Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms

LEONID OLIKER[1]
National Energy Research Scientific Computing Center
Mail Stop 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
loliker@lbl.gov

RUPAK BISWAS[2]
MRJ Technology Solutions
Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Abstract

The success of parallel computing in solving real-life computationally-intensive problems relies on their efficient mapping and execution on large-scale multiprocessor architectures. Many important applications are both unstructured and dynamic in nature, making their efficient parallel implementation a daunting task. This paper presents the parallelization of a dynamic unstructured mesh adaptation algorithm using three popular programming paradigms on three leading supercomputers. We examine an MPI message-passing implementation on the Cray T3E and the SGI Origin2000, a shared-memory implementation using cache coherent nonuniform memory access (CC-NUMA) of the Origin2000, and a multithreaded version on the newly-released Tera Multithreaded Architecture (MTA). We compare several critical factors of this parallel code development, including runtime, scalability, programmability, and memory overhead. Our overall results demonstrate that multithreaded systems offer tremendous potential for quickly and efficiently solving some of the most challenging real-life problems on parallel computers.

1. Introduction

The success of parallel computing in solving real-life computationally-intensive problems relies on their efficient mapping and execution on large-scale multiprocessor architectures. When the algorithms and data structures corresponding to these problems are intrinsically *unstructured* or *dynamic* in nature, efficient implementation on state-of-the-art parallel machines offers considerable challenges. Unstructured applications are characterized by irregular data access patterns, and are inherently at odds with cache-based systems which attempt to hide memory latency by copying and reusing contiguous blocks of data. Dynamic and adaptive applications, on the other hand, have computational workloads which grow or shrink at runtime, and require dynamic load balancing to achieve algorithmic scaling on parallel machines.

Many important applications are both unstructured and dynamic, making their efficient parallel implementation a daunting task. Examples include scientific computing, task scheduling, sparse matrix computations, parallel discrete event simulation, data mining, and web server applications. This paper presents the parallelization of a dynamic unstructured mesh adaptation algorithm using three popular programming paradigms on three state-of-the-art parallel architectures. We examine an MPI message-passing implementation on the Cray T3E and the SGI Origin2000, a shared-memory implementation using cache coherent nonuniform memory access (CC-NUMA) of the Origin2000, and a multithreaded version on the newly-released Tera Multithreaded Architecture (MTA). We compare several critical factors of this parallel code development, including runtime, scalability, programmability, and memory overhead.

Unstructured meshes for problems in computational science and engineering 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 those problems with evolving physical features. Standard fixed-mesh numerical

[1] Work supported by Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

[2] Work supported by National Aeronautics and Space Administration under contract number NAS 2-14303 with MRJ Technology Solutions.

methods can be made more cost effective by locally refining and coarsening the mesh to capture these phenomena of interest. Unfortunately, an efficient parallelization of adaptive unstructured methods is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grids. Nonetheless, it is generally believed that adaptive unstructured-grid techniques will constitute a significant fraction of future high-performance supercomputing.

Recently, three different parallel architectures have emerged, each with its own set of programming paradigms. On distributed-memory systems, each processor has its own local memory that only it can directly access. To access the memory of another processor, a copy of the desired data must be explicitly sent across the network using a message-passing library such as MPI or PVM. To run a program on such machines, the programmer must decide how the data should be distributed among the local memories, communicated between processors during the course of the computation, and reshuffled when necessary. This allows the user to design efficient programs, but at the cost of increased code complexity.

In distributed shared-memory architectures, each processor has a local memory but also has direct access to all the memory in the system. On the Origin2000, for example, each processor uses a local cache to fetch and store data. Cache coherence is managed by hardware. Parallel programs are relatively easy to implement on such systems since each processor has a global view of the entire memory. Parallelism can be easily achieved by inserting directives into the code to distribute loop iterations among the processors. However, portability may be diminished, and sophisticated "cache programming" may be necessary to enhance parallel performance.

Using multithreading to build commercial parallel computers is a radically new concept in contrast to the standard single-threaded microprocessors of traditional supercomputers. For example, the Tera MTA processors each use up to 128 threads. Such machines can potentially utilize substantially more of its processing power by tolerating memory latency and using low-level synchronization directives. This multithreaded architecture is especially well-suited for irregular and dynamic problems. Parallel programmability is simplified since the user has a global view of the memory, and need not be concerned with the data layout.

2. Unstructured Mesh Adaptation

For problems that evolve with time, mesh adaptation procedures have proved to be robust, reliable, and efficient. Highly localized regions of mesh refinement are required to accurately capture shock waves, contact discontinuities, vortices, and shear layers. This provides scientists the opportunity to obtain solutions on adapted meshes that are comparable to those obtained on globally-refined grids but at a much lower cost.

In this paper, we parallelize a two-dimensional unstructured mesh adaptation algorithm based on triangular elements; complete details of the three-dimensional procedure is given in [1]. In brief, local mesh adaptation involves adding points to the existing grid in regions where some user-specified error indicator is high, and removing points from regions where the indicator is low. The advantage of such strategies is that relatively few mesh points need to be added or deleted at each refinement/coarsening step. However, complicated logic and data structures are required to keep track of the mesh objects (points, edges, elements) that are added and removed. It involves a great deal of "pointer chasing", leading to irregular and dynamic data access patterns.

It is important to note here that mesh adaptation is a tool that is used to make the primary application more cost effective. In other words, adapting an unstructured mesh is not the final goal in itself, but simply a means to a bigger end. Thus, even though parallel mesh adaptation and the ensuing dynamic load balancing are critical components of a complete parallel adaptive application, they must be accomplished rapidly and efficiently so as not to cause a significant overhead to the actual computation.

3. Test Problem

The computational mesh used for the experiments in this paper is the one often used to simulate flow over an airfoil (see Fig. 1 for the coarse initial mesh). Mesh refinement is usually required around the stagnation point at the leading edge of the airfoil. At transonic Mach numbers, shocks form on both the upper and lower surfaces of the airfoil that then propagate to the far field. We simulate this actual scenario by geometrically adapting regions corresponding approximately to the locations of the stagnation point and the shocks. This allows us to investigate the performance of

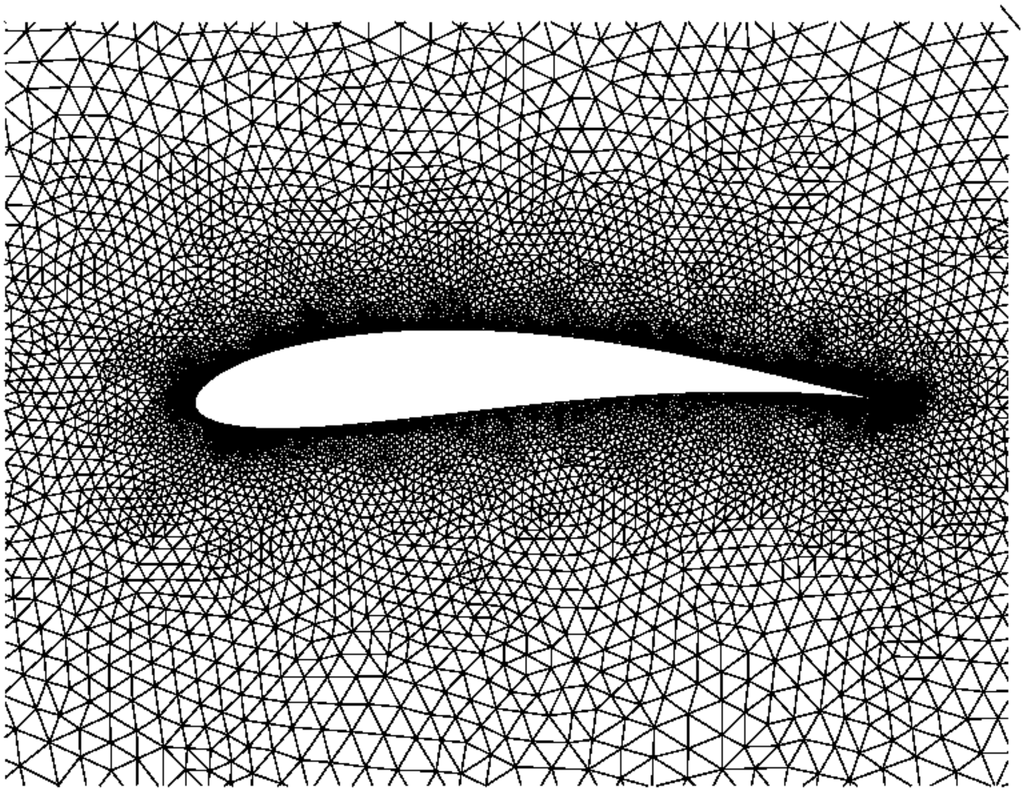

Figure 1: A close-up view of the initial triangular mesh around the airfoil.

the mesh adaptation and load balancing algorithms in the absence of a flow solver.

For such steady-state problems, the computational mesh is typically adapted every few hundred iterations of the flow solver. Adaptation usually requires about 5% of the total time. However, for unsteady time-dependent problems, adaptation is required much more frequently and can consume 30–40% of the total time. Thus an efficient parallel implementation of the adaptation algorithm is crucial for the overall efficiency.

Table 1 presents the progression of grid sizes through five levels of refinement. The computational mesh after the second refinement is shown in Fig. 2. For reference purposes, the original serial adaptation code consists of approximately 1,300 lines of C and requires 6.4 seconds to execute this simulation on a 250 MHz MIPS R10000 processor. Note that a flow solver was not run between between successive adaptations.

Mesh	Vertices	Triangles	Edges
Initial	14,605	28,404	43,009
Level 1	26,189	59,000	88,991
Level 2	62,926	156,498	235,331
Level 3	169,933	441,147	662,344
Level 4	380,877	1,003,313	1,505,024
Level 5	488,574	1,291,834	1,935,619

Table 1: Progression of grid sizes through five levels of adaptation.

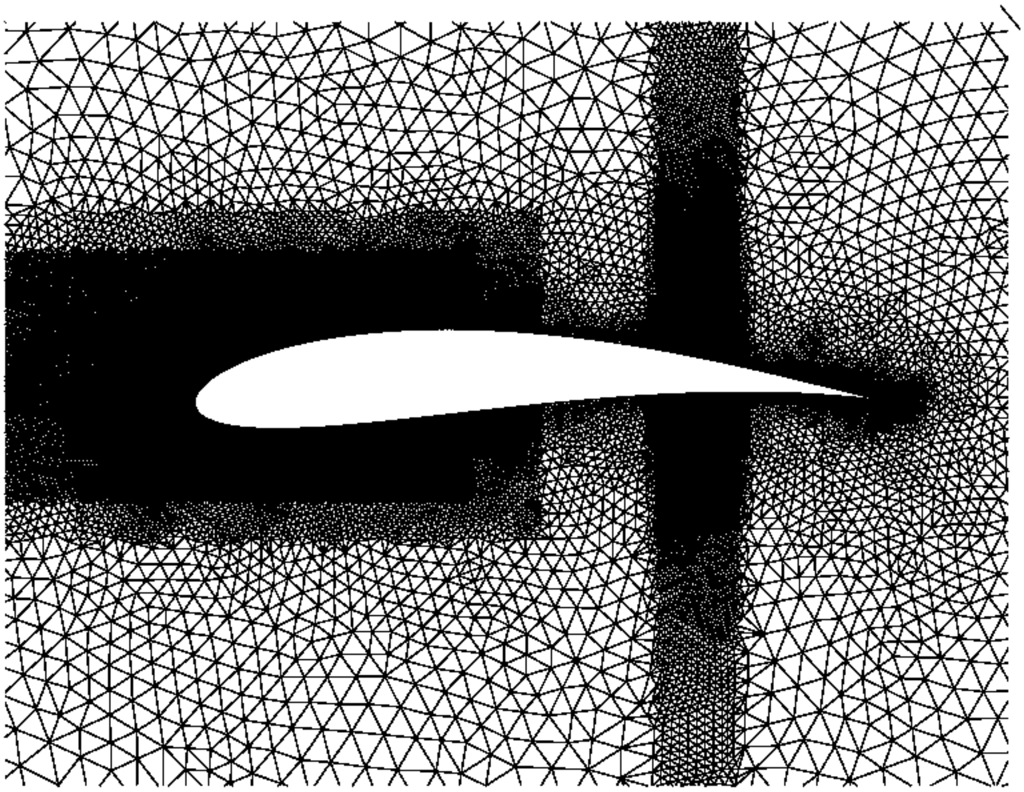

Figure 2: A close-up view of the mesh after the second refinement.

4. Distributed-Memory Implementation

The distributed-memory version of the mesh adaptation code was implemented in MPI within the PLUM framework [5]. 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.

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[3]. The mesh adaptation procedure then marks edges for refinement or coarsening based on an error indicator (geometric for the tests in this paper). Once edges have been marked, it is possible to exactly predict the new mesh before actually performing the adaptation step. Program control is thus passed to the load balancer at this time. A repartitioning algorithm, like the ParMETIS [4] parallel multilevel partitioner, is used to divide the new mesh into subgrids. All necessary data is then redistributed, the computational mesh is actually refined, and the numerical calculation restarted.

PLUM [5] is a novel method to dynamically balance the processor workloads for unstructured adaptive-grid computations with a global view. It has five salient features:

- Repeated use of the initial mesh dual graph keeps the connectivity and partitioning complexity constant during the course of an adaptive computation.
- Parallel mesh repartitioning avoids a potential serial bottleneck.
- Fast heuristic remapping assigns partitions to processors so that the redistribution cost is minimized.
- Efficient data movement significantly reduces the cost of remapping and mesh subdivision.

[3] Recall that a solver is not used for the experiments reported in this paper.

- Accurate metrics estimate and compare the computational gain and the redistribution cost of having a balanced workload after each mesh adaptation.

Several options may be set within PLUM, including predictive or non-predictive refinement, global or diffusive partitioning, and synchronous or asynchronous communication. Tables 2 and 3 present the results for the best combination of these options on a T3E and an Origin2000, through the five refinement steps shown in Table 1. Note that results are not presented for less than eight processors of the T3E because of memory constraints. The T3E used for these experiments is a 640-node machine located in the NERSC division of Lawrence Berkeley National Laboratory. Each node consists of a 450 MHz DEC Alpha processor, local memory, and some network interface hardware. The Origin2000, on the other hand, is a 64-processor SMP cluster of R10000 MIPS microprocessors, located in the NAS division of NASA Ames Research Center.

	Time (secs)				Imbalance	Remap Data Volume (MB)	
P	Refine	Partition	Remap	Total	Factor	Maximum	Total
8	4.53	1.5	12.97	19.0	1.02	68.04	286.80
16	2.41	1.6	5.98	10.0	1.03	28.11	270.18
32	1.09	1.5	3.91	6.5	1.05	16.76	281.17
64	0.78	1.5	1.81	4.1	1.07	6.88	280.30
128	0.69	1.6	0.90	3.2	1.09	4.41	290.10
160	0.61	1.7	0.69	3.0	1.14	4.24	284.41
192	0.49	2.1	0.61	3.2	1.15	3.12	292.39
256	0.31	2.6	0.48	3.4	1.16	2.34	310.82
384	0.16	3.9	0.29	4.4	1.19	1.33	301.58
512	0.14	4.7	0.25	5.1	1.25	0.99	310.40

Table 2: Performance of the MPI code on the T3E.

	Time (secs)				Imbalance	Remap Data Volume (MB)	
P	Refine	Partition	Remap	Total	Factor	Maximum	Total
2	13.12	1.3	24.89	39.3	1.00	50.11	60.64
4	11.72	1.2	16.67	29.6	1.00	35.59	88.58
8	8.31	1.4	10.23	19.9	1.02	30.21	151.75
16	5.04	1.3	5.57	11.9	1.02	13.57	121.06
32	2.28	1.7	2.82	6.8	1.05	7.77	118.55
64	1.41	2.3	1.69	5.4	1.08	4.17	132.34

Table 3: Performance of the MPI code on the Origin2000.

The general runtime trends are similar on both architectures. Notice that more than 32 processors are required to outperform the serial case, since the total runtime of the distributed version includes the load balancing overhead. The refinement time decreases as the number of processors P 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. Recall that our load balancing objective is to produce a balanced mesh for the more expensive solver phase. Partitioning times remain somewhat constant for smaller values of P, but start to grow and eventually dominate as P becomes large. This is because the amount of work and the communication overhead of the partitioner increases with P. Finally, the data remapping time decreases as P increases. This satisfies our bottleneck communication model which expresses remapping as a function of the maximum (not total) communication among processors [5]. The slight difference in the amount of data volume between Tables 2 and 3 for the same value of P is because the T3E has 8-byte integers whereas the Origin2000 has 4-byte integers.

This message-passing implementation of the adaptation algorithm required a significant amount of programming effort, effectively doubling the size of the original serial code and increasing the memory requirements by 70%. Only a fraction of the additional memory was used for tracking edges and vertices residing on partition boundaries. Most of the memory overhead was due to the send and receive buffers for the bulk MPI communication during the remapping phase.

5. Shared-Memory Implementation

The shared-memory version of the mesh adaptation code was implemented on the CC-NUMA Origin2000, which is a SMP cluster of nodes each containing two processors and local memory. The hardware makes all memory equally accessible from a software standpoint, by sending memory requests through routers located on the nodes. Access time to memory is *nonuniform*, depending on how far away the memory lies from the processor. The topology of the interconnection network is a hypercube, bounding the maximum number of memory hops to a logarithmic function of P. Each processor also has a secondary cache memory, 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. To minimize overhead, cache coherence is managed by the hardware.

This version of the parallel code was written using SGI's native pragma directives, which create IRIX threads. A rewrite to OpenMP would require minimal programming effort. The main kernel of the refinement procedure consists of looping over the list of triangles. In the shared-memory implementation, this work is split among all the processors. However, it is necessary to guarantee that multiple processors will not modify the same location in the data structure. Two basic approaches were taken to achieve this goal of preventing memory race conditions.

The first strategy (ALL_COLOR) uses graph coloring to form independent sets, where two triangles have different colors if they share an edge or a vertex. Since optimal graph coloring is NP-complete, we use a simple greedy algorithm. The processors can then work simultaneously on all triangles of the same color. Dynamic distribution of loop iterations among the processors is easily achieved through pragma compiler directives. Two algorithmic overheads are associated with this strategy: The coloring of newly-formed triangles on the fly, and a barrier synchronization at the completion of each color before processing of the next color can begin.

The second strategy (NO_COLOR) uses low-level locks instead of graph coloring. When a thread processes a given triangle, it locks the corresponding vertices and edges. Other threads attempting to access these mesh objects are blocked until the locks are released. The algorithmic overhead lies in the idle time processors must spend while waiting for blocked objects to be released. However, coloring or multiple barrier synchronizations are not required.

Finally, a hybrid method (HYBRID_COLOR) combines an edge-based coloring of the triangles with vertex-based locks. In this strategy, two neighboring triangles are guaranteed to have different colors if they share an edge. This approach requires fewer colors than ALL_COLOR, since an arbitrary number of triangles can potentially share a vertex, whereas no more than two triangles can share an edge. Vertex synchronization is handled through low-level locks as in the NO_COLOR strategy. Table 4 shows the timing results for all these approaches. ALL_COLOR required 25 colors whereas HYBRID_COLOR needed 11 colors. Note that after each adaptation, control would be passed to a shared-memory solver which uses the globally addressable mesh for performing flow computations.

For ALL_COLOR, the total runtime remains relatively constant regardless of P, although a significant performance degradation is seen between 32 and 64 processors. Several reasons contribute to these poor performance numbers. Single-processor cache performance and translation lookaside buffer (TLB) reuse are extremely poor. On one processor, the refinement runtime (20.8 secs) increases by a factor of three compared to the serial case (6.4 secs). This is because the triangles are processed one color at a time, where two triangles of the same color are never adjacent. As a result, the cache miss rate increases dramatically from the serial case where all the triangles are processed in order. Using the R10000 hardware counters, the number of cache misses (primary and secondary) increased from approximately 15 million in the original serial code to almost 85 million for the ALL_COLOR case.

Another consequence of the poor data locality is the increased miss rate of the TLB. The TLB registers store the translations between virtual and physical memory for the 64 most recently used pages. If a memory address is located

P	ALL_COLOR Refine	ALL_COLOR Color	ALL_COLOR Total	NO_COLOR Total	HYBRID_COLOR Refine	HYBRID_COLOR Color	HYBRID_COLOR Total
1	20.8	21.1	41.9	8.2	16.2	7.4	23.6
2	21.5	24.4	45.9	11.9	18.6	8.9	27.5
4	17.5	24.0	41.5	21.1	20.2	11.2	31.4
8	17.0	22.6	39.6	38.4	21.0	12.9	33.9
16	17.8	22.0	39.8	56.8	32.3	16.3	48.6
32	23.5	25.8	49.3	107.0	60.2	19.4	79.6
64	42.9	29.6	72.5	160.9	82.7	18.9	101.6

Table 4: Timings (in secs) of the CC-NUMA code on the Origin2000.

on a virtual page which resides in the TLB, the virtual-to-physical mapping is carried out in hardware with zero latency. Otherwise the operating system traps to a routine which finds the physical address in a memory table, resulting in a significant overhead. For this test case, the number of TLB misses increased from 7.3 million in the serial case to almost 53 million when using the ALL_COLOR strategy.

Poor parallel performance also stems from the structure of the code which assumes a flat shared-memory model, and does not consider data locality or cache effects. When triangles of a particular color are being processed, each processor refines distinct triangles that have non-overlapping edges and vertices. However, since there is no explicit ordering of the data structures, cache lines contain mesh objects which may be required by several processors simultaneously. This *false sharing* problem is exacerbated when new mesh objects are created during a refinement phase. Each time a new word is written to cache, all copies of that cache line residing on other nodes are invalidated. The hardware system is therefore overloaded attempting to maintain cache-coherency in this environment. Also, the automatic page migration option of the Origin2000 cannot improve parallel performance, since a single page of memory may contain data required by all processors. Note that a more intelligent graph coloring scheme could lower the number of colors and reduce graph coloring time, but would not improve the poor scalability of the refinement phase.

Relatively little effort was required to convert the serial code into the ALL_COLOR implementation. Parallelization was easily achieved by distributing loops across processors using compiler directives. The addition of the greedy graph coloring algorithm caused about a 10% increase in code size and a 5% increase in memory requirements.

Timing results for the NO_COLOR approach are also shown in Table 4. Recall that low-level locks are used here for synchronization and graph coloring is not needed. This simplifies parallelization and requires minimal memory overhead. Unfortunately, the performance of this approach is extremely poor on multiple processors. Notice that on one processor, there is a small increase in the runtime (8.2 secs) compared to the original serial code (6.4 secs). This is due to the overhead of setting, checking, and releasing low-level locks. As P increases, however, performance rapidly deteriorates due to the fine granularity of the critical sections. The processors are constantly contending to grab locks in order to modify mesh data structures. In addition, blocked threads are unable to do any useful work and spin idly waiting for access to locked data. It is obvious from these results that such fine-grain synchronization is not suitable for this type of architecture. The HYBRID_COLOR, which uses a combination of graph coloring and locks, also shows poor results for similar reasons.

One possible technique to improve performance is to order the triangles, edges, and vertices, such that each processor refines a contiguous section of the mesh. This would improve cache reuse and reduce false sharing. To accomplish this, a domain decomposition algorithm such as ParMETIS [4] or a locality-enhancing strategy such as self-avoiding walks [3] is required. All mesh objects could then be remapped or reordered accordingly. As the mesh grew after each refinement step, another round of partitioning and remapping would be required. However, this strategy would make little sense since it is equivalent to the MPI implementation, requiring a similar amount of programming effort and balancing overheads. The only major difference would be the use of a global address space instead of explicit message-passing calls for performing interprocessor communication.

6. Multithreaded Implementation

The Tera MTA is a supercomputer recently installed at the San Diego Supercomputing Center. The MTA has a radically different architecture than current high-performance computer systems. Each processor has support for 128 hardware streams, where each stream includes a program counter and a set of 32 registers. One program thread can be assigned to each stream. The processor switches among the active streams at every clock tick, while executing a pipelined instruction.

The uniform shared memory of the MTA is flat, and physically distributed across hundreds of banks that are connected through a 3D toroidal network to the processors. All memory addresses are hashed by the hardware so that apparently adjacent words are actually distributed across different memory banks. Because of the hashing scheme, it is impossible for the programmer to control data placement. This makes programmability much easier than on standard cache-based multiprocessor systems. Rather than using data caches to hide latency, the MTA processors use multithreading to tolerate latency. If a thread is waiting for its memory reference to complete, the processor executes instructions from other threads. Performance thus depends on having a large number of concurrent computation threads.

Lightweight synchronization among threads is provided by the memory itself. Each word of physical memory contains a full-empty bit, which enables fast synchronization via load and 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 may run many threads. Explicit load balancing across loops is also not required since the dynamic scheduling of work to threads provides the ability of keeping the processors saturated, even if different iterations require varying amounts of time to complete. Once a code has been written in the multithreaded model, no additional work is required to run it on multiple processors, since there is no difference between uni- and multiprocessor parallelism.

The multithreaded implementation of the mesh adaptation code is similar to the NO_COLOR strategy used on the Origin2000. Low-level locks ensure that two neighboring triangles are not updated simultaneously, since this may cause a race condition. A key difference from the CC-NUMA implementation is the multiple execution streams on each processor. The load is implicitly balanced by the operating system which dynamically assigns triangles to threads. No partitioning, remapping, graph coloring, or explicit load balancing is required, greatly simplifying the programming effort. Upon completion of the refinement phase, control would be passed to a multithreaded flow solver [2]. Table 5 shows the performance on the 255 MHz Tera MTA, through five refinement levels. The number of streams is easily changed through a compiler directive.

	Number of Streams				
P	100	80	60	40	1
1	2.04	2.22	2.72	3.82	150.1
2	1.06	1.15	1.40	1.98	
4	0.59	0.64	0.74	1.01	
6	0.40	0.43	0.51	0.69	
8	0.35	0.37	0.41	0.55	

Table 5: Refinement timings (in secs) of the multithreaded code on the Tera MTA.

Results show that for 100 streams, a single MTA processor (2.04 secs) outperforms the serial R10000 version (6.4 secs) by more that a factor of three. With four processors, performance continues to improve, achieving 86% efficiency with 100 streams and 95% efficiency with 40 streams. Using all eight processors, the fastest runtime is only 0.35 secs, although efficiency is reduced to 73% and 87% with 100 and 40 streams, respectively. As we increase the number of processors, the number of active threads increase 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).

Another potential source of the degradation in efficiency are memory *hot-spots*. A hot-spot occurs 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 a 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 2 clock ticks. Future research will examine hot-spot behavior in detail.

Notice that when the number of streams drops from 100 to 80, there is only a slight degradation in performance, indicating that 80 streams are sufficient for this problem. When the number of streams decreases to 40, however, there is a significant slowdown, although the algorithm scales better. These results indicate that there is enough instruction level parallelism in the mesh adaptation code to tolerate the overheads of memory access and lightweight synchronization, if enough streams can be used.

By setting the MTA to serial mode, we can run our code using just one stream. The runtime is almost 75 times slower than a single saturated multithreading processor. A single stream can issue only one instruction at a time, and then must wait the length of the instruction pipeline (at least 21 cycles) before issuing another. Also, no useful work can be done while memory is being fetched, since no other threads are active.

Although the Tera compiler attempts to automatically extract parallelism from a serial program, it was necessary to explicitly hand code all the parallelism for the multithreaded version of our mesh adaptation algorithm. This is because the code is highly irregular, and often uses one or more levels of indirection for memory addressing. As a result, the compiler cannot determine which sections are safe to parallelize. However, it is simple to mark the parallelizable regions with pragma directives. Thus, a trivial amount of programming was required to convert the serial mesh adaptation code into the multithreaded version. The code size increased by about 2% for compiler directives and lock placements. An additional memory requirement of 7% was used to store synchronization variables that coordinated access to data during the adaptation procedure. We look forward to continuing our experiments as more processors become available on the Tera MTA.

7. Conclusions

The goal of this work was to parallelize a dynamically adapting, unstructured mesh application. It is particularly difficult to achieve good performance on such algorithms due to their irregular and adaptive nature. We used three different programming paradigms on state-of-the-art supercomputers to achieve our objective. Table 6 summarizes our findings.

Program Paradigm	System	Best Time[4]	Code Increase	Memory Increase	Scalability	Portability
Serial	R10000	6.4 ($P = 1$)				
MPI	T3E	3.0 ($P = 160$)	100%	70%	Medium	High
MPI	Origin2000	5.4 ($P = 64$)	100%	70%	Medium	High
CC-NUMA	Origin2000	39.6 ($P = 8$)	10%	5%	None	Medium
Multithreading	MTA	0.35 ($P = 8$)	2%	7%	High[5]	Low

Table 6: Comparison among programming paradigms and architectures.

The message-passing version demanded the greatest programming effort. Data had to be explicitly decomposed across processors and special data structures had to be created for mesh objects lying on partition boundaries. We used PLUM for dynamic load balancing, which required repartitioning and data remapping between refinement phases. Significant

[4] Different programming paradigms required varying numbers of operations.
[5] Only eight processors are available on the current configuration of the Tera MTA.

additional memory was also needed, mainly for the communication buffers. Despite these drawbacks, the message-passing code showed reasonable scalability and can be easily ported to any multiprocessor system supporting MPI.

The shared-memory CC-NUMA version required an order of magnitude less programming effort than the MPI version, and had a low memory overhead. This code can be ported to any system supporting a global address space. The ALL_COLOR strategy used graph coloring to orchestrate mesh refinement, while the NO_COLOR strategy relied on low-level locking mechanisms. Unfortunately, the fine-grained nature of these computations resulted in poor cache reuse and significant overhead due to false sharing. Explicit data decomposition with remapping or a truly flat memory system would be required to improve these performance numbers.

Finally, the Tera MTA was the most impressive machine for our dynamic and irregular application. The multithreaded implementation required a trivial amount of additional code and had little memory overhead. The complexities of data distribution, repartitioning, remapping, or graph coloring were absent on this system. Our overall results have demonstrated that multithreaded systems offer tremendous potential for quickly and efficiently solving some of the most challenging real-life problems on parallel computers.

References

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

[2] S.H. Bokhari and D.J. Mavriplis, "The Tera Multithreaded Architecture and unstructured meshes," ICASE, NASA Langley Research Center, NASA/CR-1998-208953, 1998.

[3] G. Heber, R. Biswas, and G.R. Gao, "Self-avoiding walks over adaptive unstructured grids," *Parallel and Distributed Processing*, LNCS 1586 (1999) 968–977.

[4] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," Department of Computer Science, University of Minnesota, TR 96-036, 1996.

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