An Initial Evaluation of the Tera Multithreaded Architecture and Programming System Using the C3I Parallel Benchmark Suite

Sharon Brunett
Center for Advanced Computing Research
California Institute of Technology
Pasadena, California, U.S.A.

sharon@cacr.caltech.edu

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California, U.S.A.

john-t@cs.caltech.edu

Marrq Ellenbecker
Computer Science Department
California Institute of Technology
Pasadena, California, U.S.A.

marrq@cs.caltech.edu

Abstract:

The Tera Multithreaded Architecture (MTA) is a radical new architecture intended to revolutionize high-performance computing in both the scientific and commercial marketplaces. Each processor supports 128 threads in hardware. Extremely fast thread switching is used to mask latency in a uniform-access memory system without caching. It is claimed that these hardware characteristics allow compilers to easily transform sequential programs into efficient multithreaded programs for the Tera MTA. In this paper, we attempt to provide an objective initial evaluation of the performance of the Tera multithreaded architecture and programming system for general-purpose applications. The basis of our investigation is two programs from the C3I Parallel Benchmark Suite (C3IPBS). Both these programs have previously been shown to have the potential for large-scale parallelization. We compare the performance of these programs on (i) a fast uniprocessor, (ii) two conventional shared-memory multiprocessors, and (iii) the first installed Tera MTA (at the San Diego Supercomputer Center). On these platforms, we compare the effectiveness of both automatic and manual parallelization.

Keywords:

multithreaded architectures, Tera MTA, C3I Parallel Benchmark Suite, multiprocessor performance evaluation, shared-memory multiprocessors, multithreaded programming, parallel programming, automatic parallelizing compilers, lightweight threads, fine-grained synchronization, Threat Analysis, Terrain Masking, Digital Alpha, HP Exemplar, Intel Pentium Pro

1 Introduction

The Tera Multithreaded Architecture (MTA) [1] is a fascinating new approach to scalable high-performance computing. Each processor supports 128 hardware threads and each word of memory has a full-empty bit. Extremely fast thread switching is used to mask memory latency in a uniform-access shared-memory system without caching. Given a highly multithreaded program, the processor should almost never stall waiting for a memory access to complete. In addition, the performance of a highly multithreaded program should scale directly with the number of processors. The Tera designers boldly claim "the MTA overcomes issues that inhibit scaling on every other system" [1].

However, to effectively utilize the Tera MTA, a program must indeed be highly multithreaded. This requires either that the programmer writes an explicitly multithreaded program (even for a single-processor Tera MTA) or that a parallelizing compiler is used (possibly with some assistance from the programmer). Because of the difficulty of explicit multithreaded programming, the emphasis of the Tera programming system is on programmer-assisted compiler-based parallelization. Since the Tera MTA provides a flat memory structure and hardware support for extremely fine-grained multithreading, it is not unreasonable to hope that parallelizing compilers for the Tera MTA will be more successful than those for conventional multiprocessors. We might also hope that manual parallelization (where necessary) will be easier for the Tera MTA than for conventional multiprocessors. The Tera designers claim "Tera's compilers let the programmer write code in a straightforward fashion, without bothering about the usual performance hacks required for other computers" [1].

In this paper, we attempt to evaluate both the execution performance of the Tera MTA and the ease of developing efficient general-purpose programs using the Tera programming system. The first Tera MTA (with two processors) has recently been installed at the San Diego Supercomputer Center. We are fortunate to have access to that machine and consulting services of the Tera engineers. As the basis of our investigation, we have performed experiments using two programs from the USAF Rome Laboratory Command, Control, Communication and Intelligence (C3I) Parallel Benchmark Suite (C3IPBS) [2]. These programs are compact, yet involve non-trivial data and control structures typical of interesting problems from many real-world domains. Both the programs (Threat Analysis and Terrain Masking) have previously been shown to have the potential for large-scale parallelization. If the Tera MTA is to live up to the claims of its designers, it is reasonable to expect that good performance should be achieved on these two programs without excessive programmer effort.

For both the benchmark programs, we have performed experiments to compare the following:

- Sequential execution on a fast commodity uniprocessor (a 500 MHz Digital Alpha).
- Sequential execution and manual parallelization on a commodity shared-memory multiprocessor (a quad-processor 200 MHz Intel Pentium Pro).
- Sequential execution, automatic parallelization, and manual parallelization on a conventional shared-memory supercomputer (a 16-processor HP Exemplar).
- Automatic parallelization and manual parallelization on the Tera MTA.

Through these experiments we compare the performance of the Tera MTA with conventional uniprocessor and multiprocessor platforms, and assess the amount of programming effort that is required to obtain good performance on the Tera MTA. In addition, we make suggestions regarding appropriate programming techniques and practices for obtaining the best performance from the Tera MTA.

The remainder of this paper is organized as follows: in Section 2, we describe the Tera MTA in more detail; in Section 3, we describe the C3IPBS in general and the two problems that we have chosen in particular; in Section 4, we describe our experimental goals and methods; in Section 5, we describe the Threat Analysis problem, program, and performance results; in Section 6, we describe the Terrain Masking problem, program, and performance results; in Section 7, we summarize our performance results and findings; and in Section 8, we conclude.

2 The Tera MTA

The Tera MTA is a scalable, shared-memory, general-purpose parallel computer that is intended to revolutionize high-performance computing in both the scientific and commercial marketplaces. The key features of the Tera multithreaded architecture are as follows:

- Up to 256 processors per system.
- 128 hardware threads (instruction streams/register sets) per processor.
- 255 MHz clock speed.
- Switching between hardware threads in one cycle.
- Shared memory between all processors with no caching.
- 64-way interleaved memory units.
- Full-empty bit on every word of memory, enabling very fine-grained thread synchronization.

The key features of the Tera operating system and programming system are as follows:

- Fully symmetric, parallel version of Unix (not yet delivered at the time of writing).
- Dynamic allocation/adjustment of processing resources to tasks during program execution.
- Automatic parallelizing compilers for Fortran, C, and C++.
- Compiler feedback to the programmer.
- Programmer-inserted pragmas and directives for explicit parallelization.
- Explicit thread creation using futures.
- Compiler-created hardware thread creation/termination with 2 cycles overhead per thread.
- Programmer-created software thread creation/termination with 50-100 cycles overhead per thread.
- Thread synchronization in one cycle.

The most important differences between the Tera MTA and conventional architectures are (i) hardware support for extremely lightweight multithreading, and (ii) the absence of a memory hierarchy (because memory latency is instead masking by thread switching). In this paper, we investigate the practical performance and programming consequences of these architectural innovations.

3 The Benchmark Problems

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite [2] consists of eight problems chosen to compactly represent the essential elements of real C3I applications. Each problem consists of the following:

- A problem description giving the inputs and required outputs.
- An efficient sequential program written in C to solve the problem.
- The benchmark input data.
- A correctness test for the benchmark output data.

The C3IPBS is a good framework for evaluating the applicability of the Tera MTA and programming system to general-purpose applications. The C3IPBS problems are computationally intensive, compact, and involve non-trivial data and control structures. These are the kind of problems the Tera MTA will have to perform well on to make a significant impact in general-purpose parallel computing.

For this initial evaluation of the Tera MTA, we have chosen to perform experiments using the following two C3IPBS problems:

1. Threat Analysis: A time-stepped simulation of the trajectories of incoming ballistic threats, with computation of options for intercepting the threats.
2. Terrain Masking: Computation of the maximum safe flight altitude over all points in an uneven terrain containing ground-based threats.

These are among the most algorithmically straightforward of the C3IPBS problems and stand the best chance of being parallelized without excessive programmer effort. Previous work with these two problems on conventional multiprocessors has shown that they both have the potential for large-scale parallelization.

4 Experimental Goals and Methods

The goal of this paper is to provide initial answers to the following two questions:

1. What sort of performance can a multithreaded architecture such as the Tera MTA deliver, compared to conventional uniprocessor and multiprocessor architectures?
2. What are appropriate methods for developing efficient general-purpose programs on a multithreaded architecture such as the Tera MTA, and how much programmer effort is required?

For both benchmark problems, we have performed experiments on conventional uniprocessor and multiprocessor architectures, as well as on the Tera MTA. Table 1 gives the platforms used in our performance comparison.

Machine	**Processors**	**Memory**	**Operating System**
Digital AlphaStation	1 x 500 MHz Digital Alpha 21164A	500 MB	Digital Unix 4.0C
NeTpower Sparta	4 x 200 MHz Intel Pentium Pro	500 MB	Windows NT 4.0

Hewlett-Packard Exemplar	16 x 180 MHz HP PA-8000	4 GB	SPP-UX 5.3
Tera MTA	2 x 255 MHz Tera MTA-1	2 GB	Carlos

Table 1: Platforms used in our performance comparison.

On conventional architectures, we have performed experiments to measure and compare the following:

- Sequential execution on a single-processor Digital AlphaStation. This gives us a measure of "fast execution" on a top-of-the-line conventional processor.
- Sequential execution and manual parallelization (algorithmic modification and thread library calls) on a quad-processor Pentium Pro system. This gives us a measure of the potential for coarse-grained multithreading.
- Sequential execution, automatic parallelization, and manual parallelization (algorithmic modification, pragmas, and synchronization library calls) on a 16-processor HP Exemplar. This demonstrates the effectiveness of automatic parallelization for a conventional coarse-grained shared-memory multiprocessor. It also gives us a measure of the success of manual parallelization for a conventional multiprocessor supercomputer.

On the two-processor Tera MTA, we have performed experiments to measure and compare the following:

- Sequential execution without any parallelization on one Tera MTA processor. This gives us a baseline measurement for performance on the Tera MTA.
- Automatic parallelization on the Tera MTA.
- Manual parallelization (using pragmas, synchronization variables, and futures) on the Tera MTA.

The results of these experiments provide us with a set of performance numbers by which to compare the different architectures and programming systems. Just as important as the raw performance comparison are the reasons for the performance and the comparison of programmer effort required for the various programming systems. In particular, we are very interested in whether automatic parallelization of general-purpose applications is more effective for the Tera MTA than for conventional coarse-grained multiprocessors.

5 Threat Analysis

The Problem

The Threat Analysis problem is a time-stepped simulation of the trajectories of incoming ballistic threats with computation of options for intercepting the threats. The input to the problem consists of (i) the trajectories of a set of incoming threats, and (ii) the locations and capabilities of a set of weapons that can be used to intercept the incoming threats. For each threat and weapon pair, the program must compute the time intervals over which the threat can be intercepted by the weapon. The benchmark provides five different input scenarios.

The Sequential Program

Program 1 gives a slightly simplified, high-level pseudocode representation of the algorithm used in the sequential Threat Analysis program.

```
ThreatAnalysis(
    in num_threats, in threats[],
    in num_weapons, in weapons[],
    out num_intervals, out intervals[])
{
    declare threat, weapon;
    declare t0, t1, t2;

    num_intervals = 0;
    for (threat = 0 .. num_threats - 1)
        for (weapon = 0 .. num_weapons - 1) {
            t0 = initial detection time of threat;
            while (weapon can intercept threat in [t0 .. impact time of
                t1 = first time after t0 that weapon can intercept thre
                t2 = last time after t1 that weapon can intercept threa
                intervals[num_intervals] = (threat, weapon, [t1 .. t2])
               num_intervals = num_intervals + 1;
                t0 = t2 + 1;
           }
        }
}
```

Program 1: Sequential Threat Analysis.

The program computes a set of tuples of the form (threat, weapon, interval) indicating that the threat can be intercepted by the weapon over the time interval. Because of the constraints on threat interception, there can be zero, one, or more intervals associated with each (threat, weapon) pair. The `t1` and `t2` interception times within the inner loop are computed using time-stepped simulations of threat and weapon positions.

The three nested loops in the program are not immediately parallelizable, because all iterations increment the `num_intervals` count and assign to the `intervals` array. The indices that a particular iteration assigns to cannot be determined without first executing the prior iterations. However, these shared variables are the only obstacles to parallelization of the outer two loops, as computation of intervals for each (threat, weapon) pair are otherwise independent of each other. The time-stepped simulations within the inner loop are not amenable to parallelization.

Performance of the Sequential Program without Parallelization

Table 2 gives the benchmark execution time (total time for all five input scenarios) of the sequential Threat Analysis program without parallelization on the platforms described in Section 4.

Platform	Time (seconds)

Alpha	187
Pentium Pro	458
Exemplar	343
Tera	2584

Table 2: Execution time of sequential Threat Analysis without parallelization.

The relative performance of the conventional platforms is in line with their respective processor speeds and memory systems. The program is compute-bound, rather than memory-bound, so the faster processors perform best. The most interesting feature of these performance measurements is the extremely slow speed of the Tera MTA - roughly 14 times slower than the Alpha. The Tera MTA is not an efficient platform for execution of single-threaded programs. One reason is that a single thread on the Tera MTA can issue only one instruction every 21 cycles, giving roughly 5% processor utilization.

Performance of Automatic Parallelization

On both the Exemplar and Tera MTA platforms, the manufacturer-supplied automatic parallelizing compilers were unable to identify any practical opportunities for parallelization of the sequential Threat Analysis program. The automatic parallelizing compilers and analysis tools were unable make any suggestions regarding changes to the program (e.g., algorithmic modifications or the addition of pragmas) that might expose parallelism. The reason is that the algorithm is inherently sequential - the outer-loop iterations assign to shared variables and the inner loops are sequential time-stepped simulations. In addition, the program (like most general-purpose programs) contains chains of function calls, pointer operations, and non-trivial index expressions that thwart compiler analysis and make automatic parallelization extremely difficult. However, the program can be manually parallelized through relatively straightforward algorithmic modifications. This program is an example of the limitations of automatic parallelizing compilers for general-purpose applications.

A Multithreaded Program

It is relatively straightforward to manually modify the sequential Threat Analysis program to allow multithreading. Program 2 gives a slightly simplified, high-level pseudocode representation of the algorithm used in our multithreaded Threat Analysis program.

```
ThreatAnalysis(
    in num_threats, in threats[],
    in num_weapons, in weapons[],
    in num_chunks, out num_intervals[], out intervals[][])
{
    declare chunk;

    #pragma multithreaded
    for (chunk = 0 .. num_chunks - 1) {
        declare first_threat, last_threat;
```

```
            declare threat, weapon;
            declare t0, t1, t2;

            first_threat = (chunk*num_threats)/num_chunks;
            last_threat = ((chunk+1)*num_threats)/num_chunks - 1;
            num_intervals[chunk] = 0;
            for (threat = first_threat .. last_threat)
                for (weapon = 0 .. num_weapons - 1) {
                    t0 = initial detection time of threat;
                    while (weapon can intercept threat in [t0 .. impact tim
                        t1 = first time after t0 that weapon can intercept
                        t2 = last time after t1 that weapon can intercept t
                        intervals[chunk][num_intervals[chunk]] = (threat, w
                       num_intervals[chunk] = num_intervals[chunk] + 1;
                        t0 = t2 + 1;
                    }
                }
        }
    }
```

Program 2: Multithreaded Threat Analysis.

The outer loop over all threats has been replaced by a multithreaded loop in which each iteration is responsible for a different chunk (i.e., subrange) of the threats. The problem of the shared variables has been solved by modifying the algorithm so that each iteration increments its own `num_intervals` count and assigns to its own section of the `intervals` array. Declarations of other variables are localized by moving them into the inner blocks. The outer-loop iterations are now completely independent of each other and can be executed by separate threads.

The drawback of this multithreaded program is that it requires a larger `intervals` array than the sequential program. Since there is no way to determine in advance the number of intervals that each iteration will compute, each iteration's section of the `intervals` array must be generously oversized. Therefore, the larger the number of chunks, the larger the `intervals` array.

Performance of the Multithreaded Program on Conventional Multiprocessors

Table 3 gives the benchmark execution time (total time for all five input scenarios) of the multithreaded Threat Analysis program on the quad-processor Pentium Pro platform described in Section 4. The program was manually parallelized using the Caltech Sthreads library [3] implemented on top of the Win32 thread API [4] supported by Windows NT and was executed using one chunk/thread per processor.

Number of processors	Time (seconds)	Speedup
Sequential	458	N.A.
1	466	1.0
2	233	2.0

3	157	2.9
4	117	3.9

Table 3: Execution time of multithreaded Threat Analysis on quad-processor Pentium Pro.

Figure 1 shows the speedup of the multithreaded Threat Analysis program over sequential execution on the Pentium Pro platform. Excellent speedups are achieved, because the threads are completely independent and execute mostly within cache.

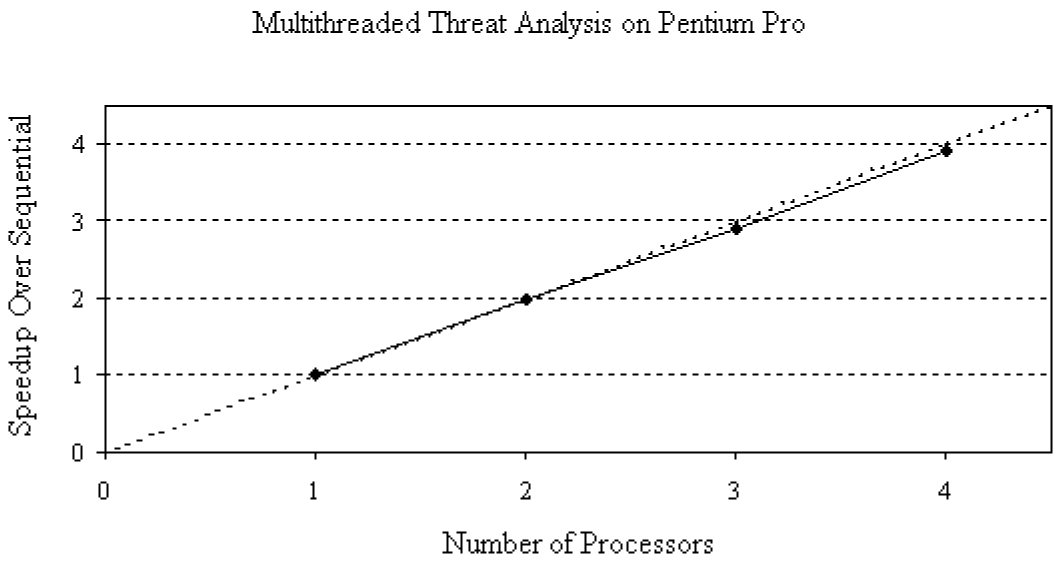

Figure 1: Speedup of multithreaded Threat Analysis on quad-processor Pentium Pro.

Table 4 gives the benchmark execution time (total time for all five input scenarios) of the multithreaded Threat Analysis program on the 16-processor Exemplar platform described in Section 4. The program was manually parallelized using the Exemplar shared-memory programming pragmas [5] and was executed using one chunk/thread per processor.

Number of processors	Time (seconds)	Speedup
Sequential	343	N.A.
1	343	1.0
2	172	2.0
3	115	3.0
4	87	3.9
5	69	5.0
6	58	5.9
7	50	6.9

8	43	7.9
9	39	8.8
10	35	9.9
11	32	10.7
12	29	11.7
13	27	12.7
14	26	13.4
15	24	14.4
16	22	15.4

Table 4: Execution time of multithreaded Threat Analysis on 16-processor Exemplar.

Figure 2 shows the speedup of the multithreaded Threat Analysis program over sequential execution on the Exemplar platform. As on the Pentium Pro, excellent speedups are achieved, because the threads are completely independent and execute mostly within cache.

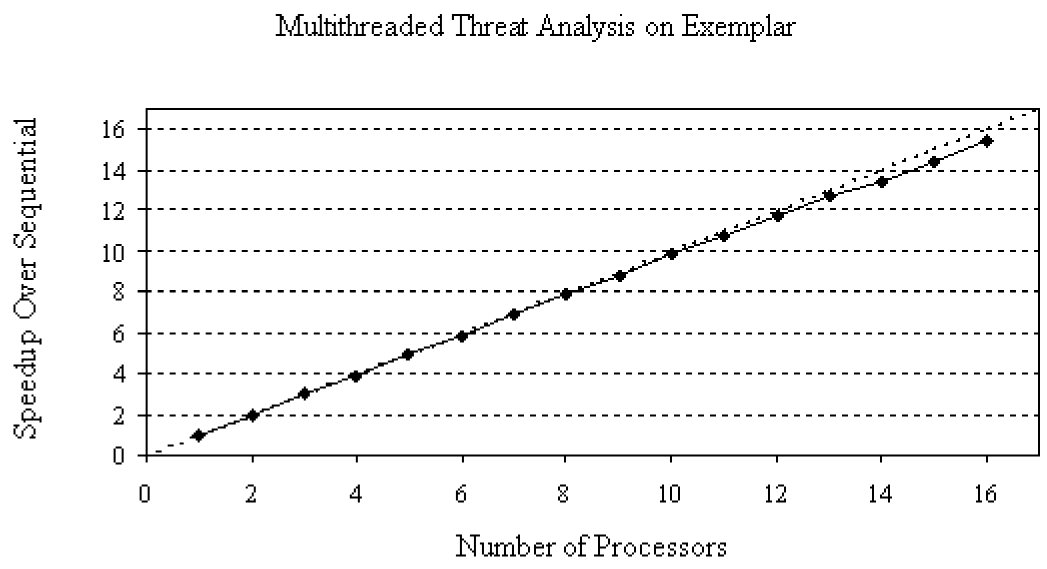

Figure 2: Speedup of multithreaded Threat Analysis on 16-processor Exemplar.

These performance measurements demonstrate that coarse-grained, outer-loop level multithreading is a straightforward, successful, and scalable approach to achieving parallelism for the Threat Analysis problem on shared-memory multiprocessors.

Performance of the Multithreaded Program on the Tera MTA

Table 5 gives the benchmark execution time (total time for all five input scenarios) of the multithreaded Threat Analysis program on the dual-processor Tera MTA platform described in

Section 4. The program was manually parallelized using the Tera parallelization pragmas [1] and was executed using 256 chunks. The Tera compiler determined the exact number of threads used.

Number of Processors	Time (seconds)	Speedup
1	82	1.0
2	46	1.8

Table 5: Execution time of multithreaded Threat Analysis on dual-processor Tera MTA.

The multithreaded program runs dramatically faster (32 times faster on one processor) than the sequential program on the Tera MTA. However, less than perfect speedup is achieved when moving from one to two processors. The less-than-ideal speedup may be a result of the development status of the current Tera MTA network or the number of threads used.

Table 6 gives the benchmark execution time of the multithreaded Threat Analysis program on the dual-processor Tera MTA with varying numbers of chunks (and hence varying numbers of threads).

Number of Chunks	Time (seconds)
8	386
16	197
32	104
64	61
128	46
256	46

Table 6: Execution time of multithreaded Threat Analysis with varying number of chunks on Tera MTA.

The program requires hundreds of threads to execute efficiently on the Tera MTA. Since each input scenario for the Threat Analysis benchmark has 1000 threats, parallelization over threats in our multithreaded program easily supplies enough threads for efficient execution. However, because of the large numbers of chunks, a large `intervals` array is required.

An alternative approach to parallelization of the Threat Analysis problem for the Tera MTA is to parallelize the outer loop over all threats without any kind of chunking. The problem of shared access to the `num_intervals` count and `intervals` array could be solved with very fine-graining locking using Tera synchronization variables. This approach does not require an oversized `intervals` array, but still requires manual localization and replication of many variables. An unwelcome consequence of this approach is nondeterministic ordering of the elements of the `intervals` array due to the race condition associated with the locking. Nondeterminacy of results complicates testing and debugging. However, it is interesting that this alternative approach is viable for the Tera MTA, but not for our conventional coarse-grained multiprocessor platforms.

Performance Comparison and Summary

Table 7 gives a comparison and summary of the benchmark execution time (total time for all five input scenarios) of the Threat Analysis program on the platforms described in Section 4. Times are given for sequential execution, automatic parallelization and manual parallelization, where appropriate on the different platforms.

Parallelization	Platform	Time (seconds)
None	Alpha	187
	Pentium Pro	458
	Exemplar	343
	Tera	2584
Automatic	Exemplar	343
	Tera	2584
Manual	Pentium Pro (4 processors)	117
	Exemplar (4 processors)	87
	Exemplar (8 processors)	43
	Exemplar (16 processors)	22
	Tera MTA (1 processor)	82
	Tera MTA (2 processors)	46

Table 7: Performance comparison for execution times of Threat Analysis.

The most interesting features of this performance comparison are (i) the failure of automatic parallelization, and (ii) the comparison of multithreaded execution on the Tera MTA with multithreaded execution on the conventional coarse-grained multiprocessor platforms. For this program, the performance of one 255 MHz Tera MTA processor is approximately equivalent to four 180 MHz Exemplar processors. Obtaining parallelism for the Tera MTA was no easier or more difficult than obtaining parallelism for the conventional multiprocessor platforms. In both cases, the automatic parallelizing compilers were not helpful and the same relatively straightforward manual modification was sufficient to obtain enough parallelism to efficiently utilize the machines. However, the Tera did offer more options for parallelization because of its support for fine-grained multithreading. Sequential execution is dramatically slower on the Tera MTA than on any of the other platforms.

6 Terrain Masking

The Terrain Masking problem is a computation of the maximum safe flight altitude over all points in an uneven terrain containing ground-based threats. The input to the problem consists of (i) the ground elevation, for all points in the terrain, and (ii) the position and range of a set of ground-based threats. The output of the problem consists of the maximum altitude at which an aircraft is invisible to all threats, for all points in the terrain. The benchmark provides five different scenarios as input.

The Sequential Program

Program 3 gives a slightly simplified, high-level pseudocode representation of the algorithm used in the sequential Terrain Masking program.

```
TerrainMasking(
    in x_size, int y_size, in terrain[][],
    in num_threats, in threats[],
    out masking[][])
{
    declare x, y;
    declare threat;
    declare temp[][];

    for (x,y = 0 .. x_size - 1, 0 .. y_size - 1)
        masking[x][y] = INFINITY;

    for (threat = 0 .. num_threats - 1) {
        for (x, y = region of influence of threat)
            temp[x][y] = masking[x][y];
        for (x, y = region of influence of threat)
            masking[x][y] = INFINITY;
        for (x, y = region of influence of threat)
            masking[x][y] = maximum safe altitude over x,y due to threa
        for (x, y = region of influence of threat)
            masking[x][y] = Min(masking[x][y], temp[x][y]);
    }
}
```

Program 3: Sequential Terrain Masking.

For each threat in turn, the program computes the maximum safe flight altitudes due to the threat over its region of influence, then minimizes these altitudes into the overall result. The maximum safe flight altitudes due to a threat cannot be computed directly into the overall result because the value at one point is computed from the values at neighboring points. For this reason, the altitudes due to a threat are minimized into the overall result only after the altitudes have been computed for all points in the region of influence.

The outer loop over all threats is not immediately parallelizable, because the regions of influence of different threats can overlap. Parallelization requires that some kind of locking scheme be used for access to the `masking` array. Another approach is to modify the algorithm so that the terrain is partitioned into non-overlapping regions, with the maximum safe flight altitude computed independently within each region. However, this requires changes to the algorithm for computing the maximum safe flight altitude due to an individual threat - since one threat might span several regions. Yet another approach, which may be efficient on a fine-grained multithreaded architecture such as the Tera MTA, is to parallelize the inner loops.

Performance on Conventional Architectures

Table 8 gives the benchmark execution time (total time for all five input scenarios) of the sequential Terrain Masking program without parallelization on the platforms described in Section 4.

Platform	Time (seconds)
Alpha	158
Pentium Pro	197
Exemplar	228
Tera	978

Table 8: Execution time of sequential Terrain Masking without parallelization.

The relative performance of the conventional platforms is in line with their respective processor speeds and memory systems. The program is memory-bound, rather than compute-bound, so processor speed is not the major determinant of execution time. As with Threat Analysis, the most interesting feature of these performance measurements is the slow speed of the Tera MTA - roughly 6 times slower than the Alpha. The speed difference is less than with Threat Analysis, because the conventional processors are not fully utilized in this memory-bound program.

Performance of Automatic Parallelization

On both the Exemplar and Tera MTA platforms, the manufacturer-supplied automatic parallelizing compilers were unable to identify any practical opportunities for parallelization of the sequential Terrain Masking program. The automatic parallelizing compilers and analysis tools were unable make any suggestions regarding changes to the program (e.g., algorithmic modifications or the addition of pragmas) that might expose parallelism. The outer loop of the program cannot be parallelized without algorithmic modifications, since its iterations assign to overlapping regions of the `masking` array. The inner loops contain opportunities for parallelization, however the program (like most programs) contains chains of function calls, pointer operations, and non-trivial index expressions that thwart compiler analysis and make automatic parallelization extremely difficult. Like the Threat Analysis program, this program is an example of the limitations of automatic parallelizing compilers for general-purpose applications.

A Coarse-Grained Multithreaded Algorithm

The outer loop over all threats of the sequential Terrain Masking program can be parallelized if locking is used to ensure that multiple threads do not assign to overlapping regions of the `masking` array at the same time. Program 4 gives a slightly simplified, high-level pseudocode representation of the algorithm used in our coarse-grained multithreaded Terrain Masking program.

```
    TerrainMasking(
        in x_size, int y_size, in terrain[][],
        in num_threats, in threats[],
        in num_blocks, in num_threads, out masking[][])
    {
        declare x, y;
        declare i, j;
        declare thread;
        declare blocks[][];
        declare locks[][];

        for (i,j = 0 .. num_blocks - 1, 0 .. num_blocks - 1)
            blocks[i][j] = bounds of block i,j in blocking of terrain;
        for (x,y = 0 .. x_size - 1, 0 .. y_size - 1)
            masking[x][y] = INFINITY;

        #pragma multithreaded
        for (thread = 0 .. num_threads - 1) {
            declare threat;
            declare x, y;
            declare temp[][];

            while (unprocessed threats) {
                threat = next unprocessed threat;
                for (x, y = region of influence of threat)
                    temp[x][y] = INFINITY;
                for (x, y = region of influence of threat)
                    temp[x][y] = maximum safe altitude over x,y due to thre
                for (i,j = blocks overlapping with threat) {
                    lock(locks[i][j]);
                    for (x,y = region of overlap between threat and block i
                        masking[i][j] = Min(masking[x][y], temp[x][y]);
                    unlock(locks[i][j]);
                }
            }
        }
    }
```

Program 4: Coarse-grained multithreaded Terrain Masking.

The outer loop over all threats has been replaced by a multithreaded loop in which each iteration dynamically processes individual threats until all threats have been processed. The problem of shared access to the `masking` array has been solved by blocking the terrain into equal-sized blocks, with a separate lock associated with each block. The role of the `temp` and `masking` arrays has been swapped in the computation of the maximum safe flight altitudes in the region of influence of a threat. The `temp` array is minimized back into the `masking` array block by block. To avoid interference between threads, blocks are locked before writing and unlocked after writing.

The drawback of the multithreaded program is that each thread requires its own `temp` array. With the Terrain Masking benchmark, the region of influence of each threat is up to 5% of the total terrain. Therefore, this approach to multithreading does not require excessive extra storage for small numbers of threads (e.g., sixteen), but may be impractical for large numbers of threads (e.g., hundreds).

Performance of the Coarse-Grained Multithreaded Program on Conventional Multiprocessors

Table 9 gives the benchmark execution time (total time for all five input scenarios) of the multithreaded Terrain Masking program on the quad-processor Pentium Pro platform described in Section 4. The program was manually parallelized using the Caltech Sthreads library [3] implemented on top of the Win32 thread API [4] supported by Windows NT and was executed using one thread per processor and ten-by-ten blocking.

Number of processors	**Time (seconds)**	**Speedup**
Sequential	197	N.A.
1	172	1.1
2	97	2.0
3	74	2.7
4	65	3.0

Table 9: Execution time of multithreaded Terrain Masking on quad-processor Pentium Pro.

Figure 3 shows the speedup of the multithreaded Terrain Masking program over sequential execution on the Pentium Pro platform. On one processor, the multithreaded program achieves an incidental speedup over the sequential program because of the memory access effect of swapping the roles of the `temp` and `masking` arrays. However, the speedup on multiple processors is considerably less than ideal, with three-fold speedup on four processors. The reason is that the program is memory-bound, causing contention between threads for access to the Pentium Pro's shared memory.

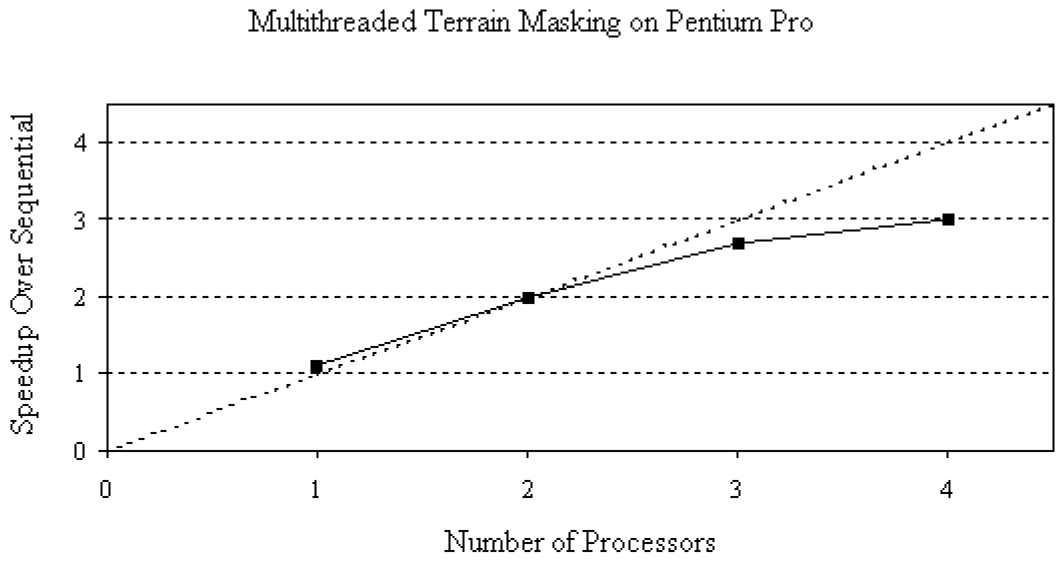

Figure 3: Speedup of coarse-grained multithreaded Terrain Masking on quad-processor Pentium Pro.

Table 10 gives the benchmark execution time (total time for all five input scenarios) of the multithreaded Terrain Masking program on the 16-processor Exemplar platform described in Section 4. The program was manually parallelized using the Exemplar shared-memory programming pragmas [5] and was executed using one thread per processor and ten-by-ten blocking.

Number of processors	Time (seconds)	Speedup
Sequential	228	N.A.
1	228	1.0
2	102	2.2
3	90	2.5
4	59	3.9
5	62	3.8
6	43	5.3
7	51	4.5
8	37	6.2
9	49	4.7
10	34	6.7
11	41	5.6
12	34	7.0
13	32	7.1
14	40	5.7
15	41	5.6
16	37	6.2

Table 10: Execution time of multithreaded Terrain Masking on 16-processor Exemplar.

Figure 4 shows the speedup of the multithreaded Terrain Masking program over sequential execution on the Exemplar platform. As on the Pentium Pro, considerably less than ideal speedups are achieved on the Exemplar because of memory contention between the threads.

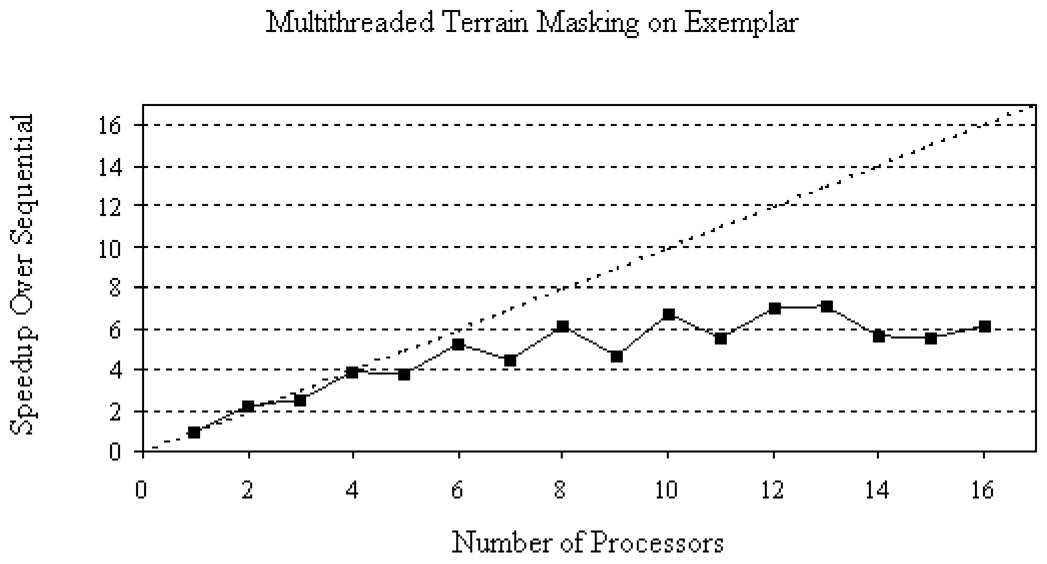

Figure 4: Speedup of multithreaded Terrain Masking on 16-processor Exemplar.

These performance measurements demonstrate that obtaining scalable multithreaded speedups for memory-bound programs such as the Terrain Masking program is difficult on conventional shared-memory multiprocessors.

Performance of a Fine-Grained Multithreaded Program on the Tera MTA

The coarse-grained multithreaded Terrain Masking program requires too much memory on the Tera MTA. Efficient utilization of the Tera MTA requires a large number of threads and each thread requires its own `temp` array. Therefore, we have implemented a fine-grained multithreaded Terrain Masking program. The inner loops that compute the maximum safe flight altitude for an individual threat are parallelized, instead of the outer loop. The loops are parallelized using the Tera parallelization pragmas and futures constructs [1]. Fine-grained parallelization was no easier or more difficult than coarse-grained parallelization. However, as with the Threat Analysis program, it is interesting that this approach is viable for the Tera MTA, but not for our conventional coarse-grained multiprocessor platforms.

Table 11 gives the benchmark execution time (total time for all five input scenarios) of the fine-grained multithreaded Terrain Masking program on the dual-processor Tera MTA platform described in Section 4.

Number of Processors	Time (seconds)	Speedup
1	48	1.0
2	34	1.4

Table 11: Execution time of multithreaded Terrain Masking on dual-processor Tera MTA.

The multithreaded program runs dramatically faster (20 times faster than on one processor) than the sequential program on the Tera MTA. However, as with the Threat Analysis program, considerably less than perfect speedup is achieved when moving from one to two processors. Again, the less-than-ideal speedup may be a result of the development status of the current Tera MTA network or the number of threads used.

Performance Comparison and Summary

Table 12 gives a comparison and summary of the benchmark execution time (total time for all five input scenarios) of the Terrain Masking program on the platforms described in Section 4. Times are given for sequential execution, automatic parallelization and manual parallelization where appropriate on the different platforms.

Parallelization	Platform	Time (seconds)
None	Alpha	158
	Pentium Pro	197
	Exemplar	228
	Tera	978
Automatic	Exemplar	228
	Tera	978
Manual	Pentium Pro (4 processors)	65
	Exemplar (4 processors)	59
	Exemplar (8 processors)	37
	Exemplar (16 processors)	37
	Tera MTA (1 processor)	48
	Tera MTA (2 processors)	34

Table 12: Performance comparison for execution times of Terrain Masking.

As with the Threat Analysis performance comparison, the most interesting features of this performance comparison are (i) the failure of automatic parallelization, and (ii) the comparison of multithreaded execution on the Tera MTA with multithreaded execution on the conventional coarse-grained multiprocessor platforms. For this program, the performance of the dual-processor 255 MHz Tera MTA processor is approximately equivalent to eight 180 MHz Exemplar processors. Obtaining parallelism for the Tera MTA was no easier or more difficult than obtaining parallelism for the conventional multiprocessor platforms. On the conventional multiprocessors, coarse-grained outer-loop parallelism was a practical approach. On the Tera MTA, fine-grained inner-loop parallelism was a practical approach. In both cases, the automatic parallelizing compilers were not helpful. Sequential execution is dramatically slower on the Tera MTA than on any of the other platforms.

7 Performance Summary

In this section, we summarize our observations from the preceding performance experiments.

Sequential Execution

The Tera MTA is not an efficient platform for execution of a single-threaded program (unless it is executed concurrently with many other programs). Sequential execution on a 255 MHz Tera MTA was much slower than on any of our conventional platforms. For both benchmark programs, sequential execution on the Tera MTA was approximately 5 times slower than sequential execution on a 200 MHz Pentium Pro. The Tera MTA was 6 times slower than a 500 MHz Alpha for the relatively memory-bound program (Terrain Masking) and 15 times slower for the relatively compute-bound program (Threat Analysis).

One reason for the Tera MTA's poor performance with single-threaded programs is that a single-thread can issue only one instruction every 21 cycles. Another reason is that the Tera MTA has no caching and relies on multiple threads to mask memory latency. A program must be highly multithreaded to obtain good performance on even a single-processor Tera MTA. The Tera engineers tell us that 80 concurrent threads are typically required to obtain full utilization of a single Tera MTA processor.

Automatic Parallelization

On both the Tera MTA and Exemplar platforms, the manufacturer-supplied automatic parallelizing compilers were unable to identify any practical opportunities for parallelization in either of the two sequential benchmark programs. Nor were they able to make any suggestions regarding changes to the program (e.g., algorithmic modifications, assertions, or pragmas) that might allow the compiler to parallelize the program.

Automatic parallelization of general-purpose programs is an extremely difficult task. There are two fundamental obstacles:

1. Efficient parallelization usually requires more than parallelization of loops in the sequential program. It involves significant modification of the underlying algorithm. This is the case with both benchmark programs. It is unreasonable to expect a compiler to deduce the high-level purpose of a program then automatically develop an alternative algorithm to solve the same problem.
2. General-purpose programs typically involve hundreds of separately compiled modules, chains of function calls, non-trivial index expressions, and operations on pointers that thwart compiler analysis of data dependencies and program flow. With both benchmark problems, the compilers were not even able to parallelize the manually transformed programs without the explicit parallel loop pragmas.

We did not see any indication that the current generation of automatic parallelizing compilers will be useful to the developers of general-purpose applications - particularly those that are typically written in languages such as C, C++, and Java - on either the Tera MTA or conventional multiprocessor platforms. Our experiments did not address the question of whether automatic parallelization is more successful for the Tera MTA than conventional multiprocessors for programs from more specialized domains, e.g., matrix-oriented programs written in Fortran.

Manual Parallelization

The Tera MTA and conventional coarse-grained multiprocessors have different strengths and weaknesses with regard to the ease of manual parallelization. These differences can be summarized as follows:

- A weakness of the Tera MTA is that it requires large numbers of threads for efficient execution. With the Threat Analysis program, splitting the outer loop into 16 threads yields over 15-fold speedup on a 16-processor Exemplar, whereas hundreds of threads are required for efficient execution on a multiprocessor Tera MTA.

  Splitting a program into many threads can be more difficult than splitting it into a few threads. The number of threads that can be obtained from the outer loop of the Terrain Masking problem is limited by the fact that the benchmark data sets contain only 60 threats per input scenario. This is plenty of threads for the Exemplar, but not enough for the Tera MTA.

  Splitting a program into many threads can require more extra memory than splitting it into a few threads, because data replication is often proportional to the number of threads. For both benchmark problems, outer-loop parallelization requires extra array storage for each thread.

- A strength of the Tera MTA is that its provides hardware support for truly fine-grained multithreading. For both benchmark problems, algorithms based on fine-grained multithreading of inner loops are practical on the Tera MTA that are not practical on our conventional multiprocessor platforms.

  On conventional multiprocessors with operating system support for threads, thread creation costs tens of thousands to hundreds of thousands of cycles and thread synchronization costs hundreds to thousands of cycles. On the Tera MTA, thread creation and synchronization cost only a few cycles.

The difficulty of exposing enough parallelism sometimes makes manual parallelization more difficult for the Tera MTA than for conventional multiprocessors. The ease of expressing efficient fine-grained parallelism sometimes makes manual parallelization easier for the Tera MTA than for conventional multiprocessors.

Multithreaded Execution

Programs with a sufficiently large number of threads run dramatically faster than equivalent single-threaded programs on the Tera MTA. For the benchmark programs, multithreaded execution on a single-processor 255 MHz Tera MTA was (i) between 2 and 3.5 times faster than sequential execution on the 500 MHz Alpha platform, (ii) approximately one third faster than multithreaded execution on the quad-processor 200 MHz Pentium Pro platform, and (iii) approximately the same speed as multithreaded execution on four processors of the 180 MHz Exemplar platform.

Since the current Tera MTA configuration has only two processors and its interconnection network is under development, our experiments do not adequately address the issue of scalability of multithreaded programs on multiprocessor Tera MTA platforms. Our limited multiprocessor experiments show speedups of 1.4 and 1.8 on two processors compared to one processor. These less-than-ideal speedups may be a result of the development status of the network or insufficient threads to fully utilize two processors.

8 Conclusion

The goal of this work was to provide an initial evaluation of the Tera MTA and programming system for general-purpose applications. Although we must be careful not to over-generalize experimental results for two applications from the same problem domain running on a prototype dual-processor Tera MTA, we are now in a position to highlight some clear strengths and weaknesses of the Tera MTA.

The most obvious weakness of the Tera MTA is its extremely poor performance for single-threaded programs. While the designers may claim that the Tera MTA is not intended for sequential programs, this is clearly a practical problem for users porting sequential programs from other platforms. It often takes a considerable amount of time to parallelize a sequential program, and the user may want some kind of acceptable performance from the sequential program while that parallelization is under way. In addition, some programs or parts of programs are inherently sequential. Our experiments indicate that one Tera MTA processor is approximately as powerful a four Exemplar processors. A sequential program runs at 25% peak performance on four Exemplar processors, but runs at only 5% of peak performance on one Tera MTA processor.

A related weakness of the Tera MTA is its reliance on large numbers of threads to obtain good performance - even on a single processor. Not all programs have the potential for hundreds of threads of control. This makes parallelization of some programs difficult for the Tera MTA. For example, a program in which the only opportunity for parallelism is an outer loop consisting of 16 independent iterations with equal workload will perfectly utilize a 16-processor Exemplar. However, this program contains only a small fraction of the parallelism necessary to fully utilize even a single-processor Tera MTA. Even programs that naturally split into large numbers of threads may contain sequences of execution that do not parallelize well. These sequences will become execution bottlenecks on the Tera MTA.

A major strength of the Tera MTA is its ability to efficiently execute truly fine-grained multithreaded programs. Thread creation, scheduling, and synchronization operations are many orders of magnitude less costly on the Tera MTA than on conventional multiprocessor platforms. It is very exciting for the programmer to be able to exploit fine-grained inner-loop parallelism as well as coarse-grained outer-loop parallelism. Similarly, synchronization on every element of a large data structure is practical, instead of requiring that the programmer artificially subdivide the structure solely for synchronization purposes. For many problems, there are more options for easy and efficient parallelization for the Tera MTA than for a conventional multiprocessor platform.

It is a shame that the flip side of the Tera MTA's ability to efficiently execute hundreds of fine-grained threads seems to be its inability to execute efficiently with a small or moderate number of threads. The Tera MTA would be a much more appealing platform if it could execute efficiently with both large and moderate numbers of threads and provide reasonable performance for single-threaded programs.

A potential strength of the Tera MTA that we were unable to investigate on a dual-processor configuration is scalability to large numbers of processors. Our experiments demonstrate that memory contention is sometimes a major obstacle to achieving scalability on conventional shared-memory multiprocessor platforms. It is possible that the Tera model of large numbers of fine-grained threads and no memory hierarchy may be effective in overcoming this obstacle. If this is the case, it would be major breakthrough in scalable supercomputing. We look forward to investigating this issue when Tera MTAs with large numbers of processors are installed in the near future.

The Tera automatic parallelizing compiler does not appear to offer much help to programmers developing general-purpose multithreaded applications for the Tera MTA. Automatic parallelization of large, general-purpose applications - particularly those written in C and C++ - is an extremely difficult problem that will probably not be solved anytime soon. Developers of general-purpose applications for the Tera MTA should expect that they will have to manually parallelize their programs in the forseeable future. However, parallelizing compilers may offer some assistance for applications from more specific domains, e.g., scientific programs written in Fortran that deal mostly with regular matrices.

It is difficult to compare of the raw performance of the Tera MTA with that of conventional processors. A single-processor Tera MTA appears to be approximately three times as fast as a cutting-edge commodity uniprocessor and a little faster than a slightly outdated commodity quad-processor shared-memory multiprocessor. In a cost-performance comparison, the Tera MTA is very clearly the loser. However, this comparison is not entirely fair to the Tera MTA, as the commodity processors are mass-produced and based on decade-old processor families. There is a huge cost to developing a new processor and producing it is small quantities. Unfortunately, we have no means of estimating the "true cost" of a mature, mass-produced Tera MTA processor. The eventual legacy of the Tera MTA may be the incorporation of hardware support for multithreading in future commodity processors.

This paper presents an initial evaluation of the Tera MTA. There is need for further objective evaluation using more problems, problems from different domains, and Tera MTA configurations with larger numbers of processors. We look forward to the installation of larger and more powerful Tera MTA platforms so that we and others can continue to explore this exciting new approach to high-performance computing.

Availability

The programs described in this paper are available from the Caltech Structured Multithreaded Programming Project web site at http://threads.cs.caltech.edu/.

Acknowledgments

Thanks to Gail Alverson, John Feo, Kristi Maschhoff, and Allan Porterfield at Tera Computer Company and Wayne Pfeiffer at San Diego Supercomputer Center for their help developing these programs and their comments on this paper. Particular thanks to John Feo for developing the fine-grained solution to the Terrain Masking problem for the Tera MTA. Thanks to Sean Suchter in the Computer Science Department at Caltech for investigating and explaining the Tera MTA to us in the early stages of this work. Thanks to Eric Bogs and Jason Roth in the Computer Science

Department at Caltech for developing the Sthreads solutions to the Threat Analysis and Terrain Masking problems on the Pentium Pro platform. Thanks also to our other colleagues in the Computer Science Department and Center for Advanced Computing Research at Caltech for their support of this work. This research was supported by DARPA grant DABT63-97-C-0028, Air Force Office of Scientific Research grant AFOSR-91-0070, and the NSF Center for Research on Parallel Computation (CRPC) under Cooperative Agreement No. CCR-912008.

References

1. **Tera Computer Company Homepage**. http://www.tera.com/.

2. Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Rakesh Jha, Wing Au, Minesh Amin, David A. Catanon, and Vipin Kumar. **The C3I Parallel Benchmark Suite - Introduction and Preliminary Results**. Supercomputing '96, Pittsburgh, Pennsylvania, November 17-22, 1996.

3. John Thornley, K. Mani Chandy, and Hiroshi Ishii. **A System for Structured High-Performance Multithreaded Programming in Windows NT**. 2nd USENIX Windows NT Symposium, pages 67-76, Seattle, Washington, August 3-4, 1998.

4. Jim Beveridge and Robert Wiener. **Multithreading Applications in Win32: the Complete Guide to Threads**. Addison-Wesley Developers Press, Reading, Massachusetts, 1997.

5. **Exemplar Programming Guide**, Third Edition. Hewlett-Packard Company. June 1996.