Evaluating Titanium SPMD Programs on the Tera MTA

Carleton Miyamoto, Chang Lin
{miyamoto,cjlin}@cs.berkeley.edu
EECS Computer Science Division
University of California, Berkeley

Abstract

Coarse grained threading, such as in the SPMD model of parallel programming, has been shown to be an effective and well-understood programming model for performance-oriented languages such as Titanium [2]. Some parallel architectures, however, such as the Tera MTA, prefer light-weight threading in order to achieve high performance. This paper describes a port of the Titanium compiler to the Tera MTA and the mapping of its SPMD constructs to a more dynamic, threaded model. The Titanium language was enhanced to allow the programmer to specify non-SPMD-style loop-level parallelism. To evaluate the platform and language interface, a few Titanium applications were modified to take advantage of the language changes and were measured for performance. They show that thread overhead and synchronization may still be a limiting factor for scaling SPMD programs. With automatically detected loop-level parallelism, associating threads with a specific processor can greatly reduce the thread overhead for some applications. Finally, the data suggests that having only-coarse grained parallelism may not keep the Tera processors busy so parallelizing compiler optimizations are a necessity.

1 Introduction

As parallel computing has become more widely used, the inherent difficulties of expressing computations have given rise to a variety of parallel languages. Some of these, such as HPF and Split-C [1], have taken traditional, familiar languages and extended the semantics for parallelism. By extending known language abstractions and porting these compilers to new platforms, much of the initial investment of the parallel programmer, in both the learning curve and code bases, can be preserved. Although some program modifications are often needed, the details of synchronization and communication constructs are largely abstracted away by the languages.

This paper describes the port of the Titanium [2] reference compiler to the Tera MTA [3]. The parallel constructs of the language were mapped to the Tera's native mechanisms for parallelism. In order to understand the mapping and the interaction between the Tera and Titanium, several Titanium applications were evaluated on the architecture. The results provide an insight into the resources required by the applications and the Tera's ability to provide them.

1.1 The Tera MTA

The Tera Multi-Threaded Architecture provides hardware support specifically targeted at the multi-threaded programming style. The architecture supports multiple logical register files and thread states active simultaneously on a single CPU. The Tera MTA currently specifies 128 *streams* in a processing unit. Each *stream* owns a register file and thread state and is the basic unit of scheduling. These streams share the CPU's other resources, such as the ALU, FPU, and other parts of the processor pipeline. Because only one stream is active at a time, to maintain a fine grain of concurrency, streams are context switched on every processor cycle. Streams, in effect, provide hardware support for multiple lightweight threads.

Unlike current trends in memory subsystems, the Tera's memory architecture is flat. No data memory caches exist on the processor. Instead, multiple memory banks are connected to the processor by an interconnection network. On a memory access, the address is hashed, which is an attempt to randomize its actual placement in the memory banks. The network then retrieves the desired memory contents within the order of one to two hundred cycles. The MMU is non-blocking, allowing multiple memory requests to be outstanding on a single processor. This memory architecture design is an effort by the Tera designers to free programmers from having to deal with a multi-level memory hierarchy, which can greatly complicate their algorithms. In fact, the almost-uniform memory access latencies nullify the effect of many data placement optimizations by the programmer or compiler.

Along with the flat memory architecture, each memory word contains, among other information, a full-empty bit. These bits provide a fast method of synchronization and communication among threads. When the bit is not in the proper state, accessing the memory location causes the stream to block until the state changes. The Tera native compilers [4] utilize them to provide higher-level synchronization primitives as well as threading constructs.

Streams and the memory architecture provide a unique cost model for programming for the Tera MTA. Instead of programming for data placement, which reduces data latencies, the programmer *invests* some of the parallelism in the code to hide data latencies. This investment is profitable only if the code has sufficient parallelism to both invest and keep the processors busy during execution. The programmer, however, benefits from reduced program complexity, which improves code maintenance costs, debugging, and readability. In order to help find this parallelism, the Tera native C, C++, and Fortran compilers feature strong support for automatically detecting parallel opportunities in sequential code. They also provide explicit synchronization variables and a threading model based on futures. In the former, a **`sync`** keyword along with several specialized functions, provide the means to build complex high level constructs. In the latter, the programmer can easily specify a semantic fork/join style of thread control.

1.2 Titanium

Titanium [2] is a Java-based, Single Program Multiple Data (SPMD) language that contains explicit communication and synchronization constructs for parallel programming. These language constructs consist of barriers, broadcasts, exchanges, and the Java synchronized qualifier. The Java class libraries, which provide services such as I/O, strings, and containers, are used from the standard distributed Java implementation from Sun. The compiler generates C code, which is then fed into a machine's native compiler, to generate an executable. At startup, a Titanium executable creates a fixed number of Titanium processes, each of which executes the same code, but with logically separate address spaces. The reference compiler supports NOW clusters [5], SMPs, and MPPs. In the NOW case, logical Titanium processes map onto separate individual processes running on each machine of the cluster. With SMPs, Titanium uses the operating system's native thread support (such as the Posix pthread library) in a single address space, typically using one thread per Titanium process. Multiple address spaces are then simulated for specific variables when necessary. Finally, on MPPs, the natural parallel constructs of the machine or runtime libraries are used.

Porting Titanium to a new architecture requires implementing the small, well-defined section of the Titanium runtime library that contains native synchronization and multiprogramming routines. These functions provide the mapping from the Titanium SPMD constructs and abstractions to the machine. On threaded platforms, such as SMPs, for example, memory accesses are optimized for the shared memory address space, and the usual overhead of supporting remote memory accesses on distributed memory platforms is fully avoided. This flexibility allows the compiler to easily take advantage of the unique benefits of each platform.

1.3 Applications

To evaluate the appropriateness of the Titanium SPMD model to the Tera multithreaded architecture, a few Titanium applications were tested. They were designed and optimized for either the shared memory or for the distributed memory model.

1.3.1 Parallel 3D Adaptive Mesh Refinement

The 3D AMR application [6] solves Poisson's equation within a discrete problem domain, a *patch*. Multiple domains spread across multiple Titanium processors over a 3D space provide a high level of parallelism, with communication only necessary where domains abut. AMR is similar to the multigrid algorithm for adaptively refined grids, with additional support to allow fine grid levels to not completely cover coarser levels. The Titanium AMR is implemented to scale on both shared and distributed memory machines, using ghost cells when necessary. A red-black ordering is kept when computing the values within a patch, which consists of multiple double-precision floating-point values. Patches are distributed in groups among all of the Titanium processes involved in the computation, as specified by the input. Parallelism within each patch in a group or within the patch data items is only implicit within the program's loops. Additional details on the algorithm can be found in [6].

1.3.2 EM3D

The second application tested is a Titanium version of the EM3D kernel. It was written in a naïve fashion for shared memory machines and uses Titanium's **`foreach`** looping structure to traverse the nodes and edges in a linear manner. No special care is taken in data placement. The initial input is a randomized set.

1.3.3 FFT

FFT is a Titanium version of the FFT in the Stanford SPLASH-2 [14] benchmarks for shared memory systems. It is a complex 1D version that minimizes inter-processor communication. The loops in this code, as with EM3D, were not parallelized.

1.4 Paper Organization

The rest of this paper is organized into several sections. Section 2 shows the mapping chosen between Titanium SPMD parallel constructs and the Tera's. Section 3 describes the same for synchronization. In section 4, the results of the application evaluations are presented. Finally, some related and future work is discussed.

2 Parallelization

Implementing the Titanium runtime backend for the Tera requires writing two types of routines. The first involves forming a mapping between the Titanium parallel constructs and those provided by the Tera architecture. The Tera supports two types of parallelism - explicit threads, and implicit loop-level, both of which are exploited.

3.1 Explicit

Explicit parallelism is parallelism specified explicitly by the program. On the Tera, futures and future variables provide this feature to the programmer. This works similarly to fork/join parallelism. A future statement specifies a fork point in the program. This statement is tied to a future variable. The full-empty bit on this variable is asserted when the future thread completes. So, simply reading the future variable specifies a join point. Titanium SPMD processes are implemented as Tera threads in this manner. As in an SMP machine implementation, the use of threads implies that all of the Titanium processes share a single address space.

In order to allow the number of Titanium processes to scale to large numbers, the spawning function implements a tree structure. Actually a flat binary tree, each Titanium process, a Tera future, spawns two children at startup before starting its normal execution. Each thread can also optionally be bound to a specific Tera processor or be allowed to migrate throughout the system.

3.2 Implicit

In the Tera system, implicit parallelism is not necessarily specified directly by the programmer, but is discovered by a parallel compiler. The form most often found is loop-level parallelism, where multiple iterations of a loop are independent and, so, can be executed in parallel. The Tera compiler chain aggressively detects loop-level parallelism and includes a series of directives for the programmer to help the compiler in this respect.

In addition to Java looping structures, Titanium supports a **`foreach`** loop that iterates over a multi-dimensional space of points. Roughly equivalent to multiple nested **`for`** loops, they are unordered but semantically operate serially. Strength reduction and loop-invariant code motion reduce overhead in the body of the loop. Because Titanium produces C code, however, aliasing, worsened by the loop optimizations, sufficiently confuses the Tera compiler and its ability to optimize loops [7]. Instead of relying on alias analysis, which, in pointer (and reference) based languages, has been shown to be either hard or inaccurate [8, 9, 10], a new **`foreachp`** construct was added to the language. Similar to the **`foreach`** loop, this new construct does not specify, but instead hints, that no loop carried dependencies exist, and the loop can be executed in parallel.

3 Synchronization

In addition to the parallel constructs, implementing the runtime backend for the Tera involves mapping Titanium's synchronization constructs to the Tera architecture. The Tera C compiler allows program variables of word length or smaller to be declared as synchronization variables, which means that their state is changed to empty on a read and to full on a write. Reads of empty variables will block, while writes of full variables may block, depending on whether the variable is a *sync* variable (for blocking writes) or a *future* variable (for non-blocking writes). The compiler also contains intrinsic functions that can explicitly manipulate the full/empty states of program variables, as well as perform blocking or non-blocking reads and writes to those variables. We used these low-level, fine-grained synchronization constructs to build locks and block-and-broadcast-release mechanisms, on top of which we built the higher-level, coarse-grained constructs present in the Titanium language.

Locks were implemented in a natural manner, by mapping the lock's state directly onto the full-empty state of a word in memory. To acquire a lock, the thread reads from this word, which clears its state to empty, causing subsequent reads to block. When the thread releases the lock, it writes to the word, changing the state back to full. Only one thread is allowed to perform a synchronized read on a word at a time, so only one of the waiting threads is allowed to acquire the lock. We did not need to implement a spin-wait for performance, because the Tera hardware has a built-in mechanism that spin-waits a thread for a certain interval. Block-and-broadcast-release was implemented using two alternating synchronization variables. A thread wishing to block reads from one variable, and the thread which releases the others toggles a switch so that further blocking threads reads from the other variable. This prevents a release from allowing future blocks to pass through unchallenged.

When these constructs had been implemented, the two higher-level Titanium synchronization constructs, barriers and monitors, were simple to implement. Two types of barriers were reviewed. The first was implemented by using our block-and-release mechanism and an atomic counter, supported in hardware with an atomic fetch-and-increment operation (**`INT_FETCH_ADD`**). This is termed the *counter barrier*. The second, a *tree barrier*, uses a flat binary tree structure with parents waiting for their children to finish, then blocking waiting for the root node to finish. Both use a dual state approach and a toggle to simplify cleanup of the barrier mechanisms. Monitors were implemented using locks to protect the desired critical sections.

4 SPMD Programs

The Tera machine, on which the test programs were evaluated, is a four-processor configuration, running at 260MHz with 4GB of main memory, running MTX OS v0.4.52. The AMR Titanium executables are built with the Tera C compiler's (**`tcc`** v0.4) whole program optimization to ensure better automatic loop parallelization by the optimizer. The other programs used separate compilation, which should be sufficient because they were not loop-parallelized. Implicit parallelism optimizations, **`-par`** and **`-par1`** in the Tera compilers, were activated where noted. The former uses a *crew*, a more heavyweight but more flexible thread mechanism, to perform loop parallelism across multiple CPUs, while the latter relies on *frays*, which have fewer overheads but are limited to living on a single processor. Titanium's runtime garbage collection system was turned off for these tests.

The AMR program was modified to parallelize loops using the new **`foreachp`** construct. All loops that dealt with traversing the points within a patch were marked parallel for the Tera C compiler to optimize. Loop parallelism among patches was not specified, allowing for Titanium's SPMD parallelism. Four different input sets were presented to AMR in the testing. They varied the patch size present on each processor from a 16x16x16 patch to a 64x64x64 one. Most of the Titanium runtime itself was not parallelized, including the Java class libraries. The array copy function, however, was optimized to execute in parallel.

4.1 Tree or Counter Barriers

In many shared memory machines, tree based barrier code reduces consistency communication among the processors, especially under high contention. This provides a better degree of scalability at the cost of using a vector of synchronization variables on the order of the number of processes. On the other hand, they are more complicated than counter based barriers and may introduce some unnecessary overhead.

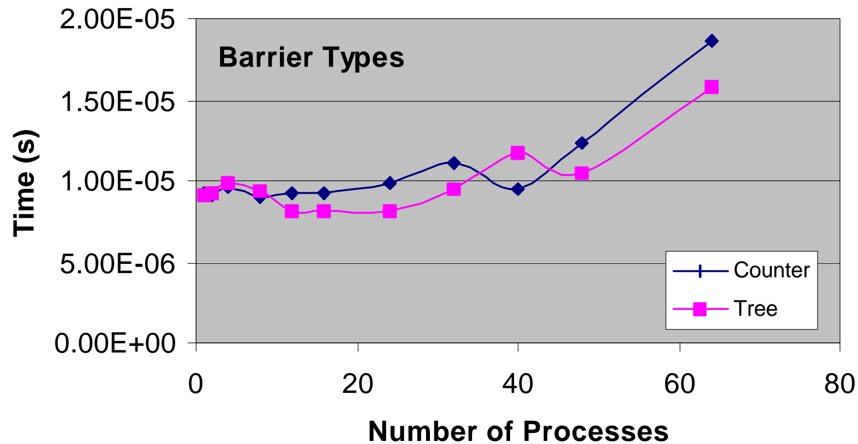

Figure 1. Counter and Tree barrier timings under EM3D. A flat line would indicate linear speedup.

Both types of barriers were implemented and evaluated on the Tera. The results (Fig. 1) show that the timings are relatively close on EM3D. EM3D executes a fixed number (determined at runtime) of stages with barriers in between each stage and no other synchronization. This should make it a stable benchmark to test the scaling of barriers. The tree timings are consistently better than the timings for the counter variable, though only by a small degree. The differences, however, are within the noise of the machine from run to run, although most of these values were repeatable. Additional scaling numbers for higher processes are not available due to bugs in the Tera runtime. So, the data suggests that the tree barrier may still be more efficient on the Tera but only by a small, possibly a fixed constant amount.

4.2 Processor Affinity

Normally in SPMD Programs, an SPMD process is closely tied to a specific processor. This allows the programmer to reason about the CPU resources available and to program with those assumptions. On the Tera, however, each SPMD process corresponds to a Tera thread. A question arises whether these threads should be bound to a specific processor or not. Figure 2 shows the results of this experiment.

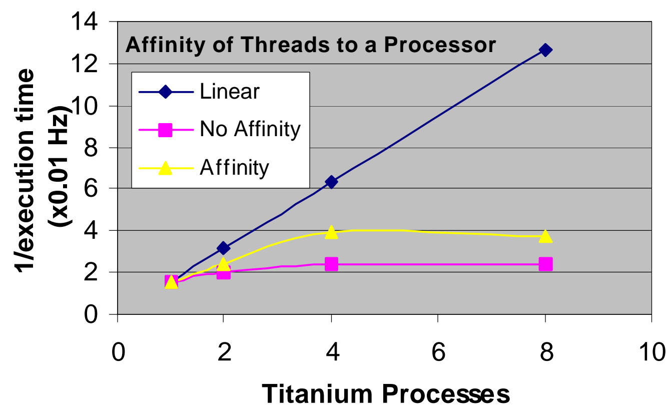

Figure 2. Effects of binding a Tera thread to a specific processor. The *Linear* line shows linear scaling. This is AMR executing with 32x32x32 sized patches. There are a total of 9 patches that are assigned to the processors in a round-robin fashion.

AMR performs better if each Titanium process is tied to a specific processor (i.e. the thread has *processor affinity*). This can be attributed to a few factors. First, each Titanium AMR process has enough loop level parallelism to keep a single, entire Tera processor busy, and the patches are balanced. As a result, there are no benefits to a more dynamic scheduling and load balancing. The balanced parallelism can be seen in the trace information for single process runs of the program. Also, when a thread is tied to a processor, loop-level parallelism can be implemented with frays instead of crews to synchronize those mini-threads, resulting in a large reduction in overhead. The much larger number of total streams issued by the crew case reflects this, again, in the trace information.

The Tera two- and four-process speedup is significantly less than the two- and four-process configuration reported in [6]. One possible source is the serial code in both AMR and the Java class library, which Titanium programs use extensively. Because of the Tera's poor serial performance (as observed in [12]), this code takes up a larger portion of the runtime, causing it to dominate more quickly, lowering the overall speedup.

4.3 Scalability

A key issue for AMR, scalability allows for larger problems to be solved as the hardware itself becomes more powerful. Figs. 3 and 4 show the peak rates attained for issuing threads and MFLOPS.

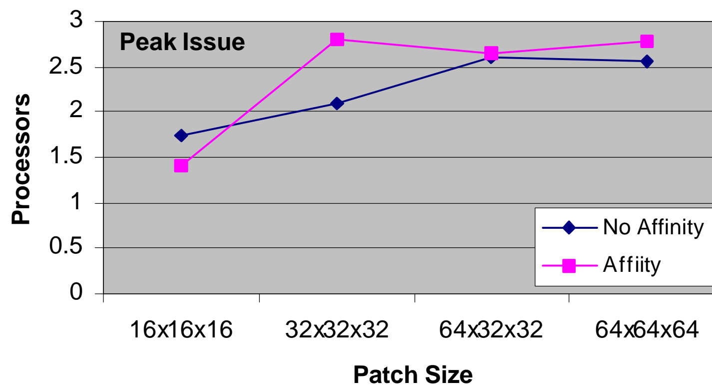

Figure 3. The peak thread issue rate in AMR while computing a patch of that size. The Y-axis is normalized to 1, where 1 signifies all of the resources of a single processor. In the legend, *No Affinity* means crew threading was used on all four processors. *Affinity* means that fray threading was used with two Titanium processes bound to each processor. In each case, eight Titanium SPMD processes were created.

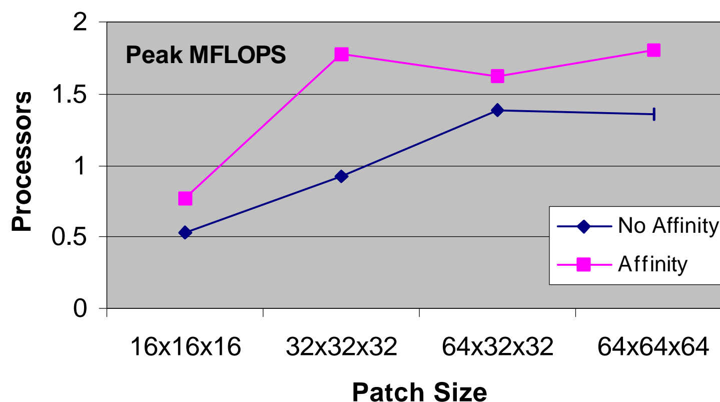

Figure 4. Peak MFLOPS achieved in AMR. A 1 on the Y-axis is the MFLOPS rating for a single processor.

Both of these graphs for all cases begin to level off at the larger patch sizes. As the patch size increases, the amount of loop-level parallelism also increases, so although the number of Titanium processes is fixed, the total amount of parallelism increases. The bound SPMD processes (*Affinity*) seem to saturate more quickly in both cases. Assuming that the saturation was caused by machine bottlenecks (the limited memory network or a shortage of streams in parallelized loops—both of which were reported by earlier works), this mechanism would most benefit from advances in the hardware, as well as intermediate problem sizes. Note that the issue rate peaks at about 2.7, which, on the four processor system, results in a utilization of only 67.5%, much lower than those measured by [11]. Several factors may have contributed to this, including not parallelizing the Java portion of the Titanium runtime library and the greater use of pointers and references that arise from an object oriented language.

Figs. 5 and 6 show the average thread issue rate and the average floating point performance when averaged over the entire run of the program.

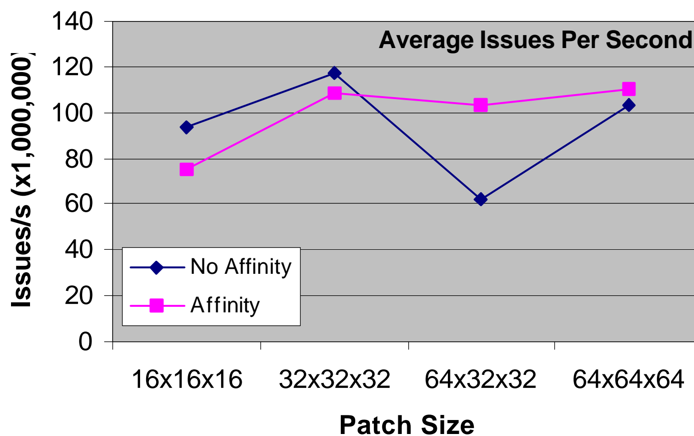

Figure 5. Average thread issues per second across the entire run of AMR.

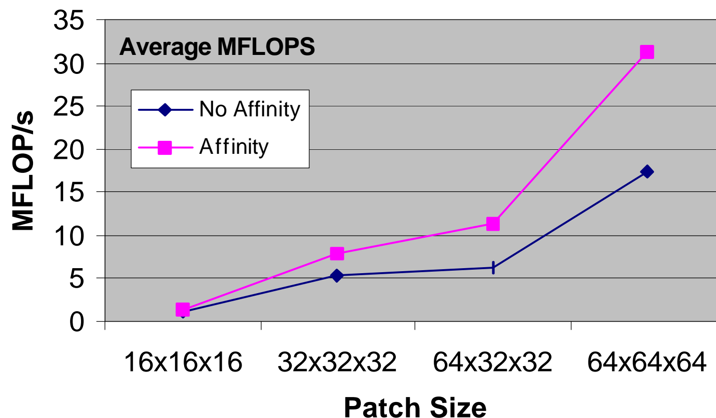

Figure 6. Average MFLOPS across the entire run of AMR.

Although the previous graphs showed that the execution time for the slower crew based threading was much longer, these graphs show that the CPUs were, on average, doing a similar amount of work per unit time. An explanation for this may be the high overhead of creating and maintaining crew threading.

4.4 Numbers of Threads

In addition to problem size scalability, the number of SPMD Processes that can be supported was also measured. This looks into the opposing effects of synchronization overhead against the greater number of processor resources and scheduling freedoms revealed with more coarse-grained parallelism.

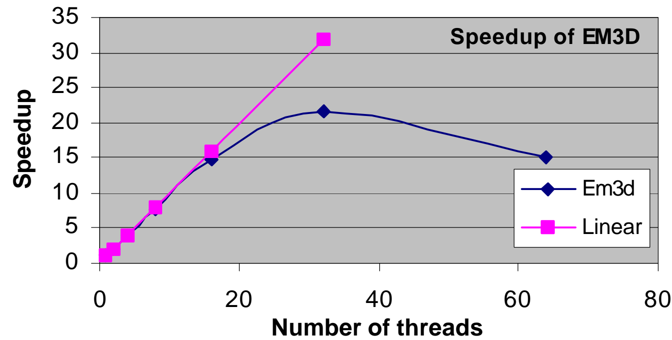

Figure 7. Thread speedup on the EM3D application. This was measured without affinity.

We ran the EM3D application on a single CPU with varying numbers of threads, to test the scalability of the system (Fig. 7). The results are as expected; our data shows speedup to around 32 threads, at which point the CPU becomes saturated and no more parallelism is exploited.

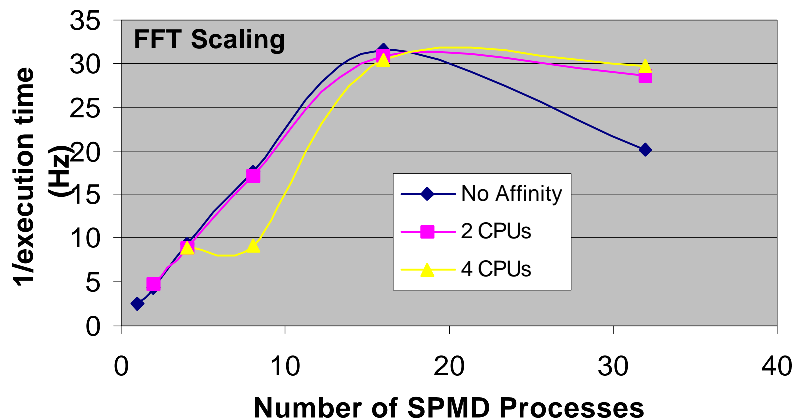

Figure 8. Scaling of FFT as the number of SPMD processes is increased. This was measured without affinity, and affinity using just 2 and 4 Tera CPUs.

The FFT test shows how FFT, which was not parallelized, scales with the number of SPMD processes. The 32-processor case shows a dip for the no affinity measurements. A possible explanation for this may be poor scheduling on the part of the runtime system, causing a bottleneck to form. The affinity cases (2 and 4 CPU) may be forcing the runtime to spread the threads out enough to avoid the bottleneck. Higher processor numbers may help to reveal the cause.

5 Related Work

Two of the more recent studies on the Tera have looked at standardized benchmarks and parallel kernels on the system. Snavely, Carter, et al. [11] reviewed the NAS kernel benchmarks on a two processor Tera system. They reported problems that limited parallel efficiency stemming from insufficient memory bandwidth due to a faulty memory network. Also, load imbalances within the streams may also have been a contributing factor. Brunett et al. [12] evaluated the C3I Parallel Benchmark Suite, comparing the Tera against a few other systems, such as the HP

Exemplar. They noticed slow sequential performance, many times slower than on competing systems, but more competitive parallel performance, which often outpaced the other machines. They also showed modest scaling speedups when running on both of the Tera's processors.

Another Tera study was done by Bokhari et al. [13] on unstructured adaptive meshes. Running on a 255MHz two-processor machine, they reported a peak rate of 210Mflop/s on a processor. Running a Fortran version of the EUL3D unstructured grid solver, they showed good scaling going from one processor to two as well as good stream efficiency.

6 Future Work

Currently, a few more Titanium applications are in line to be evaluated. They include matrix oriented kernels, parts of the Stanford SPLASH benchmark suite, another multigrid algorithm, and a particle simulation. These applications should present the Tera with different workloads and system demands.

Additionally, in the future, other scalable architectures, such as the Cray T3E and the IBM SP-2, will be evaluated and compared against the Tera. It would be interesting to see how systems that more closely map onto the SPMD computing model perform compare to a system that does not, such as the Tera. Also, this would help derive a model of the scalability behavior of the Tera, with a focus on its synchronization constructs and floating point performance. The Tera may also have a high sensitivity to the frequency of memory accesses, due to its lack of data caching. Measuring this sensitivity would help to determine if the memory subsystem is efficient enough.

Due to bugs in the runtime and compiler, some aspects of the Tera could not be adequately tested at this point. The runtime did not handle creating 128 or more Titanium processes, which, although may have high synchronization overhead, may be useful in programs with low synchronization requirements. Also, the EM3D application, although it contains a lot of loop-level parallelism, was not optimized because of a bug in the C compiler.

7 Conclusion

The SPMD programming model has been shown to be an effective tool in describing and thinking about various parallel programs in an understandable and efficient way. Even on a multithread-centric machine like the Tera, perhaps thinking about programming in terms of SPMD can be effective for some programs when tuning for performance. Allowing threads to be tied to specific processors helps to reduce inefficiencies in highly parallel constructs such as crews. The coarse grained SPMD parallelism also helps to expose additional levels of coarse-grained parallelism not detectable by the Tera compilers. AMR benefited from both intra- and inter-patch level parallelism that falls naturally from the mixing of the SPMD and thread based models. Tera frays are a simple, lightweight way of extending the SPMD model of programming, while keeping the control of the sharing of resources with the programmer. In addition, a high number of SPMD processes may still not be practical due to synchronization overheads. The EM3D case only scaled linearly up to 16 nodes and peaked at 32. So, having only SPMD style coarse-grained parallelism may not be enough to keep the Tera processors busy. Parallelizing optimizations from the C compiler are still necessary to mask the high memory latencies of the machine. Finally, there is a possibility that a tree barrier still performs better than a counter barrier on the Tera machine, though more data need to be taken to confirm this result.

References

[1] D.E. Culler et al., *Parallel Programming in Split-C*, Supercomputing 1993, Portland Oregon, November 1993.

[2] K. Yelick et al., *Titanium: A High-Performance Java Dialect*, ACM 1998 Workshop on Java for High Performance Network Computing, Stanford, CA, February 1998.

[3] Alverson, G. et al., *The Tera Computer System*, 1990 ACM International Conference on Supercomputing, Amsterdam, Netherlands, June 1990.

[4] Alverson, G. et al., *Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor*, 1992 ACM International Conference on Supercomputing, Washington, DC, July 1992.

[5] T.E. Anderson et al., *A Case for Networks of Workstations: NOW*, IEEE Micro, February 1995.

[6] G. Pike et al., *Parallel 3D Adaptive Mesh Refinement in Titanium*, Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, March 1999.

[7] B. So et al., *Measuring the Effectiveness of Automatic Parallelization in SUIF*, 1998 ACM International Conference on Supercomputing, July 1998, pp 212-219.

[8] S. Muchnick, *Advanced Compiler Design and Implementation*, ISBN 1-55860-320-4, © 1997 by Morgan Kaufmann Pulishers, Inc., pp. 641-656.

[9] R. Wilson et al., *Efficient Context-sensitive Pointer Analysis for C Programs*, ACM 1995 Programming Language Design and Implementation, June 1995, pp. 1-12.

[10] R. Hasti et al., *Using Static Single Assignment Form to Improve Flow-insensitive Pointer Analysis*, ACM 1998 Programming Language Design and Implementation, June 1998, pp. 97-105.

[11] A. Snavely et al., *Multi-processor Performance on the Tera MTA*, International Conference of High Performance Computing and Communications, Orlando, FL, November 1998.

[12] S. Brunett et al., *An Initial Evaluation of the Tera Multithreaded Architecture and Programming System Using the C3I Parallel Benchmark Suite*, International Conference of High Performance Computing and Communications, Orlando, FL, November 1998.

[13] S. Bokhari et al., *The Tera Multithreaded Architecture and Unstructured Meshes*, NASA/CR-1998-208953, ICASE Interim Report No. 33, December 1998.

[14] S. Woo, M. Ohara, et al., *The SPLASH-2 Program: Characterization and Methological Considerations*, Proceedings of the 22nd International Symposium on Computer Architecture, June 1995, pp. 24-36.