Microbenchmarking the Tera MTA

E. Jason Riedy
ejr@cs.berkeley.edu

Rich Vuduc
richie@cs.berkeley.edu

May 21, 1999

Abstract

The Tera Multithreaded Architecture, or MTA, addresses scalable shared memory system design with a different approach; it tolerates latency through providing fast access to multiple threads of execution. The MTA employs a number of radical design ideas: creation of hardware threads (*streams*) with frequent context switching; full-empty bits for each memory word; a flat memory hierarchy; and deep pipelines. Recent evaluations of the MTA have taken a top-down approach: port applications and application benchmarks, and compare the absolute performance with conventional systems. While useful, these studies do not reveal the effect of the Tera MTA's unique hardware features on an application. We present a bottom-up approach to the evaluation of the MTA via a suite of microbenchmarks to examine in detail the underlying hardware mechanisms and the cost of runtime system support for multithreading. In particular, we measure memory, network, and instruction latencies; memory bandwidth; the cost of low-level synchronization via full-empty bits; overhead for stream management; and the effects of software pipelining. These data should provide a foundation for performance modeling on the MTA. We also present results for list ranking on the MTA, an application which has traditionally been difficult to scale on conventional parallel systems.

1 Introduction

The Tera Multi-Threaded Architecture, or MTA, is a new supercomputer with hardware multithreading. It is designed to tolerate latency through switching between multiple threads of execution rather than to hide latency through a memory hierarchy. The Tera's designers hope both to provide a scalable, shared memory system, and to remove the burden of locality management from parallel programmers [2, 1].

The machine configuration at SDSC, the San Diego Supercomputing Center, has four 260 MHz Tera processors. There are four memory modules with one GB of memory each. The instruction pipeline is 21 cycles deep. Due to the experimental nature of the machine, reliable timings on multiple processors proved difficult to obtain. All results in this paper are for single-processor

benchmarks except where noted. Preliminary results on multiple processors suggest the results extend gracefully.

There have been a number of preliminary evaluations of the Tera MTA. Snavely, *et al.*, ported the serial versions of the NAS Parallel Benchmarks to the Tera [10]. They demonstrated that data parallel kernels can be easily ported to a Tera dual processor system (*i.e.*, hardware and compiler) with minimal modification. They were able to achieve execution rates on the Tera that were comparable to those on a 16 processor Cray T90. Brunett, *et al.*, recently ported the US Air Force C3I Parallel Benchmark Suite [3] to the MTA. They showed that an inherently sequential, compute-bound application performed poorly on the Tera, while a memory bound, data parallel application could achieve comparable or superior performance (faster absolute execution times) than a number of small scale HP, Pentium Pro, and Alpha-based SMP systems.

The thrust of these evaluations has been that data parallelism and many threads are sufficient for achieving high performance on the Tera. These conclusions, while true, are not surprising. Moreover, these studies are not of much use to computer architects because they do not expose to what degree particular design features of the underlying hardware – *e.g.*, the lack of caches, full-empty bits, fast hardware context switching – impact which aspects of program execution. Finally, they neither suggest how future applications can benefit from the Tera design, nor what new or existing programming models for high performance computing on such an architecture should look like.

In a third study, Snavely, *et al.* [11] ported a ray-tracing style application for medical imaging to the Tera, and compared it to a Cray T3E port. However, this article, which was targeted at a non-computer science audience, also does not assess the underlying hardware.

To obtain a better understanding of the unique and radical hardware features of the MTA, we construct a series of microbenchmarks that measure specific machine characteristics relevant to systems and application designers. The goal is to provide a set of basic parameters with which programmers can construct better performance models for their applications.

We also discuss implementations of two algorithms for parallel list ranking (or more generally, list scan) which have been traditionally difficult to parallelize on current systems.

2 Threads, Streams, and Teams

The primary abstraction for an application designer is a *thread*. It has the familiar high-level interpretation as the unit of work specified within a program. Threads are created and maintained through the Tera runtime system.

One or more threads execute in a *team*. Logically, a team corresponds to a CPU. An application may request or maintain one or more teams through the runtime system. Unless directed otherwise, the runtime can expand the number of teams associated with a process if it detects excess, unused parallelism.

The threads within a team execute on *streams*, which is a single instruction

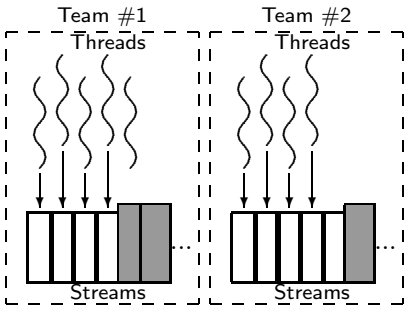


Figure 1: The basic threading abstractions in the Tera MTA include teams, threads, and streams.

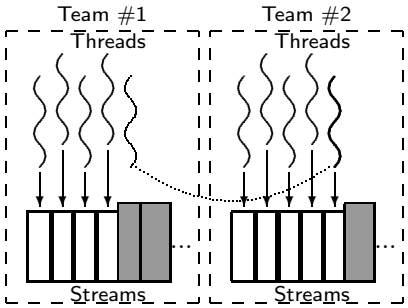


Figure 2: The runtime can transparently migrate threads between teams.

issuing slot on the CPU. Streams are created in the hardware, and each stream has its own complete register file. The CPU issues one instruction each cycle, picking an instruction word from a single ready stream (one not waiting for memory) in some unspecified order.

These terms are illustrated in Figure 1. We will measure the overheads incurred in creating switching between threads.

When there is excess parallelism (*i.e.*, a large number of unblocked threads are not mapped to streams) assigned to a team, the runtime system may migrate threads transparently among teams. Teams are mostly anonymous, and the overheads incurred by thread migration are not known and cannot be measured directly.

3 Microbenchmarks

To study the performance of different aspects of the Tera MTA, we created a series of microbenchmarks. First, we examine the memory and instruction latencies. We then time combinations of the hardware synchronization support.

A multi-threaded machine must have support for thread creation and scheduling, so we explore those mechanisms. The exceptional per-processor bandwidth of the Tera MTA is our final target.

3.1 Latencies

To interpret later timing results, we need an estimate of how many instruction issues or memory operations can occur in a given time. The Tera MTA tolerates latency rather than hiding it, so the individual memory latencies will be quite large. Given the number active streams, the per-stream instruction word latency will also seem large.

3.1.1 Memory Latency

Memory and processor nodes are separate in the Tera. They are arranged in a toroidal 3-D network. An interesting aspect of the memory system is the use of *hot-spot caches*. Each memory module has a small cache which is used to provide faster access to highly contended variables, *e.g.*, when multiple threads are reading a shared flag.

We ran the memory read latency test of the lmbench [7] microbenchmark package to obtain the average round-trip network latency for small transfers. The basic benchmark maintains an array of addresses. The array is initialized so that when an array element is loaded, the value returned is the address of the next element to be read. The inner loop is unrolled a thousand times to produce a sequence of exactly 1000 load instructions. We disabled all parallel compiler optimizations.

Figure 3 shows the results of running the benchmark on various array sizes using various strides. Our findings can be summarized as follows:

- When streaming through an array with a stride that is much less than the array length, we see a load latency of approximately 170 cycles. This is independent of array sizes, as expected.
- When the stride equals the memory size (*i.e.*, we are reading the same address over and over again, we observe a minimum in the load latency), we see a minimum in the execution time. The difference (approximately 60 cycles) demonstrates the presence of hot-spot caches. The hardware manuals suggest that the access time from the hot-spot cache is two cycles, as opposed to 60-70 cycles from main memory [12].
- The drop-off in load latency begins when there are 256 or fewer different accesses within the array. This suggests that the aggregate hot-spot cache holds 256 words.
- The minimum latency roughly represents the latency through the network, assuming the hot-spot caches respond within a negligible number of cycles.

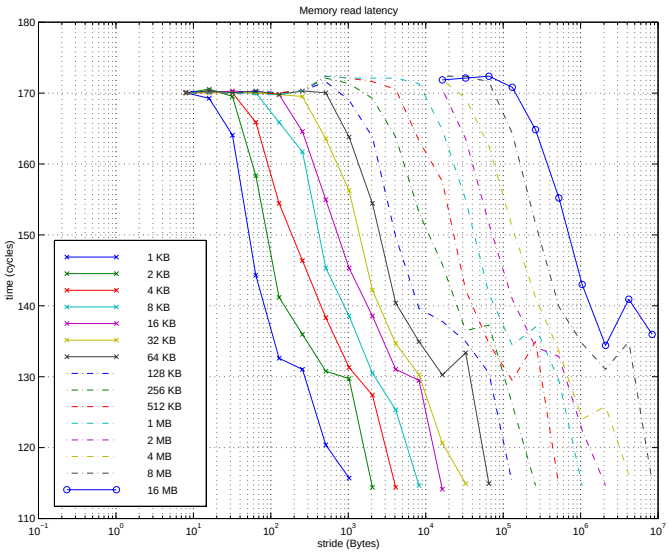


Figure 3: Memory read latency is around 170 cycles. Hot-spot access is around 115 cycles.

The slight bump for array sizes over 128 KB is due to having made measurements for the larger array sizes at a time when the machine had a slightly higher load.

3.1.2 Instruction Latency

The Tera has a VLIW design; each instruction word consists of three RISC-style instructions [13]. The three instruction word pieces correspond to memory, arithmetic, and control instructions, although many common arithmetic operations can be placed in both the arithmetic and control positions [12]. The Tera C compiler packs instructions very well within loops and seems to average two instructions per word in routine, book-keeping sections of code.

The processor pipeline is 21 stages deep, so each instruction word has a minimum latency of 21 instructions [13]. However, each stream can only issue one instruction word per cycle. With $k > 21$ streams, a purely round-robin scheduling order would give each stream's instruction words a latency of k cycles. In reality, some streams will wait on memory or experience exceptions, so the instruction word latency is not entirely deterministic.

To get a feeling for the instruction word latency, we created a simple program that creates a number of computation threads, locks those threads to streams, and measures the time to complete 8192 TERA_CLOCK instructions. We then divide by the number of instructions to determine the clocks per instruction. The TERA_CLOCK operation is allowed only in the control section of the instruction word, and so only one is issued per word. Because the runtime system

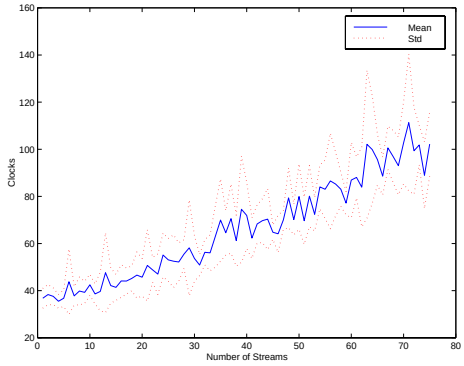


Figure 4: The clocks per instruction varies greatly and grows with the number of streams.

uses a few streams (less than ten) internally, the total number of streams active on the processor is slightly higher than the number we create.

Figure 4 shows the per-instruction execution time as a function of the number of computation streams. The time shown is the average over five runs with a given number of streams on a quiescent machine. The dots trace one standard deviation from the mean and show a large variation from run to run. A good rule of thumb to estimating a stream's instruction word latency is to add 20 to the estimated number of active streams. Even with a full 128 streams, the individual instruction word latency is not enough to completely hide memory latency.

3.2 Synchronization

The Tera MTA provides hardware support for synchronization through full-empty bits on every word in memory. Figure 5 shows the contents on each word in memory. The full-empty bits literally denote whether the word should be considered to be holding useful data (full) or waiting for useful data (empty). The forwarding bit instructs the CPU to retry the memory fetch with the pointer given in the data portion of the word. One of the two trap bits is used for the full-empty system, and the other is currently used to 'poison' unallocated words and help programmers find pointer bugs quickly. The Tera provides byte and half-word addressing, but the full-empty, trap, and forwarding bits apply only to whole words.

On a data access, the address is interpreted as a *pointer* with the structure shown in Figure 6. While 47 bits are available for memory addresses, only 42 are currently used. There is a four-way associative, 512 entry TLB on the MTA. Segment sizes can vary from 8 KB to 256 MB; access privilege level is maintained per segment.

The compiler provides read and write primitives to empty and fill words

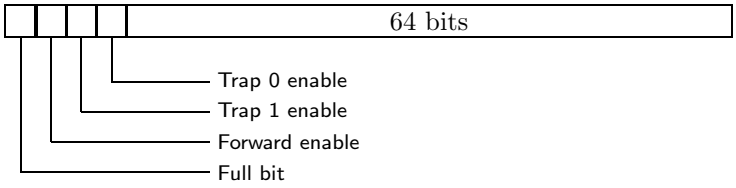


Figure 5: Each memory word holds data and access qualifiers.

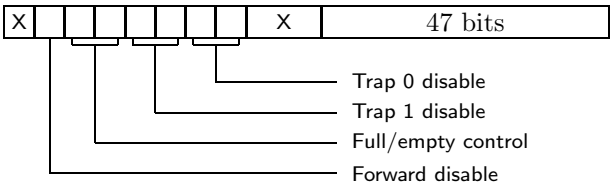


Figure 6: Pointers hold both a destination address and access control information.

in all sensible combinations. When a full-empty abiding read on an empty word occurs, the CPU retries the access some number of times and then calls a trap handler. The trap handler then builds a continuation structure, copies the word's value into that structure, and sets the word to trap on all future full-empty accesses and forward other accesses to the saved value. All future full-empty reads will immediately trap on access to this word. Trapped threads have their streams remapped and are placed in a waiting queue. If no stream has trapped on a given empty word, a filling write to that word is a simple write. A write to a trapped location will transfer the value and re-start as many threads as possible.

Given the selectable hardware counters available in the Tera, the number of retries is not difficult to determine. Figure 7 shows a piece of the measurement code and gives the general flavor of the Tera C extensions and runtime calls. The measuring program prints 1023, implying a 10-bit hardware retry counter starting. We will use 1024 as the number of memory accesses before a thread traps out. Note that the first access will see the full 170-cycle latency, while the remaining 1023 could see the reduced, 115-cycle latency. For later estimations, we will assume multiple accesses to the same location experience a 115-cycle latency.

3.2.1 Full-Full v. Full-Empty

One problem that frequently occurs with heavyweight synchronization primitives is known as the thundering herd problem. If dozens or hundreds of threads are waiting at one synchronization point, say a condition variable, waking all threads immediately can overload the software thread scheduler and cause horrible performance. In software, this is avoided by waking only a few of the

```
sync int retry_cnt[2];

void retry_inner (void* v) {
  /* Wait to be triggered.  Must trap out. */
  readff (retry_cnt);
  /* Write counter. */
  writeef (retry_cnt+1,
           terart_get_task_counter(RT_MEM_RETRY));
}

int count_retries (void) {
  sync int retry_cnt[2];
  int inner_cnt, outer_cnt;

  /* Set the f-e bits to empty. */
  purge (retry_cnt); purge (retry_cnt + 1);
  /* Begin counting retry events. */
  terart_reserve_task_event_counter (RT_ANY_COUNTER,
                                     RT_MEM_RETRY);
  /* Create the inner stream. */
  terart_create_stream (NULL, retry_inner, NULL);
  /* Spin until the stream has blocked. */
  spinout ();
  /* Trigger the inner stream. */
  writeef (retry_cnt,
           terart_get_task_counter(RT_MEM_RETRY));
  /* Spin until the inner stream has completed. */
  spinout ();
  /* Retrieve counts, will block if empty. */
  return retry_cnt[1] - retry_cnt[0];
}
```

Figure 7: The routines for measuring the number of retries gives the flavor of the Tera C extensions.

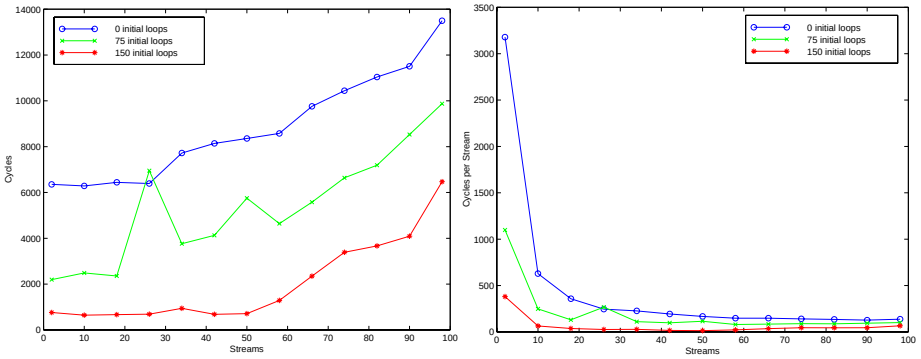


Figure 8: The total time to trigger a group of streams with a single write *decreases* as they trap out. The time per stream converges quickly as the number of streams grows, however.

necessary threads at a time. It is natural to ask if the Tera thread scheduler experiences similar problems. Because the CPU stream scheduler only issues one instruction per cycle, any problems must come from higher levels.

Many threads waiting on a single, empty memory location provides an appropriate test. If the threads all use the readff primitive, then each will leave the word in the full state. All threads can be continued immediately, given enough streams. To test this, we guarantee enough streams for all threads to immediately resume by initially starting a stream per thread and disabling the runtime's auto-retire facility. The main timing microbenchmark thread

1. starts all the streams,
2. spins in an initial, single-operation loop,
3. starts timing,
4. fills the sync variable,
5. spins until all streams have completed, and
6. records the final time.

Each thread simply performs a readff on the sync variable followed by an atomic addition to inform the timing thread of its completion. The main thread is locked to a stream. The benchmark was run only on a single team due to difficulties in obtaining reliable multi-team results.

Figure 8 shows both the total number of clocks until completion and the number of clocks per stream for different numbers of initial loops. The test was run only once, so the odd spikes are simply noise from other processes. The overhead in the times is a single atomic addition. Polling in the CPU is not the most efficient mechanism for triggering waiting streams in this artificial

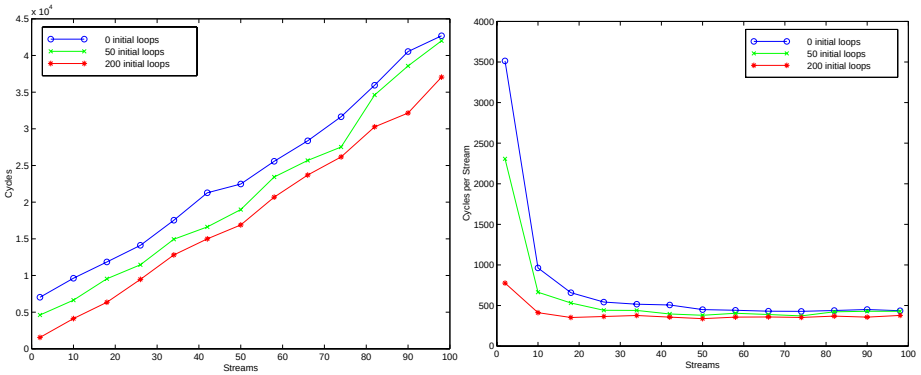


Figure 9: The trap handler begins threads triggered by a full-empty transition more efficiently, as well.

environment. The polling streams must wait for a memory latency after the write to see the state change. The trap handler can re-start all the threads that will leave the variable full at once. The hardware scheduler's simplicity in executing a single instruction from a ready stream every cycle allows the runtime's remapping system to take the simple route of enabling every thread possible without penalty. Polling also adds a small amount of traffic to the interconnect. Whether the amount is significant remains to be seen.

At 50 threads and 150 initial loops, the total execution time per stream is impossibly low. At around 14 cycles per stream, this is not consistent with either the latency in the atomic addition or the latency in restoring the thread's state to the stream. We do not have an explanation for the missing atomic addition, but the thread's state may be lazily stored to memory. If blocked threads tend to be re-enabled before another thread needs a stream, this would be a good optimization.

Figure 9 shows results from a similar test. The difference is that each thread empties the memory word and then fills it, allowing only one thread to execute at a time. The time grows linearly with the number of threads, implying that the trap handler only enables a single thread. The best time again occurs when all the threads have trapped. The best time per thread is also close to the three memory accesses per thread (readfe, writeef, and int_fetch_add). If all threads were enabled and immediately mapped to streams, they would have to trap out again, and the time would be larger. So the trap handler only allows a single emptying read to occur, and it leaves the word set to trap.

When a stream traps on a memory access, the trap handler receives the pointer used to access the word. As shown in Figure 6, the pointer contains the full-empty state its access will transfer to the memory word. The trap handler must keep two lists of threads, one of threads that will leave the word full and one of threads that will empty the word. When a filling write appears, all the threads in the leaving-full queue can be mapped to streams followed by one from

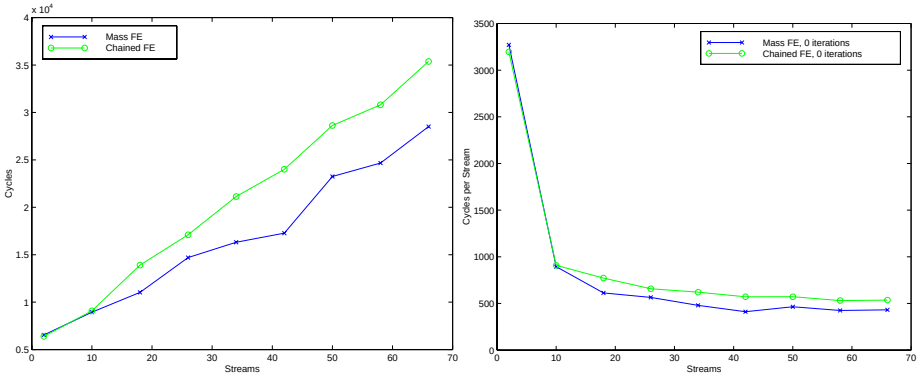


Figure 10: The hot-spot cache provides a slight advantage to synchronizing on the same location in a computation-less test.

the emptying queue.

3.2.2 Memory Bottlenecks and Synchronization

Synchronization variables typically need to have some non-local arrangement to avoid ping-pong cache interactions in standard SMPs. Because the Tera MTA does not use CPU-local caching, this odd programming requirement should not apply.

Placing threads in a synchronization chain tests one simple variant. Each thread in the chain waits to empty one variable and then fills another to continue the chain. Either each thread waits on and fills the same location (as in the full-empty version of the previous benchmark), or thread i waits on some synchronization variable array at index i and fills index $i + 1$. Each of the following tests was performed with no initial loop, so no threads have trapped on synchronization variables initially.

The hot-spot cache actually gives a bit of an advantage to threads triggered from the same location, shown in Figure 10. Each thread in this test immediately begins the next, then performs an atomic add to notify the timing thread of its completion. The per-thread results are appropriate for two memory latencies with some retries. No streams encountered synchronization traps.

When each thread runs a computation loop before triggering the next thread, the difference between multiple locations and single locations disappears. Synchronization traps cause the inflection point in the per-thread times. With a single-instruction loop of 200 iterations, streams retry beyond the trap limit after the tenth. As expected, half the iterations require twice as many streams. For 200 iterations, the time per stream stabilizes around 50 streams. With more data points, the time probably goes down given the trap handler's efficiency. The variable instruction latency per stream and length of the retry cycle makes determining the number of retries difficult from this data.

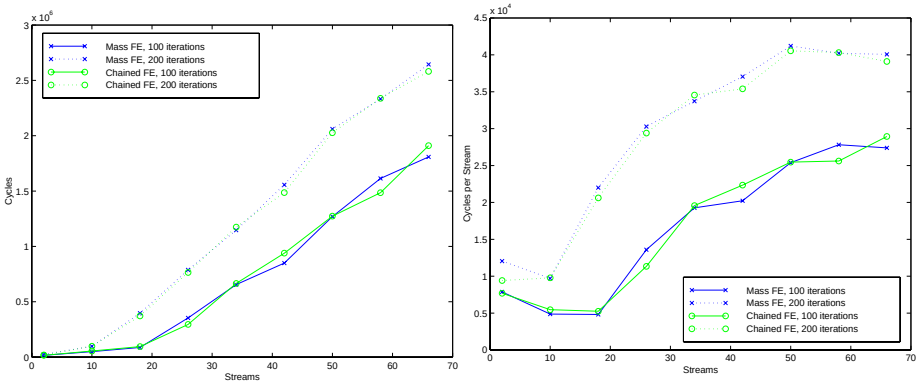


Figure 11: With a computational load, there is no apparent difference between waiting on a single location or chaining synchronization across multiple locations.

3.3 Threads and Streams

The difference between threads and streams, described in Section 2, is subtle. The programmer should deal primarily with threads and let the runtime grow or shrink the number of streams according to the apparent parallelism. Performance modeling requires a bit more control over the allocation of streams and their mapping to threads. The Tera runtime library provides this control, although its use is discouraged. The runtime procedures to create threads and the synchronization trap handler's remapping function provide measurements useful for evaluating the cost of giving control to the runtime.

3.3.1 Runtime Thread Creation

The Tera runtime libraries provide a simple stream- and thread-creation mechanism. The syntax is vaguely similar to POSIX thread creation routines but provides additional flexibility in assigning threads to streams and teams. Most users need not worry about thread creation; the parallelizing compiler creates streams efficiently. The first stream in a crew (a team-migratable group of threads) is created through the runtime, but subsequent streams in the frays (streams that cannot migrate, used for loop-level parallelism) are created through the appropriate hardware instruction. The time to create a stream in hardware should be on the order of an instruction latency, but lack of a working assembler at SDSC prevented our measuring it. Table 1 gives the times to create a stream and a mapped thread through the runtime. This time is exceedingly fast compared to typical SMP thread libraries, but also exceedingly slow compared to the Tera hardware instruction.

	Clocks	μseconds
Mean	2.72×10^4	1.04×10^{-1}
Std	9.72×10^2	3.74×10^{-2}

Table 1: Starting a stream through the runtime takes little time. These measurements include one memory write overhead of around 2×10^2 clocks.

```
void remap_inner (void* v) {
  writeef ((sync unsigned*) v, TERA_CLOCK (0));
}

unsigned time_remap (void) {
  unsigned initial, final;
  sync unsigned arg;
  purge(&arg);
  /* Do not allow the runtime to add streams. */
  terart_disable_growth ();
  terart_create_thread_on_team (NULL, inner_thread,
                                (void*)&arg);
  spinout ();
  initial = TERA_CLOCK (0);
  final = readfe (&arg);
  return final - initial;
}
```

Figure 12: Timing remapping through a trap is relatively simple. However, the runtime has the option of adding additional, unrequested streams unless directed otherwise.

3.3.2 Stream Remapping

When a thread currently mapped to a stream traps or enters a potentially high latency OS call, it can have its stream usurped by a ready, unmapped thread. The code necessary to time the remapping is shown in Figure 12. Table 2 shows the time required to unmap the current thread and map a new one to the available stream. Adjusting the mean time to remove a average 115-cycle latency for the 1024 retries reduces it substantially to 5.39×10^4 clocks. If other threads have already caused a trap on the synchronization variable, the remapping time would be the smaller, adjusted one.

When a thread is forcibly remapped, its stream's state must be saved, and the new thread's saved state restored. Table 3 shows a general accounting of obvious external costs in the remapping process. The 70 words of state is a conservative upper bound. Each stream has 32 registers, plus condition registers

	Clocks	μseconds
Mean	1.72×10^5	6.60×10^{-1}
Std	1.55×10^4	5.95×10^{-2}
Adj. Mean	5.39×10^4	2.07×10^{-1}

Table 2: After adjusting for the memory retries, the remapping cost is on the order of the cost to create a new stream through the runtime.

Category		Clocks
Total		172 000
Retries	- 1024 accesses $\times$ 115 cycles / access =	53 900
State Write	- 70 words $\times$ 170 cycles / word =	42 000
State Read	- 70 words $\times$ 170 cycles / word =	30 100
Insn words	$\div >$ 30 cycles / instruction =	<1 010

Table 3: With loose bounds and to three significant digits, stream remapping uses no more than 1010 instruction words in the stream.

and a program counter. The runtime stores additional information about the thread as well. The loosely estimated number of instruction words is reasonably close to the claimed few hundred instructions [13]. Remapping is expensive, but not prohibitively so.

When the user function invoked in a new, runtime-allocated thread ends, its stream should be remapped to another waiting thread. We attempted to time this and ran across a likely bug in the runtime system. The waiting thread was not mapped onto the available stream. The time required by a functional runtime to start a new thread on an available stream should be on the order of the time given for remapping through the trap handler, if not smaller.

3.4 Bandwidth

By using thread parallelism to tolerate memory latencies rather than memory hierarchies to hide it, the Tera MTA achieves an excellent bandwidth for a relatively slow processor. The standard STREAM benchmark provides bandwidth numbers possible when letting the compiler control the parallelism. We then investigate the response of bandwidth with respect to the number of memory loads in flight, the run-length between load groups, and the number of streams.

3.4.1 The STREAM Bandwidth Benchmark

We ported and ran the STREAM memory bandwidth microbenchmark [6] on the Tera MTA. The benchmark consists of a series of simple data parallel vector operations such as copy, add, and scale. We allowed the compiler to perform

Teams	Bandwidth (GB/s)
1	1.7
4	5.8
Speedup	3.4

Table 4: The STREAM-measured bandwidth is quite large for a single CPU and scales to multiple CPUs well.

```
for i = 1 to n,                          for i = 1 to n step 2,
   read A[i]                                read A[i]
   loop                          —          read A[i+1]
      operate on A[i]            —          loop
                                                operate on A[i]
                                                operate on A[i+1]
```

Figure 13: One step of basic software pipelining unrolls a loop once and gathers instructions to minimize latency.

automatic parallelization of this code (*i.e.*, automatic stream creation and management), and restricted only the number of teams executing simultaneously to one team and four teams. Table 4 shows the results. We see that the bandwidth delivered to a single processor is very large at 1.7 GB/s, and that the multiprocessor speedup in aggregate bandwidth at four processors is very reasonable at 3.4.

3.4.2 Software Pipelining Effects

Each stream on the Tera can have up to eight outstanding memory requests. The requests are temporarily stored in memory request registers in the stream's register file. Each instruction word has a three-bit lookahead field specifying the number of words until its results are needed. Only memory instructions use this field. A sequence of eight memory requests with appropriate lookaheads result in having eight memory requests outstanding. To measure the effects of having multiple outstanding memory requests, we timed a read loop after applying different amounts of software pipelining. Software pipelining combines loop unrolling and instruction scheduling, rearranging instructions within an unrolled loop to reduce latency as shown in Figure 13.

With an unrolling depth of two, each iteration of the read loop can have two reads outstanding. A depth of eight results in eight outstanding loads. We have checked the disassembled version of our read loop and verified that indeed n reads are outstanding at a unrolling depth of n. However, we also include an operation loop after the loads. The only way to prevent the compiler from optimizing a very simple loop away without including more memory references is to include floating point instructions and require the compiler not to optimize

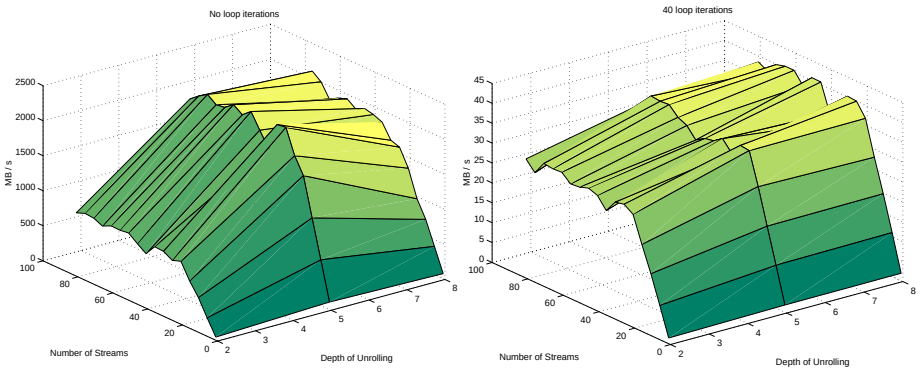


Figure 14: The processor's saturation point varies with the achievable load depth and frequency of loads, but 30 streams gives reliably good results.

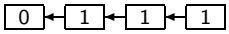
them. This also prevents the lookaheads from progressing into the loop, so all reads are completed by the time the loop begins.

Figure 14 gives the bandwidth achieved for copying 10 MB by threads and unrolling depth with no and 40 iterations of the simple loop. The maximum bandwidth is clearly achieved by the loops with no work iterations. The maximum bandwidth is achieved with 26 threads and a unrolling depth of five, reading over 2.3 GB/s. For a depth of eight, the maximum bandwidth occurs with 76 streams. However, only 30 streams are necessary at depths of five or eight and for both tested work iterations to achieve within 5% of the maximum.

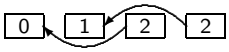
4 A Parallel Challenge: List-Ranking

Given a linked-list L, we compute a *list ranking* of L by computing the distance of each node in L from the head (or tail). Parallel list ranking (and more generally, list scanning) algorithms have been traditionally difficult to implement efficiently. First, the data structure is highly irregular, leading to irregular, high-volume communication patterns. Furthermore, it is difficult to achieve good load balance. Finally, the serial algorithm is so straightforward that the constant work factors are extremely low (only two or three load/stores required per element). Many proposed parallel algorithms do not succeed in practice because the high constants make them uncompetitive with the serial algorithm.

In this section, we review two parallel algorithms: Wyllie's algorithm [4] and the Reid-Miller algorithm [8]. We then discuss implementations on both an Ultra Enterprise 5000 and the Tera MTA. Due to technical difficulties, we could not complete an evaluation of the Tera algorithms, but we do present partially available data and draw preliminary conclusions.



(a) First Iteration



(b) Second Iteration

(c) Third Iteration

Figure 15: Wyllie's algorithm. (a) Initial state. (b) Each node accumulates its neighbor's value and changes its successor to point to its neighbor's successor. (c) The algorithm repeats until no links are left.

4.1 Wyllie's Algorithm

Figure 15 illustrates Wyllie's algorithm. Suppose each node in the linked list has been assigned to a unique processor. Initially, each node has a value of 1, except for the tail who is assigned a value of 0, as in Figure 15 (a). Then, each processor synchronously reads the value of the next node and accumulates that value into its own value. Each processor also adjusts its next pointer so that it points to its successor's successor (Figure 15 (b). This process repeats until no nodes have any remaining successors (Figure 15 (c)).

Although this algorithm is simple to describe and implement, it is not work-optimal, *i.e.*, it requires asymptotically more work $O(n \log n)$ than the serial algorithm $O(n)$. Futhermore, the algorithm as described requires explicit synchronization at each node to ensure that each node reads both the value and the next pointer of its successor before that successor changes itself.

There is one algorithmic improvement which we can make to Wyllie's algorithm to alleviate the need to do fine-grained synchronization that is essentially a bulk-synchronous approach. At a cost of doubling the memory requirements, we can maintain two lists. We use one list to do all reads, and perform updates to the second list. When all processors have updated their nodes during one round, we then swap the role of the two lists.

4.2 The Reid-Miller Algorithm

One problem with list ranking in parallel is contention at the nodes. The Reid-Miller approach alleviates this contention by randomly breaking up the list of n

elements into m sublists that can be processed independently. The list ranking then proceeds in three phases as shown in Figure 16:

1. The list begins in the same initial state as Wyllie's algorithm. (Figure 16 (a))
2. Randomly divide the list into m sublists, and compute the sum of the sublist nodes independently. (Figure 16 (b))
3. Find the list scan of this sublist. (Figure 16 (c))
4. Fill in the scan of the original list using the reduced list results. (Figure 16 (d))

The first and third phases can be performed in parallel. The second can be implemented as any list scan, *e.g.*, a serial scan, Wyllie's algorithm, or a recursive call. One problem with the approach is that randomly dividing the sublist often leads to imbalanced sublists. The original authors periodically rebalance the list during the first and third phases by partially computing the sums of the sublists and packing out sublists that have completed. Reid-Miller demonstrated that the final algorithm achieves high performance and good scalability on the Cray C90, but requires a great deal of tuning to get the "right" implementation. Additional details can be found in [8]. Strictly speaking, the algorithm is slightly worse than work-optimal, but in practice is currently the fastest known algorithm.

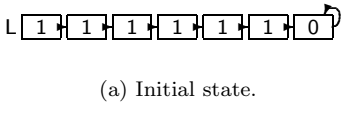
While the Reid-Miller algorithm is essentially the current state-of-the-art for the list ranking problem, one notable modification by Helman and JaJa [5] reduces the required number of memory accesses in half. While the original Reid-Miller was targeted for vector multiprocessors, the Helman-JaJa algorithm is tuned for SMPs. In our experiments, we focus on the original algorithm only.

4.3 SMP Implementations

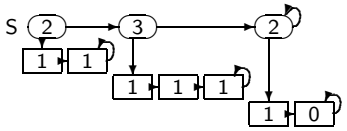
We first implemented the serial, Wyllie, and Reid-Miller algorithms on a eight-way Sun Enterprise 5000 system based on the 167 MHz Ultra1 processor. We use POSIX threads for the parallel algorithms. The execution times per list element as the list length varies is shown in Figure 17.

The best Wyllie implementation uses the two-list technique described above. This requires a barrier operation between phases, when we swap the read and write lists. Even after extensive tuning, Wyllie's algorithm could not beat the serial algorithm for any array input size. Furthermore, we can see the $\log n$ asymptotic factor as the list sizes increase.

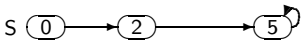
We achieve much better performance and scaling results for the Reid-Miller algorithm, eventually outperforming the serial algorithm for lists of length greater than $O(10^4)$. We performed an extensive search of the tuning parameter space to get the results shown. The increase for a list of length 16 million elements is due to TLB misses.



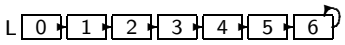
(a) Initial state.



(b) Phase 1: Randomly divide into sublists, and compute the sublist sums.



(c) Phase 2: List scan on the sublist list.



(d) Phase 3: Expand the sublist scan to the full list.

Figure 16: The Reid-Miller algorithm.

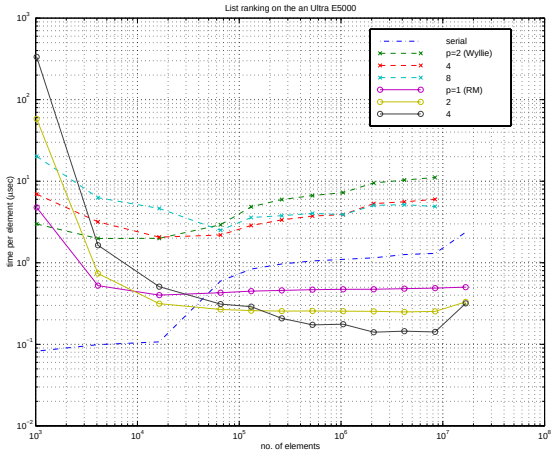


Figure 17: Best list ranking performance on the Sun Enterprise 5000.

4.4 Tera MTA Implementations

Tera's automatically parallelizing compiler uses low-level hardware instructions to create and manage streams in each team. This bypasses much of the runtime overhead and produces better results. One version of this benchmark uses the compiler's automatic parallelization. Determining exactly the right idioms and pragmas to use can be problematic, so we also examine a version that explicitly creates and manages threads through the runtime.

4.4.1 Automatically parallelized code

We first consider direct ports of the single-threaded versions of the parallel algorithms (*i.e.*, data parallel versions that do not explicit create or manage threads), and allow the compiler to do automatic, single-team parallelization, possibly with tweaking. For instance, in the case of Wyllie's algorithm we add fine-grained synchronization (*e.g.*, use full-empty bits in Wyllie's algorithm to serialize access to a list element) and insert a compiler pragma to force parallelization of the inner for loop. The resulting execution times per element are shown in Figure 18. The compiler requested 40 streams for the parallel regions in the Wyllie algorithm, and 120 streams in the Reid-Miller algorithm. Forcing the compiler to use other values did not yield faster execution.

Qualitatively, this duplicates the relative performance trends of the three algorithms that we saw in the SMP case. As expected, the serial execution time per element is independent of the list length. The actual time (2.1 μs) is almost exactly three memory references ($\approx$ 700 ns), which accounts for the three references in a list scan. Interestingly, the absolute execution times are also very similar between the two platforms. While it is not entirely fair to do a direct comparison, this suggests that there is nothing magical about Tera that would yield much higher performance on a traditionally difficult parallel problem such as list ranking. However, it also suggests that it is possible to obtain performance without much effort if the code can be written in a particular compiler-friendly style (e.g., data parallel fashion). For example, it is much easier to write down the sequential, data parallel version of Reid-Miller than the explicit POSIX threads version, yet the two versions performed somewhat comparably.

4.4.2 Explicitly parallel Wyllie

Next, we consider an explicitly multithreaded version of the Wyllie algorithm based on the two-list approach. In all our explicitly parallel algorithm implementations, we use the runtime system to explicitly allocate streams and execute directly on the streams, *i.e.*, do not go through the runtime system to create non-mapped threads. We also implemented our own counter-based barrier using the Tera int_fetch_add instruction and a toggling block-and-release mechanism. As shown in Figure 19, we see that with a sufficient number of streams, Wyllie's algorithm can outperform the serial algorithm, albeit by a modest factor. Recall that execution is only on a single team (processor).

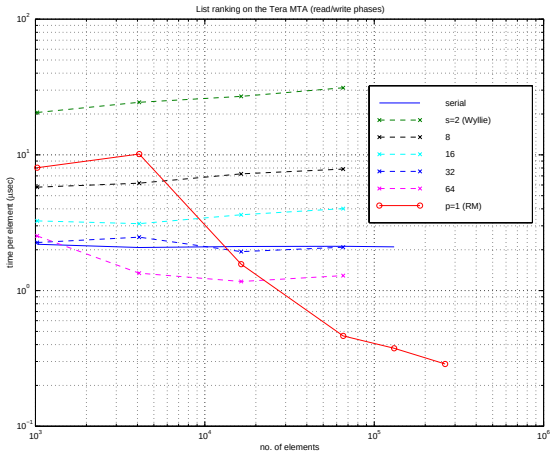


Figure 18: Automatically parallelized single-team list ranking implementations on the Tera MTA.

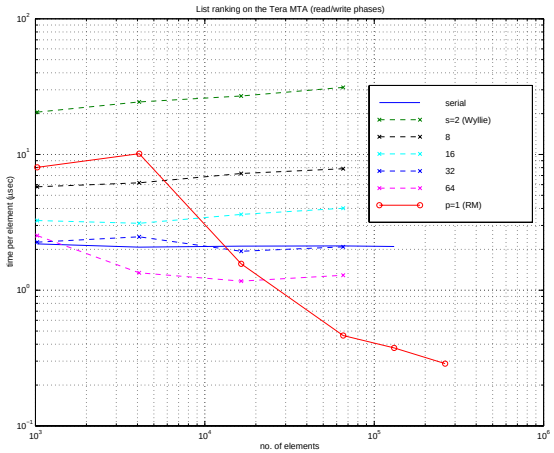


Figure 19: Two-list, explicitly parallel implementation of Wyllie's algorithm. Serial, and auto-parallel Reid-Miller versions are shown for reference.

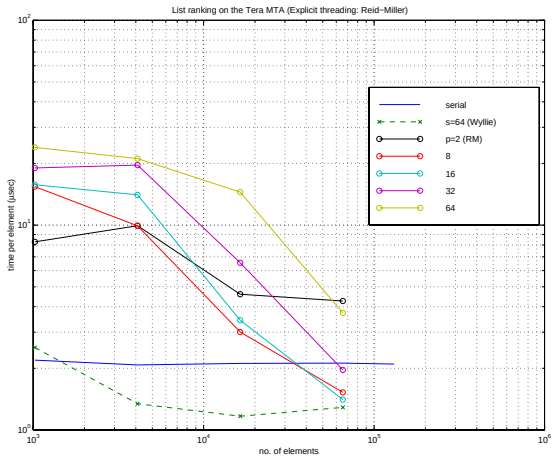


Figure 20: An explicitly parallel version of the Reid-Miller algorithm. Serial, and the best two-list Wyllie implementation are shown for reference.

4.4.3 Explicitly parallel Reid-Miller

For completeness, we present some preliminary data for an explicitly multi-threaded version of the Reid-Miller algorithm (see Figure 20). The algorithm was not tuned as extensively as the SMP version. The trend indicates that it eventually outperforms the serial algorithm, and possibly the Wyllie algorithm as well. It also suggests that the 'ultimate' list ranking algorithm will use all three in some form.

5 Conclusions

The Tera MTA effectively tolerates its large memory latency and varying instruction word latencies. Data locality does not matter in a single-processor setting. Incomplete timings suggest this extends to multi-processor jobs as expected. The programmer need not be concerned with data non-locality in synchronization, either. The Tera pseudo-randomly distributes successive user-level addresses evenly across memory modules, and the hot-spot cache is tiny, so trying to use that for speeding memory accesses would be ill-advised. With around 30 streams, it is not necessary anyways. With an available machine, it should be possible to evaluate analytical models of multithreaded machines like the one proposed in [9].

The number of retries before trapping appears too large in microbenchmarks. The effects of long polling intervals will vary by application and should be monitored by checking the Tera's retry counter. As the number of streams increases to the amount needed to achieve maximum bandwidth, however, the difference between triggering polling and trapped threads becomes small.

Any synchronization libraries should implement the primitives in obvious

ways, and they will have good performance. One major advantage to the full-empty bits is that synchronization can be piggy-backed onto normal data access, occasionally removing the need for separate mutexes and condition variables altogether. This overlaps the synchronization overhead with useful memory accesses and should be used whenever appropriate.

Unfortunately, this data on list ranking is thus far inconclusive. The problems inherent in early versions of hardware and software restrained the collection of reliable data for larger problem sizes. The more highly tuned implementations of the Reid-Miller algorithm was effected particularly strongly, which is unfortunate given the SMP results. What we have demonstrated is that with modest effort, we can extract the same levels of performance for the list ranking problem on a single, multithreaded processor that we could on a conventional SMP system. We did not see a huge jump in total performance, but we are also comparing a single processor to multiple processors. We also demonstrated that a combination of algorithmic changes and use of built-in hardware synchronization primitives for Wyllie's algorithm did lead to better relative performance on the Tera than on the conventional SMP.

One interesting microbenchmark not attempted would measure the memory interconnect impact of various synchronization styles. Each stream on each CPU can be polling an empty variable, and this will use interconnect bandwidth. The interplay between the maximum number of retries and the bandwidth impact is interesting and deserves future study.

6 Acknowledgments

We would like to thank the people at Tera Computer Corporation for their assistance, especially Gail Alverson. We would also like to thank the operator at SDSC who turned the Tera back on after their unannounced, site-wide shut down.

References

[1] Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, and Rich Korry. Tera hardware-software cooperation. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997* ACM/ IEEE SC97 *Conference: November 15–21, 1997, San Jose, California*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, 1997. ACM Press and IEEE Computer Society Press.

[2] Gail A. Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton J. Smith. Exploiting heterogeneous parallelism on a multi-threaded multiprocessor. In *6th* ACM *International Conference on Supercomputing*, pages 188–197, Washington, DC, July 1992.

[3] Sharon M. Brunett. An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite. In Acm, editor, Sc *'98: High Performance Networking and Computing: Proceedings of the 1998* Acm/Ieee Sc98 *Conference: Orange County Convention Center, Orlando, Florida, November 7–13, 1998*, New York, Ny 10036, and 1109 Spring Street, Suite 300, Silver Spring, Md 20910, 1998. Acm Press and Ieee Computer Society Press.

[4] T. Cormen, C. Leisersen, and A. Rivest. *Introduction to Algorithms*. Mit Press, 1990.

[5] David R. Helman and Joseph JaJa. Prefix computations on symmetric multiprocessors. Technical Report CS-TR-3915, University of Maryland, College Park, July 1998.

[6] J. D. McCalpin. The Stream memory bandwidth benchmark. `http://www.cs.virginia.edu/stream`.

[7] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In Usenix, editor, Usenix *1996 Annual Technical Conference, January 22–26, 1996. San Diego,* Ca, pages 279–294, Berkeley, Ca, January 1996. Usenix.

[8] Margaret Reid-Miller and Guy E. Blelloch. List ranking and list scan on the Cray c90. Technical Report CS-94-101, Carnegie Mellon University, School of Computer Science, February 1994.

[9] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten Von Eiken. Analysis of multithreaded architectures for parallel computing. Report UCB/CSD 90/569, University of California, Berkeley, Computer Science Division, Berkeley, CA, USA, April 1990. To appear in the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Crete, Greece, July 1990.

[10] A. Snavely. Multi-processor performance on the Tera Mta. In Acm, editor, Sc *'98: High Performance Networking and Computing: Proceedings of the 1998* Acm/Ieee Sc98 *Conference: Orange County Convention Center, Orlando, Florida, November 7–13, 1998*, New York, Ny 10036, and 1109 Spring Street, Suite 300, Silver Spring, Md 20910, 1998. Acm Press and Ieee Computer Society Press.

[11] A. Snavely, G. Johnson, and J. Genetti. Data intensive volume visualization on the tera Mta and cray T3e. In Astc *'99 Proceedings*, February 1998.

[12] Tera Computer Company. *Tera Principles of Operation*, 1998 edition.

[13] Tera Computer Company. *Tera Programming Guide*, 12 March, 1999 edition.