again. Another cost of consistency protocols is the round-trip latency associated with the required acknowledgment of invalidate or update messages. SAM provides dynamic caching of data, but does not have any such consistency problems. Because of their single-assignment property, values in SAM can easily be replicated for faster access without consistency problems. There are no consistency problems associated with accumulators, since only one process accesses an accumulator at a time (except for chaotic accesses, which don’t guarantee any consistency properties).

SAM supports a global address space at the level of user-defined types, like many other systems [1, 4, 5, 9, 24]. Such an approach provides maximum flexibility in expressing communication and avoids the problem of false sharing. In addition, because it has the type information necessary to do the proper data conversion, SAM can operate in a heterogeneous environment. In contrast, shared virtual memory (SVM) systems [3, 13, 15, 17] provide a flat, global address space. SVM uses the virtual memory hardware to implement a page-based coherence protocol. The page-fault hardware automatically detects accesses to remote or invalid pages, and the page-fault handler is used to move or copy pages from remote processors. The advantage of the SVM approach is that the shared memory abstraction is provided completely transparently to the user. However, this can also be a disadvantage, since the user cannot control when communication occurs. The fixed and comparatively large size of the page (the unit of coherence) increases the possibility of problems due to false sharing, especially for the dynamic data structures in some of the applications we have considered. Both Munin [3] and TreadMarks [13] attempt to minimize this problem by allowing multiple processors to write to a page and merging changes at the next synchronization point. SVM systems have also been mainly limited to homogeneous systems (but see [25]), because they typically lack the type information to do the proper conversions between machines with differing data representations.

4.2 Expressing Data Relationships

Our choice to provide primitives to express data relationships explicitly has a number of implications. The main result is that SAM primitives capture the high level information about the data relationships among processes in a parallel application. This high-level information allows for efficient implementation of the communication required by these relationships. First, these primitives tie the necessary synchronization and communication with the data usage. Therefore, the synchronization can be incorporated into the communication process, without adding any extra overhead. This contrasts with many other DSM systems, where the synchronization (locking) is separate from data access and may entail additional message overhead. Second, these primitives encompass both types of data relationships, precedence constraints and mutual exclusion. Many systems provide direct support for expressing one of these relationships, but not both.

Midway [4] ties data objects to locks, so that it can fold in synchronization operations with the necessary communication when modifying objects with mutual exclusion. Acquiring a lock causes the data associated with the lock to be transferred in the same message. However, Midway has no support for expressing precedence relationships; such relationships must be built up using additional locks and synchronization operations. The Linda system [7] provides distributed shared memory in the form of a global tuple space. Processes interact via operations that insert, read, and remove tuples from tuple space. These low-level operations can be used to implement both precedence constraints and mutual exclusion. However, compiler analysis is necessary to determine exactly what relationship is being expressed and choose the most efficient implementation of the operation. VDOM [11] is an object-based DSM system that allows for explicit naming of and access to successive versions of shared objects, and can therefore express precedence constraints. However, neither VDOM nor Linda has support for explicit memory management or communication optimizations.

The SAM primitives are intended to capture the data relationships of a program without unnecessarily introducing any extra constraints. In contrast, programs expressed using imperative shared memory often over-specify the data relationships in a program. In particular, by storing many values in a single location, an imperative program may introduce anti-dependences that limit the parallelism in a program. That is, a dependence may result between a write to a location and a preceding read of the location, even though no actual dataflow relationship exists. These anti-dependences are enforced by the imperative semantics, even in the case where the processes involved in the anti-dependence execute on different processors and therefore are actually operating on distinct copies of the data. These anti-dependences can be eliminated using SAM, because all values in the system are named separately.

A number of systems have supported the notion of chaotic access. Hutto [12] advocates the use of “slow memory” in distributed shared memory systems which allows the effects of writes to propagate slowly through the system. Agora [5] supports a shared 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 [16] allows a process to access an “inconsistent”, read-only copy of a virtual-memory page which is not kept consistent as the page is modified. Clouds [17] supports an operation for getting a copy of a segment that will not be kept coherent even if the contents of the segment are changed. We believe that we are the first to use such chaotic operations to attain higher performance in complex scientific algorithms.

Finally, as we showed in Section 3, the SAM primitives provide the precision to build a variety of data usage models. The SAM primitives can be used for emulating imperative shared memory, implementing a variety of distributed data structures, and modeling communication patterns unique to particular parallel languages or data structures.

5 Conclusion

In this paper we have presented the design of SAM, a distributed shared memory system that simplifies the process of programming complex applications in a distributed memory environment. SAM provides a portable layer of software which provides a useful set of functionalities to higher-level applications while hiding the details of the hardware. Data items in SAM are accessed via a shared address space and are automatically replicated and cached to exploit the locality of reference in programs. SAM is based on the principle of exposing all data relationships in parallel programs explicitly. Data relationships are cleanly expressed via access to two different kinds of shared data, values and accumulators. Values have a single-assignment semantics; access to values result in precedence constraints. Accumulators may be updated multiple times in any order, but mutual exclusion is enforced.

Our experience with the system suggests that SAM is versatile in supporting different parallel programming paradigms and that it greatly simplifies programming and provides portability. We demonstrated SAM's flexibility by using it in a variety of systems, including a parallel programming language, distributed data structures, and several irregular scientific algorithms. We have shown that it can be used to support the familiar model of imperative data, common idioms such as chaotic data accesses, and custom synchronization schemes made possible by high-level language semantics. Because SAM provides a global name space and automatic data caching, programming in SAM is much easier than using explicit message passing, and the programs are easier to understand and maintain. By sharing the same SAM software implementation, individual SAM applications have substantially fewer lines of code. We have implemented SAM on a variety of systems: the CM-5, the Intel iPSC/860 and Paragon, the IBM SP1, and a network of workstations. The same programs written using SAM can run without modifications on all these platforms. We have achieved good performance on these platforms on a variety of large, complex applications.

References

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

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

[3] John Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *Proceedings of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[4] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[5] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[6] Bruno 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 and D. Gelernter. Applications Experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming*, pages 173–187, July 1988.

[8] Soumen Chakrabarti and Kathy 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] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard 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, December 1989.

[10] D. E. Culler et al. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

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

[12] Phillip W. Hutto. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Tenth International Conference on Distributed Computing Systems*, pages 302–309, June 1990.

[13] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report Rice COMP TR93-214, Rice University, November 1993.

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

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

[16] Ronald G. Minnich and David 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.

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

[18] Martin C. Rinard, Daniel J. Scales, and Monica. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.

[19] Ed Rothberg and Anoop Gupta. An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization. In *Proceedings of Supercomputing '93*, pages 503–512, November 1993.

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

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

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[23] 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, January 1988.

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

[25] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. Technical Report CSRI-244, Computer Systems Research Institute, University of Toronto, September 1990.

A SAM Implementation

In this section we briefly discuss the details of our implementation of SAM that runs on the CM-5, Intel iPSC/860 and Paragon, and networks of workstations.

At any one time, many processors in the system may hold a copy of a particular value. However, the SAM implementation uniquely identifies one copy of the value as the *master copy*. The master copy is the copy of the value on the processor that originally created the value. Also, for each value, there is a distinguished processor, called the *directory server*, that handles requests for the value from processors that don't know where the master copy is. The identity of the directory server is based only on the name of the value, so any processor that knows the name of the value can determine its directory server. The directory server queues up requests for a value until the location of the master copy is known, then forwards requests to that processor.

On each processor, there is a *value table* that stores information on values in the system and is indexed by the name of the values. The information stored for each value includes such things as the current state of the value, the location of the master copy, queues of processes and processors that are waiting to access the value, and a pointer to any local copy of the value.

When a process attempts to access a value that is not locally available, the implementation adds the current process to the queue associated with the value and tries to fetch the value remotely (if it has not already done so). If the location of the master copy is known, then the value is requested from that processor; otherwise, a request is made to the directory server. When a copy of a value arrives at a processor in response to a request, all processes waiting for the value are notified that the value is now available locally. Similarly, when a value is created locally, all waiting processes are notified. In addition, the existence and location of the new value are announced to the directory server for the value.

The value table entry for each value also includes a reference count. If the reference count of a particular value drops to zero, then that value is placed on a "free" list. A value on the free list can still be accessed; if it is accessed, it is taken off the free list. However, whenever all available memory has been used up, the least recently used values on the free list are removed from the value table and their memory freed up so that it can be reused for other values. The reference count is maintained based on calls to `begin_use_value` and `end_use_value`. In addition, for the master copy, SAM uses information supplied by the user to ensure that the master copy is not freed until all possible accesses to the value have occurred.

Accumulators are implemented using an efficient, specialized protocol. The current version of an accumulator exists only on one processor at a time. When all waiting processes on one processor have updated the accumulator, it is passed on to the next processor that has processes waiting to update the accumulator. However, each processor keeps one "old" version of the accumulator which is used to satisfy chaotic read requests for the accumulator. Thus, when a processor passes an accumulator on to another processor, it actually keeps its copy of the current version of the accumulator, but marks the copy as old (since the accumulator will continue to be updated by other processes). There are a variety of strategies for determining the current location of an accumulator. In the simplest scheme, requests for the accumulator are sent to the directory server for the accumulator whenever the accumulator is not available locally. Periodically, the processor that holds the current version of the accumulator communicates with the directory server and receives these requests in batch from the directory server.

On machines that allow it, the SAM implementation makes use of the facility for interrupting the main computation on a processor when a message arrives. On other systems, the implementation periodically polls for incoming messages.

A preprocessor is supplied with the SAM system that automatically builds the necessary packing functions for transmitting SAM data items which are complex C data types in messages. The programmer must use a special allocator for building these complex data types, so that the SAM run-time system can determine the number of elements in dynamically allocated arrays.