Characterizing Applications on the Cray MTA-2 Multithreading Architecture

Sadaf R. Alam Richard F. Barrett Collin B. McCurdy
Philip C. Roth Jeffrey S. Vetter

Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA 37831
{alamsr,rbarrett,cmccurdy,rothpc,vetter}@ornl.gov

ORNL is currently evaluating several strategic applications on the Cray MTA-2 platform to better understand massively parallel multithreading as an architectural choice beyond scalar, vector, and multicore architectures. In this paper, we describe our initial experiences with several applications including molecular dynamics, finite difference methods, a fast multipole method and a discrete event simulation engine.

1 Introduction

Several vendors of high-end computing (HEC) systems have announced or deployed systems that include processing devices from a number of computing paradigms, including microprocessors, vector processors, multi-threaded processors, and other hardware accelerators like FPGAs. Dramatic speedups are possible for applications whose parts are mapped to appropriate compute devices in multi-paradigm systems. For instance, Cray has recently announced their Adaptive Supercomputing strategy that will incorporate microprocessors, vector processors, multi-threaded processors, and hardware accelerators into a single HEC system.

To better understand massively parallel multi-threading as an architectural choice compared to scalar, vector, and multi-core processors, Oak Ridge National Laboratory is evaluating several strategic applications on Cray's current *multi-threaded architecture* platform, the Cray MTA-2 [4, 13]. The MTA-2 takes a non-traditional approach to addressing the ever-widening gap between the rate at which processors can execute instructions and the rate at which data can be transferred between processors and memory.

Similar to the approach of overlapping communication with computation in a message passing parallel program, clever instruction scheduling can be used to *hide* some of the memory access latency with computation. For instance, a compiler can arrange the instructions of a program so that the instruction that loads data into a processor register is executed as far in advance as possible before the instruction that uses the register's value. However, modern processors are able to execute tens or even hundreds of instructions in the time it takes to load one data word from memory into a processor register, and instruction scheduling alone is insufficient to hide the time required to access data in main memory.

The conventional approach for dealing with the memory access latency problem is to insert data caches between the processor and main memory that contain data that is likely to be accessed either because it has recently been accessed, or because it is near some data that has been recently accessed. Because the time required to access data from a cache is less than that to access data from main memory, instruction scheduling can be effective in hiding the latency of accesses that are satisfied from the data cache(s). Unfortunately, data caches are small compared to the size of main memory, and sometimes accesses must be satisfied from main memory (a "cache miss"). Algorithms for solving certain types of important problems such as graph theory problems tend to produce memory access streams that cause a large number of cache misses, resulting in poor performance on systems with traditional cache-based designs.

The MTA-2 takes a non-conventional approach to addressing the gap between processor and memory access performance. Instead of data caches to reduce the latency of some memory accesses, the MTA-2 uses a high degree of thread-level concurrency to hide memory access latency. If there are enough threads available to an MTA-2

processor, the time required to satisfy the memory accesses for any individual thread running on that processor is completely hidden by the execution of instructions from other threads.

Over the past decade, there have been several studies investigating the performance of this approach as implemented in the Cray MTA-2 and its predecessors [2, 3, 9, 10, 19, 20, 22]. The primary question we are investigating is whether some types of algorithms *require* multithreading to demonstrate substantial performance relative to systems with more conventional processors. Also, unlike the previous studies, we are also investigating how best to map an application to systems containing both conventional processing elements and multi-threaded processors when only some of the application kernels require the multi-threaded processors.

1.1 Strong versus Weak Scaling

Parallel computing was developed with the end goal of solving computational problems faster by applying more processing power. Realized speed up is bounded by the amount of parallelism that can be captured computationally, analytically described by Amdahl's Law [5] as

$$S = \frac{s+p}{\left(s+p/N\right)} = \frac{1}{\left(s+p/N\right)}$$

for s and p defined to be the serial and parallel portions, respectively, and N parallel processes. The implication is that a fundamental bound on performance is inherent for any given problem. The ability of a computer to solve such problems given a range of process counts is called strong scaling. The user is probably looking to discover some "sweet spot" where the time to solution is balanced with some resource management requirement.

However, some scientists realized another use for the processing power that was being made available to them. They understood that for problems that they could not fully capture in the available computing resource, they could instead increase the size of their problem in proportion to the number of processors and amount of memory. While not fully solving the ultimate problems of interest, insight can be gained, which at the least can be used to help clarify the issues that must be addressed once the full experiment can be executed.

This so-called weak scaling was first described analytically by Gustafson [16] as

$$S = \frac{(s+pN)}{(s+p)} = s+pN = N+(1-N)s$$

for s and p defined to be the serial and parallel portions, respectively, and N parallel processes. Good scaling here means that the time to solution remains fairly constant as the size of the problem is increased.

Strong scaling problems are characterized by a change in problem definition, or complexity, as the problem size changes. The Community Climate Systems Model (CCSM) [7] is a well-known example of a strong scaling model. The NAS Benchmarks [6], derived from fluid dynamics experiments, define fixed problem sets.

Weak scaling problems are characterized by an improvement in problem definition as the problem size increases. The radiation-hydrodynamics application SAGE [18] is a well-known example of a weak-scaling application code. The LINPACK benchmark creates a linear system of dimension limited only by the available memory.

Large scale parallel processing architectures, while often good at weak scaling, are also often not very good at strong scaling. The MTA-2 provides a strong exception to this, a claim supported throughout this paper.

2 Cray MTA-2 Overview

In this section, we present an overview of the Cray MTA-2 system organization, compute node processor, and front-end system. We conclude the section by comparing MTA-2 with Eldorado, Cray's forthcoming system with multi-threaded architecture processors. We leave a detailed description of the system's programming environment for Section 3.

2.1 System Organization

The MTA-2 consists of a collection of compute nodes connected by an interconnection network. The topology of the MTA-2 interconnection network is described in the literature as a "modified Cayley" topology [14] and as a 3D torus that is missing some links [4].

Unlike most modern parallel systems, memory is not collocated with the compute nodes. Instead, memory is located in separate memory units

attached directly to the interconnection network. All memory requests traverse the network from the originating processor across the network to a memory unit, and (in the case of memory loads) back to the originating processor. Because all memory requests traverse the interconnection network, the time required to access memory is nearly no matter which processor originates the request and which memory unit satisfies it. To alleviate contention caused by the spatial and temporal locality of memory accesses with some algorithms (e.g., sequential access to an array), virtual addresses are hashed to memory units so that sequential addresses in a process' address space do not refer to locations in the same memory unit, freeing the compiler from the need for careful data placement and memory access instruction scheduling.

2.2 Processor

The MTA-2 processor tolerates memory access latency by supporting many concurrent threads of execution. The processor uses 64-bit VLIW instructions. Each instruction can contain one fused multiply-add instruction, one add or control operation, and one memory load or store operation. To be able to hide the memory access latency of the worst-case scenario in which a thread issues one memory access per instruction), the MTA-2 processor supports 128 concurrent instruction streams and can switch between streams on each clock cycle. To enable such rapid switching between streams, the processor maintains a complete thread execution context for each of its 128 streams.

Although the MTA-2 processor does not use a data cache, it does include an instruction stream shared between all of its hardware streams.

2.3 Front-end System

Like many high-end computing systems, the MTA-2 uses a separate front-end system. Users log in to the front-end system for software development and to run programs on the MTA-2 compute nodes. In contrast to the MTA-2 compute nodes, the system's front-end system uses a SPARC processor and runs the Solaris version of UNIX from Sun Microsystems.

2.4 Cray Eldorado

The MTA-2 is no longer an active product in the Cray product line. However, multi-threaded processors seem likely to be a feature of future systems from Cray. Multithreaded processors are explicitly mentioned in the company's Adaptive Supercomputing strategy overview, and are the primary processor in the forthcoming Eldorado system [14]. Although Eldorado uses multithreaded processors similar to the MTA-2, there are several important differences. Unlike the MTA-2, Eldorado will use a complete 3D mesh/torus network similar to that used in the Cray XT3 parallel processing system, and each compute processor will have an associated local memory. Consequently, Eldorado will not have the MTA-2's nearly uniform memory access latency, so data placement and access locality will be an important consideration when programming Eldorado. Eldorado's multithreaded processors will operate at 500 MHz, over twice the clock rate of the MTA-2 processor. Finally, the Eldorado design will allow systems with up to 8192 processors, whereas the largest possible MTA-2 system contains only 256 processors.

3 Programming the Cray MTA-2

Because of its use on multi-threading instead of data memory caches to address the gap between processor speed and memory access latency, the Cray MTA-2 platform is significantly different from other modern architectures. These differences are reflected in the MTA-2 programming model and, consequently, its software development environment.

3.1 Programming Model

The key to obtaining high performance on the MTA-2 is to keep its processors saturated, so that each processor always has a thread whose next instruction can be executed. If the collection of threads presented to a given processor is not large enough to ensure this condition (where "large enough" depends on the timing and frequency of memory accesses performed by the threads in the collection) the processor will be under-utilized. For a given program, the size of this thread collection depends on how well the program exposes its parallelism to the system. Because the MTA-2 is a shared memory system, its programming model involves collections of threads that synchronize their access to shared data in memory, rather than separate address spaces communicating via messages as in the MPI programming model.

In the high-level language source code of an MTA-2 program, parallelism can be expressed both implicitly and explicitly [12]. Implicit parallelism is expressed using the source language's loop constructs, such as a C `for` loop or Fortran `DO` loop. The MTA-2 compilers automatically parallelize the

body of such loops so that a collection of threads executes the loop, with each thread executing some of the loop iterations. Ideally, this collection of threads will be interleaved on one or more MTA-2 processors so that the loop iterations are executed in parallel.

There are some restrictions on the types of loops the MTA-2 compilers can parallelize automatically, and sometimes compiler directives must be used to indicate that a given loop can be parallelized. For instance, the number of loop iterations must be determinable before the loop executes, and the loop must not contain complex data dependencies between loop iterations (though the compilers can handle linear recurrences). Directives may be needed to indicate that a given loop can be parallelized, especially for C and C++ programs where the compiler cannot tell whether two pointer variables used within a loop body do not actually point at the same memory location.

In addition to implicit expressions of parallelism in MTA-2 programs, parallelism can be expressed explicitly using *futures*. A future is best suited for expressing task parallelism and recursion, though it could be used for data parallelism. A future is similar to a traditional high-level language function, in that it contains a body comprised of a sequence of statements, can accept input parameters, and can return a result. Unlike a traditional function, executing a future causes a new thread to be spawned; the body of the future executes in the context of this new thread. Because the new thread could execute concurrently with the original thread, futures provide a convenient mechanism for the original thread to synchronize with the future. If the future produces a result, it is assigned to a *future variable* with the same name as the future. The future variable must be a simple type like integer or double, and must be declared using the future keyword before the future itself is declared. Once the future has begun executing, the full/empty bit of the associated future variable is marked empty. If the original thread (or any other thread) accesses the future variable while it is marked empty, the accessing thread will block until the variable is marked full. When the future executes its return statement, the value it returns is assigned to the future variable and the variable is marked full, allowing any threads that were blocked on the future variable to resume.

Whenever multiple threads execute concurrently and access the same location in memory, thread synchronization is usually required to ensure correct operation. For example, if a thread updates a memory location by writing a value to it, and another thread wants to obtain that updated value by reading the memory location, the two threads must synchronize so that the reading thread does not access the location until the writing thread has finished updating the value. The MTA-2 full/empty bits provide a natural mechanism for synchronizing threads.

A future variable is one approach for using the MTA-2 full/empty bits for synchronization between a thread executing the future that will produce the future variable's value and other threads that consume the value. Future variables can also be declared independent of a future and used for synchronization. When used in this manner, a collection of MTA-2 *generic functions* are used in conjunction with simple variable reads and writes for synchronizing access to the future variable's value. MTA-2 generic functions explicitly and atomically manipulate a future variable's value and its associated full/empty bit. For example, the `readfe()` generic function tests the full/empty bit for a memory location, and if the full/empty bit is marked empty, the function blocks till some other thread marks it as full. Then the function reads the memory location, sets the full/empty bit to empty, and returns the location's value. A *synchronization variable* (denoted using the `sync` qualifier in the variable's declaration) can be used for synchronization like a future variable without a future, but simple read and write accesses performed on synchronization variables have slightly different semantics than simple reads and writes to a future variable.

3.2 Programming Environment

The MTA-2 provides a traditional programming environment that includes compilers, a linker, build management software (i.e., a `make` command), and a debugger in addition to the analysis tools described in Section 3.1. The development environment is hosted on the MTA-2 login system, called the "Programming Environment server" or in the Cray documentation [11]. This login node contains SPARC processors and runs Solaris, a version of UNIX from Sun Microsystems. Widespread support for Solaris in the open source community makes it relatively easy to augment the default MTA-2 development environment with tools that operate on source files or control the build system (e.g., the Eclipse Integrated Development Environment). However, because the processor and operating system differ between the MTA-2

compute nodes and its login node and because the MTA-2 compute node processor is not widely supported in the open source community, it is significantly more difficult to deploy open source packages that operate on MTA-2 object files or executables.

The MTA-2 development environment includes C, C++, and Fortran compilers. The Fortran compiler accepts both Fortran 77 and Fortran 90 programs, though support for some Fortran 90 features (e.g., modules) lacks maturity. According to the MTA-2 Programmer's Guide [12], the C compiler accepts both programs that adhere to the 1989 ANSI C language standard and the traditional Kernighan and Ritchie syntax. The C++ compiler accepts programs that comply with a draft of the ISO 14882 C++ standard specification; it is unclear how this draft differs from the final specification approved in 1998. These compilers share a language-independent back-end.

The MTA-2 development environment also includes a debugger called mdb, based on the GNU gdb debugger, but extended to support parallel programs and the MTA-2 processor architecture.

As noted above, the key to obtaining good performance on the MTA-2 is to ensure that each processor always has a thread whose next instruction can be executed (i.e., is not blocked waiting for a memory access to complete), and that one reason that threads are created is for executing the iterations of a loop. However, because program loops are an implicit representation of parallelism, it is difficult for a non-expert to determine whether the MTA-2 compilers will automatically parallelize a loop, and if not, why. To provide insight into the compiler's automatic parallelization, the MTA-2 provides a compiler analysis tool called `canal` that produces an annotated program listing indicating which program loops have been parallelized, which haven't, and why. The `canal` tool also reports the compiler's estimate of how much parallelism has been exposed for each loop, in terms of the number of processor streams that will be requested when the loop's threads are executed.

In some cases, the compiler under-estimates the number of streams required to keep the processor saturated when executing a given loop. A program's run time on the MTA-2 is fairly predictable if the processors are saturated, so situations where the compiler under-estimated the stream requirements are indicated when the program's run time greatly exceeds the predicted run time. When performance falls short of expectations (or when no prediction has been made to establish the expectation), two MTA-2 development environment tools can provide insight into the program's dynamic behavior. The `BPROF` tool is a traditional basic block profiler that supports both a graphical user interface and a text-only report mode. `BPROF` is useful for drawing the user's attention to portions of the code that took a long time to execute, but does not necessarily provide insight into why each portion took as long as it did. The `traceview` tool does provide that insight by showing how well the program used the hardware streams available to it during its execution. Using a program trace produced when the program was run, the `traceview` tool presents a graphical display that shows a timeline of the program's execution. The timeline shows how many hardware streams were allocated to the program at each point during its execution, and how many hardware streams were actually in use. A large gap between the two indicates a portion of the code (e.g., a loop) that did not make efficient use of the processors allocated to it. With a click on the timeline display, `traceview` can provide detailed information about which code was executing at any given time during the program run, including a listing of the program source code. If the compiler under-estimated the number of streams required to saturate the available processors for a loop, the user can use information from `traceview` to estimate how many additional streams to request for the loop, and add a directive to the program source to indicate the number of streams the compiler should request.

4 Applications

We investigated applications and application kernels from several problem domains on the Cray MTA-2. In this section, we detail our evaluation of the MTA-2 for molecular dynamics applications, a fast multipole method, finite difference methods, and a discrete event simulation engine.

4.1 Molecular Dynamics

Molecular Dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating the equations of motion. In the Newtonian interpretation of dynamics, the translational motion of a molecule is caused by force exerted by some external agent. The motion and the applied force are explicitly related through Newton's second law:

$F_i = m_i a_i$.

m_i is the atom's mass,

$a_i = \frac{d^2 r_i}{dt^2}$ is its acceleration, and

F_i is the force acting upon it due to the interactions with other atoms. MD techniques are extensively used in many areas of scientific simulations including biology, chemistry and materials. Atoms in an MD simulation are expected to behave in a way atoms in a real substance would do. For instance, in a MD computer simulation, atoms within a simulated system will move and bump into each other, wander around if the system is fluid, oscillating in waves with its neighbors, in some cases evaporate away from the system if there is a free surface.

MD simulations are computationally very expensive. Typically the computational cost is proportional to N^2, where N is the number of atoms in the system. In order to reduce the computational cost, techniques such as cutoff limit are used. It is assumed that atoms within a cutoff limit contribute to the force and energy calculations on an atom. As a result, the MD simulations do not exhibit a cache friendly memory access pattern. An atom and its neighbors continuously move during a simulation run, and an atom does not interact with a fix pair of atoms. Since, usually the positions of atoms are stored in arrays, multiple accesses to the position arrays in a random manner is required to calculate the cutoff distance, and subsequently to perform force calculations.

The MTA-2 architecture provides an optimal mapping to the MD calculations because of its uniform memory latency architecture. In other words, there is no penalty for accessing atoms outside the cutoff limit or the cache boundaries as in the microprocessor-based systems.

Our MD kernel contains two important parts of an MD calculation: force evaluation and integration. Calculation of forces between bonded atoms is straightforward and less computationally intensive as there are only a very small numbers of bonded interactions as compared to the non-bonded interactions. The effect of non-bonded interactions are modeled by a 6—12 Lennard-Jones (LJ) potential model: $V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$.

LJ potential combines large distance attractive forces (r^{-6} term) and short distance repulsive force (r^{12} term) between two atoms. The integration in our kernel is implemented using the velocity Verlet algorithm, which calculates the trajectories of atoms from the forces. The Verlet algorithm uses positions and acceleration at time t and positions from time $t + \delta t$ to calculate new positions at time $t + \delta t$. The pseudo code for our implementation is given in Figure 1. Steps are repeated for n simulation time steps. n depends on the time-scale of the simulated system and the value of δt.

```
1.  advance velocities
2.  calculate potential energy and forces
        for i=1 to N atoms
          for j=1 to N-1 atoms
            if(i & j in cutoff limits)
               compute force
3.  complete velocities update
4.  calculate new kinetic and total energies
```

Figure 1: MD kernel implemented on MTA-2 (velocity Verlet method)

The most time consuming part is step 2, which is parallelized by the MTA compiler after a simple modification to a reduction operation. Rest of the kernel is parallelized by the MTA compiler without any code modification. We ran two tests using the MD kernel. First, in the strong scaling mode where the problem size is fixed and the number of MTA processors are increased. Second, we keep the problem size per processor fixed by increasing the number of atoms and the number of MTA processors by a factor of two. Results of the experiments are shown in Figure 2 and Figure 3 respectively. In the strong scaling mode, the results are consistent with parallel implementations of the MD kernel, which typically does not scale beyond 8 or 16 processors on an SMP cluster.

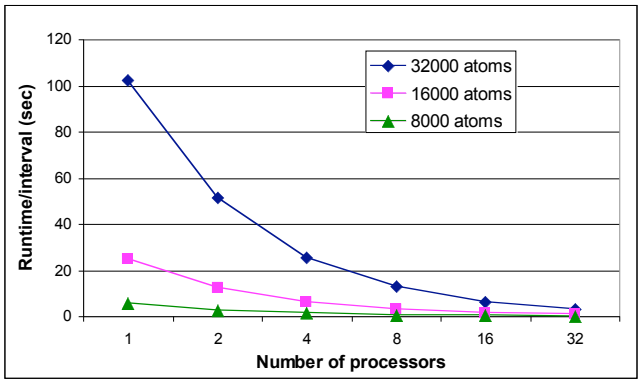

Figure 2: MD simulation in strong scaling mode

The weak scaling mode experiments on the MTA-2 system result in relatively higher

performance ratios as compared to microprocessor based systems. Note that the computation cost increase by N*(average number of atoms in cutoff limit), therefore, the simulation time is not constant as expected in the weak scaling mode. On parallel systems, the inter-processor communication cost is proportional to the number of atoms, therefore, the performance ratios in the weak scaling mode are even lower than the MTA-2 system. As shown in Figure 3, the runtime cost on a single microprocessor system increases rapidly by increasing the number of atoms in an MD simulation.

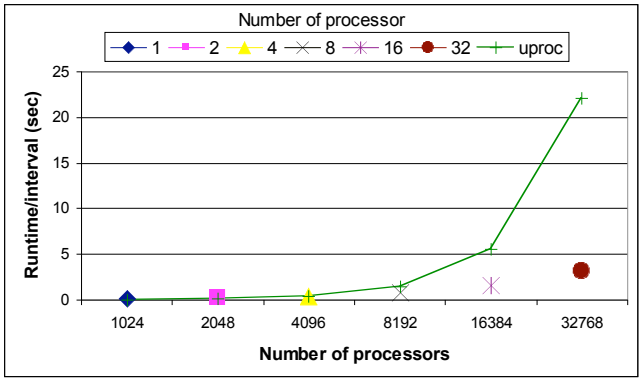

Figure 3: MD simulation with fix workload per processor

4.2 Finite difference solution of an elliptic PDE

A broad range of physical phenomena in science and engineering can be described mathematically using partial differential equations. Determining the solution of these equations on computers is commonly accomplished by mapping the continuous equation to a discrete representation. One such technique is the finite differencing method, letting us solve the equation using a difference stencil, updating the solution at each point as a function of the point and its neighbors, given some discrete time step. Notationally, this is represented by

$$u_{i,j}^{t+1} = \frac{u_{i,j-1}^{t} + u_{i-1,j}^{t} + u_{i,j}^{t} + u_{i+1,j}^{t} + u_{i,j+1}^{t}}{5},$$

for i, j = 1, …, n, for time step t. The Fortran implementation of the above equation is shown in Figure 4.

```
REAL :: GRID ( M+2, N+2 )
          ! Extra space for ghost boundaries.

DO J = 2, N+1
  DO I = 2, M+1
    GRID_NEW(I,J) = &
                  GRID(I-1,J)+              &
      GRID(I,J-1)+GRID(I,J)+GRID(I,J+1)+   &
                   GRID(I+1,J) )            &
              * 0.2
  END DO
END DO
GRID OLD = GRID NEW
```

The strong scaling performance of the 5-point stencil is shown in Figure 5. Our experiments show that regardless of the size of the grid (as long as each processor has a reasonable amount of work), the ability of the MTA-2 to compute the solution scales almost linearly. Further, performance almost perfectly tracks that predicted by the MTA-2 Canal performance tool. That is, for the 5-point stencil loop, Canal shows that five floating point instructions will be executed with eight memory references per loop iteration, for a predicted performance of (5/8)*220MFLOPS*numpes.

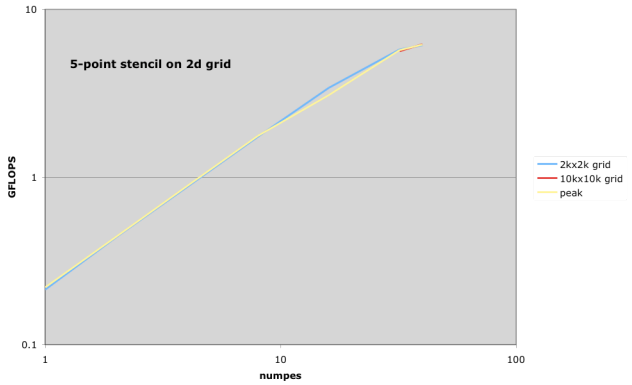

Figure 5: This graph shows the scaling performance of the 5-point difference stencil executed on the MTA-2. Whether operating on a 2,000 x 2,000 grid or a 10,000 x 10,000 grid, the performance is the same, staying within 90% of theoretical peak up to 40 processors.

4.3 Fast Multipole Method

The Adaptive Fast Multipole Method (FMM) [15] approximates the solution to the $O(n^2)$ n-body problem in O(n) time. The FMM is an interesting candidate for the MTA because the combination of its reliance on a tree data structure, by definition a global object, and the adaptive nature of the algorithm make it extremely

challenging to parallelize effectively on distributed memory platforms. Keeping the ratio between interaction computation and tree construction and maintenance low is particularly challenging.

At a very high level the adaptive version of the algorithm proceeds as follows:

- Insert particles into an adaptive oct-tree.
- Traverse tree to create interaction lists.
- Traverse tree from the leaves up to propagate information summarizing particles below.
- Traverse tree to compute interactions.
- Traverse tree from root down to propagate potential to particles below.

A great deal of parallelism can be obtained simply by parallelizing the tree traversals. Our initial MTA implementation of the traversals used the "future" construct to spawn new threads at each internal node of the tree to explore succeeding nodes. Since the upward and downward passes both require synchronization between parents and children, those traversals spawned "named futures" whereby the spawner awaits the completion of spawnees. Traversals that did not require synchronization instead spawned "anonymous futures", touching a synchronization variable to ensure that the futures actually executed.

We found that the use of futures was quite expensive and the insight that there was no need to actually traverse the tree yielded a substantial improvement in performance. For non-synchronized traversals we simply parallelized a `forall` loop over an array containing all the nodes. We used a similar `forall` loop to implement synchronized traversals, with the addition of read and write operations to empty/full bits to ensure that parents or children had already executed.

The only major algorithmic phase that required a different parallelization strategy was the initial tree construction phase. The obvious solution is to parallelize the loop that inserts particles into the tree. This approach, however, requires substantial synchronization to ensure that new tree nodes are uniquely created and properly linked to their parents. Unnecessary locking of boxes at upper levels of the tree in our initial implementation serialized particle insertion: we locked each box, *and* its parent, on the way down to the leaf in order to ensure that when a thread found the leaf box in which to insert a particle, the leaf was not transformed into a parent box by another thread.

The improved version instead uses synchronizing reads of parent boxes to get to a leaf, and only then locks the leaf. A retry mechanism ensures that if the leaf has been modified between the last read and obtaining the lock, then the lock is released and traversal to the new leaf continues. The mechanism is similar to a `cmpxchg` atomic instruction in which if the content of the targeted memory location is not equal to the expected content, then the exchange is not performed.

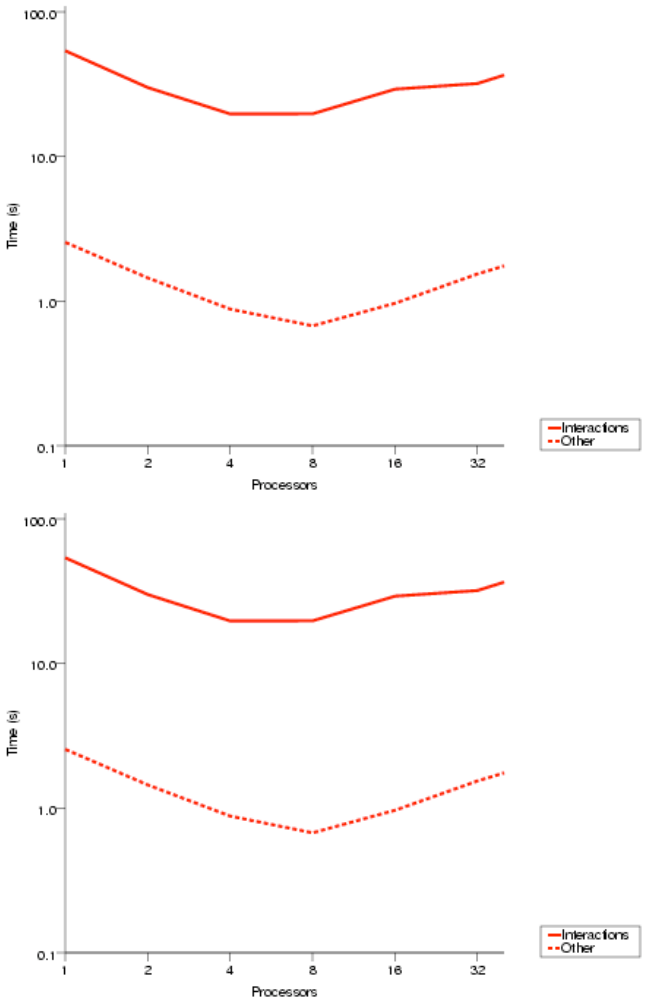

Figure 6: Weak (top, 64K particles/processor) and strong (bottom, 64K total particles) scaling performance of the initial FMM implementation.

Figure 6 demonstrates that while the implementation described above does quite well under weak scaling assumptions, it does not fare as well when the number of particles is held constant as processors are added.

The problem appears to be the result of the coarse granularity of our parallelization strategy.

As described above, parallelism comes from tree traversals, and is therefore limited by the number of nodes in the tree, low compared to the number of particles. Additionally the amount of work per node is not consistent in the adaptive algorithm, depending as it does on the number of particles a node contains (if a leaf), and the number and quality of its neighbors. The very slow single-thread performance of the MTA makes it particularly susceptible to this kind of load imbalance.

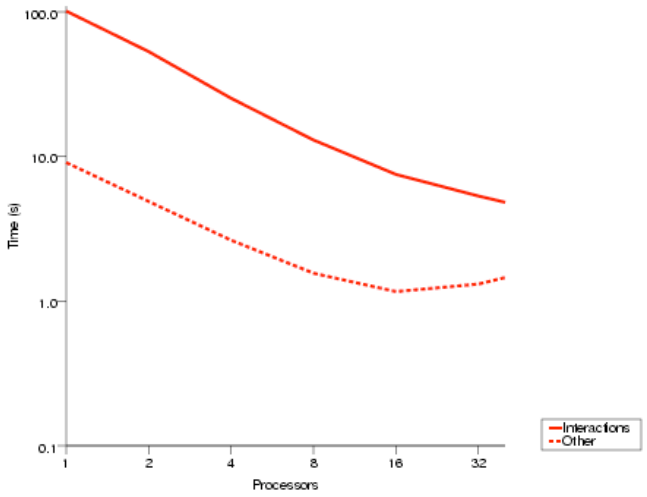

Figure 7: Improved strong scaling due to lowering the maximum particles per leaf from 128 to eight.

We have found that one way to address both problems at once is to adjust the maximum number of particles per leaf. This is a runtime parameter to the algorithm, allowing the following tradeoff: more particles per leaf means a shallower tree, fewer nodes, thus fewer interactions between nodes and greater accuracy; fewer particles per leaf means a deeper tree, more interactions between nodes, more savings but less accuracy, and for our purposes, more parallelism. An additional benefit is more consistency in the work per box, since that work is dependent on the particles in a box, and as the maximum particles per box decreases the variance in the number of particles per box also decreases. Figure 7 demonstrates the improvement in scaled performance that results from decreasing the particles per leaf from 128 down to eight.

Finally, we describe a completely different approach to parallelizing interaction computation that is likely feasible only on the MTA: we move the granularity of parallelism from the nodes participating in the interactions down to the interactions themselves.

In the original formulation of the adaptive FMM algorithm, and in every implementation of it we have seen, the interaction lists are parameterized by the source node, that is, the node for which the interactions are to be computed. This leads to a straightforward computation partitioning strategy for parallelizing interactions: divide nodes across processors based on the total work as determined by each node's list contents, then compute interactions for the nodes in parallel. A major advantage of this strategy is the absence of the need to synchronize updates to the fields of a node after each computation.

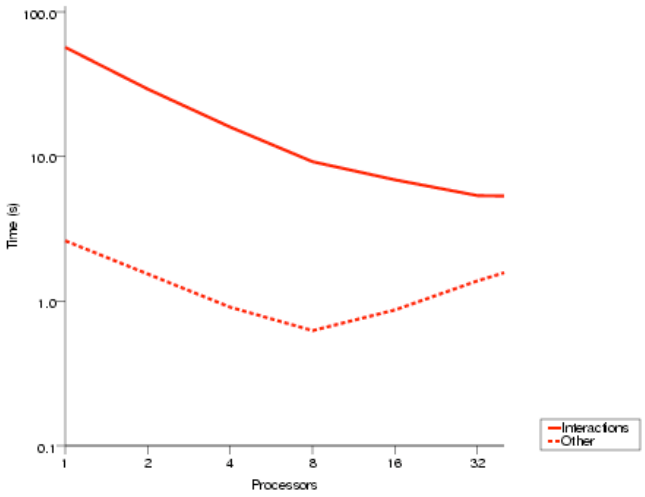

Figure 8: Improved strong scaling due to more fine-grained parallelization of interaction computation.

We instead modify the list creation phase to create a single global interaction list, and then parallelize the loop that goes through the list. Each thread computes a single interaction and then executes a `readfe/writeef` combination to atomically update the appropriate field. Figure 8 demonstrates the effectiveness of this approach. Note that it enables significantly more parallelism even when the number of particles per box is high. It is unclear whether the drop in efficiency at high processor counts is the result of contention or residual load imbalance due to variance in the time to compute interactions. Another question for future exploration is whether we can similarly crack open the computation in the upward and downward passes.

4.4 Discrete Event Simulation

Discrete-event simulations (DES) are a special class of computer systems simulation in which the

ordering and timing of events is the main focus of interest. These systems primarily focus on the timing or time stamp of an activity, when it commences or ceases within a system. For example, in simulating computer networks to estimate effective system capacity or queue sizes, the important parameters may be the start time and duration of job processing rather than details of the signal transmission in the network. Thus, in such a problem, it is not efficient to advance time in small fixed time steps but to advance to the time of the next event. Since events can occur at any time, the time advances in non-uniform and can be either very small or very large. Typical applications of the discrete-event simulation systems include factory layout and process planning, transport systems, telecommunication networks and office management systems.

The model development process in a DES system begins with identifying the important events that occur in a real system. Typically, the events and their consequences that influence the progress of a simulation are stored in a 'list' or a 'queue'. The simulation is progressed by working through the time-stamped 'event list'. These calculations are inherently sequential and exhibit very low memory locality. Optimization techniques such as look-ahead and roll-back have been used to process the events in the event list out-of-order. But the overheads involved in implementing let alone parallelizing these techniques are very large.

We have ported and optimized a simple DES kernel onto the MTA-2 system. The kernel consists of a priority queue implementation and two loops: the first loop generates random events and inserts them into the queue; the second removes events from the queue in timestamp order.

A straightforward parallelization strategy for the two loops simply serializes insertion and removal from the queue via a lock is shown in Figure 9:

```
For 1 to MAX_ELEMENTS in Parallel
   Create an event with a random timestamp
   Lock()
   Insert event in Priority Queue
   Unlock()

For 1 to MAX_ELEMENTS in Parallel
   Lock()
   Remove the event with minimum timestamp
   Unlock()
```

However, this code performs extremely poorly on the MTA, even on a single processor, due to the lack of actual parallelism. Our goal for this initial exploration was to attempt to improve the performance of the kernel by taking advantage of the extremely fine-grained synchronization mechanisms offered by the MTA to enable simultaneous insertions *or* removals from the priority queue.

The priority queue implementation is binary tree-based. Both insertion and removal operations maintain the invariant that a parent is smaller than both of its children -- thus the smallest element is located at the root – and, moreover, do so in O(log n) operations. The sequential operations are implemented as follows:

- Insert: add the element as a leaf; move the element up the tree, swapping with its parent, until its timestamp is greater than that of the parent.

- Remove: remove the root and replace it with the last leaf; move the new root down the tree, swapping with the smallest child, until both children are larger.

Insertions can occur in parallel so long as 1) each newly inserted element is given a unique slot, and 2) the test for whether a swap is necessary occurs atomically with the actual swap, which also must be an atomic operation, so two threads do not attempt to swap the same parent with different children at the same time. We ensure the first condition is met through the use of the `int_fetch_add()` intrinsic procedure, which atomically increments an integer variable, to find the next available slot in the tree. We lock the parent and child before checking, and potentially swapping, to ensure the second condition holds.

Similarly, removals can occur in parallel so long as 1) a unique leaf is found and moved to replace the removed root, 2) the new root is not removed prior to a necessary swap with one of its new children, and 3) the tests for swaps, and the swaps, occur atomically. Again, the int_fetch_add() intrinsic enables us to meet the first condition, while locks around the parent and each of its children ensure the remaining two conditions hold.

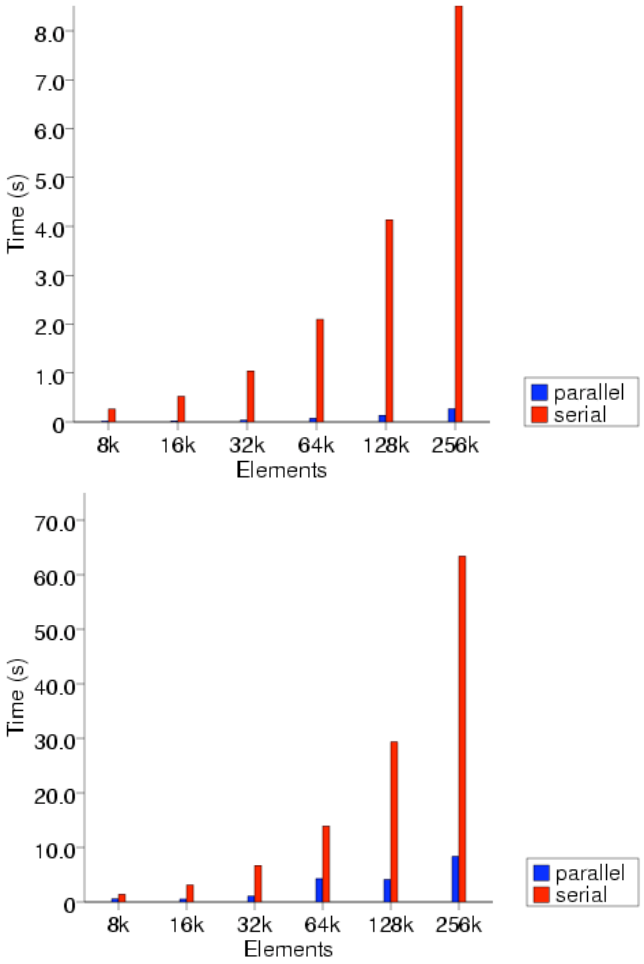

Figure 10: Performance of the parallelized insertion (top) and removal (bottom) loops on a single processor.

To evaluate the effectiveness of our fine-grained locking strategy on MTA hardware we compare the running times of a kernel using our improved insert and remove routines with a version that serializes insertions and removals with a coarse-grained lock.

Figure 10 compares performance of the two implementations on a single processor as the number of elements inserted and removed increases. The fine-grained locking in the insertion algorithm appears to allow hardware threads to fully utilize the resources of the processor: the speedup over the serial execution is nearly 40X. The improvement for the removal algorithm, on the other hand, tops out at 8X. While a significant improvement, the lower factor suggests that contention is preventing threads from fully utilizing the resources available even on a single processor.

Figure 11, which demonstrates the performance of multiple processors operating on a fixed (256K) number of elements, confirms these suspicions: while the insertion routine exhibits near-perfect scaling as the number of processors increases, the removal routine does not scale at all.

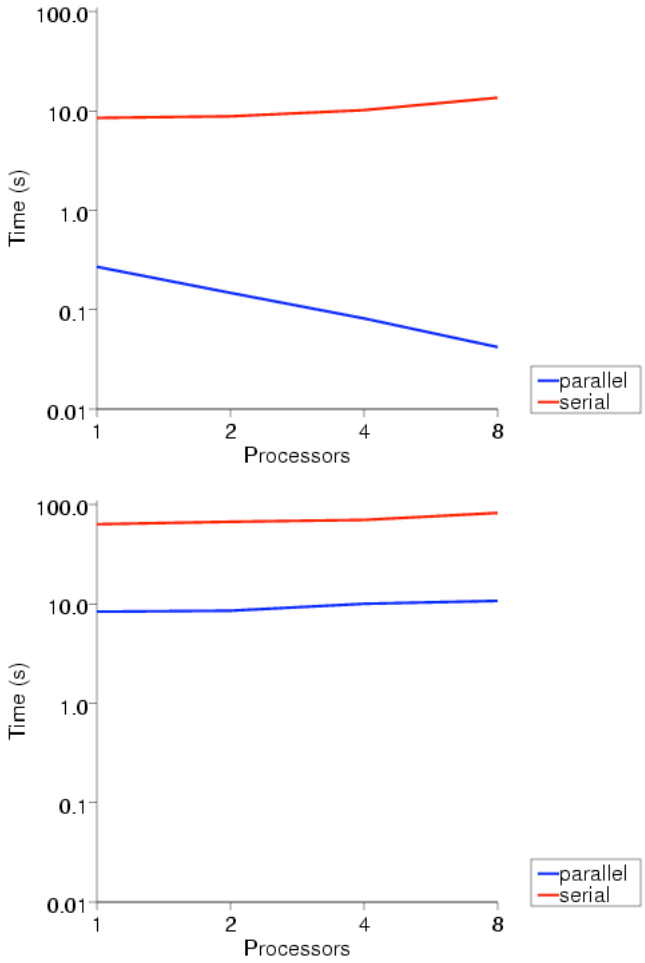

Figure 11: Multiprocessor scaling performance of the insertion (top) and removal (bottom) loops. The number of elements is fixed at 256K.

At this point we do not see any obvious means of improving our locking strategy for removals. Perhaps the best path forward is to determine whether removal is actually on the critical path of a more realistic discrete event simulation, and if so to look into different data structures and/or implementations of the priority queue. Additionally, we still need to ensure that insertions can occur simultaneously with removals under our current locking strategy.

5 Inferring Eldorado Performance

Because the MTA-2 is no longer a commercial product, perhaps the most important question we set out to answer with this work is: What do these experiments on a 40-processor MTA-2 tell us about the potential performance of our applications on Eldorado?

A major change from the MTA-2 to the Eldorado architecture is the move away from uniform memory access latency. MTA-2 programmers are encouraged not to be concerned with memory access locality, but the extent to which such locality must be considered on Eldorado is not yet known. One encouraging observation is that the FMM algorithm provides a great deal of locality that we have not exploited in our current implementation. On the other hand, the tree construction algorithm and the cracked open interaction computation of our implementation are both heavily dependent on low latency synchronization provided by MTA-2. Similarly, our discrete event simulation algorithms rely heavily on the low-latency synchronization provided by the MTA-2. It is yet not clear how these algorithms will perform on Eldorado if the synchronization cost is higher than that on the MTA-2, or if the synchronization cost depends on the placement of synchronized objects within the system's memory.

6 Related Work

This research follows several earlier investigations into the suitability of multithreading (as implemented by the MTA-2 and its predecessor) for scientific computing. Snavely et al [22] investigated the performance of several kernels and applications on a Tera MTA system, the predecessor to the MTA-2. Oliker and Biswas [21] considered the suitability of the MTA approach for irregular, dynamic applications. Miyamoto and Lin [19] considered SPMD programs written in the Titanium programming language [23] on the MTA platform. Our work complements this previous work by examining applications and kernels from problem domains that were not previously considered, on a recent incarnation of the architecture.

Henry et al [17] present an implementation and preliminary performance results of a discrete-event simulation kernel on the MTA. This particular kernel is designed for modeling large-scale computer networks. Performance results show that the implementation is competitive with a single processor system and scales only if the number of timelines is large, i.e. the timelines are greater than the number of streams available on the MTA system.

Bokhari and Sauer [8] investigated dynamic programming sequence alignment algorithms for DNA sequences on the Cray MTA-2 system. Their algorithms are reported to scale to up to eight MTA-2 processors and the implementation relies extensively on the use of full/empty bits in MTA-2 memory to facilitate parallel execution in the dynamic programming algorithms. They do not discuss performance comparisons with their algorithm on other systems, or with other algorithms.

7 Conclusions

General purpose platforms like the Cray XT3 [1] successfully address a broad set of scientific computing requirements. However, some important types of algorithms, such as graph theoretic and pointer-chasing algorithms, cause data access patterns that do not map well to traditional cache-based architectures. The MTA-2 uses an approach that is strikingly different from this traditional system architecture, an approach that can result in extremely high utilization and excellent scalability for these troublesome algorithms. In this report, we have shown quantitatively that applications from several areas, including molecular dynamics, fast multipole methods, and discrete event simulations can perform well on the MTA-2 architecture.

The MTA-2 was not a commercially successful product for Cray. Nevertheless, because of the Eldorado platform, the highly multi-threaded model remains a viable option to support high performance computing. When Eldorado systems become available, we expect to perform a similar evaluation to the one described in this report as part of our continuing system evaluation efforts, and anticipate an interesting and insightful comparison of that new architecture's performance with the results presented in this report.

References

[1] S.R. Alam, R.F. Barrett *et al.*, "Evaluation of the Cray XT3 at ORNL: a status report," Proc. Cray User Group 2006 Annual Technical Conference, 2006.

[2] G. Alverson, R. Alverson *et al.*, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in *Proceedings of the 6th international conference on Supercomputing*. Washington, D. C., United States: ACM Press, 1992

[3] G. Alverson, P. Briggs *et al.*, "Tera hardware-software cooperation," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*. San Jose, CA: ACM Press, 1997

[4] R. Alverson, D. Callahan *et al.*, "The Tera computer system," in *Proceedings of the 4th international conference on Supercomputing*. Amsterdam, The Netherlands: ACM Press, 1990

[5] G.M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Computer Design*, 6(12):39–40, 1967.

[6] D.H. Bailey, E. Barszcz *et al.*, "The NAS Parallel Benchmarks," *The International Journal of Supercomputing Applications*, 5(3):63-73, 1991.

[7] M.B. Blackmon, B. Boville *et al.*, "The Community Climate System Model," *BAMS*, 82(11):2357–76, 2001.

[8] S. Bokhari and J. Sauer, "Sequence alignment on the Cray MTA-2," *Concurrency and Computation: Practice and Experience (Special issue on High Performance Computational Biology)*, 16(9):823–39, 2004.

[9] P. Briggs, "Sparse matrix manipulation," *SIGPLAN Not.*, 31(8):5-7, 1996.

[10] S. Brunett, J. Thornley, and M. Ellenbecker, "An initial evaluation of the Tera Multithreaded Architecture and programming system using the C3I parallel benchmark suite," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. San Jose, CA: IEEE Computer Society, 1998

[11] Cray Inc., "Cray MTA-2 Computer System User's Guide," Cray Inc. S-2317-10, 2005.

[12] Cray Inc., "Cray MTA-2 Programmer's Guide," Cray Inc. S-2320-10, 2005.

[13] Cray Inc., *Cray MTA-2 System - HPC Technology Initiatives*, http://www.cray.com/products/programs/mta_2/, 2006.

[14] J. Feo, D. Harper *et al.*, "ELDORADO," in *2nd conference on Computing Frontiers*. Ischia, Italy: ACM Press, 2005

[15] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Physics*, 73:325–48, 1987.

[16] J.L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, 31(5):532–3, 1988.

[17] R.R. Henry, S.H. Kahan *et al.*, "An implementation of the SSF Scalable Simulation Framework on the Cray MTA," Proc. 17th Workshop on Parallel and Distributed Simulation (PADS'03), 2003, pp. 77–85.

[18] D.J. Kerbyson, H.J. Alme *et al.*, "Predictive performance and scalability modeling of a large-scale application," Proc. IEEE/ACM Conference on Supercomputing (SC'01), 2001.

[19] C. Miyamoto and C. Lin, "Evaluating Titanium SPMD programs on the Tera MTA," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. Portland, Oregon, USA: ACM Press, 1999

[20] L. Oliker and R. Biswas, "Parallelization of a dynamic unstructured application using three leading paradigms," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. Portland, Oregon, United States: ACM Press, 1999

[21] L. Oliker and R. Biswas, "Multithreading for dynamic irregular applications," in *First SIAM Conference on Computational Science and Engineering*. Washington, D.C., USA, 2000

[22] A. Snavely, L. Carter *et al.*, "Multi-processor performance on the Tera MTA," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Orlando, Florida, USA: IEEE Computer Society, 1998

[23] K. Yelick, L. Semenzato *et al.*, "Titanium: a high-performance Java dialect," *Concurrency: Practice and Experience*, 10(11–13):825–36, 1998.