Journal of Parallel and Distributed Computing **62**, 241–266 (2002)
doi:10.1006/jpdc.2001.1777, available online at http://www.idealibrary.com on IDEAL®

A Comparison of Three Programming Models for Adaptive Applications on the Origin2000

Hongzhang Shan and Jaswinder Pal Singh

Department of Computer Science, Princeton University, 35 Olden Street, Princeton, New Jersey 08544
E-mail: shz@cs.princeton.edu, jps@cs.princeton.edu

Leonid Oliker

National Energy Research Scientific Computing Center, Mail Stop 50F, Lawrence Berkeley National Laboratory, Berkeley, California 94720
E-mail: loliker@lbl.gov

and

Rupak Biswas

NASA Advanced Supercomputing Division, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, California 94035
E-mail: rbiswas@nas.nasa.gov

Received August 18, 2000; accepted May 17, 2001

Adaptive applications have computational workloads and communication patterns that change unpredictably at runtime, requiring dynamic load balancing to achieve scalable performance on parallel machines. Efficient parallel implementations of such adaptive applications is therefore a challenging task. In this paper, we compare the performance of and the programming effort required for two major classes of adaptive applications under three leading parallel programming models on an SGI Origin2000 system, a machine that supports all three models efficiently. Results indicate that the three models deliver comparable performance; however, the implementations differ significantly beyond merely using explicit messages versus implicit loads/stores even though the basic parallel algorithms are similar. Compared with the message-passing (using MPI) and SHMEM programming models, the cache-coherent shared address space (CC-SAS) model provides substantial ease of programming at both the conceptual and program orchestration levels, often accompanied by performance gains. However, CC-SAS

0743-7315/02 $35.00

currently has portability limitations and may suffer from poor spatial locality of physically distributed shared data on large numbers of processors.

Key Words: parallel programming; shared address space; message passing; dynamic mesh adaptation; *N*-body problem.

1. INTRODUCTION

Architectural convergence and software tools have made it possible for different programming models to be supported on the same platform. At present, the three leading programming models are explicit message passing, one-sided communication using symmetric private address spaces, and cache-coherent shared address space (CC-SAS). The message-passing paradigm is perhaps the most popular, and commonly implemented by using the MPI library. The SHMEM library is similar to MPI but uses symmetric address spaces for the individual processes. Thus, communication in SHMEM requires only one process to be explicitly involved, and any process can specify remote data using their local name and the process identifier. CC-SAS, on the other hand, assumes a global shared address space, leverages hardware cache-coherency features, and accesses remote data implicitly via ordinary loads and stores.[1]

Unfortunately, it is not obvious how these three programming models compare in terms of parallel performance and ease of programmability. Our previous studies [16, 17] have shown that even for nonadaptive applications, using different programming models significantly affects overall performance and requires varying amounts of programming effort. In this paper, we focus on adaptive applications in which the computational workloads and/or the communication patterns/volumes change at runtime, requiring dynamic load balancing to achieve scalable performance on parallel machines. Applications that exhibit such irregular unpredictable memory accesses and communication patterns have become increasingly important in scientific and engineering fields, as more complex phenomena and domains are studied. However, obtaining scalable performance for this class of applications on current state-of-the-art multiprocessor systems is a challenging task.

Several researchers have investigated the parallel performance of various adaptive applications on different computer platforms. Martonosi and Gupta examined a wire routing program and found that, compared to a shared-memory implementation, the message-passing model reduced communication volume at the cost of compromising solution quality [8]. Singh *et al.* found CC-SAS, when implemented efficiently on the Stanford DASH machine, to provide substantial programming ease and likely performance advantages for hierarchical *N*-body applications [19] and a number of graphics algorithms [18]. Dikaiakos and Stadel conducted performance comparisons of cosmological simulations on the Intel Paragon and KSR-2 machines, and found that the shared-memory version running on the KSR-2 outperformed the message-passing code running on the Paragon [4]. More

[1] The words *process* and *processor* are used synonymously throughout this paper.

recently, Oliker and Biswas examined the performance of a dynamic unstructured mesh adaptation algorithm using three different programming models and concluded that a multithreaded implementation on the Cray (formerly Tera) MTA was the simplest and showed the most promise [11]. However, each programming paradigm in the last study was implemented on a different platform, making direct performance comparisons rather difficult.

Most of these studies did not compare the algorithmic and coding implications of using various programming paradigms. The studies also differ substantially from ours since we use a common high-performance computer with sophisticated implementations of the three different programming models. The use of a single platform makes the performance data and the code comparisons very relevant. Our overall research goal is to study the problem of programming models for adaptive applications in a layered framework. The top layer is called the *application layer*. The applications selected for this work needed to satisfy the following criteria:

- require irregular and unpredictable communication, as well as dynamic load balancing,
- have wide applicability to problem domains that require high-performance computing,
- require the use of large numbers of processors, and
- be nontrivial to obtain scalable performance.

Based on these requirements, we selected two typical applications: *dynamic remeshing* and *N-body* as our test cases. Details of these applications are given in Section 2. For each application, we develop one or more programs for the different programming models using well-known algorithms.

The middle layer in our framework is the *programming model layer*, which provides different programming interfaces to the application layer. The two dominant parallel programming paradigms are message passing and cache-coherent shared address space. However, there exists another programming model called SHMEM, which lies between these two extremes. A brief description of these models is given in Section 3.

The bottom layer is called the *communication layer* and consists of the computation and communication hardware, and low-level software. In this work, we focus on tightly coupled distributed shared-memory multiprocessors. In particular, we selected the SGI Origin2000 platform, which has an aggressive communication architecture and provides full hardware support for the CC-SAS model. The MPI and SHMEM programming models are built in software but leverage the underlying hardware for a shared address space and efficient communication. In fact, the performance of the latter two models on this machine is comparable to or better than that on most systems not supporting the CC-SAS model in hardware.

This layered approach allows us to confine our investigations to each individual layer, isolate any problems, and find suitable solutions. For example, there are two different considerations in the application layer: at the algorithmic level and at the implementation or program orchestration level. We examine whether the algorithms

that deliver the best performance for each programming model are similar. For nonadaptive applications, we found that the best-performing algorithms were similar across models [16], which is a positive indicator for application developers. If this is true for the two chosen adaptive applications, we will investigate if there are substantial implementation differences, and how they affect overall performance. In the programming model layer, we compare the conceptual and programming complexities of the different paradigms, although these issues are sometimes quite subjective. We need to determine if the performance difference between programming models is caused by this layer. For example, in our previous study using regular applications [13], we found that the performance of the MPI implementation could be made competitive with the SHMEM and CC-SAS versions by eliminating an extra data copy and by optimizing the communication buffer management, although this had some programming implications. Finally, since all the programming models are built on the same communication layer, their performance is directly comparable.

We find that all three programming models for both adaptive applications can achieve scalable performance on the Origin2000 (at least up to 64 processors). The algorithms needed by the different programming models for best overall performance are similar, but the implementations differ significantly at the conceptual and program orchestration levels, far beyond whether explicit messages or implicit loads/stores are used. Results indicate that compared with MPI and SHMEM, the CC-SAS strategy provides substantial ease of programming, often accompanied by performance gains. However, CC-SAS may suffer from poor spatial locality of physically distributed shared data on large numbers of processors.

More generally, if the applications are properly programmed, the parallel performances for these three programming models are quite similar. CC-SAS programs usually require less time to develop; however, many naive implementations will not achieve high performance. Significant insights about the application are needed to structure the programs to obtain better data locality and reduce synchronization. In fact, the technologies applied are often similar to those used in message-passing programs. For example, we build a high-level locally essential tree to reduce the page faults in the *N*-body application and use a partitioner for load balancing in the dynamic remeshing application. The important practical advantage of the CC-SAS model is that it allows simpler implementations of these complex algorithms because of the implicit naming and communication features. A big disadvantage of CC-SAS is its loss of portability. Many supercomputing platforms still do not directly support this model. Compared with MPI, SHMEM is relatively easier to implement due to its one-sided communication and often delivers higher performance, but it is also limited by the portability problem.

The remainder of this paper is organized as follows. Section 2 gives an overview of the two adaptive applications being investigated in this work. In Section 3, we briefly describe the three parallel programming models: message passing using MPI, SHMEM, and CC-SAS. The implementation details for the two applications and the programming differences among the three models are described in Section 4. Performance results are presented and critically analyzed in Section 5. Finally, Section 6 summarizes our key conclusions.

2. ADAPTIVE APPLICATIONS

In this section, we give an overview of the two applications being investigated in this work. Dynamic remeshing and *N*-body are two typical adaptive problems that satisfy the application layer criteria mentioned in Section 1. For these kinds of irregular dynamic applications, the processor workloads and the interprocessor communication can change dramatically with time; thus, dynamic load balancing is a critical component. Communication also tends to be naturally finegrained, which can be challenging. In designing efficient parallel implementations for these adaptive applications, data locality is another important consideration. In Section 4, we will focus on these two issues to analyze the algorithmic and implementation differences among the different programming models, and discuss the programming effort required for each application and paradigm.

2.1. Dynamic Remeshing Problem

Dynamic local mesh refinement [3] is required to efficiently capture physical features of interest that evolve with time. It provides users the opportunity to obtain solutions that are comparable to those on globally refined grids but at a much lower cost. Adaptive unstructured meshing is a powerful tool in the numerical modeling of physical phenomena on complex irregular domains. The mesh used in our experiments is that often used to simulate flow over an airfoil. Mesh refinement is usually required around the leading edge of the airfoil, and along the shocks that form on the upper and lower surfaces at transonic Mach numbers. This physical scenario is simulated by geometrically refining the grid in these regions. Further details about this application can be found in [11].

The flowchart for the solution process is shown in Fig. 1. The initial mesh is first partitioned, and a submesh is assigned to each process. An initial matrix is then generated from each submesh by assigning a random value in (0,1) to each (i, j)

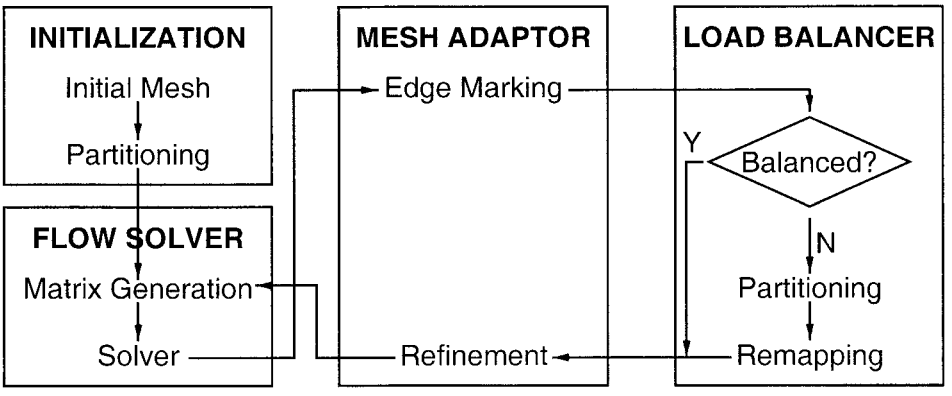


FIG. 1. Flowchart of the dynamic remeshing problem.

entry corresponding to the vertex pair (v_i, v_j) of the edges in the submesh. All other off-diagonal entries are set to 0. The matrix is made positive definite by setting the diagonal entries to a large value (diagonally dominant). The publicly available Aztec library [21] is then used to solve the sparse linear system. Details of the matrix generation process and the solution phase are given in [13]. After a specified convergence is attained, the mesh adaptor is invoked. Based on an error tolerance or geometric information, it marks the edges in the regions that need to be refined. However, the actual refinement is delayed until after the load balancer in order to reduce the data movement overhead and achieve better load balance in the adaptation phase. If the marked mesh will cause the current partitions to become unbalanced, the load balancer assumes responsibility for repartitioning the mesh and remapping the data. After refinement, the matrices for the submeshes are regenerated and passed onto the solver. The entire cycle is then repeated until the computation is done. Extensive details about this mesh adaptation procedure and the dynamic load balancing strategy are given in [10, 12].

2.2. *N-Body Problem*

The N-body problem is a classical one, and arises in many areas of science and engineering such as astrophysics, molecular dynamics, and computer graphics. Having specified the initial positions and velocities of the N interacting bodies, the problem is to find their positions after a certain amount of time. The Barnes–Hut method [2] is widely used to solve this problem today. It has three phases within each iteration of the simulation. In the tree-building phase, an octree is constructed to represent the distribution of the bodies. It is implemented by recursively partitioning the three-dimensional space into eight subspaces until the number of bodies in each subspace is below a certain threshold. In the second phase, the force interactions between individual bodies are computed. Each body traverses the octree starting from the root. If the distance between a body and the visited subspace (cell) is large enough, the entire subtree rooted there is approximated by the cell; otherwise, the traversal continues recursively with the children. In the third and final phase, each body updates its position and velocity based on the computed forces.

3. PROGRAMMING MODELS

We chose an SGI Origin2000 machine as the common platform to compare and contrast the different programming models. The Origin2000 is a scalable, hardware-supported cache-coherent nonuniform memory access (CC-NUMA) system, with an aggressive communication architecture. It therefore automatically supports the CC-SAS programming model. The MPI message-passing and SHMEM models are built in software but leverage the machine's shared address space and the efficient communication features. We give here a brief description of all three programming models; further conceptual comparisons can be found in [1, 9].

3.1. Message Passing Using MPI

In the message-passing model, each process has only a private address space, and must communicate explicitly with other processes to access their (private) data. Communication is performed via send–receive pairs, so processes on both sides are explicitly involved. The sender specifies whom to send the data but not the destination address; these are specified by the matching receiver whose address space they are in. The data typically are packed and unpacked at each end for efficient transfer. This model is perhaps the most difficult to program for irregular applications; however, the benefits lie in enhanced performance for coarse-grained communication and implicit synchronization through blocking communication.

We used an improved version of MPICH [16], the portable implementation of MPI for our experiments. It uses the Origin2000 shared address space and fast communication support to accelerate message passing. The MPICH performance was much better than the vendor-supplied implementation of MPI. We suspect that this is because MPICH uses one copy (instead of two), and lock-free queue management. It also allows the programmer to instrument the implementation so as to distinguish between wait time and time to copy remote data. MPICH is consistent with the message-passing model in that application data structures are only allocated in private per-process address spaces. However, the communication buffers for the send and receive operations are allocated in the shared address space during the initialization phase; they include a shared packet pool for exchanging control information (all messages) and data (short messages), and buffer space for data (large messages). All copying of data to and from the buffers is done with the `memcpy` function. Note that while hardware support for load/store communication is very useful, an invalidation-based coherence protocol, such as on most cache-coherent machines including the Origin2000, can make such producer–consumer communication inefficient compared to an update protocol or a hardware-supported but noncoherent shared address space.

3.2. SHMEM

The SHMEM library provides the fastest interprocessor communication because of its lower protocol overhead. Basically, each process has its personal address space as in message passing, but the address spaces are symmetric. Any process can name the variables in another process's address space by using the local name and the remote process identifier. The major primitives are the `put` and `get` commands. The `get` operation is used to copy a variable amount of data from another process (using `bcopy` that is similar to the `memcpy` used in message passing) and explicitly replicate it locally. A `put` is the dual of `get`; however, each is an independent and complete way of performing data transfer. Only one is used per communication, unlike explicit message passing, which requires send–receive pairs. Thus, the communication becomes one-sided but remains explicit.

In SHMEM, there is no concept of a uniformly addressable shared address space that all processes can access. However, the private address spaces of processes that

contain the logically shared data structures are identical in their data allocation. By providing a global segmented address space and avoiding the need for matching send–receive operations, the SHMEM model delivers significant programming simplicity over MPI, even though it too does not provide fully transparent naming or replication.

3.3. Cache-Coherent Shared Address Space (CC-SAS)

In this model, remote data are accessed just like locally allocated data (or data in a sequential program), using loads and stores. A load/store cache miss causes the data to be communicated in hardware at cache line granularity, and automatically replicated in the local cache. Unlike the `get/put` operations in SHMEM, ordinary load/store operations are used to fetch/send data. The transparent naming and replication provides programming simplicity, particularly for dynamic fine-grained applications. In our parallel CC-SAS implementations, the parent process uses the Unix `fork` command to spawn off child processes, one for each additional processor, at the beginning of the program. These cooperating processes are then assigned chunks of work, while locks and barriers are used for synchronization. The child processes are finally terminated at the end of the last parallel section.

4. IMPLEMENTATION DETAILS

In this section, we describe specific implementation details of our two adaptive applications using the different programming models. We compare only the MPI and CC-SAS versions, since SHMEM is similar to MPI except primarily for its one-sided communication. In other words, instead of using send–receive pairs as in MPI, the SHMEM model only needs either a `put` or a `get`. However, there is one other important difference between MPI and SHMEM in the way they are used to dynamically allocate memory. In MPI, dynamic memory allocation is performed by invoking the `malloc` utility locally and independently in different processes. Instead, in SHMEM, if the program needs to allocate memory for symmetric variables, it must reserve exactly the same size of memory in all processes.

4.1. Dynamic Remeshing Problem

For this application, we focus on the three main modules: mesh adaptor, load balancer, and flow solver, and then compare and contrast the MPI and CC-SAS implementations and program orchestrations that arise because of the nature of the programming models. We show that some of the differences are due to reasons beyond using explicit communication messages rather than loads/stores, even though the same basic partitioning algorithm for load balancing and communication reduction is used. Overall, the CC-SAS model provides substantial ease of programming.

Mesh adaptor. In the mesh adaptor module, all the edges of the unstructured mesh are first marked to indicate whether they need to be bisected, either based on geometric information or solution-driven error tolerance. However, the actual mesh

refinement is delayed until after the load balancer module is executed. This delay has three beneficial side effects: (i) it improves the load balance of the refinement phase since a larger fraction of the processors participate in the mesh adaptation, (ii) it reduces the communication volume needed for data remapping after the repartitioning since refinement is performed by the destination processors, and (iii) it increases data locality since the flow solver works on the newly partitioned refined mesh. Detailed explanations of these side effects are given in [10, 12].

In the MPI implementation, each process owns a submesh and maintains the necessary local data structures to represent it. Thus, each mesh object (vertex, edge, element) has a local index. These local data structures and indices provide good data locality for the MPI program. However, in order to exchange information with other processes, each process also maintains a mapping between the local index and the global index (which is the index of the mesh object in the global mesh).

In CC-SAS programs, a complete shared mesh is maintained. A potential drawback of this strategy is that the shared data structures cannot be easily changed without synchronization. Unfortunately, mesh refinement involves modifying several data structures by inserting new mesh objects and altering their relationships. This need for synchronization can be dramatically reduced by letting each process precompute its number of new vertices, edges, and elements, and applying the range to the global data structures. This enables the process to modify its partition of the data structures with only a few synchronizations, but at the cost of some additional complexity. However, MPI programs must maintain a lot of extra data structures to track ownership of mesh objects and to orchestrate communication.

Load balancer. Dynamic repartitioning to balance processor workloads is an essential phase in any parallel adaptive mesh computation. As the numerical simulation evolves, various regions of the mesh are dynamically refined, leading to load imbalance that hurts the overall performance [10, 12]. Significant research has been done on parallel partitioning algorithms, and several state-of-the-art MPI software packages are currently available on the World Wide Web [5, 22]. We chose ParMETIS [5, 6] as the basic partitioner for this work because of its good overall performance and wide availability.

ParMETIS is a multilevel partitioner that consists of three main phases: coarsening the graph to be partitioned, partitioning the coarse graph, and projecting the partitioned graph back to the given initial graph. The coarsening is implemented by using a vertex-matching scheme where the "heaviest" edge incident on a vertex is collapsed. To find a match for vertices on partition boundaries, a try–confirm strategy is used. This is because in the MPI model, a message must be received from the remote process to confirm the matching, as other processes may also try to match their own boundary vertices with the same vertex. After the coarsening phase is complete, the coarsest graph is partitioned and the ownership of its vertices projected back to the initial graph. During this uncoarsening projection phase, each process reconsiders the ownership of its boundary vertices to reduce the overall edge cut and to further balance the workload. Due to private address spaces in MPI and the lack of global information, these decisions are made based on an incomplete local view.

A load balancer is simpler to implement in CC-SAS programs since all processes share the same global view. In addition, a load balancer enhances data locality, thereby reducing contention as well as the number of cache misses and page faults. The MPI try–confirm process in ParMETIS is no longer required as the communication to check for matchability is replaced by synchronization. When a process finds a matching vertex, it first locks it and checks whether it has already been matched. This procedure is obviously much easier to implement. The initial partitioning is straightforward because of the shared address space. Finally, when updating the ownership of boundary vertices in the uncoarsening phase, CC-SAS enables a decision to be made based on a consistent global view, which helps generate more balanced partitions.

After the partitioning, data must be remapped among all the processes for the MPI implementation. In other words, each process may have to send/receive messages to obtain the data that corresponds to its assigned partition. This remapping phase is very expensive for large computational meshes. In our application, remapping is performed in bulk fashion, as opposed to communicating several small individual messages. The advantages include the amortization of message start-up costs and good cache performance. The disadvantages are complexity and some extra work. Basically, data leaving a partition are first stripped out and placed in a buffer, then appropriately communicated, and finally integrated into the corresponding data structure of the destination processor.

No explicit data remapping is necessary for program orchestration in the CC-SAS implementation. However, the quality of the partitioning nonetheless affects data locality. For instance, if data movement is minimized, most mesh objects will remain assigned to the same process before and after a partitioning, thereby enhancing data locality. Moreover, the new objects that are created by subdivision are not used by other processes during the refinement stage.

Using a partitioner like ParMETIS rather than simply splitting the data structures in CC-SAS guarantees that each process is assigned a continuous submesh to work on, and that synchronization is only needed on the subdomain boundaries. This greatly reduces the number of synchronization operations, and allows each process to obtain good temporal and spatial data locality. Otherwise, the CC-SAS implementation is unable to achieve scalable performance [11].

Flow solver. The flow solver module consists of matrix generation and the actual numerical solver. The matrix generation step is application dependent, and as described in Section 2.1, a diagonally dominant positive definite matrix is generated for each submesh. In the message-passing model, this requires collecting information from all neighbors of boundary vertices. In our implementation, the process owning a boundary vertex is responsible for gathering this data. In CC-SAS, there is no need for any explicit communication.

Numerical solvers constitute an immense area of research. Our objective in this paper is to study only the effects of the various programming models. We have therefore selected the Conjugate Gradient (CG) algorithm, which is the best-known Krylov subspace method for solving the linear system $Ax = b$. Algorithmic details

about CG are given in [14]. In our work, we use the publicly available Aztec library [21]. In the MPI version, the matrix A is partitioned by rows (each row corresponds to a vertex in the mesh) among the processes, based on the partitioning given by the load balancer. Each process prepares a list of the row indices of A that it owns, as well as those of the vectors x and b.

The solver has been separated into two phases: matrix transformation and iterative solution. In the transform stage, the vertices are grouped as internal vertices (those that do not need communication with other processes), border vertices (those that need communication with other processes), and external vertices (those owned by other processes). Each process reorders its submatrix (based on internal, border, and external vertices in that order) into a nearly block diagonal form to obtain good data locality for the time-consuming iterative solution phase. In MPI programs, this involves expensive hashing, searching, and broadcast operations, due to all the data being in private address spaces. In Aztec, a large number of small messages are used for the communication. However, in our implementation, much of the vertex ownership information can be provided during matrix generation without additional cost. Thus, the Aztec interface can be modified accordingly. Till date, we have accomplished these modifications only partially so that most of the matrix transformation work is still left within Aztec. In CC-SAS programs, a shared array is used to provide all the information needed by the reordering. Compared to MPI, the conceptual/orchestration complexity and programming effort are greatly reduced.

The kernel of the CG iterative solver consists of a sparse matrix–vector multiply (SPMV), three vector updates, and three dot products. However, for many practical applications, the SPMV dominates the operation count. The basic solver algorithm is similar across programming models except for the differences in explicit messaging versus implicit loads/stores.

4.2. N-Body Problem

The Barnes–Hut method [2] for solving the N-body problem consists of three main phases: tree-building, force calculation, and particle update. For each of these modules, we compare below the MPI and CC-SAS implementations.

Tree-building. Tree-building is the most complex step of the MPI implementation. In this phase, each processor builds a *locally essential tree*, which allows the force calculation phase itself to proceed without communication. In the first iteration, the domain is partitioned into a fixed number of particles that are distributed equally among the processors. Subsequent iterations use the previous distribution as the starting point. A popular message-passing implementation strategy uses the orthogonal recursive bisection (ORB) partitioner [23]. We use a different approach. A cost distribution tree is computed in parallel, requiring the use of global communication. This cost represents the expected amount of work required to perform force calculation for the particles within a cell, and is used as the load balancing metric. If a cell's cost is greater (less) than a specified threshold, its space is recursively subdivided (collapsed) into eight (one) subspaces. Thus a limited

global tree that represents the cost distributions is created. This tree is partitioned using the *costzones* [19] technique, which assigns each processor a contiguous range of cells of approximately equal cost in Peano–Hilbert order. A data remapper uses the computed partitioning to distribute the cells and their corresponding particles, thereby creating a cost balanced local tree on each processor. A communication step is finally required to appropriately distribute the particle and cell information, thus allowing each processor to build its locally essential tree.

We also implemented the ORB version in a manner similar to that reported in [7, 15], and found no significant performance differences with our costzones approach. Instead, using costzones allowed us to make easier comparisons with the CC-SAS implementation of the *N*-body problem.

The CC-SAS version of the *N*-body simulation is obtained from the SPLASH-2 suite [24] and further optimized. The tree-building phase varies dramatically from the MPI implementation since only one global octree is created. Each process is responsible for those particles assigned to it based on the costs in the previous iteration. The global octree is built by concurrently adding particles to the single shared tree, using synchronization locks if necessary. When the cost (defined in costzones) of a cell exceeds a specified limit, that cell is dynamically subdivided into eight new subcells. To guarantee correctness, a synchronization lock is placed on a cell whenever a particle is inserted into it, or during cell subdivision. Unlike MPI, explicit communication is not required to compute the shared cost distribution tree. The particles are then partitioned using the costzones technique, by assigning each processor a contiguous section (in the Peano–Hilbert ordering sense) of the global tree. This ordering strategy ensures cost balanced partitions and good data locality during the subsequent force calculation phase. Note that this partitioning approach is algorithmically similar to that used in the MPI version; however, a data remapping phase is not required in CC-SAS. Since all the bodies are globally addressable, they can be reassigned to the processors without the need for explicit communication.

Force calculation and particle update. The force calculation is the most expensive phase of the *N*-body problem. In this step, each body computes its force interaction with every other body (or cell) by recursively traversing the octree. The MPI implementation uses the locally essential tree, created in the tree-building phase, to perform a load balanced and communication-free force calculation. Each particle's cost is also kept track of, for building the cost distribution tree in the subsequent iteration. In the third and final phase, each body updates its position and velocity based on the results of the force calculation. The message-passing version of the update phase is communication free, but suffers from some load imbalance. This is because the costzones partitioning scheme used in tree-building is based on the cost, not the number, of bodies. However, the computational overhead of the update phase is a function of the total number of bodies in each partition. An additional redistribution step to load balance the updates is not worthwhile, since this phase constitutes a relatively small portion of the overall *N*-body simulation time. The SHMEM version of this algorithm was transformed directly from the MPI code, by replacing two-sided communications with one-sided communications.

TABLE 1

The Number of Essential Source Code Lines for the Two Adaptive Applications

	Dynamic remeshing				*N*-body
	Mesh adaptor	Load balancer	Flow solver	Total	Total
MPI	5,337	4,615	6,603	16,015	1,371
SHMEM	5,579	4,100	5,906	15,585	1,322
CC-SAS	2,563	2,142	3,725	8,430	1,065

In CC-SAS, once the global shared tree has been built, the force calculation is computed in parallel without the need for synchronization. However, unlike the MPI version, implicit communication is required during this phase since the global tree is physically distributed among the processors. The particle update then proceeds in parallel, using the results of the force calculation. Once again, this step is synchronization free, but requires implicit communication. The CC-SAS update phase is also somewhat load imbalanced, for the same reasons as the imbalance in the MPI update. To increase data locality for the next iteration, bodies are reordered based on their processor assignment. The reordering step constitutes a small fraction of runtime, which is dominated by the force calculation.

Overall, the CC-SAS implementation and conceptual orchestration are much simpler than MPI. Using synchronization locks to build a global tree in a shared address space is much less complex than creating a locally essential tree in a distributed-memory environment. However, the MPI force calculation and particle update proceed with only the use of local memory unlike the CC-SAS version which requires implicit communication.

In Table 1, we list the number of essential source code lines for all three programming models for these two applications. Code sections for preparing test data, debugging, and comments are not included. SHMEM is very similar to MPI since they both use explicit communication; however, the one-sided SHMEM library requires fewer lines of code. The CC-SAS codes require far fewer lines due to their implicit communication that obviates the need to set up and maintain special data structures and communication buffers. This also leads to substantial ease in programming.

5. PERFORMANCE RESULTS

The Origin2000 machine used for the experiments reported in this paper contains 64 300-MHz R12K MIPS microprocessors, and is located at Princeton University. Each processor has separate 32KB primary instruction and data caches, and a unified 8 MB secondary cache with two-way associativity and a 128-byte block size. The entire machine has 16 GB of main memory, with a page size of 16KB. There are two processors in each node sharing a noncoherent bus. Pairs of nodes are connected to a network router, and the interconnect topology across the 16 routers is a hypercube.

TABLE 2

Sequential Runtimes (in s) for Each Test Case of the Two Adaptive Applications

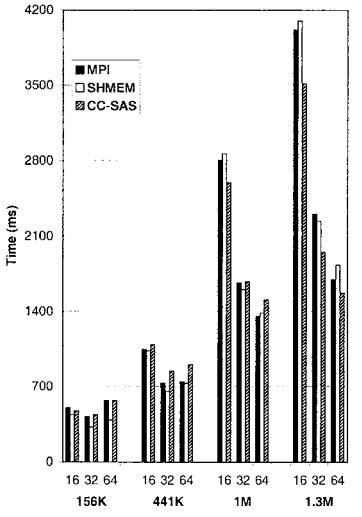
Dynamic remeshing				*N*-body			
Number of triangles				Number of particles			
156K	441K	1M	1.3M	16K	64K	256K	1M
6.41	24.85	69.17	97.13	3.39	15.04	67.69	329.81

Table 2 presents the sequential runtimes for both adaptive applications. For the dynamic remeshing problem, we simulate flow over an airfoil and geometrically refine regions corresponding to the locations of the stagnation point and the shocks [11]. The original mesh contains 28K triangles, and grows to approximately 59K, 156K, 441K, 1M, and 1.3M triangles, through five levels of refinement. For this work, we study only the last four levels in detail. For example, referring to the 1.3M test case implies that the mesh adaptor increased the 1M-triangle mesh to 1.3M triangles. The load balancing, matrix generation, and ensuing iterative solution were then based on this newly generated 1.3M mesh. For the *N*-body problem, we also tested four cases: 16K, 64K, 256K, and 1M particles. These data sets compose two neighboring Plummer model galaxies that are about to undergo a merger [20]. However, unlike our dynamic remeshing simulation, each *N*-body problem represents an independent experiment. One interesting parallel performance result is the superlinear speedups demonstrated by both applications in some of the test cases. This occurs partly because as the number of processors increases, a larger fraction of the problem fits in cache. The superlinear effect may continue until the entire problem is accommodated in the combined caches of the processors.

Furthermore, we analyze the per-process wall-clock time by decomposing it into four parts: BUSY (time spent in computation), LMEM (time waiting for local cache miss), RMEM (time waiting for remote communication), and SYNC (time for synchronization). The BUSY time is obtained using SpeedShop, an integrated package of tools that runs performance experiments on executables and lets one examine the results of those experiments. The SYNC and RMEM times are obtained by instrumentation. Finally, the LMEM time is derived by subtracting the BUSY, SYNC, and RMEM times from the application's total wall-clock time, which is measured using an efficient hardware clock. In CC-SAS programs, we cannot differentiate between LMEM and RMEM times using the available tools. Thus, we lump them together as MEM time.

5.1. Dynamic Remeshing Problem

The performance of the dynamic remeshing problem for varying numbers of triangles is presented in Fig. 2. In this rapidly adapting flow simulation, mesh refinement and the ensuing load balancer are invoked after 10 iterations of the



Time (ms)

	156K data set			441K data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	499	418	570	1047	732	746
SHMEM	436	323	388	1034	654	728
CC-SAS	472	434	568	1092	844	904

	1M data set			1.3M data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	2805	1665	1355	4015	2305	1694
SHMEM	2865	1605	1389	4100	2239	1830
CC-SAS	2597	1678	1508	3518	1949	1574

FIG. 2. Runtimes (in ms) for the dynamic remeshing problem on 16, 32, and 64 processors for different data sets.

numerical solver. Future research will investigate the performance of this application under varying flow solver iteration requirements. It is important to note that a realistic application will consist of many mesh adaptations, and thus the solver → adaptor → load-balancer cycle in Fig. 1 will be executed that many times. Overall, the three programming methods show similar performance for the entire application across all test cases. For the smaller data sizes, MPI and SHMEM generally outperform CC-SAS. However, for our largest test case consisting of approximately 1.3M triangles, the CC-SAS implementation has runtimes lower than those of the message-passing versions.

Figure 3 presents the runtime breakdown for the 1.3M test case. CC-SAS has a lower BUSY time than MPI or SHMEM, but suffers from higher SYNC

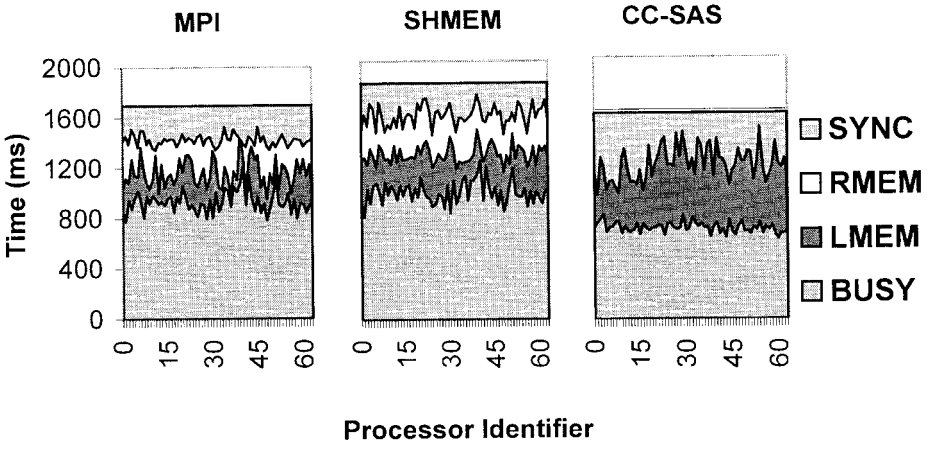
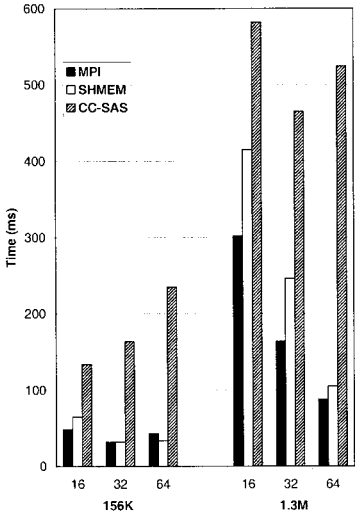


FIG. 3. Time breakdown for the 1.3M data set size on 64 processors.

overheads. Also notice that the MEM time under all three programming models is relatively high. In order to understand the overall runtime behavior, each of the dynamic remeshing components must be examined individually.

The mesh adaptor module is responsible for refining the mesh in specified regions in order to investigate localized flow phenomena in finer detail. As described in Section 4.1, this phase consists of edge marking and mesh subdivision. The parallel workload of the mesh adaptor is generally load imbalanced because the partitioning process is designed to optimize the performance of the costly flow solver phase. In MPI and SHMEM, each processor is responsible for refining its local region of the mesh. To build a consistent final mesh, coarse-grained communication is used across partition boundaries. CC-SAS, however, maintains a single shared mesh that is concurrently refined. The global address space allows a reduction in the programming complexity, but introduces a large volume of implicit communication for such irregularly structured computations. CC-SAS also incurs an additional algorithmic cost since synchronization locks are required to avoid possible race conditions during the subdivision phase. The number of synchronization points is minimized by precomputing the location of newly created triangles.

Figure 4 presents the mesh adaptor runtimes for the 156K- and 1.3M-triangle test cases. The distributed-memory implementation significantly outperforms CC-SAS for both data sizes due to its data locality and coarse-grained communication. We experimented with the CC-SAS version by reorganizing the data structures in a localized fashion as in the message-passing cases. With the modifications, the mesh adaptor runtimes improved; however, the overall performance was unaffected as it had the same MPI/SHMEM data remapping bottleneck. The performance difference between MPI and SHMEM is primarily in the local operations, and is probably caused by changes in the cache behavior due to the particular memory



Time (ms)

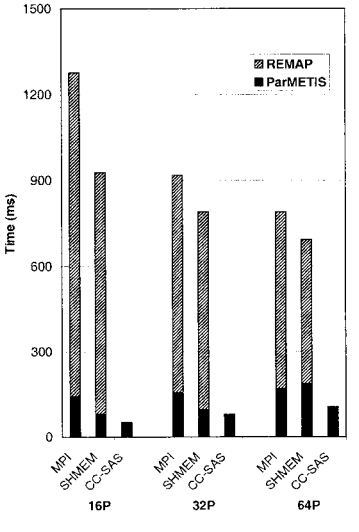
	156K data set			1.3M data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	48	32	43	302	164	88
SHMEM	65	32	34	415	246	105
CC-SAS	134	164	235	582	465	524

FIG. 4. Runtimes (in ms) for the mesh adaptor module on 16, 32, and 64 processors for the 156K and 1.3M data sets.

allocation method used. In fact, Fig. 3 indicates that SHMEM has higher LMEM time, which may simply be a function of cache conflicts. For the 1.3M data set, the performance of the MPI and SHMEM mesh adaptor improves with larger numbers of processors. However, for almost every test case shown, the CC-SAS performance degrades as the number of processors increases. The irregular nature of unstructured mesh subdivision is inherently at odds with the globally shared mesh and thus causes an increase in the volume of implicit communication, memory latency, false sharing, and TLB misses [11]. In addition, the use of low-level synchronizations hurts performance for larger numbers of processors. In conclusion, the distributed-memory implementation of the mesh adaptor offers significant performance advantages over CC-SAS.

The load balancing module is invoked after each iteration of the mesh adaptor, to rebalance the processor workloads and minimize interprocessor communication for the costly solver phase. Note that the partitioning is always performed on the initial dual graph, which keeps the connectivity and partitioning complexity constant throughout the adaptive computation [10]. In the load balancing module, there is a significant algorithmic difference between CC-SAS and MPI. The MPI implementation calls ParMETIS to compute a new partitioning, followed by a data-remapping phase that appropriately distributes the mesh. This message-passing remapping phase incurs both the communication cost and the computational overhead for breaking down and rebuilding the data structures. In CC-SAS, a partitioning phase is used but data remapping is not required. Each processor is assigned its proper subdomain, but the actual data redistribution is performed during the transform phase of the flow solver. Thus, CC-SAS has a significant advantage during load balancing.

Figure 5 presents the load balancing runtimes for the 1.3M data set. The total CC-SAS time is significantly lower than that of MPI or SHMEM since it does not



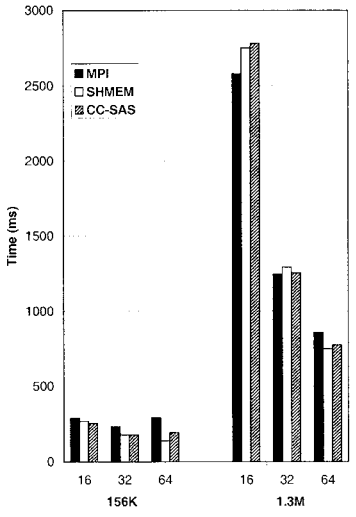
Time (ms)

	Partitioning			Remapping		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	143	156	170	1135	762	621
SHMEM	81	95	187	848	696	508
CC-SAS	52	79	105	0	0	0

Partitioning Quality

	Load Imbalance Factor			Edge Cut		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	1.04	1.07	1.13	3169	5447	8668
SHMEM	1.04	1.07	1.13	3169	5447	8668
CC-SAS	1.02	1.05	1.05	3935	6472	10497

FIG. 5. Runtimes (in ms) for load balancing (ParMETIS partitioning and remapping), and partitioning quality on 16, 32, and 64 processors for the 1.3M data set.



Time (ms)

	156K data set			1.3M data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	291	234	293	2579	1246	859
SHMEM	270	179	139	2750	1292	749
CC-SAS	255	179	193	2779	1253	776

FIG. 6. Runtimes (in ms) for the solver on 16, 32, and 64 processors for the 156K and 1.3M data sets.

perform data remapping. Interestingly, the CC-SAS ParMETIS implementation is itself also substantially faster than the original MPI version, but partitioning time alone is not sufficient to rate the performance of a partitioner; one needs to investigate partitioning quality as well. Partitioning quality is usually defined in two ways: the computational load imbalance factor[2] and the edge cut. Figure 5 shows that the CC-SAS code gives better workload balance among the processors (upto an 8% improvement) at the cost of a larger number of cut edges. Future research will examine shared-memory partitioning in detail. Note that for all three paradigms, the ParMETIS time increases with larger numbers of processors, due to the partitioners' increased volume of computation and communication overheads. Finally, the data-remapping time decreases with more processors. This is because remapping time is a function of the maximum communication among processors [10].

The flow solver is the most expensive module of our dynamic remeshing simulation. After each adaptation and load balancing phase, the newly generated mesh is converted to a matrix, as described in Section 2.1. A transform step then rearranges the matrix to improve data locality for the time-consuming CG iterative solution procedure. Recall that the number of rows and nonzeros in our matrix corresponds respectively to the number of vertices and edges in the underlying mesh. For example, the matrix generated from the 1.3M data set contains more than 488K rows and 1.9M nonzeros. The solution to this matrix requires 22 CG iterations. Figure 6 presents the runtimes of the numerical solver using the three programming paradigms for 156K and 1.3M triangles. These runtimes do not include the matrix generation overhead (which is about 11% of the total execution time for the largest test case running on 64 processors), since it is application dependent and beyond the scope of this paper.

[2] The load imbalance factor is the ratio of the workload on the most heavily loaded processor to the average load across all processors.

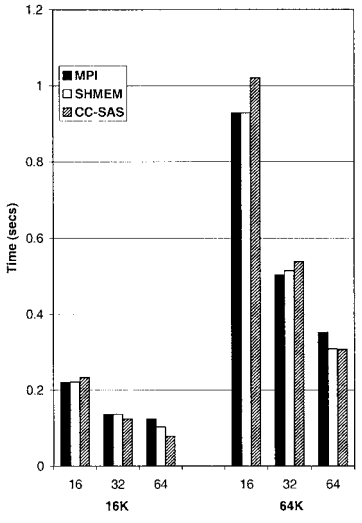
Overall, the runtimes of the flow solver are quite similar on all three programming models since their underlying algorithms are essentially identical. The implementation details, however, vary significantly. For the 1.3M case, there is a dramatic improvement in performance with increasing numbers of processors. Partitioning the matrix into more (smaller) subdomains results in improved cache reuse and reduced solver times. The overhead of the transform in MPI and SHMEM is substantially higher than CC-SAS. The distributed-memory matrix transformation involves complex reordering based on internal, border, and external vertices. This is necessary for efficient communication during the CG algorithm. The shared-memory transform, however, simply assigns a block of rows to each processor since no explicit communication is required. Nevertheless, SHMEM and/or MPI slightly outperform CC-SAS for the 1.3M test case. This is due to the efficient performance of the distributed-memory CG, which has better data locality and lower communication volume than the CC-SAS implementation.

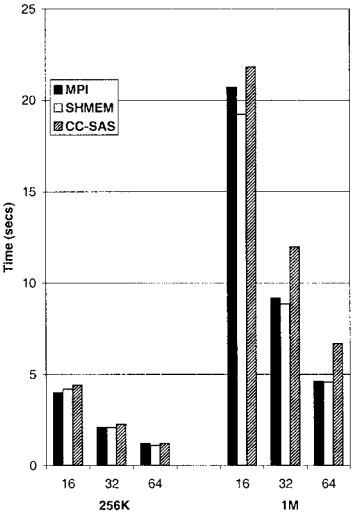
In summary, dynamic remeshing is an irregularly structured, dynamically adapting application, consisting of several distinct modules, each with its own performance characteristics. None of the programming models presented in this paper is best-suited for all these modules. The distributed-memory versions generally have better data locality and consequently outperform CC-SAS during the mesh adaptation and iterative solution phases, because all the data are already in local memory. However, CC-SAS has the advantage of a global address space, which reduces programming complexity and improves performance for a number of the modules. For example, during load balancing, the CC-SAS version of ParMETIS partitions the mesh faster than MPI or SHMEM. More importantly, CC-SAS does not require a remapping phase, which accounts for a significant overhead on large data sets. Finally, the CC-SAS transform phase of the flow solver outperforms both MPI and SHMEM.

The total runtimes in Fig. 2 show that the advantages of MPI and SHMEM lead to better overall performance for the three smallest data sets. The 1.3M test case shows the crossover point where CC-SAS becomes the fastest implementation. For this largest data set, the benefits of CC-SAS outweigh the advantages of MPI and SHMEM. Thus, none of the programming paradigms is a clear winner for this application in terms of overall performance. However, even though all three models use similar high-level algorithms, CC-SAS offers an inherent advantage by reducing the programming and orchestration overheads.

5.2. N-Body Problem

The performance of the *N*-body simulation for varying data sizes is presented in Fig. 7. The MPI and SHMEM implementations show similar behavior across all processor counts and data sets, since their underlying algorithms are the same. In fact, on 16 processors, all three programming paradigms have similar runtimes across all data sets. However, on 64 processors, performance differences between the two basic programming schemes begin to emerge. For the 16K data set, CC-SAS has a runtime advantage compared to MPI and SHMEM. All three implementations benefit from larger data sizes, but the effect is more dramatic





Time (secs)

	16K data set			64K data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	0.221	0.136	0.124	0.929	0.503	0.352
SHMEM	0.222	0.136	0.103	0.929	0.514	0.308
CC-SAS	0.234	0.125	0.079	1.021	0.538	0.307

	256K data set			1M data set		
	P=16	P=32	P=64	P=16	P=32	P=64
MPI	3.995	2.107	1.210	20.71	9.173	4.639
SHMEM	4.183	2.077	1.103	19.22	8.869	4.576
CC-SAS	4.421	2.275	1.223	21.82	11.97	6.688

FIG. 7. Runtimes (in s) for the *N*-body problem on 16, 32, and 64 processors for different data sets.

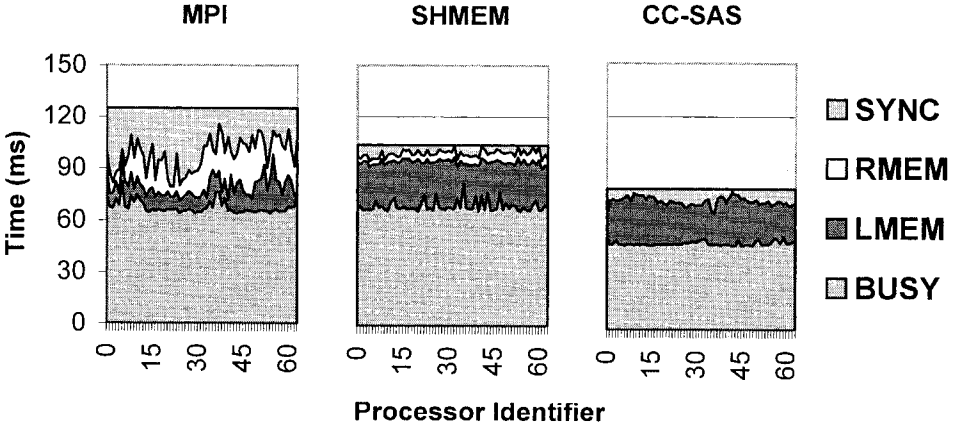


FIG. 8. Time breakdown for the 16K data set on 64 processors.

for message passing. At 1M bodies, the MPI and SHMEM versions significantly outperform CC-SAS.

Figure 8 shows the runtime breakdown for the 16K test case using 64 processors. Here the BUSY times for MPI and SHMEM are significantly higher than CC-SAS. The message-passing versions require complex data movement and computations to build the locally essential tree, compared to the CC-SAS global tree implementation. For this test case, the CC-SAS paradigm not only simplifies the programming overhead, but also results in a reduced runtime. Figure 8 also shows that the MPI version suffers from high and imbalanced RMEM and SYNC times, compared to SHMEM. This is due to the disadvantage of having send/receive pairs in MPI, which cause a higher communication overhead than the one-sided approach.

Figure 9 shows the time breakdown for the 1M data set. The results are quite different compared to the 16K example. Since the data size is relatively large, the total execution time is dominated by the force calculation. The BUSY time for all three approaches is very similar. However, the MEM and SYNC times for CC-SAS are much higher than MPI and SHMEM. This is because in the message-passing implementations, the locally essential tree-building time is now negligible, and the force calculation proceeds without the need for interprocessor communication. CC-SAS, on the other hand, uses a global shared tree, which is physically distributed among all the processors. This results in implicit communication during the force calculation, which causes page faults (TLB misses) and increases memory latency.

We can improve the performance of the CC-SAS implementation for this largest test case by locally duplicating a subset of the remote cells. Note that this would not be a natural programming style for CC-SAS, and brings us closer to the message-passing style of data replication. Each processor explicitly creates a local copy of the remote cells that are frequently used during the force calculation. From our experiments, we found that the duplication can be limited to the first four levels of the tree, which is approximately 590 (out of more than 366K) cells for the 1M data

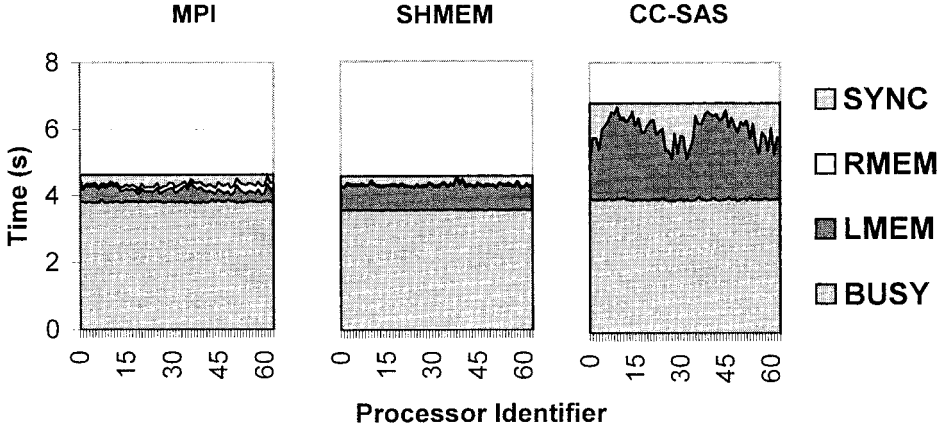
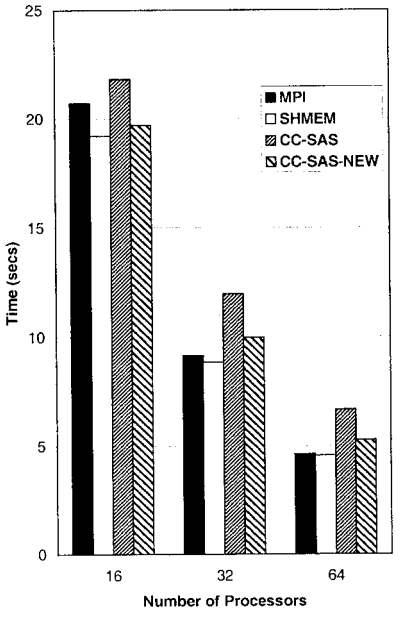


FIG. 9. Time breakdown for the 1M data set on 64 processors.



Time (secs)

	P=16	P=32	P=64
MPI	20.71	9.173	4.639
SHMEM	19.22	8.869	4.576
CC-SAS	21.82	11.97	6.688
CC-SAS-NEW	19.70	9.983	5.287

FIG. 10. Runtimes (in s) on 16, 32, and 64 processors for the 1M data set, including CC-SAS-NEW performance.

set. The improved implementation, called CC-SAS-NEW, is presented in Fig. 10. Note that CC-SAS-NEW outperforms the original CC-SAS implementation but is still slower than the message-passing versions. If all the remote cells required in the force calculation (not just a subset) were duplicated in CC-SAS-NEW, this programming strategy would effectively be the same as the locally essential tree of MPI and SHMEM. Thus for the *N*-body problem, all three programming paradigms use the same algorithm to achieve best performance; however, their corresponding implementations are quite different.

6. CONCLUSIONS

In this paper, we studied the performance of and the programming efforts for two different adaptive applications (dynamic remeshing, *N*-body) under three leading programming models (MPI, SHMEM, CC-SAS) on an SGI Origin2000 system. In order to keep our investigation tractable and modular, we used a layered approach. Results indicated that all three models mostly achieve similar performance; however, the implementations differ significantly even though the same basic parallel algorithms are used. CC-SAS provides substantial ease of programming, and is often accompanied by performance gains. Unfortunately, CC-SAS currently has portability limitations and may suffer from poor spatial locality of the physically distributed shared data, in which case some of the programming advantages must be given up to obtain comparable performance. These observations are consistent with those in our previous study on regular applications [16].

The *N*-body simulation successfully achieved scalable performance across all three programming methodologies. The CC-SAS runtime on the largest test case was approximately 44% slower than MPI; however, the CC-SAS implementation required 22% less code. With some explicit data replication, we demonstrated that CC-SAS performance can be substantially improved to be within 20% of the message-passing versions. Further improvements are expected through more explicit control of data management. For certain projects, CC-SAS's programming advantages may outweigh the performance deficiencies.

The dynamic remeshing problem showed comparable performance across all three programming models, but did not scale well on a 64-processor machine even for the largest problem size considered. Previous research examined the mesh adaptation and load balancing algorithms across various programming paradigms and architectures [11]. The dynamic remeshing simulation in this paper extended that work by creating a complete adaptive application that combines a numerical solver with the original parallel mesh adaptation module. To understand the overall runtime behavior, each of the components had to be examined individually. The solver phase was the most computationally expensive step and achieved scalable performance across all three programming models. However, the runtimes of the other critical modules did not decrease with increasing processor counts, creating a potential performance bottleneck. This was true of the parallel partitioner, whose computation and communication overheads grew with the number of processors across all three programming models. Another dramatic example of this slowdown behavior was seen in the CC-SAS mesh adaptor. For our largest test case running on 16 processors, this phase accounted for 16% of the overall runtime. However, on 64 processors, the CC-SAS mesh adaptation consumed more than 33% of the total execution time. We expect this trend to continue as the number of processors increases. These poor performance characteristics will be magnified in unsteady simulations that require fewer solver iterations between mesh adaptations.

Achieving scalable performance for dynamic irregular applications is eminently challenging. Private address space methodologies have been making steady progress toward this goal; however, they suffer from complex implementation requirements. The use of a global address space greatly simplifies the programming task, but can degrade the performance of dynamic adaptive applications. Previous work [11] attempted to implement a dynamically evolving mesh adaptation code using shared-memory algorithms and OpenMP-style directives. This naive programming strategy resulted in extremely poor performance, compared to the MPI counterpart, since data locality issues were not properly addressed. In this paper, we have shown that it is possible to achieve message-passing performance using the CC-SAS programming technique by carefully following the same high-level strategies. This approach focuses on spatial locality through methods such as data remapping and replication, which are traditionally not considered a part of the shared-memory programming paradigm. In addition, fine-grained synchronizations are generally nonscalable, and may need to be completely eliminated on massively parallel systems. Future work with adaptive irregular applications will investigate whether CC-SAS can remain competitive with message-passing codes on larger numbers of processors.

ACKNOWLEDGMENTS

The work of the first two authors was supported by the National Science Foundation under Grant ESS-9806751. The second author was also supported by a Presidential Early Career Award for Scientists and Engineers (PECASE) and a Sloan Research Fellowship. The work of the third author was supported by the Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under Contract DE-AC03-76SF00098. This manuscript won the Best Student Paper Award at the 2000 Supercomputing Conference (SC2000), held November 4–10 in Dallas, Texas.

REFERENCES

1. R. J. Anderson and L. Snyder, A comparison of shared and nonshared memory models of parallel computation, *Proc. IEEE* **79** (1991), 480–487.
2. J. E. Barnes and P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm, *Nature* **324** (1986), 446–449.
3. R. Biswas and R. C. Strawn, A new procedure for dynamic adaption of three-dimensional unstructured grids, *Appl. Numer. Math.* **13** (1994), 437–452.
4. M. D. Dikaiakos and J. Stadel, A performance study of cosmological simulations on message-passing and shared-memory multiprocessors, *in* "Proc. 10th ACM International Conference on Supercomputing, ACM SIGARCH, Philadelphia, PA, 1996," pp. 94–101.
5. G. Karypis and V. Kumar, "ParMETIS: Parallel graph partitioning and sparse matrix ordering," Department of Computer Science, University of Minnesota, Minneapolis, MN. [Available at http://www-users.cs.umn.edu/ ~ karypis/metis.]
6. G. Karypis and V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs, *SIAM Rev.* **41** (1999), 278–300.
7. P. Liu and S. N. Bhatt, Experiences with parallel N-body simulation, *in* "Proc. 6th ACM Symposium on Parallel Algorithms and Architectures, ACM SIGACT and SIGARCH, Cape May, NJ, 1994," pp. 122–131.
8. M. Martonosi and A. Gupta, Tradeoffs in message passing and shared memory implementations of a standard cell router, *in* "Proc. 18th International Conference on Parallel Processing, Pennsylvania State University, University Park, PA, 1989," pp. III:88–III:96.
9. T. A. Ngo and L. Snyder, On the influence of programming models on shared memory computer performance, *in* "Proc. Scalable High Performance Computing Conference, Williamsburg, VA, 1992," pp. 284–291.
10. L. Oliker and R. Biswas, PLUM: Parallel load balancing for adaptive unstructured meshes, *J. Parallel Distrib. Comput.* **52** (1998), 150–177, doi:10.1006/jpdc.1998.1469.
11. L. Oliker and R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, *IEEE Trans. Parallel Distributed Systems* **11** (2000), 931–940.
12. L. Oliker, R. Biswas, and H. N. Gabow, Parallel tetrahedral mesh adaptation with dynamic load balancing, *Parallel Comput.* **26** (2000), 1583–1608.
13. L. Oliker, X. Li, G. Heber, and R. Biswas, Ordering unstructured meshes for sparse matrix computations on leading parallel systems, *in* "Parallel and Distributed Processing" (J. Rolim *et al.*, Eds.), Lecture Notes in Computer Science, Vol. 1800, pp. 497–503, Springer-Verlag, Berlin, 2000.
14. Y. Saad, "Iterative Methods for Sparse Linear Systems," PWS, Boston, MA, 1996.
15. J. K. Salmon, "Parallel Hierarchical *N*-body Methods," Ph.D. thesis, California Institute of Technology, 1990.
16. H. Shan and J. P. Singh, A comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessor, *Intl. J. Parallel Programming* **29** (2001), 283–318.
17. H. Shan and J. P. Singh, Parallel sorting on cache-coherent DSM multiprocessors, *in* "Proc. Supercomputing '99, ACM SIGARCH and IEEE Computer Society, Portland, OR, 1999."

18. J. P. Singh, A. Gupta, and M. Levoy, Parallel visualization algorithms: Performance and architectural implications, *IEEE Computer* **27** (1994), 45–55.

19. J. P. Singh, J. L. Hennessy, and A. Gupta, Implications of hierarchical *N*-body methods for multiprocessor architectures, *ACM Trans. Comput. Systems* **13** (1995), 141–202.

20. J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, Load balancing and data locality in adaptive hierarchical N-body methods: Barnes–Hut, fast multipole, and radiosity, *J. Parallel Distrib. Comput.* **27** (1995), 118–141, doi:10.1006/jpdc.1995.1077.

21. R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid, "Aztec: A massively parallel iterative solver library for solving sparse linear systems," Sandia National Laboratories, Albuquerque, NM. [Available at http://www.cs.sandia.gov/CRF/aztec1.html.]

22. C. Walshaw, M. Cross, and M. G. Everett, "Jostle: Parallel graph/mesh partitioning and load-balancing software," University of Greenwich, UK. [Available at http://www.gre.ac.uk/ ~ jjg01.]

23. R. D. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, *Concurrency: Practice and Experience* **3** (1991), 457–481.

24. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, *in* "Proc. 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995," pp. 24–36.

HONGZHANG SHAN is currently working for Storage Network. He received his Ph.D. in computer science from Princeton University in June 2001, and his B.S. and M.S. in computer science from Tsinghua University, Beijing, China, in 1991 and 1993. His research interests include parallel programming models, applications, and architectures, as well as storage networks and file systems.

JASWINDER PAL SINGH is an associate professor in the Department of Computer Science at Princeton University, and the director of the Program in Integrative Information, Computer and Computational Sciences (PICaSso), a multidepartment, university-wide interdisciplinary program at the boundary of computer science and a broad range of application areas. He obtained his Ph.D. from Stanford University in 1993, and his B.S.E. from Princeton in 1987. At Stanford, he participated in the DASH and FLASH multiprocessor projects, leading the applications efforts there. He has led the development and distribution of the SPLASH and SPLASH-2 suites of parallel programs, which are widely used in parallel systems research. At Princeton, he leads the PRISM research group, which does application-driven research in scalable parallel and distributed computing as well as algorithmic research in applying high-performance computing to areas such as computational protein structure prediction, computational immunology, computer graphics, and information services. He has co-authored a graduate textbook titled "Parallel Computer Architecture: A Hardware-Software Approach." He received the Presidential Early Career Award for Scientists and Engineers in 1997, and is a Sloan Research Fellow.

LEONID OLIKER is currently a computer scientist in the Future Technologies Group at the National Energy Research Scientific Computing Center (NERSC), located at Lawrence Berkeley National Laboratory (LBNL). In 1991, he received a B.S.E. in computer engineering from the University of Pennsylvania and a B.S. in finance from the Wharton School of Business. In 1998, he received his Ph.D. in computer science from the University of Colorado. Since then, he has held positions as a visiting researcher and a postdoctoral scientist at the Research Institute for Advanced Computer Science (RIACS) at NASA Ames Research Center, and as a postdoctoral fellow in the Scientific Computing Group at LBNL. Dr. Oliker has published over 30 technical papers in the area of high-performance computing. His research interests include the study of adaptive algorithms on advanced parallel architectures, job scheduling for effective system performance, resource management for mobile computing, and performance of processor-in-memory clusters.

RUPAK BISWAS received his B.Sc. (Honours) in physics (1982) and his B.Tech. in computer science (1985), both from the University of Calcutta, India, and his M.S. (1988) and Ph.D. (1991) in computer science from Rensselaer Polytechnic Institute. He is currently a senior computer scientist with NASA

Ames Research Center. In the past, he was a senior research scientist with Computer Sciences Corporation and with Veridian/MRJ and a staff scientist with the Research Institute for Advanced Computer Science (RIACS), all at NASA Ames. He is the Group Leader of the Algorithms, Tools, and Architectures (ATA) Group that performs research in computer science technology for high- performance scientific computing. The ATA Group is part of the NASA Advanced Supercomputing (NAS) Division of NASA Ames. Dr. Biswas has published over 85 technical papers in journals and major conferences, and has given several invited talks at home and abroad. He has guest edited special issues of Parallel Computing, Journal of Parallel and Distributed Computing, and Applied Numerical Mathematics. His current research interests are in dynamic load balancing for NUMA and multithreaded architectures, analyzing and improving single-processor cache performance for irregular applications, scheduling strategies for heterogeneous distributed multiresource servers in NASA's Information Power Grid (IPG), mesh adaptation for mixed-element unstructured grids, resource anagement for mobile computing, and the scalability and latency analysis of key NASA algorithms and applications.