Parallel Computing Strategies for Irregular Algorithms

RUPAK BISWAS
NASA Ames Research Center

LEONID OLIKER and HONGZHANG SHAN
Lawrence Berkeley National Laboratory

Parallel computing promises several orders of magnitude increase in our ability to solve realistic computationally-intensive problems, but relies on their efficient mapping and execution on large-scale multiprocessor architectures. Unfortunately, many important applications are irregular and dynamic in nature, making their effective parallel implementation a daunting task. Moreover, with the proliferation of parallel architectures and programming paradigms, the typical scientist is faced with a plethora of questions that must be answered in order to obtain an acceptable parallel implementation of the solution algorithm. In this paper, we consider three representative irregular applications: unstructured remeshing, sparse matrix computations, and N-body problems, and parallelize them using various popular programming paradigms on a wide spectrum of computer platforms ranging from state-of-the-art supercomputers to PC clusters. We present the underlying problems, the solution algorithms, and the parallel implementation strategies. Smart load-balancing, partitioning, and ordering techniques are used to enhance parallel performance. Overall results demonstrate the complexity of efficiently parallelizing irregular algorithms.

Keywords: Unstructured mesh adaptation, sparse matrix computations, N-body problems, dynamic load balancing, data ordering, message passing, shared-memory directives, multithreading, PC cluster

1. INTRODUCTION

Parallel computing promises several orders of magnitude increase in our ability to solve realistic computationally-intensive problems, but relies on their efficient mapping and execution on large-scale multiprocessor architectures. Unfortunately, many important applications are irregular and dynamic in nature, making their effective parallel implementation a daunting task. Irregular applications are characterized by non-uniform data access patterns that are inherently at odds with cache-based systems which attempt to hide memory latency by copying and reusing contiguous blocks of data. Dynamic irregular applications are even more challenging since they have computational workloads which grow or shrink at runtime, and require dynamic load balancing to achieve algorithmic scaling on parallel machines.

Authors' addresses:
Rupak Biswas, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035, U.S.A.
Leonid Oliker, National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory, Mail Stop 50F, Berkeley, CA 94720, U.S.A.
Hongzhang Shan, National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory, Mail Stop 50F, Berkeley, CA 94720, U.S.A.

In this paper, we consider three representative irregular applications: unstructured remeshing, sparse matrix computations, and N-body problems.

In addition to these application-specific issues, the proliferation of parallel architectures and programming paradigms requires the typical scientist to answer a plethora of questions in order to obtain an acceptable parallel implementation of the solution algorithm. This paper examines four popular parallel programming paradigms: message passing using MPI, shared memory using OpenMP-style directives, hybrid MPI+OpenMP, and hardware-supported multithreading. The different parallel implementations are tested on a wide spectrum of computer platforms: Cray T3E, SGI Origin2000, IBM SP, Cray (formerly Tera) MTA, and a PC cluster.

We present the underlying problem, the solution algorithm, and the various parallel implementation strategies for each of the three irregular applications. Most modern computer architectures, based on deep memory hierarchies, show acceptable performance for irregular computations only if users care about the proper distribution and placement of data. Single-processor performance depends critically on the exploitation of locality, and parallel performance degrades significantly if inadequate data partitioning causes excessive communication. As a result, smart load-balancing, partitioning, and ordering techniques are required to enhance parallel performance; however, the exact nature depends on the programming paradigm and the architecture.

The remainder of this paper is organized as follows. In Section 2, we give a brief description of the three irregular algorithms that we investigated. The various programming paradigms and computational platforms are described in Section 3. Specific implementation details and performance results are presented in Section 4. Finally, Section 5 concludes the paper with some closing remarks and observations.

2. IRREGULAR ALGORITHMS

In this section, we give a brief overview of the three irregular algorithms that we investigated as part of this work. *Unstructured Remeshing* and *N-Body Problems* are two typical irregular applications that are also dynamic in that the processor workloads and the interprocessor communication can change drastically over time; thus, dynamic load balancing is a critical component. *Sparse Matrix Computations* constitute our third irregular application and are essentially static unless the underlying computational mesh undergoes adaptation; however, performance can be significantly enhanced via smart ordering of the matrix elements. In any case, all three applications are characterized by irregular data access patterns that are inherently at odds with cache-based systems that attempt to hide memory latency by copying and reusing contiguous blocks of data.

2.1 Unstructured Remeshing

Unstructured meshes for computational science and engineering problems allow robust and automatic grid generation around highly complex geometries. Furthermore, the ability to dynamically adapt such unstructured meshes is a powerful tool for efficiently solving problems with evolving physical features. Standard fixed-mesh numerical methods can be made more cost effective by locally refining and coarsening the mesh to capture these phenomena of interest. Highly localized refinement regions are required to accurately capture shock waves, contact discontinuities, vortices, and shear layers. Unfortunately, an efficient parallelization of adaptive unstructured remeshing is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grids.

In this work, we consider a two-dimensional unstructured remeshing algorithm based on triangular elements; complete details of the three-dimensional procedure are given in [5], [21]. Briefly, local mesh adaptation involves adding vertices to the existing grid in regions where some user-specified error indicator is high, and removing vertices from regions where the indicator is low. The advantage is that relatively few vertices need to be added or deleted at each remeshing step; however, complicated logic and data structures are required to keep track of the mesh objects (vertices, edges, elements). It involves a great deal of pointer chasing, leading to irregular and dynamic data access patterns.

A triangular element can be refined in different ways; however, the most popular strategy is to bisect all three of its edges. This type of subdivision is called isotropic; however, a refined mesh will be nonconforming unless all its triangles are isotropically subdivided. To obtain a consistent triangulation without global refinement, anisotropic subdivision is allowed. That is, a triangle can be subdivided into two (three) smaller triangles by bisecting one (two) edge(s). Isotropic and anisotropic refinements of a triangle are shown in Figure 1. The process of creating a consistent triangulation is called a closure operation, which may require several iterations if refinement propagation is allowed.

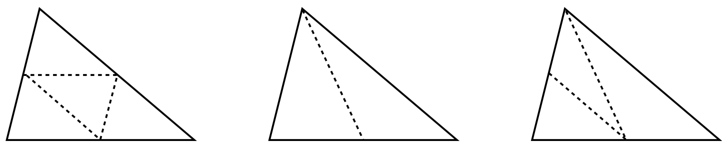

Fig. 1. Isotropic and anisotropic refinements of a triangle.

In a parallel environment, sophisticated dynamic load balancing techniques must be employed as the computational workload and the communication volume grow and shrink nonuniformly at runtime depending on the adaptation. We used the METIS graph partitioner [16] for this work because of its good overall performance and wide availability. It uses a multilevel algorithm consisting of three main steps: coarsen the graph to be partitioned, partition the coarse graph, and project the partitioned graph back to the original graph. During the coarsening phase, METIS gradually reduces the size of the graph by collapsing vertices using a heavy edge matching scheme. A greedy graph growing algorithm is then used to partition the coarsest graph. This partitioned coarse graph is finally uncoarsened back to the original using a combination of boundary greedy and Kernighan-Lin refinement [17] to further reduce the overall edgecut and improve the load balance.

2.2 Sparse Matrix Computations

Unlike the adaptive dynamic unstructured remeshing algorithm discussed in the previous section, our second application is static but still irregular as it deals with the process of obtaining numerical solutions for sparse linear systems defined over unstructured meshes. In a way, unstructured remeshing is an enabling tool that allows the efficient parallel solution of the governing partial differential equations (PDEs) modeling the underlying physical problem of interest. A discretization of the PDEs usually leads to large sparse matrices, for which special solution techniques are normally used whenever the large number of zero elements are not stored.

Conjugate Gradient (CG) utilizes Krylov subspaces and is perhaps the most popular iterative algorithm [24] to solve large symmetric positive-definite sparse linear systems of the form $Ax = b$. The method starts from an initial guess x_0 of the vector x. Since the convergence rate of CG depends on the spectral condition number of the coefficient matrix A, it is typically used with a preconditioner for ill-conditioned systems. One broad class of effective preconditioners is based on incomplete LU (ILU) factorizations of A. For this work, we use the ILU(0) preconditioner where all fill elements not at the nonzero positions of A are discarded. For most practical applications, the sparse matrix-vector multiply and the triangular solves are the most expensive operations within preconditioned CG (PCG). An outline of the PCG algorithm is given in Figure 2.

```
For an initial guess $x_0$, compute $r_0 = b - Ax_0$, $p_0 = z_0 = M^{-1}r_0$
for $j = 0, 1, \ldots$, until convergence
        $\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
        $x_{j+1} = x_j + \alpha_j p_j$
        $r_{j+1} = r_j - \alpha_j Ap_j$
        $z_{j+1} = M^{-1}r_{j+1}$
        $\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
        $p_{j+1} = z_{j+1} + \beta_j p_j$
endfor
```

Fig. 2. The preconditioned Conjugate Gradient algorithm.

Partitioning the sparse matrix is required on distributed-memory architectures, but can be beneficial even on shared-memory machines by enforcing data locality. With graph partitioning, some level of data locality is indirectly achieved by minimizing interprocessor communication, but not at the cache level. On the other hand, special ordering strategies can be used to improve the profile of a matrix, thereby enhancing the efficiency of the solution algorithms. We investigated both these techniques in our experiments.

Partitioners attempt to improve data locality by minimizing interprocessor communication; improving cache performance is usually not an objective. We used the METIS multilevel graph partitioner [16] for the experiments in this paper. Details about METIS have been given earlier in Section 2.1.

Linearization is extremely effective in enhancing cache performance. Special numbering techniques have considerably improved the efficiency of sparse matrix computations. Cuthill and McKee [8] proposed a simple algorithm, called CM, based on ideas from graph theory. Levels of increasing distance from a pseudoperipheral vertex are first constructed. The enumeration is then performed level-by-level with increasing vertex degree within each level. In this work, we used a popular variation called reverse Cuthill-McKee (RCM) [10] that further improves the matrix profile by reversing the CM ordering.

Unlike the CM algorithms that operate on a pure graph, one could use space-filling curves (SFCs) to linearize objects in a higher dimensional space. For example, an SFC can easily order all the hexahedral elements of a three-dimensional structured grid. Furthermore, the serialization of hierarchically adaptive structured grids using an SFC enhances data locality, and therefore improves cache reuse, for mesh and graph problems. However, for unstructured grids, an SFC introduces an artificial structure in that the construction depends on the embedding. To overcome this drawback, a novel approach called self-avoiding walk (SAW) [12] that uses a mesh-based technique has been recently developed.

In two dimensions, a SAW visits all the triangular elements exactly once such that two consecutive triangles (in the SAW) share an edge or a vertex. A SAW goes over vertices only when the triangles following one another in the enumeration do not share an edge. The construction complexity is linear in the number of triangles; however, SAWs can be easily generated in parallel for hierarchical adaptation as they can be rebuilt purely locally. Note that a SAW is not a Hamiltonian path; however, Hamiltonicity of the underlying dual graph implies the existence of a SAW that goes only over edges.

2.3 N-Body Problems

The N-body problem is a classical one, and arises in many areas of science and engineering such as astrophysics, molecular dynamics, computer graphics, and fluid dynamics. Having specified the initial positions and velocities of N interacting particles, the problem is to find their positions after a certain period of time. Since there are potentially $O\left(N^2\right)$ interactions every time step, any naive algorithm will have extremely poor performance. In addition, it may incur memory limitations, unbalanced workloads, and poor computation-to-communication ratios. It is an excellent example from the class of dynamic irregular applications.

The Barnes-Hut algorithm [1] is a widely-used method to solve this problem. It reduces the number of interactions to $O\left(N \, log \, N\right)$ by partitioning the particles into space-separated clusters. It is able to accomplish this improvement by taking advantage of the locality inherent in space partitioning: groups of particles far from one another can approximate their effect on each other instead of calculating it precisely for every particle. Figure 3 shows a simple two-dimensional example of 36 particles partitioned among 4 processors with each leaf cell containing at most 4 particles.

There are three primary phases within each iteration of the Barnes-Hut algorithm. In the tree building phase, an octree is constructed to represent the spatial distribution of the particles. It is implemented by recursively partitioning the three-dimensional space into eight subspaces until the number of particles in each subspace is below a certain threshold. In the second phase, the force interactions between individual particles are computed. Each particle traverses the octree starting from the root. If the distance between a particle 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

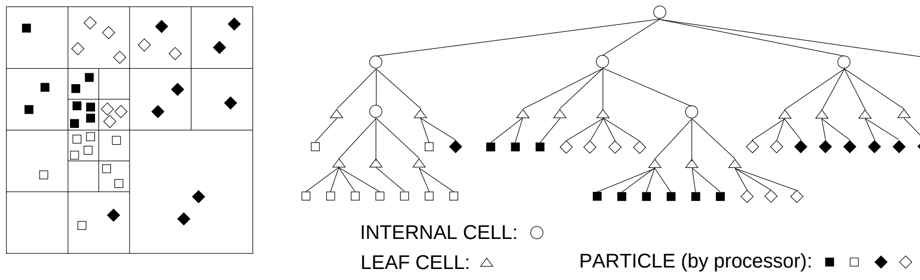

Fig. 3. A simple example of the Barnes-Hut algorithm applied to 36 particles that are partitioned among 4 processors.

in preparation for the next iteration.

Basically, the algorithm requires all-to-all communication for global reduction operations, and demonstrates unpredictable send/receive patterns. The force calculation phase dominates the other phases; however, its performance is critically dependent on the quality of the tree building phase because the space partitioning must successfully expose all reasonable simplifications.

3. PROGRAMMING PARADIGMS AND COMPUTATIONAL PLATFORMS

Over the last five years, a few different parallel architectures have emerged, each with its own set of programming paradigms. Even though many of these computing platforms support multiple programming models, they each have a preference for the one that naturally maps to the underlying architecture. In the following subsections, we give a brief description of the four leading programming paradigms and the five computational platforms that we used for the experiments reported in this paper.

3.1 Programming Paradigms

At this time, the four most popular programming paradigms are message passing, shared-memory programming, a hybrid model that combines the message-passing and shared-memory paradigms, and multithreading. We briefly discuss each of these programming models below.

3.1.1 *Message Passing.* Parallel programming with message passing is the most common and mature approach for high-end parallel computers. On distributed-memory architectures, each processor has its own local memory that only it can directly access. All other accesses require a copy of the desired data to be explicitly transferred across the network using a message-passing library such as MPI [18]. To run codes on these machines, programmers must decide how data should be distributed among the local memories, communicated between processors during the course of the computation, and reshuffled when necessary for dynamic problems. This model increases code complexity, particularly for irregular applications; however, the benefits lie in enhanced performance for coarse-grained communication and implicit synchronization through blocking communication.

3.1.2 *Shared-Memory Programming.* Using a single-image shared-memory system can immensely simplify the programming task compared to message-passing implementations. In distributed shared-memory architectures, each processor has local memory but also can directly access any memory location in the system. Thus, parallel programs are relatively easier to implement by inserting compiler directives into the code to distribute loop iterations and computational threads among the processors. Currently, OpenMP [23] is the standard model that enables programmers to develop shared-memory parallel applications. However, performance may suffer from poor spatial locality of physically distributed shared data, and portability is compromised as fine tuning is sometimes necessary to achieve optimal efficiency.

3.1.3 *Hybrid Programming.* Recent advances in technology have led to the development of parallel architectures that contain a larger number of networked SMPs. Pure MPI codes can be easily ported to these clustered systems, since message passing is required among the SMP nodes. However, message passing within each SMP may not be the most effective use of the system. A novel programming paradigm combines two layers of parallelism: OpenMP shared-memory codes within each SMP, and MPI message passing among the SMPs. This hybrid strategy allows codes to potentially benefit from both fine-grained loop-level and coarse-grained domain-level parallelism. It offers an advantage on systems where MPI is unoptimized due to issues such as a poorly implemented communication layer within an SMP, or because of hardware limitations. Unfortunately, it is currently unclear whether the performance gains of this approach compensate for the increased programming complexity and the loss of performance portability even though it may be the best mapping to the underlying architecture.

Figure 4 shows a schematic of this hybrid programming paradigm. Two MPI tasks ($task_1$ and $task_2$) are initiated on processors p_1 and p_2 of an 8-way SMP node. Each MPI task then spawns four OpenMP threads, where each individual thread is assigned to a processor. For example, $task_2$ spawns $thread_{23}$ which is assigned to processor p_7. For the experiments reported in this paper, each processor ran exactly one thread. Threads spawned by the same task communicate implicitly through shared memory using OpenMP constructs. But threads spawned by different tasks communicate explicitly using MPI.

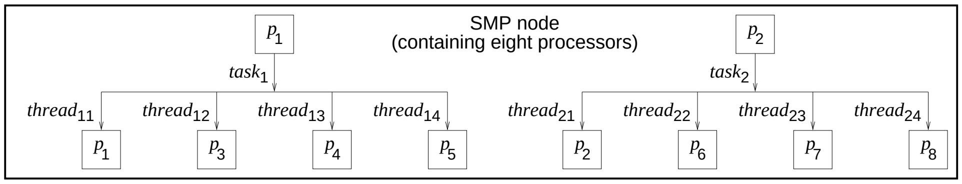

Fig. 4. A schematic of hybrid programming where an SMP node containing eight processors runs two MPI tasks and four OpenMP threads per task.

3.1.4 *Multithreading.* Multithreading is available both in software and in hardware. Hardware-supported multithreading has lately received considerable attention as a promising way to hide memory latency in high-performance computers. By tolerating memory latency and using low-level synchronization directives, multithreaded machines can potentially utilize a larger fraction of their processing power while providing access to a large and uniform shared memory. Parallel programmability is significantly simplified since the user has a global view of the memory, and need not be concerned with data placement. These machines are therefore particularly well-suited for irregular and dynamic applications.

3.2 Computational Platforms

We conducted our experiments on five different computing platforms. A brief description of each machine is given below.

3.2.1 *Cray T3E.* The Cray T3E is a distributed-memory machine and only supports the message-passing programming paradigm. Most of the MPI results presented in this paper were obtained on the 644-processor T3E at Lawrence Berkeley National Laboratory. Each T3E node consists of a 450 MHz DEC Alpha processor (900 Mflops peak theoretical floating-point speed), 256 MB of main memory, a 96 KB secondary cache, and is connected to other nodes through a three-dimensional torus.

3.2.2 *SGI Origin2000.* The SGI Origin2000 is a scalable, hardware-supported cache-coherent nonuniform memory access (CC-NUMA) system, with an aggressive communication architecture. The hardware makes all memory equally accessible from a software perspective by sending memory requests through routers located on the nodes. Memory access time is nonuniform, depending on how far away the word lies from the processor. The interconnection network is a hypercube, bounding the maximum number of memory hops to a logarithmic function of the number of processors. The machine supports MPI, OpenMP, and the hybrid MPI+OpenMP programming paradigms. Our shared-memory implementations of the irregular algorithms use SGI's native pragma directives, which create IRIX threads. Rewrites to OpenMP are straightforward, but have not yet been done.

The performance results presented here were obtained on the 64-processor Origin2000 at NASA Ames Research Center. Each Origin2000 node is a symmetric multiprocessor (SMP) containing two 250 MHz MIPS R12000 processors and 512 MB of local memory. Each processor also has separate 32 KB primary instruction and data caches, and a 2-way set-associative 4 MB secondary cache where only it can fetch and store data. If a processor refers to data that is not in cache, there is a delay while a copy of the data is fetched from memory. When a processor modifies a word of data, all other copies of the cache line containing that word are invalidated.

3.2.3 *IBM SP.* The hybrid architecture used in our experiments with sparse matrix computations was the IBM SP system at San Diego Supercomputing Center (SDSC). The machine is a cluster of 144 SMP nodes, containing a total of 1,152 processors. Each SMP is equipped with 4 GB of memory shared among its eight 222 MHz Power3 processors, and connected via a crossbar. Compared to traditional shared-bus designs, the crossbar technology reduces bandwidth contention to main memory. Each processor has a 64 KB 128-way set-associative primary cache, and a 4 MB 4-way set-associative secondary cache with its own private bus. The SMPs are connected to one another via an omega-type switching interconnect. Currently, only a maximum of four MPI tasks are available within each SMP when using this fast switch. The machine supports pure MPI and the hybrid MPI+OpenMP programming paradigms. OpenMP is only available within each 8-processor SMP node; thus, message passing is necessary to utilize all the processors.

For the N-body simulations, we used the IBM SP system in the NERSC Division at Lawrence Berkeley National Laboratory. This machine is a cluster of 184 nodes, each containing 16 processors and 16 GB of memory. Each processor is a 375 MHz Power3+ with an 8 MB level 2 cache. Unlike the SDSC SP, all 16 MPI tasks within an SMP can be used across the fast switch. There are no other significant architectural differences between the two SP systems used in our study.

3.2.4 *Cray MTA*. Cray (formerly Tera) has designed and built the radically different MTA system that can accommodate up to 256 custom multithreaded processors. At every cycle, each processor context switches with zero overhead among as many as 128 instruction streams, choosing from only those that are ready to execute their next instruction. Furthermore, each stream can have up to eight outstanding memory references, thereby increasing the processors' memory latency tolerance to 1,024 cycles. Performance thus largely depends on having a large number of concurrent computation threads.

The 8-processor 255 MHz MTA machine used for this work was installed at SDSC in 1998. It has a 8 GB flat uniform shared memory physically distributed across several banks that are connected through a toroidal network to the processors. Because of a memory hashing scheme, logically adjacent words are placed on different memory banks. Each word contains a full/empty bit which enables synchronization among the threads via load/store instructions without operating system intervention. Synchronization among threads may stall one of the threads, but not the processor on which the threads are running since each processor can run multiple threads. Explicit load balancing is avoided since the dynamic scheduling of work to threads provides the ability of keeping the processors saturated. Finally, once a code is written in the multithreaded model, no additional work is required to run it on multiple processors as there is no difference between uni- and multiprocessor parallelism.

3.2.5 *PC Cluster*. The fifth and final platform that is becoming increasingly attractive for high-end scientific computing are PC clusters. The one used for this study is a cluster of eight 4-way 200 MHz Pentium Pro SMPs (a total of 32 processors) located at Princeton University. Each of the 32 processors has separate 8 KB data and instruction primary caches, and a 4-way set-associative 512 KB secondary cache. Each node has 512 MB of main memory, runs Windows NT 4.0, and is connected to other nodes via either the Myrinet [6] or the Giganet [11] network.

For such PC-SMP clusters, the preferred programming paradigm is not obvious. Currently, message passing and software distributed shared memory (DSM) are the two popular paradigms for these systems. The message-passing programming model is built in software on top of Giganet by the VIA interface [31] using MPI/Pro from MPI Software Technology, Inc. The selection of MPI/Pro allows us to avoid a potentially poor implementation of the communication layer.

Page-based shared virtual memory (SVM) is one of the most common ways to support software DSM on clusters. SVM provides replication and coherence at the page granularity, uses a relaxed memory consistency model to alleviate problems with false sharing and fragmentation, and provides multiple writer protocols to enable more than one processor to locally modify copies of a page between synchronizations. Recently, a new promising protocol for SVM called GeNIMA [2] has been developed that uses general-purpose network interface support to significantly reduce overheads. Our shared address space (SAS) programming model [14] uses GeNIMA which is built on VMMC, a high-performance user-level virtual memory mapped communication library [9]. VMMC itself runs on top of Myrinet.

However, since each node of the PC cluster has four processors, a more natural programming paradigm could be SAS within an SMP while using MPI among the SMP nodes. Here, the hardware supports cache coherence for the SAS codes, while communication between SMPs relies on the network through message passing. The benefits and drawbacks of this mixed programming strategy is similar to those discussed for hybrid MPI+OpenMP in Section 3.2.3.

4. IMPLEMENTATION DETAILS AND PERFORMANCE RESULTS

In this section, we provide specific implementation details of our three irregular applications using the different programming paradigms mentioned in Section 3.1. Performance results are then presented on a variety of platforms that were described in Section 3.2.

4.1 Unstructured Remeshing

The unstructured mesh used for the experiments in this paper is the one often used to simulate flow over an airfoil (see Figure 5 for the coarse initial mesh containing 14,605 vertices and 28,404 triangles). At transonic Mach numbers, shocks form on both the upper and lower airfoil surfaces that then propagate to the far field. Mesh refinement is usually required in these regions as well as around the stagnation point at the leading edge of the airfoil. This actual scenario is modeled by geometrically adapting regions corresponding approximately to the locations of the stagnation point and the shocks. This strategy allows us to investigate the performance of the mesh adaptation and load balancing algorithms in the absence of a numerical solver. After five levels of refinement, the mesh was more than 40 times larger and consisted of 488,574 vertices and 1,291,834 triangles. The computational mesh after the second refinement is shown in Figure 5.

4.1.1 *Message Passing on the T3E.* The message-passing version of the unstructured remeshing algorithm was implemented in MPI within the PLUM framework [3], [19]. PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured-grid applications. It differs from most other load balancers in that it dynamically balances processor workloads with a global view. It repeatedly uses the dual graph of the initial mesh to keep the connectivity and partitioning complexity constant during the course of an adaptive computation. In addition, a fast heuristic remapping technique and an

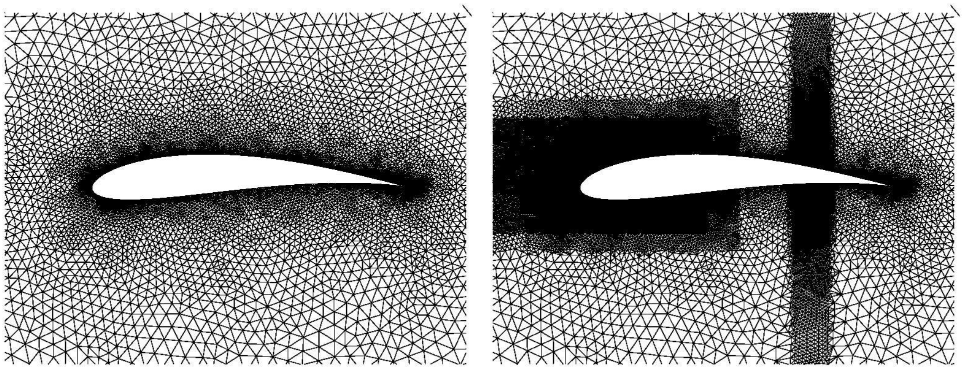

Fig. 5. A close-up view of the initial triangular mesh around the airfoil (left) and after the second refinement (right).

efficient bulk data movement strategy help PLUM minimize the data redistribution cost.

PLUM consists of a partitioner and a remapper that load balance and redistribute the computational mesh when necessary. After an initial partitioning and mapping of the unstructured mesh, a solver executes several iterations of the application. However, in the experiments reported in this paper, a solver was not used as we wanted to focus only on the remeshing algorithm. The adaptation procedure then marks edges for refinement or coarsening based on an error indicator. Again, only refinement was performed in this work because of two reasons. First, the test case is a steady-state problem and does not really require mesh coarsening; and second, coarsening per se does not bring in any additional irregularity to the problem. At this point, it is possible to exactly predict the new mesh without performing the adaptation, resulting in several beneficial side effects [19], [26]. Program control is thus passed to the load balancer which uses a repartitioning algorithm to divide the new mesh into subgrids. A variety of partitioners can be used within PLUM [4]; in this work, we use the METIS [16] parallel partitioner. All necessary data is then redistributed, the computational mesh actually refined, and the simulation restarted. In short, there are three major steps in parallel remeshing: refinement, repartitioning, and remapping. Figure 6 gives a high-level overview of the simplified version of the entire PLUM iterative process.

In the MPI implementation (extensive details can be found in [21]), each processor owns a submesh and maintains the local data structures to represent it. Thus, each mesh object (vertex, edge, element) has a local index. To exchange information with neighbors, each processor also maintains a mapping between its local index and the global index, which is the index of the mesh object in the global mesh. Generating and maintaining these indices for a dynamic irregular application is non-trivial. The coarsening phase of the parallel METIS partitioner is also somewhat complicated. To find a match for vertices on partition boundaries, a try-confirm strategy is used. This is because a message must be received from the remote processor to confirm the matching as other processors may also be trying to match their own boundary vertices with the same vertex. Finally, during the METIS uncoarsening phase, each processor reconsiders the ownership of its boundary vertices to reduce the overall

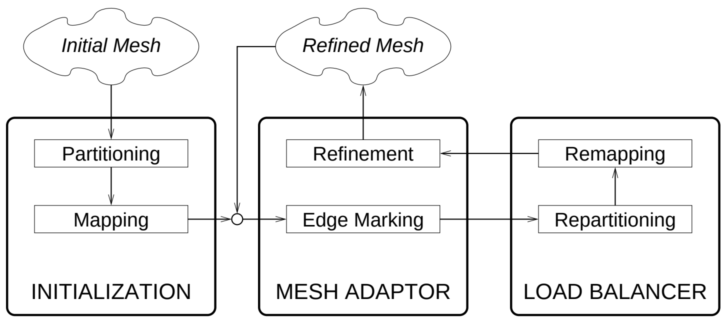

Fig. 6. Unstructured remeshing with dynamic load balancing using the PLUM framework.

edge cut and improve the quality of load balance. Due to private address spaces in MPI and the lack of global information, these decisions are made based on an incomplete view.

Several options may be set within PLUM, including predictive or non-predictive refinement, global or diffusive partitioning, and synchronous or asynchronous communication. Table 1 presents the results for the best combination of these options (predictive refinement, global partitioning, asynchronous communication) on a T3E, through five refinement levels.

Table 1. Runtimes (in seconds) using MPI on the T3E.

	REMESH			
P	Refine	Partition	Remap	Total
32	0.604	1.029	2.018	3.651
64	0.307	0.912	0.969	2.188
128	0.162	1.045	0.525	1.732
256	0.088	1.784	0.279	2.151

Observe that the refinement time decreases as the number of processors increases, since there is less work per processor and little communication in the refinement phase. However, the speedup values become progressively poorer due to the uneven distribution of refinement workloads across the processors. This is expected since our load balancing objective is to produce a balanced mesh for the more expensive solver phase. On the other hand, partitioning times eventually increase with the number of processors because the amount of work and the communication overhead of the partitioner increases superlinearly. A scalable repartitioner would obviously improve the overall parallel performance significantly. However, load balancing quality is excellent, with imbalance factors ranging from 1.05 to 1.16 between 32 and 256 processors. Finally, the data remapping time decreases with the number of processors while satisfying our bottleneck communication model which expresses remapping cost as a function of the maximum (not total) communication among processors [19].

4.1.2 *Shared-Memory Programming on the Origin2000.* Unlike the MPI implementation, a single complete shared mesh is maintained in the shared-memory version. A potential drawback of this strategy is that the shared data structures cannot be easily changed without synchronization. This is a critical issue since mesh refinement involves modifying several data structures by inserting new mesh objects and altering their relationships. The synchronization need can be dramatically reduced by letting each processor precompute its number of new mesh objects, and applying the range to the global data structures. This enables the processor to modify within its own range with only a few synchronizations, but at the cost of some additional complexity.

The shared-memory implementation on the Origin2000 must also take advantage of the machine's CC-NUMA feature in order to achieve scalable performance [20]. Using a partitioner rather than simply splitting the data structures guarantees that each processor 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 processor to obtain good temporal and spatial data locality.

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

No explicit data remapping is necessary for program orchestration in the CC-NUMA implementation. We therefore allowed the unstructured remeshing process to progressively become more unbalanced. This strategy would not work in a hybrid environment since eventually all the memory could be exhausted on a single node although the total requirements are satisfiable. Even on distributed shared-memory systems, this strategy could disproportionately increase the mesh refinement times. However, it does show the possible flexibility of shared-memory platforms since one has the choice of remapping the mesh or not. A more general shared-memory implementation would include a remapping phase, but it has not been done at this time.

Runtimes for the CC-NUMA code on the Origin2000 are presented in Table 2. For the purpose of comparison, results for the MPI version running on the Origin2000, with data remapping after each repartitioning, are also presented. Notice that the refinement times are significantly higher for the shared-memory code. This is because the quality of load balance deteriorates in the absence of explicit data remapping. Refinement speedups are also poorer because the load balance problem is exacerbated with an increase in the number of processors. However, the partitioning times are better than those for the original MPI version of METIS; a remarkable feat given the modest amount of effort that has gone into developing the shared-memory version. The load imbalance factors range from 1.02 to 1.05

Table 2. Runtimes (in seconds) on the Origin2000.

	REMESH							
	CC-NUMA				MPI			
P	Refine	Partition	Remap	Total	Refine	Partition	Remap	Total
16	1.725	0.559	0.000	2.284	0.610	0.568	2.110	3.288
32	1.727	0.705	0.000	2.432	0.302	0.764	1.467	2.533
64	2.233	1.196	0.000	3.429	0.225	1.533	1.146	2.904

between 16 and 64 processors for CC-NUMA compared to 1.04 to 1.13 for MPI while the number of edge cuts is about 20% higher for CC-NUMA. Overall, the total runtimes are comparable even though the implementations are fundamentally different.

4.1.3 *Multithreading on the MTA*. Our multithreaded version of the mesh adaptation code uses low-level locks to prevent potential race conditions [20]. Basically, when a thread processes a given triangle, it locks the corresponding vertices and edges to prevent neighboring triangles from being simultaneously updated. The load is implicitly balanced by the operating system, which dynamically assigns triangles to threads. No partitioning, remapping, or graph coloring is therefore required, greatly simplifying the programming effort.

Table 3 presents runtimes using 60 streams per processor on the MTA. The number of streams is easily changed through a compiler directive. Results show efficiencies of 97%, 92%, and 83% using 2, 4, and all 8 processors. As we increase the number of processors, the number of active threads increases proportionately while the runtimes become very small. As a result, a greater percentage of the overall time is spent on thread management, causing a decrease in efficiency (Amdahl's Law). Increasing the number of streams per processor reduces the overall runtime but is also less efficient.

Table 3. Runtimes (in seconds) using multithreading on the MTA.

P	REMESH
1	2.72
2	1.40
4	0.74
8	0.41

Another potential source of the degradation in efficiency are memory hot-spots that occur when many streams attempt to access the same memory location. This may be happening frequently within the mesh adaptation code, since multiple streams are sharing edges and vertices of the triangles. Since an MTA memory bank can only handle one reference approximately every 36 clock ticks, multiple references to the same bank effectively become serialized. To help alleviate this problem, a limited number of special hot-spot caches are provided in hardware, which allow consecutive references to a single location every two clock ticks. Overall however, we found that there is sufficient instruction- and thread-level parallelism in the unstructured mesh adaptation code to tolerate the overheads of memory access and lightweight synchronization, if enough streams can be used.

4.2 Sparse Matrix Computations

The sparse matrix A used for our experiments was generated from a two-dimensional mesh containing 661,054 vertices and 1,313,099 triangles. Matrix element a_{ij} was set to a random value in $(0,1)$ if the distance between mesh vertices v_i and v_j was less than 4. Here, the distance between two vertices is defined to be the number of edges on the shortest path connecting them. All other off-diagonal entries were set to zero. This models a local discrete operator where each vertex communicates with all of its neighbors that are no more than three edge lengths away. The diagonal entries of A were set to 40 to make it positive definite. The CG algorithm converged in exactly 13 iterations (tolerance set to 10^{-15}), with the unit vector as the right-hand side b and the zero vector as the initial guess for x. For the PCG experiments, the diagonal entries of A were reduced to 10, causing the original CG to fail. However, the ILU(0) PCG algorithm successfully converged in exactly 18 iterations (tolerance again set to 10^{-15}), given the modified matrix.

4.2.1 *Message Passing on the T3E and the SP.* On the T3E, we use the parallel CG routine, called `AZ_cg`, in the Aztec library [30], which is implemented in MPI. The matrix A is partitioned into blocks of rows, with each block assigned to one processor. The associated components of vectors x and b are distributed accordingly. Communication may be needed to transfer some components of x; packing several components into one message minimizes the total number of messages. However, this optimization can be performed in a pre-processing phase.

Timing results for `AZ_cg` in Table 4 show that SAW is always more than twice as fast as RCM and METIS, which are themselves comparable. Both RCM and METIS, in turn, are almost twice as fast as ORIG (the default ordering of the mesh generator) when using 16 or more processors. However, METIS, RCM, and SAW, all demonstrate excellent scalability (more than 75% efficiency) up to the 64 processors that were used for these experiments, but ORIG seems less scalable (only about 56% efficiency). ORIG has another critical drawback: it requires almost two orders of magnitude more time to initialize the data structures and the communication schedule [22]. This indicates that the ORIG ordering is too inefficient and unacceptable on distributed-memory machines.

Table 4. Runtimes (in seconds) of CG using MPI on the T3E.

	CG			
P	ORIG	METIS	RCM	SAW
8	8.652	7.662	6.185	2.916
16	5.093	2.909	3.198	1.491
32	3.167	1.468	1.662	0.795
64	1.929	0.961	0.882	0.462

Although Aztec is a powerful iterative library, it does not provide a global ILU(0) factorization routine. Thus we had to use the BlockSolve95 software library [15] to conduct the message-passing PCG experiments. BlockSolve95 itself reorders the input matrix to achieve scalable performance. First, the graph is reduced by extracting cliques and identical nodes (i-nodes), allowing the use of higher-level BLAS. Next, the reduced graph is colored, and the vertices grouped and ordered sequentially by color. We therefore investigate what effect, if any, our ordering strategies have on the parallel performance of PCG.

We could not port BlockSolve95 to the T3E because of the large number of MPI tags it requires; hence, our message-passing PCG experiments were conducted on the SP machine. Table 5 presents the runtimes of the BlockSolve95 `BSpar_solve` PCG routine using various partitioning and ordering strategies. Results clearly show that the initial ordering of the matrix plays a significant role in PCG performance, even though the matrix is further reordered by the BlockSolve95 library. Notice that RCM and SAW have an advantage over METIS; however, all three schemes are about an order of magnitude faster than ORIG.

Table 5. Runtimes (in seconds) of PCG using MPI on the SP.

	PCG			
P	ORIG	METIS	RCM	SAW
8	96.41	22.44	14.63	11.60
16	64.23	9.67	7.44	7.29
32	14.75	5.97	4.14	3.88
64	15.11	3.67	2.36	2.46

4.2.2 *Shared-Memory Programming on the Origin2000.* As in Section 4.1.2, we exploit the CC-NUMA feature of the Origin2000 by performing an intelligent initial data distribution. Sections of the sparse matrix A are appropriately mapped onto the memories of their corresponding processors using the default "first touch" data distribution policy. However, the computational kernel is still much simpler to implement than the MPI version. Table 6 shows the CG runtimes using the ORIG, RCM, and SAW orderings of the mesh. As a basis for comparison, we also present runtimes for an MPI implementation on the Origin2000 with the SAW ordering. The PCG algorithm has not yet been implemented in shared-memory mode as it would require sophisticated graph dependency analysis similar to that in the BlockSolve95 library.

Table 6. Runtimes (in seconds) of CG on the Origin2000.

	CG			
	CC-NUMA			MPI
P	ORIG	RCM	SAW	SAW
8	9.824	5.575	5.516	3.815
16	6.205	2.845	2.872	1.926
32	3.584	1.548	1.514	1.075
64	2.365	0.885	0.848	0.905

Observe that the RCM and SAW schemes reduce the runtimes compared to ORIG, indicating that an intelligent ordering algorithm is necessary to achieve good performance and scalability on distributed shared-memory systems. There is little difference between RCM and SAW performance because both techniques reduce the number of secondary cache misses and the non-local memory references of the processors. Recall however from Section 4.2.1 that on the T3E, SAW was about twice as fast as RCM. This performance difference is probably due to the larger cache size of the Origin2000 that reduces the individual effects of the two ordering strategies.

The last two columns of Table 6 compare the CC-NUMA and MPI implementations of CG on the Origin2000 using the SAW ordering. The runtimes are quite comparable, even though the programming methodologies are very different. This indicates that for this class of applications, it is possible to achieve message-passing performance using shared-memory constructs, through careful data ordering and distribution.

4.2.3 *Hybrid Programming on the SP.* For the hybrid implementation of the CG algorithm on the SP, we added OpenMP directives to the Aztec MPI library [30]. A total of eight key loop nests were identified via profiling and subsequently parallelized. To achieve the best possible OpenMP performance, dense vector operations were performed with the threaded vendor-optimized BLAS from IBM's Engineering Scientific Subroutine Library (ESSL). For the same reasons as mentioned in Section 4.2.2, a hybrid PCG implementation is not considered in this paper.

Results are presented in Table 7 for varying numbers of SMP nodes, MPI tasks, and OpenMP threads. Due to limitations in the current switch architecture of the SDSC's SP, the maximum number of MPI tasks is limited to four on each SMP. In addition to ORIG, METIS, RCM, and SAW, a new hybrid scheme comprised of METIS+SAW is presented. This new approach is particularly promising for hybrid architectures. Here, the graph is first partitioned into the appropriate number of MPI tasks using METIS. Next, a SAW ordering is performed on each individual subdomain in parallel.

Table 7. Runtimes (in seconds) of CG using MPI+OpenMP on the SP.

				CG				
P	Nodes	Tasks	Threads	ORIG	METIS	RCM	SAW	METIS+SAW
16	2	1	8	3.375	1.926	1.217	1.139	1.118
	2	2	4	4.125	1.366	1.071	1.018	0.992
	2	4	2	4.782	1.472	1.084	1.019	1.006
	4	4	1	5.186	0.965	0.986	0.914	0.902
32	4	1	8	2.973	0.870	0.651	0.678	0.617
	4	2	4	3.608	0.709	0.618	0.593	0.581
	4	4	2	4.067	0.723	1.120	0.680	0.649
	8	4	1	4.267	0.586	0.607	0.580	0.569
64	8	1	8	2.992	0.473	0.391	0.390	0.372
	8	2	4	3.557	0.452	0.690	0.442	0.407
	8	4	2	3.963	0.466	0.798	0.495	0.460

The METIS+SAW strategy consistently outperforms all others; however, it is only marginally better than pure SAW linearization since cache behavior is significantly more important than interprocessor communication for the CG algorithm [22]. Nonetheless, we expect algorithms with higher communication requirements to benefit from this dual partitioning/ordering approach. Unfortunately, the hybrid MPI+OpenMP implementation offers almost no noticeable advantage over pure MPI while increasing programming complexity. We believe a pure MPI implementation to be a more effective strategy for iterative sparse solvers on clusters of SMPs. Similar conclusions have recently been drawn for other architectures and application domains [7], [13], [22], [27].

The results in Sections 4.2.1, 4.2.2, and here, show that if the underlying mesh were dynamically adapted, a new reordering would be required for efficient parallel performance. Furthermore, a remapping would be necessary to appropriately redistribute the corresponding submatrix onto the processors. A significant overhead is associated with these reordering and remapping phases [19], [20], [26], making the CC-NUMA and the MPI+OpenMP strategies comparable to an MPI implementation. The major difference would be the use of a shared address space (global on an Origin2000, local within an SMP node of an SP) instead of explicit message-passing calls for interprocessor communication.

4.2.4 *Multithreading on the MTA*. The multithreaded implementation of CG on the MTA required only compiler directives. Since the data structures are dynamically allocated pointers, special pragma assertions were used to indicate that there are no loop-carried dependencies. Load balancing is implicitly handled by the operating system which dynamically assigns rows to threads. Results using 60 streams per processor are presented in Table 8. An efficiency of over 90% was achieved for CG using the ORIG ordering, indicating that there is enough thread- and instruction-level parallelism to tolerate the relatively high overhead of memory access. Performance degrades slightly between four and eight processors because with an increasing number of active threads, the runtimes become very small and a greater percentage of the total time is spent on thread management.

Table 8. Runtimes (in seconds) using multithreading on the MTA.

	CG		PCG
P	ORIG	SAW	ORIG
1	9.86	9.74	80.34
2	5.02	5.01	50.02
4	2.53	2.64	29.18
8	1.35	1.36	17.29

Notice that SAW ordering has a negligible effect on CG performance for this cache-less machine. Thus, the programming and runtime overheads associated with partitioning and linearization schemes are absent. Furthermore, reordering and remapping are not required even if the underlying mesh is adapted. Thus, the MTA has a distinct advantage over distributed-memory systems for irregular adaptive applications.

For the PCG algorithm, we developed a multithreaded version of the lower and upper triangular solves. Matrix factorization times are not reported since it is performed only once outside the inner loop. Our multithreaded strategy uses low-level locks to perform a dynamic dependency analysis. The lightweight synchronization of the MTA allows locks to be effectively used at such a fine granularity. Specific implementation details can be found in [22]. Table 8 shows PCG performance with the ORIG ordering, again using 60 streams per processor. A speedup of only 4.6X was achieved on eight processors. This limited scalability is due to the lack of available thread-level parallelism in our dynamic dependency scheme. A large fraction of the computational threads were blocked at any given time, preventing a full saturation of the MTA processors. Further optimizations were not possible due to limitations in the current system software. We plan to revisit the multithreaded PCG once a more mature runtime system becomes available.

4.3 N-Body Problems

For the experiments in this paper, our data set consisted of one million particles and simulates two neighboring Plummer model galaxies that are about to merge [29]. Figure 7 shows three snapshots during the collision and evolution process. In the tree building phase of the Barnes-Hut algorithm, the dimensions of the root cell of the octree are determined from the current positions of the particles. Whenever the number of particles in a cell exceeds a given fixed number k ($k = 8$ in our experiments), the cell is subdivided into eight smaller subcells. The particles in the parent cell are then inserted into the appropriate subcell. In addition, each particle keeps track of the number of operations it required in the force calculation phase during the previous timestep. This information is used as a measure of the workload. The initial workload is set to unity.

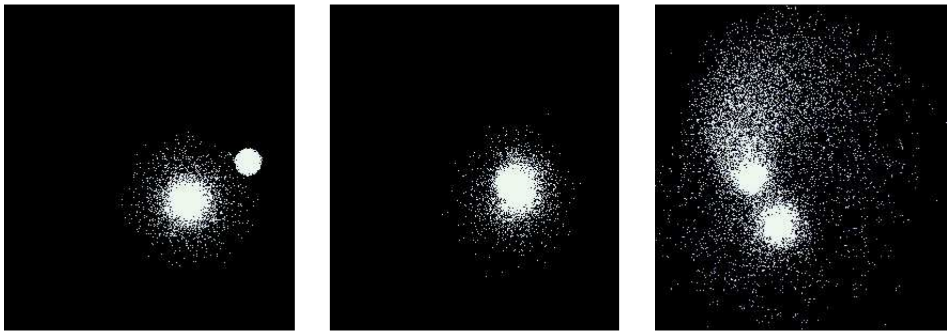

Fig. 7. Three snapshots during the collision and evolution of two Plummer model galaxies.

4.3.1 *Message Passing on the T3E.* The tree building phase is the most complex step in the MPI implementation. To make the subsequent force calculation phase communication free, each processor needs to build a locally essential tree. The domain is first subdivided into cells, each containing a maximum number of particles, that are distributed equally among the processors. A cost distribution tree is then computed in parallel, requiring global communication. The cost represents the expected workload of performing the force calculation for the particles within a cell, and is used (instead of the number of particles) 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. This limited global tree is partitioned using the costzones technique [28], which assigns each processor a contiguous range of cells of approximately equal cost in Peano-Hilbert order [12]. A data remapper uses the computed partitions 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. Subsequent iterations use the previous distribution as the starting point.

As mentioned in Section 2.3, the force calculation is the most expensive phase of the N-body problem. The MPI implementation uses the locally essential tree to perform a load-balanced and communication-free force calculation. Each particle's cost is recorded in order to build the cost distribution tree in the subsequent iteration. The message-passing version of the final update phase is also communication free, but suffers from some load imbalance because the costzones partitioner is based on the cost, not the number, of particles. However, the computational overhead of the update phase is a function of the total number of particles in each partition. Fortunately, the time spent updating each particle's position and velocity is negligible, and a repartitioning is not worthwhile.

Runtimes for the MPI version of the N-body simulation on the T3E are presented in Table 9. The efficiency is a high 93% when going from 64 to 128 processors; however, it drops to about 76% between 128 and 256 processors. This is because as the problem size remains fixed while using an increasing number of processors, the communication-intensive tree building phase dominates the force computations.

Table 9. Runtimes (in seconds) using MPI on the T3E.

P	N-BODY
64	10.95
128	5.88
256	3.88

4.3.2 *Shared-Memory Programming on the Origin2000.* The shared-memory version of the N-body simulation was obtained from the SPLASH-2 suite [32], but is further optimized. The tree building phase is significantly different from the MPI implementation since only one global shared octree is created by concurrently inserting particles using synchronization locks, if necessary. When the cost of a cell (defined in the cost distribution tree) exceeds a specified limit, the cell is dynamically subdivided into eight new subcells. Note that on the Origin2000, explicit communication is not required to compute the shared cost distribution tree. The particles are then partitioned using the costzones technique [28] in the manner described in Section 4.3.1. This strategy ensures cost-balanced partitions and good data locality during the subsequent force calculation phase. The approach is algorithmically similar to that used in the MPI version; however, a data remapping phase is not required in the shared-memory implementation. Each processor is responsible for the particles assigned to it based on the costs calculated in the previous iteration. Since all the particles are globally addressable, they can be reassigned to the processors as necessary without the need for explicit communication.

Once the global shared tree is built, the force computations are performed in parallel without the need for synchronization. However, unlike the MPI version, implicit communication is required since the global tree is physically distributed among the processors. The update phase can also proceed synchronization free in parallel, but it too requires implicit communication. The CC-NUMA update phase is somewhat load imbalanced, for the same reasons as the imbalance in the MPI update. To increase data locality for the next iteration, particles are reordered based on their processor assignment. The reordering step constitutes a small fraction of the total runtime, which is dominated by the force calculation. Overall, the CC-NUMA implementation is much simpler than that in MPI.

Sample performance results for the N-body simulation are presented in Table 10; detailed information can be found in [26]. The runtimes for the CC-NUMA code using a global shared tree are reported in the column titled GLOB_T. Runtimes for an MPI implementation on the Origin2000 are also reported in the last column as a basis for comparison. A thorough analysis of the results show that the total execution time is dominated by the force calculation. However, the global shared tree required for the CC-NUMA implementation is physically distributed across the processors and requires implicit communication during the force calculation. This causes page faults (TLB misses) and increases the memory latency. Instead, in the MPI version, the time required to build the locally essential tree is negligible, and the force computations are free of communication. Thus the MPI runtimes are better than those for CC-NUMA when using 32 or more processors.

Table 10. Runtimes (in seconds) on the Origin2000.

	N-BODY		
	CC-NUMA		MPI
P	GLOB_T	REPL_T	
16	21.82	19.70	20.71
32	11.97	9.98	9.17
64	6.69	5.29	4.64

The performance of the GLOB_T shared-memory code can be improved dramatically by locally duplicating a subset of the remote cells. However, this would not be natural for shared-memory programming, but bring us closer to the message-passing style of data replication. Each processor explicitly creates a local copy of the remote cells which 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 contain approximately 590 (out of more than 366,000) cells. Results for this improved implementation, called REPL_T, are also presented in Table 10.

Note that these are better than those for GLOB_T, and much closer to the MPI results. This indicates that for N-body problems, a shared-memory code can attain message-passing performance through a smart implementation strategy.

4.3.3 *Hybrid Programming on the SP.* The current hybrid implementation was obtained by adding OpenMP directives into the force calculation loop and the position (and velocity) update loop. Other than these differences, the hybrid code is identical to the MPI version. One could also use OpenMP directives when new subspaces are computed for the particles. However, since the time spent in this stage is relatively negligible and the operations are primarily on shared variables, the performance improvement is only marginal. In one of our early hybrid implementations, we incorporated the shared-memory tree building method into the MPI code. Though the complexity of the code increased significantly, we did not obtain any performance improvement.

Results are presented in Table 11 for varying numbers of SMP nodes, MPI tasks, and OpenMP threads on the SP machine. Notice that the best runtimes are obtained (except for 256 processors) when the number of OpenMP threads is one; that is, when the parallelization is purely MPI based. In fact, performance steadily improves as the number of threads decreases for a fixed number of processors. We therefore conclude that our hybrid implementation offers no advantage over pure MPI for the N-body problem.

Table 11. Runtimes (in seconds) using MPI+OpenMP on the SP.

P	Nodes	Tasks	Threads	N-BODY
64	4	1	16	7.29
	4	2	8	5.82
	4	4	4	5.12
	4	8	2	4.07
	4	16	1	3.79
128	8	1	16	3.92
	8	2	8	3.10
	8	4	4	2.44
	8	8	2	2.16
	8	16	1	2.02
256	16	1	16	2.11
	16	2	8	1.54
	16	4	4	1.41
	16	8	2	1.30
	16	16	1	1.46

4.3.4 *Message-Passing, Shared-Memory, and Hybrid Programming on the PC Cluster.* The MPI program used for the experiments in Section 4.3.1 was ported without any changes to our PC cluster described in Section 3.2.5. However, the shared-memory implementation required major modifications from the one described in Section 4.3.2 since it suffered from a high synchronization overhead during the shared-tree building phase on the PC cluster. A new tree building method, called

Barnes-spatial [25], has been developed to completely eliminate the expensive synchronization operations.

We were unable to run the data set consisting of one million particles on our PC cluster because of memory limitations. Instead, we used a smaller case consisting of 128K particles. Results presented in Table 12 show that the MPI version performs much better than the shared address space (SAS) implementation, and is almost twice as fast when executed on 32 processors. An analysis of the time breakdown on 32 processors indicates that the synchronization overhead dominates the overall SAS runtime. This is because at each synchronization point, many `diffs` and `write` notices are processed by the page coherence protocol. Almost 82% of the barrier time is spent on protocol handling. Trying to maintain coherence requires pages to be propagated to their home processors, leading to increased network traffic and memory contention. In addition, a large number of shared pages are invalidated, thereby dilating the synchronization interval. Together, they cause a severe degradation in SAS performance.

Table 12. Runtimes (in seconds) on the PC cluster.

	N-BODY		
P	MPI	SAS	MPI+SAS
16	6.21	8.20	8.09
32	3.24	6.10	6.08

We also tested a hybrid MPI+SAS implementation of the N-body problem on the PC cluster. Two layers of parallelism are combined by implementing SAS codes within each PC-SMP while using MPI among the SMP nodes. Here, the hardware directly supports cache coherence for the SAS code segments, while inter-SMP communication relies on the network through message passing. However, code complexity increases dramatically since we incorporated the shared-memory tree building method into the MPI code. Results presented in Table 12 show that the hybrid implementation offers a small performance advantage over pure MPI. This is due to tradeoffs between the two approaches. For example, while SAS programming reduces the intra-SMP communication compared to MPI, it requires the additional overhead of explicit synchronizations. Although this hybrid programming methodology is the best mapping to our underlying architecture, it is debatable whether the performance gains of this approach compensate for its drawbacks. Nonetheless, PC clusters provide a cost-effective platform for solving realistic problems with reasonable performance.

5. CONCLUSIONS

The goal of this paper was to present, compare, and contrast different parallel computing strategies for irregular algorithms. Such algorithms exhibit non-uniform data access patterns that are particularly challenging for cache-based systems with several levels of memory hierarchy. In addition, some of these algorithms are dynamic in that their computational workloads vary at runtime. We selected three representative irregular applications: unstructured remeshing, N-body problems, and sparse matrix computations, and parallelized them using various programming paradigms for a wide range of computer platforms ranging from state-of-the-art supercomputers to PC clusters. We used four popular parallel programming models: message passing using MPI, shared memory using OpenMP-style directives, hybrid MPI+OpenMP, and hardware-supported multithreading. The parallel implementations were then tested on a Cray T3E, SGI Origin2000, IBM SP, Cray (formerly Tera) MTA, and a PC cluster.

The MPI versions demanded the greatest programming effort even though message passing is the most common and mature approach for high-performance systems. Data had to be explicitly decomposed across processors and special data structures had to be created to manage shared information along partition boundaries. Significant additional memory was also needed for the communication buffers. Despite these drawbacks, the message-passing codes showed good scalability and can be easily ported to any parallel system supporting MPI.

The CC-NUMA shared-memory versions required considerably less programming effort, and also had low memory overhead. However, explicit data decomposition with smart remapping or replication was required to improve performance of our irregular applications; a strategy that is counter-intuitive for shared-memory systems. As a result, some of the programming advantages over MPI must be given up to obtain comparable performance.

The hybrid programming paradigm was the most complex as it combined two layers of parallelism by implementing OpenMP shared-memory constructs within an SMP while using MPI among the SMP clusters. However, the performance improvements over pure MPI codes were negligible. Overall results clearly indicate that in order to obtain good performance, the explicit data decomposition model must be followed; it mattered little whether one used a shared address space or explicit message-passing calls for interprocessor communication. It should also be noted that smart local data ordering enhanced performance for all these three programming paradigms.

The MTA with its hardware-supported multithreading was the most impressive machine for our test suite of irregular algorithms. The implementations required a trivial amount of additional code and had little memory overhead. The complexities of data distribution, repartitioning, remapping, and load balancing were absent on this system; however, the application must possess significant instruction- and thread-level parallelism to extract all the benefits multithreading has to offer.

Finally, we must acknowledge that PC clusters are becoming increasingly attractive for high-end scientific computing and that software DSM is gaining popularity as a paradigm to operate distributed clusters as shared memory systems. Even though absolute performance results are not quite competitive with those obtained on commercial supercomputers, they provide a cost-effective platform for solving realistic problems with reasonable performance.

ACKNOWLEDGMENTS

The work of the second 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.

The work of the third author was supported by the National Science Foundation under grant ESS-9806751 while he was at Princeton University.

REFERENCES

[1] J.E. Barnes and P. Hut, A hierarchical O(N log N) force-calculation algorithm, *Nature*, 324 (1986) 446–449.

[2] A. Bilas, C. Liao, and J.P. Singh, Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems, in: *Proc. 26th International Symposium on Computer Architecture* (Atlanta, GA, 1999) 282–293.

[3] R. Biswas, S.K. Das, D.J. Harvey, and L. Oliker, Parallel dynamic load balancing strategies for adaptive irregular applications, *Applied Mathematical Modelling*, 25 (2000) 109–122.

[4] R. Biswas and L. Oliker, Experiments with repartitioning and load balancing adaptive meshes, in: *Grid Generation and Adaptive Algorithms*, Springer, New York, NY, 1999, 89–111.

[5] R. Biswas and R.C. Strawn, A new procedure for dynamic adaption of three-dimensional unstructured grids, *Applied Numerical Mathematics*, 13 (1994) 437–452.

[6] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su, Myrinet: A gigabit-per-second local area network, *IEEE Micro*, 15 (1995) 29–36.

[7] F. Cappello and D. Etiemble, MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks, in: *Proc. Supercomputing'00* (Dallas, TX, 2000).

[8] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: *Proc. 24th ACM National Conference* (New York, NY, 1969), 157–172.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, VMMC-2: Efficient support for reliable, connection-oriented communication, in: *Proc. 5th Hot Interconnects Symposium* (Stanford, CA, 1997).

[10] A. George, *Computer Implementation of the Finite Element Method*, Technical Report STAN-CS-208, Stanford University, Stanford, CA, 1971.

[11] Giganet, Inc., *URL:* `www.giganet.com`.

[12] G. Heber, R. Biswas, and G.R. Gao, Self-avoiding walks over adaptive unstructured grids, *Concurrency: Practice and Experience*, 12 (2000) 85–109.

[13] D.S. Henty, Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, in: *Proc. Supercomputing'00* (Dallas, TX, 2000).

[14] D. Jiang and J.P. Singh, Scaling application performance on cache-coherent multiprocessors, in: *Proc. 26th International Symposium on Computer Architecture* (Atlanta, GA, 1999) 305–316.

[15] M.T. Jones and P.E. Plassmann, *BlockSolve95 User's Manual*, Technical Report ANL-95/48, Argonne National Laboratory, Chicago, IL, 1995.

[16] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal of Scientific Computing*, 20 (1998) 359–392.

[17] B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell System Technical Journal*, 49 (1970) 291–308.

[18] Message Passing Interface Forum, *URL:* `www.mpi-forum.org`.

[19] L. Oliker and R. Biswas, PLUM: Parallel load balancing for adaptive unstructured meshes, *Journal of Parallel and Distributed Computing*, 52 (1998) 150–177.

[20] L. Oliker and R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, *IEEE Transactions on Parallel and Distributed Systems*, 11 (2000) 931–940.

[21] L. Oliker, R. Biswas, and H.N. Gabow, Parallel tetrahedral mesh adaptation with dynamic load balancing, *Parallel Computing*, 26 (2000) 1583–1608.

[22] L. Oliker, X. Li, P. Husbands, and R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, *SIAM Review*, 44 (2002) 373–393.

[23] OpenMP Programming, *URL:* `www.openmp.org`.

[24] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS, Boston, MA, 1996.

[25] H. Shan and J.P. Singh, Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance, in: *Proc. 12th International Parallel Processing Symposium* (Orlando, FL, 1998).

[26] H. Shan, J.P. Singh, L. Oliker, and R. Biswas, A comparison of three programming models for adaptive applications on the Origin2000, *Journal of Parallel and Distributed Computing*, 62 (2002) 241–266.

[27] H. Shan, J.P. Singh, L. Oliker, and R. Biswas, Message passing vs. shared address space on a cluster of SMPs, in: *Proc. 15th International Parallel and Distributed Processing Symposium* (San Francisco, CA, 2001).

[28] J.P. Singh, J.L. Hennessy, and A. Gupta, Implications of hierarchical N-body methods for multiprocessor architectures, *ACM Transactions on Computer Systems*, 13 (1995) 141–202.

[29] 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, *Journal of Parallel and Distributed Computing*, 27 (1995) 118–141.

[30] R.S. Tuminaro, M. Heroux, S.A. Hutchinson, and J.N. Shadid, *Official Aztec User's Guide*, Technical Report SAND99-8801J, Sandia National Laboratories, Albuquerque, NM, 1999.

[31] Virtual Interface Architecture, *URL:* `www.viarch.org`.

[32] 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) 24–36.