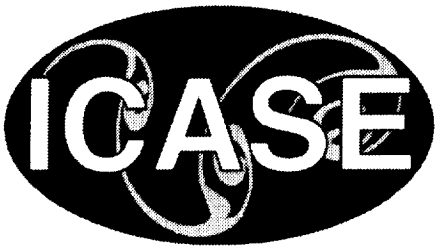
NASA/CR-1998-208953
ICASE Interim Report No. 33



The Tera Multithreaded Architecture and Unstructured Meshes

Shahid H. Bokhari
University of Engineering and Technology, Lahore, Pakistan

Dimitri J. Mavriplis
ICASE, Hampton, Virginia

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

December 1998

Available from the following:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

THE TERA MULTITHREADED ARCHITECTURE AND UNSTRUCTURED MESHES*

SHAHID H. BOKHARI† AND DIMITRI J. MAVRIPLIS‡

Abstract. The Tera Multithreaded Architecture (MTA) is a new parallel supercomputer currently being installed at San Diego Supercomputing Center (SDSC). This machine has an architecture quite different from contemporary parallel machines. The computational processor is a custom design and the machine uses hardware to support very fine grained multithreading. The main memory is shared, hardware randomized and flat. These features make the machine highly suited to the execution of unstructured mesh problems, which are difficult to parallelize on other architectures.

We report the results of a study carried out during July-August 1998 to evaluate the execution of EUL3D, a code that solves the Euler equations on an unstructured mesh, on the 2 processor Tera MTA at SDSC.

Our investigation shows that parallelization of an unstructured code is extremely easy on the Tera. We were able to get an existing parallel code (designed for a shared memory machine), running on the Tera by changing only the compiler directives. Furthermore, a *serial* version of this code was compiled to run in parallel on the Tera by judicious use of directives to invoke the "full/empty" tag bits of the machine to obtain synchronization. This version achieves 212 and 406 Mflop/s on one and two processors respectively, and requires no attention to partitioning or placement of data issues that would be of paramount importance in other parallel architectures.

Key words. parallel computing, multiprocessors, supercomputing, multithreaded architectures, Tera computer, unstructured meshes

Subject classification. Computer Science

1. Introduction.

The Tera Multithreaded Architecture (MTA) is a new parallel supercomputer currently being installed at San Diego Supercomputing Center (SDSC). This machine has an architecture quite different from those of other contemporary parallel machines. It has a flat, shared memory without locality and has hardware support for very fine grained multithreading. The machine and its associated parallelizing compiler promise great ease in scalable parallel computing.

We report the results of a study carried out during July-August 1998 in which we evaluated the porting of an unstructured mesh code to the Tera. Algorithms based on unstructured meshes are ordinarily very difficult to parallelize efficiently on conventional parallel machines. Our results show that code can be ported with great ease to the Tera and that the performance achieved is very promising.

We first discuss, in Section 2, how the Tera architecture attempts to compensate for the limitations of conventional parallel machines. We describe the architecture of the machine in some detail in Section 3. In Section 4 we describe our unstructured mesh solver and how it was ported to the Tera. Two variants of the code were ported: the measured performance of these codes is presented in Sections 4 and 5 respectively. In Section 7, we conclude with a discussion of the problems we encountered and our plans for future research.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199. This research was made possible by access to the Tera MTA at the San Diego Supercomputer Center, which receives major support from the National Science Foundation.

†Department of Electrical Engineering, University of Engineering and Technology, Lahore, Pakistan (shahid@icase.edu).

‡Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199 (dimitri@icase.edu).

2. The State of Parallel Computing. Despite nearly half a century of research and development, truly general purpose parallel computing remains an elusive goal. Very careful programming and a good knowledge of the target computer's architecture are required to achieve even modest performance. At the same time, the wide diversity in available parallel architectures means that a program successfully ported to one machine may require considerable reworking to run on another. This discourages practitioners from exploiting parallel computing and confines the field to experts, academicians and researchers. Finally an inordinate effort is required to successfully parallelize an algorithm and even then the achieved performance is poor compared with the theoretical peak.

There are a number of reasons for the above mentioned state of affairs. Firstly, with currently available distributed memory machines, parallel computing involves a never-ending battle to match computation to architecture. Parallel machines necessarily involve large numbers of interconnected processors. The utilization of these processors is inevitably linked to how well the structure of the computation matches (or can be transformed to match) the structure of the machine. The process of transformation may involve partitioning, mapping and reordering of data, as well as reformulation of the computation. These transformational requirements lead to major combinatorial problems that are often more difficult than the actual problem being solved. The programmer is required to have extensive knowledge of the interconnect network, cache hierarchy, arithmetic unit etc.

Figure 2.1 sketches how the quest for utilization has evolved over time on uniprocessors. The checkered rectangles in this figure represent the hardware-time products for the indicated architectures–the higher the utilization, the larger the fraction of grey blocks in this rectangle. A simple, primitive processor's hardware could be utilized only to a limited extent. Among the first developments in computer architecture was the evolution of pipelined processors that could deliver higher utilization for certain types of operations. This higher utilization required additional investments in 'performance enhancing' hardware, that is hardware that did not contribute to actual computation but was required to improve the utilization of the 'productive' hardware. A modern pipelined processor improves utilization by considerable investment in such performance enhancing hardware as well as in sophisticated compilers. At the same time, the programmer may have to make some investment in transforming his program, or even the underlying algorithm, to better utilize the specific hardware. Figure 2.1 also shows that, in a contemporary pipelined machine, some of the work done by the hardware may be wasted because of speculative execution.

A modern parallel processor requires relatively larger hardware and software investments to obtain adequate utilization. Figure 2.2 illustrates how the productive hardware (that carries out the actual computations for our program) has to be augmented with additional hardware and software. A contemporary high-performance parallel machine requires performance enhancing hardware in the form of caching units, high speed interconnect, synchronization mechanism, etc. Furthermore, considerable investment may need to be made in compilers, in an operating system, and for analysis tools. Parallel programming platforms such as PVM [5, 6], MPI [7], PARTI [1], and PETSc [4] constitute part of the software overhead. Finally the programmer needs to invest considerable effort in developing his program, rethinking his algorithm and, of course, in the difficult issues of partitioning, mapping, scheduling, etc. Despite these overheads, the utilization achieved by such processors is low; indeed, there are large classes of problems for which these machines are considered unsuitable.

One approach to improving utilization is to invest in additional hardware and software to support parallelism, possibly at the expense of additional compiler overhead. Figure 2.3 illustrates how specially designed hardware can be used to offload the burden placed on the programmer and on parallelism support software.

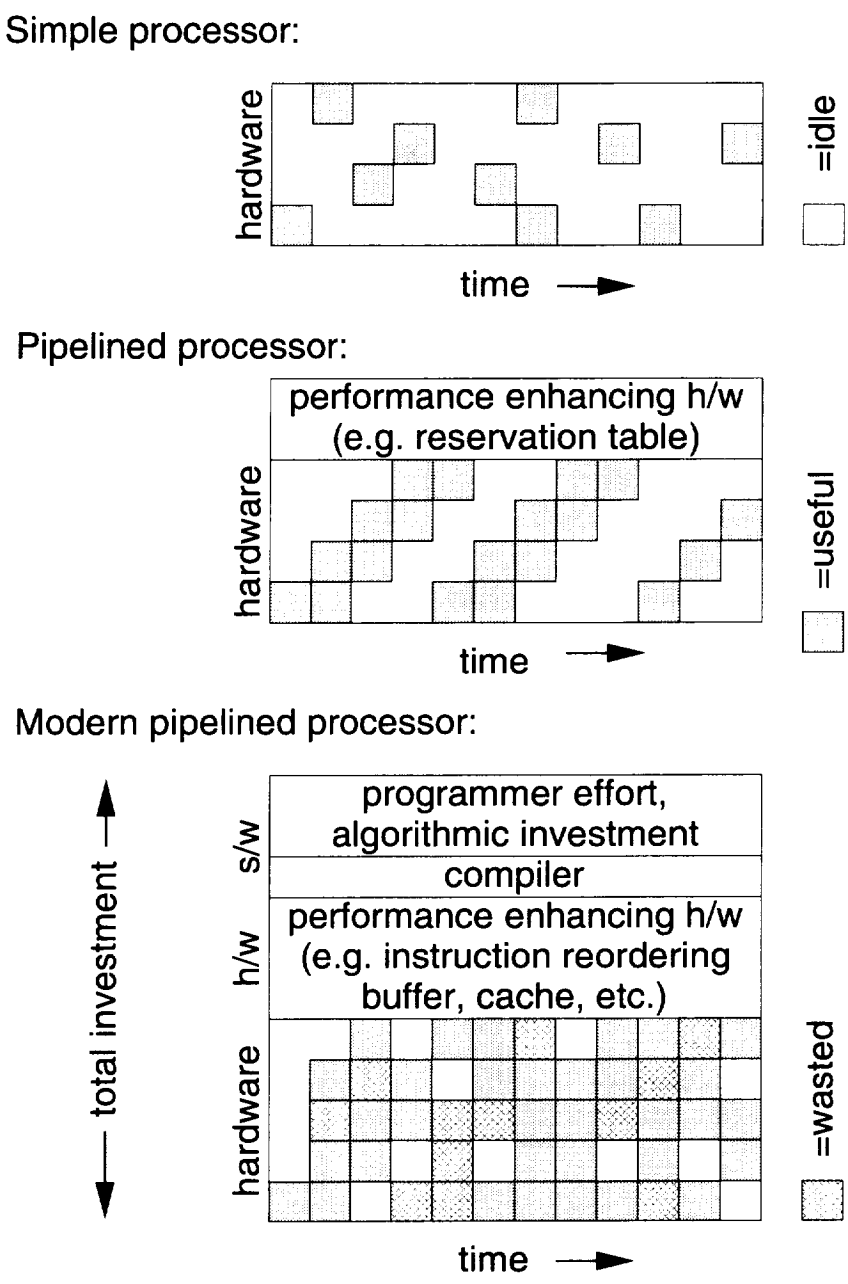


FIG. 2.1. *The Quest for Utilization. As uniprocessors have evolved over time, the investment in non-productive 'performance enhancing' hardware has increased. A modern machine also requires considerable investment in compiler development.*

This proposal rules out the possibility of using commodity microprocessors for parallel processing and requires a protracted cycle of development and production. However the potential benefits are very attractive. The Tera Multithreaded architecture (MTA) uses this path, as described in Figure 2.4. By investing heavily in performance enhancing hardware, the Tera is able to eliminate the issues of parallelism support and data partitioning, etc. Higher investment in hardware reduces the effort required by the programmer and also increases the utilization of the productive hardware.

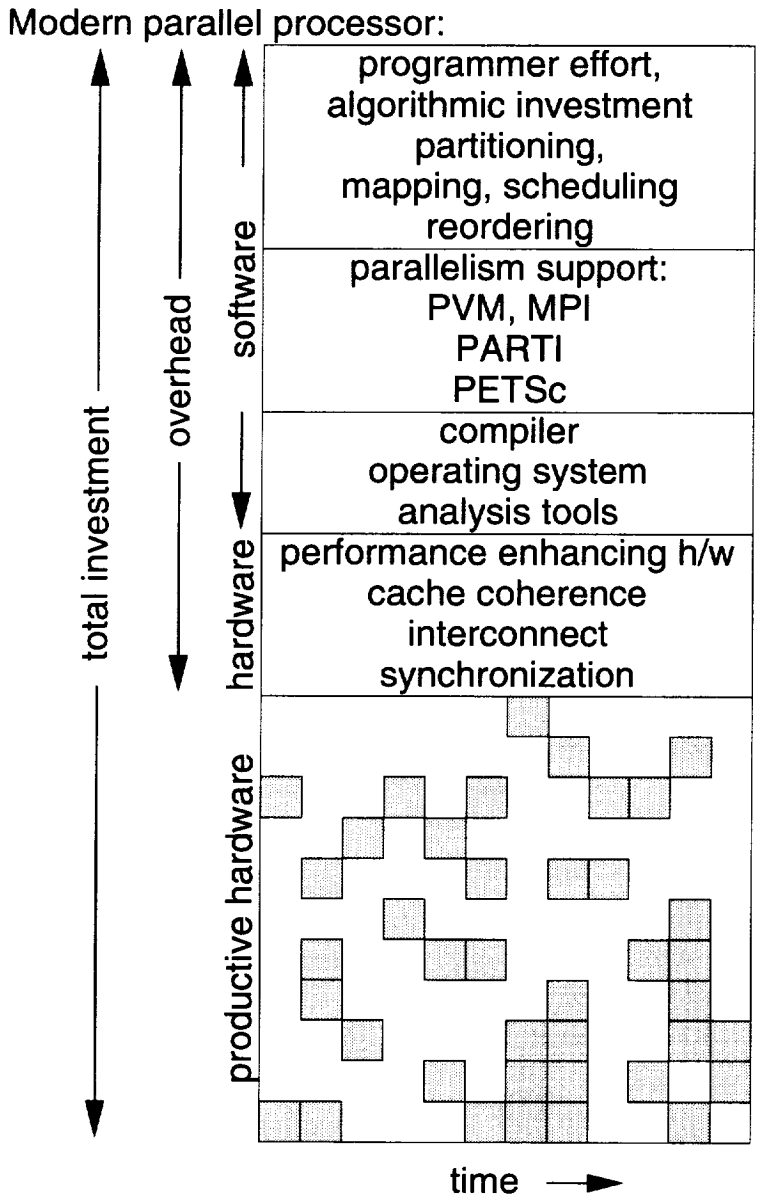


FIG. 2.2. *Parallel Computing: Investment and Return. A modern parallel processor achieves low utilization despite considerable investment in hardware and software.*

3. Key Features of the Tera Architecture. Detailed information on the Tera architecture may be found in [2, 3] and at the Tera web site[1]. We present a brief overview.

3.1. Zero overhead thread switching. The Tera has special purpose hardware (*streams*) that can hold the state of up to 128 threads (per processor). On each clock cycle, each processor switches to a different resident thread and issues one instruction from that thread. A blocked thread (for example, one waiting for word from memory or for a synchronization event) causes no overhead, the processor executes the instruction

[1] `www.tera.com`

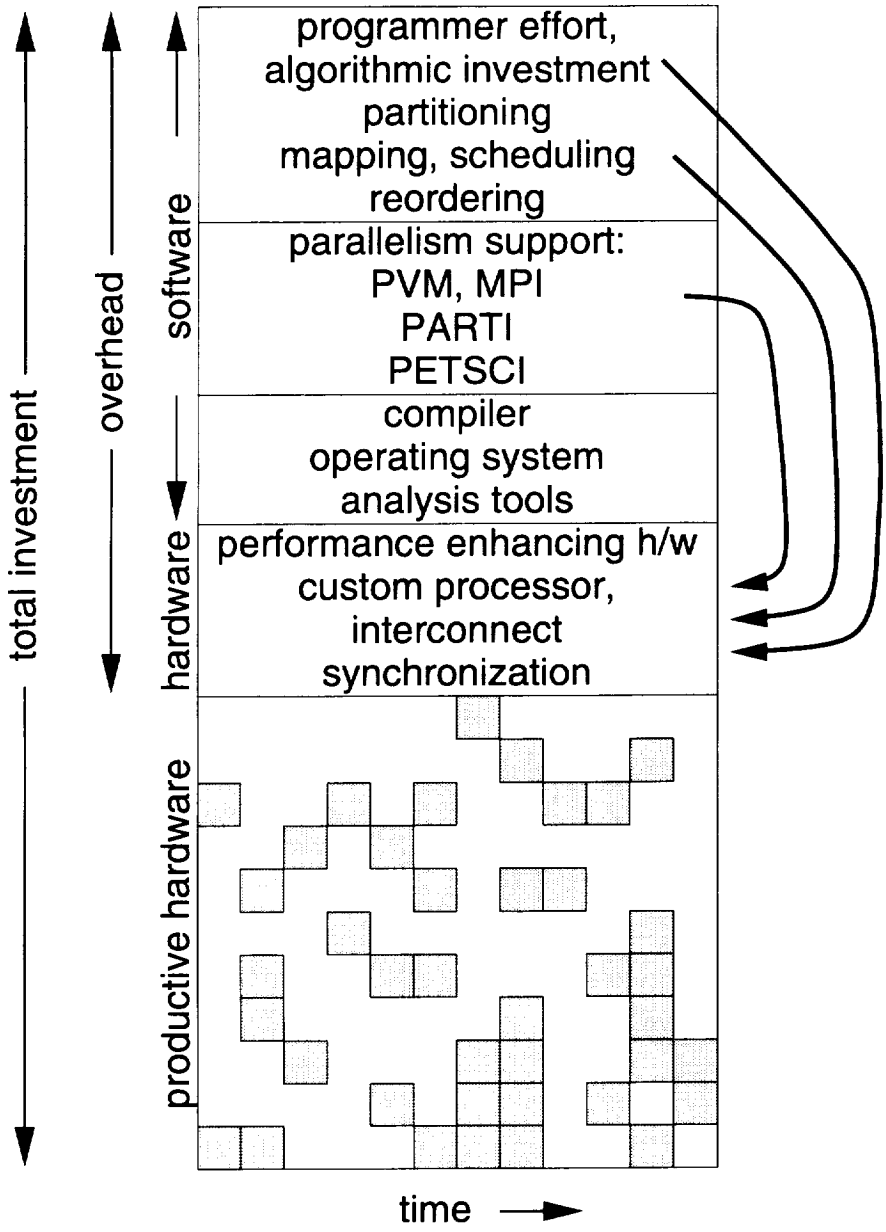


FIG. 2.3. *Additional investment in hardware reduces software overhead (functions migrate into hardware).*

of some other ready thread.

3.2. Pipelined Processors. Each processor in the Tera system has 21 stages. As each processor accepts an instruction from a different stream at each clock tick, at least 21 ready threads are required to keep it fully utilized. Since the state of up to 128 streams is kept in hardware, this target of 21 ready threads is easy to achieve.

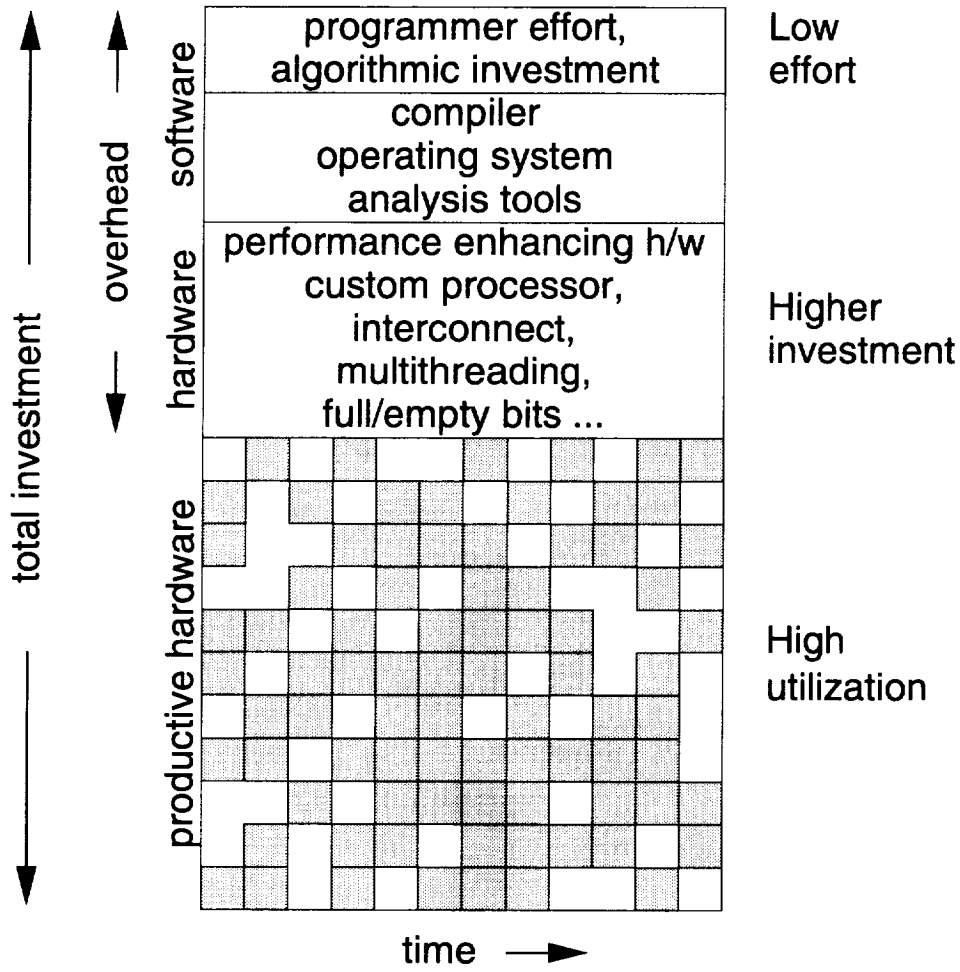


FIG. 2.4. *The Tera Idea: Higher investment in hardware yields improved utilization and also reduces software overhead.*

3.3. Flat Shared Memory. All memory locations on the Tera are 64 bit words. Addresses are hashed by hardware to randomly scatter them across memory banks. The cycle time per memory bank is 35 clock ticks. The access time varies from 150 200 clock ticks, depending upon the size of the system. The 21 stage processor pipeline is dwarfed by the $\approx$ 150 cycles of latency to memory. This mismatch is overcome by having more than 21 threads, each with lookahead or performing non-memory operations. A processor will typically have hundreds of memory references outstanding. As a result of these features, the memory has no locality and there are no issues of partitioning or mapping on the machine.

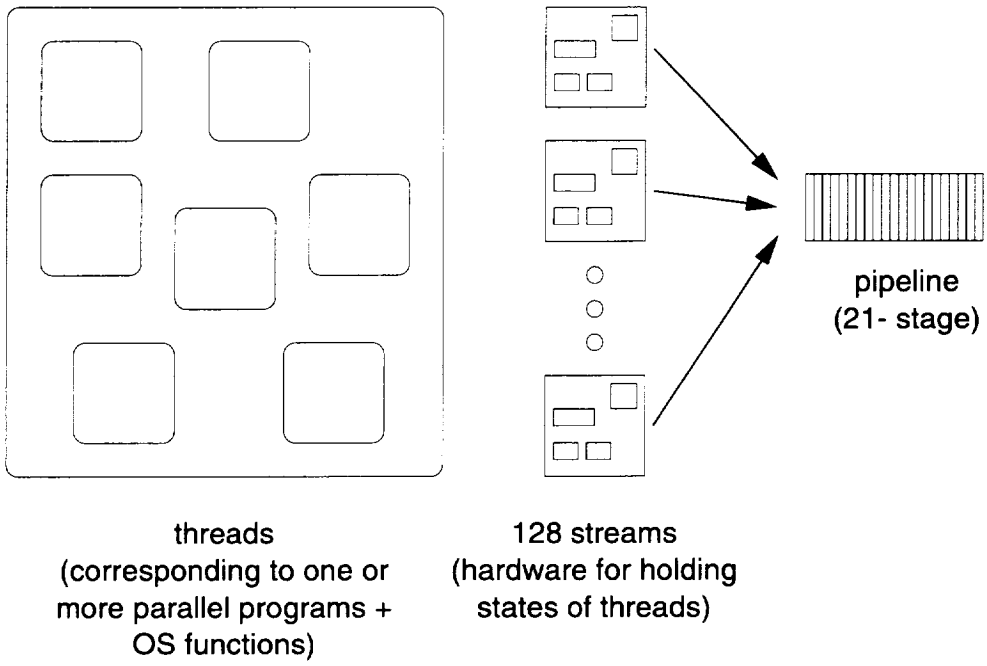


FIG. 3.1. *The Tera Architecture (1 processor)*

3.4. Extremely fine-grained synchronization. Each 64 bit word of memory has an associated full/empty bit. A memory location can be written into or read out of using ordinary loads and stores, as in conventional machines. We can also do load and stores under the control of the full/empty bit. For example, a "read-full set-empty" instruction will read data from a location only if that location's full/empty bit is set. It will set the full/empty bit to empty after successfully executing the read. If the full/empty bit is not set, the thread executing the read will be suspended (by hardware) and will resume only when the bit is set full by some other thread. This feature allows extremely fine-grained synchronization and is detailed in Section 5.1.

3.5. Tera Performance Characteristics. The Tera is designed to operate on a 300 MHz Clock. At the present time the clock is running at 255 MHz. There are three units in each processor, all of which may be active during a single cycle:

unit	Operation	flop
M (Memory)		0
A (Arithmetic)	fused multiply-add	2
C (Control)	add	1
	Total	3

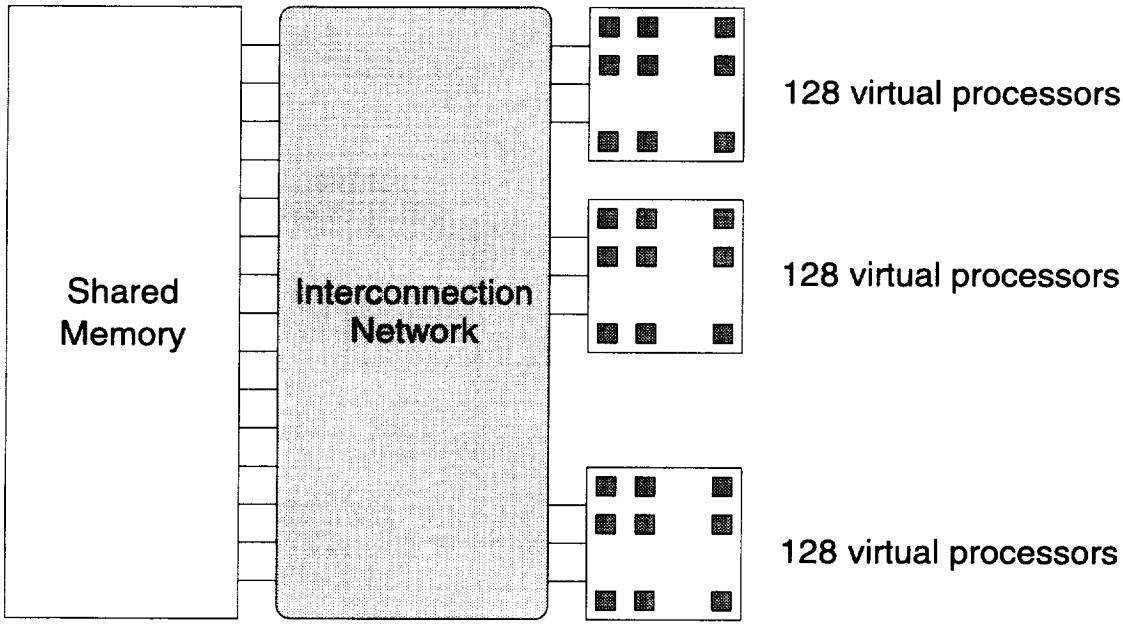


FIG. 3.2. *A View of the Tera Multiprocessor. Each stream may be thought of as a virtual processor. Some streams may be needed to execute OS functions a user may not be able to use all 128 streams per processor.*

Thus "peak" performance is $3 \times 300 = 900$ MFlop/s. We have measured 210 MFlop/s on actual code at 255 MHz, this extrapolates to 274 MFlop/s at 300 MHz.

4. The Numerical Solver. The code that we chose to implement on the Tera is a representative kernel from EUL3D, a 3D unstructured grid Euler solver. This code uses vertex based variables and an edge-based loop for residual construction. The kernel reproduces edge based flux loops and vertex based updates.

Unstructured mesh problems have traditionally been difficult to parallelize because of their need for partitioning, mapping and load balancing. Furthermore, because of the indirect access to the grid data, such problems are hard to compile.

On the Tera these become non-issues because

1. Partitioning and mapping are not needed because of the flat shared memory which has no locality, and
2. Load balancing is not needed because of very fine grained multithreading: loops can be dynamically scheduled across processors with very little overhead.

The specific problem we experimented with has 53961 nodes and 353476 edges. This is considered to be a medium-sized problem in the aerodynamics community – a large problem would have 0.3 million nodes

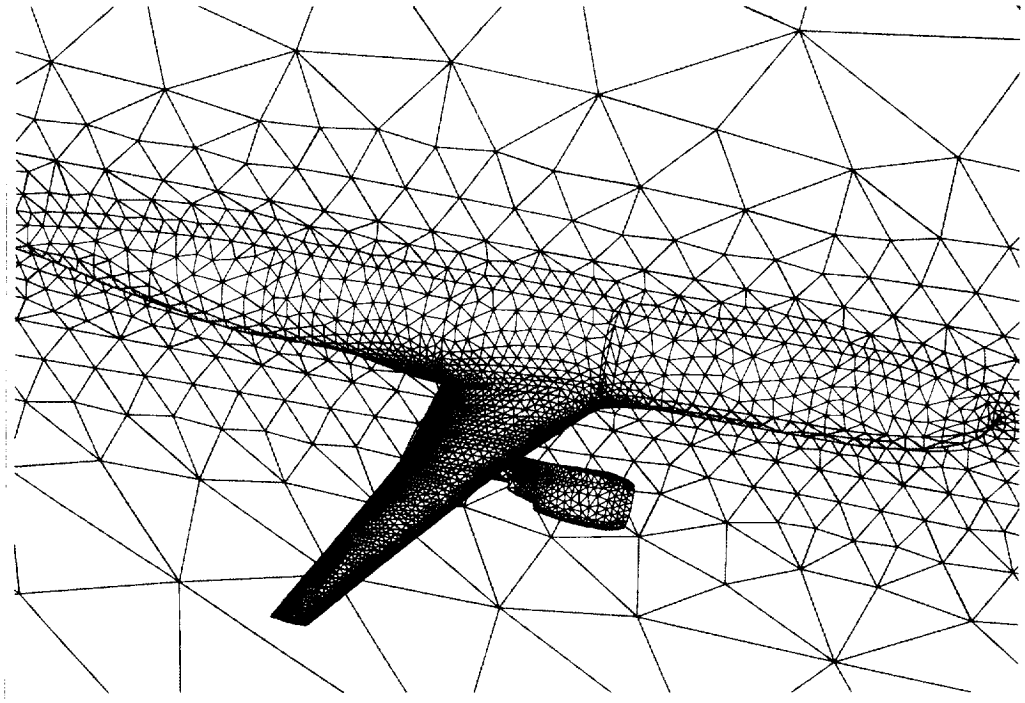


FIG. 4.1. *Unstructured meshes are widely used in aerodynamic and structural analysis codes. Because of the enormous, irregular variations in density, algorithms based on such meshes are difficult to parallelize on conventional multiprocessors.*

and 3 million edges.

At each node of our mesh we store density, momentum (x, y, z components), energy, pressure, plus some scratch space. This results in approximately 10 variables per node.

For each edge we need to store the identity of the 2 nodes at its end points plus a vector describing the orientation of the edge. We thus have ≈ 5 variables per edge.

The movement of data in the edge based loop is described in Figure 4.2. Pseudocode corresponding to this loop is given below.

```
do i=1, totalNodes
  initialize variables
enddo

do cycle=1, totalCycles
   do i=1, totalNodes
     clear residuals
   enddo

   do i=1, totalEdges
     compute residuals
   enddo
```

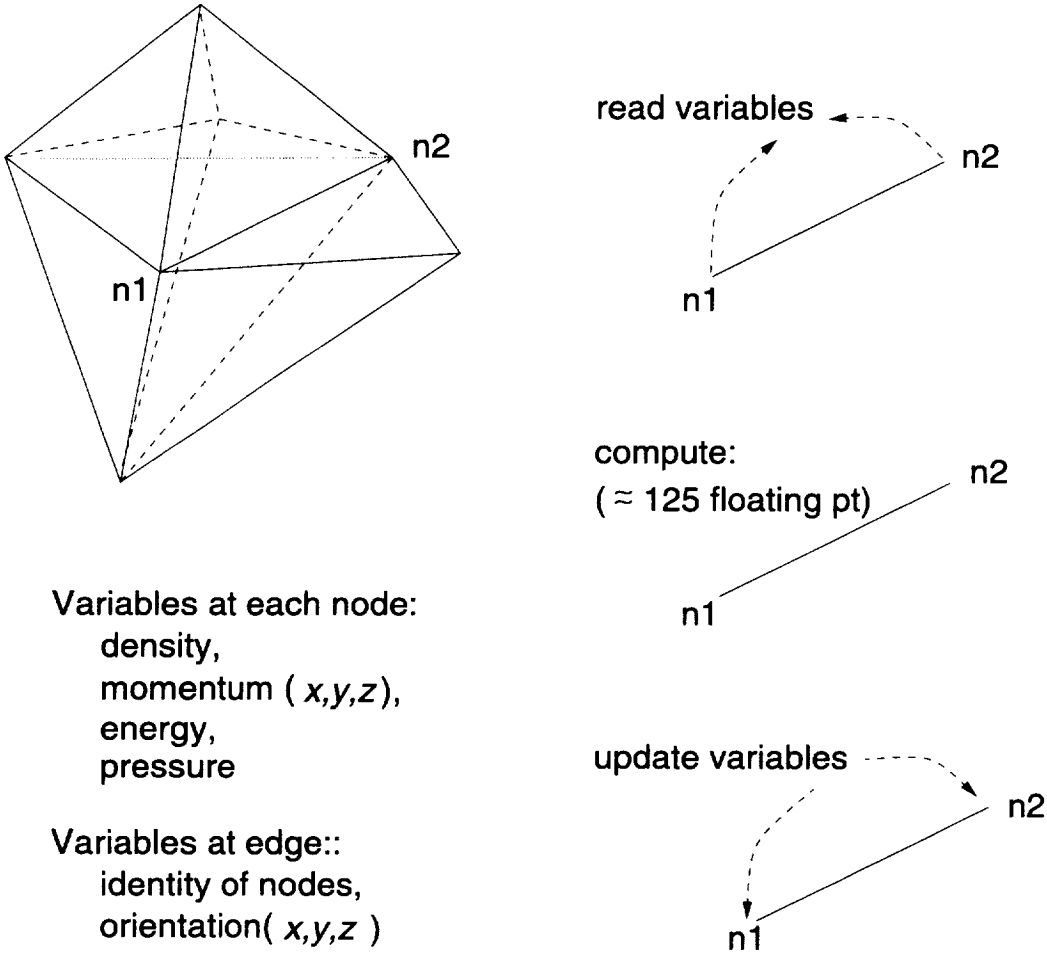


FIG. 4.2. *Computation in the Edge Based Loop*

```
   do i=1, totalNodes
     update variables
   enddo
enddo
```

4.1. Parallel Implementation. When executing the edge based loop in parallel, it is important to ensure that two threads do not attempt to update the same node at the same time. A simple way of ensuring

this is to color the edges of the graph so that no edges incident on same node have the same color. Once this has been done, all edges with the same color can be processed in parallel.

Although the problem of finding the minimum color edge coloring of a graph is intractable, our primary objective is to obtain a coloring with a reasonable number of colors. A simple greedy algorithm is fast and effective for our purposes. On our sample problem, which has average degree 14, our algorithm yields 24 colors.

In the pseudocode for the edge colored algorithm, given below, the compiler has to be told to parallelize the edge loop. This is because it has no way of knowing about the coloring, and cannot establish that it is safe to parallelize the loop just by looking at the code. The `C$TERA ASSERT PARALLEL` compiler directive is used for this purpose.

```
do i=1, totalNodes
  initialize variables
enddo

do cycle=1, totalCycles
   do i=1, totalNodes
     clear residuals
   enddo

   do i=1, totalColors
C$TERA ASSERT PARALLEL
      do (for each edge of color i)
        compute residuals
      enddo
   enddo

   do i=1, totalNodes
     update variables
   enddo
enddo
```

4.2. Performance of Colored Algorithm. The performance of the colored algorithm was measured by

- Varying number of streams (1 to 100)
- Varying number of processors (1 to 2)

The Tera compiler normally selects the number of streams for each parallel loop, based on estimated grain size and expected number of iterations. It is difficult to vary streams under programmer control, but can be done. The procedure is to insert the compiler directive `C$TERA USE` *n* `STREAMS`, before *every* loop in the program, and then recompile. This is a tedious and time consuming procedure, and we hope that Tera will provide a more convenient alternative in the near future.

It is possible to select any subset of processors to run on, using a bit vector supplied on the command line. Thus, on a four processor system `-p 0011` would use the 3rd and 4th processors only. This is a run time option: no recompilation is required.

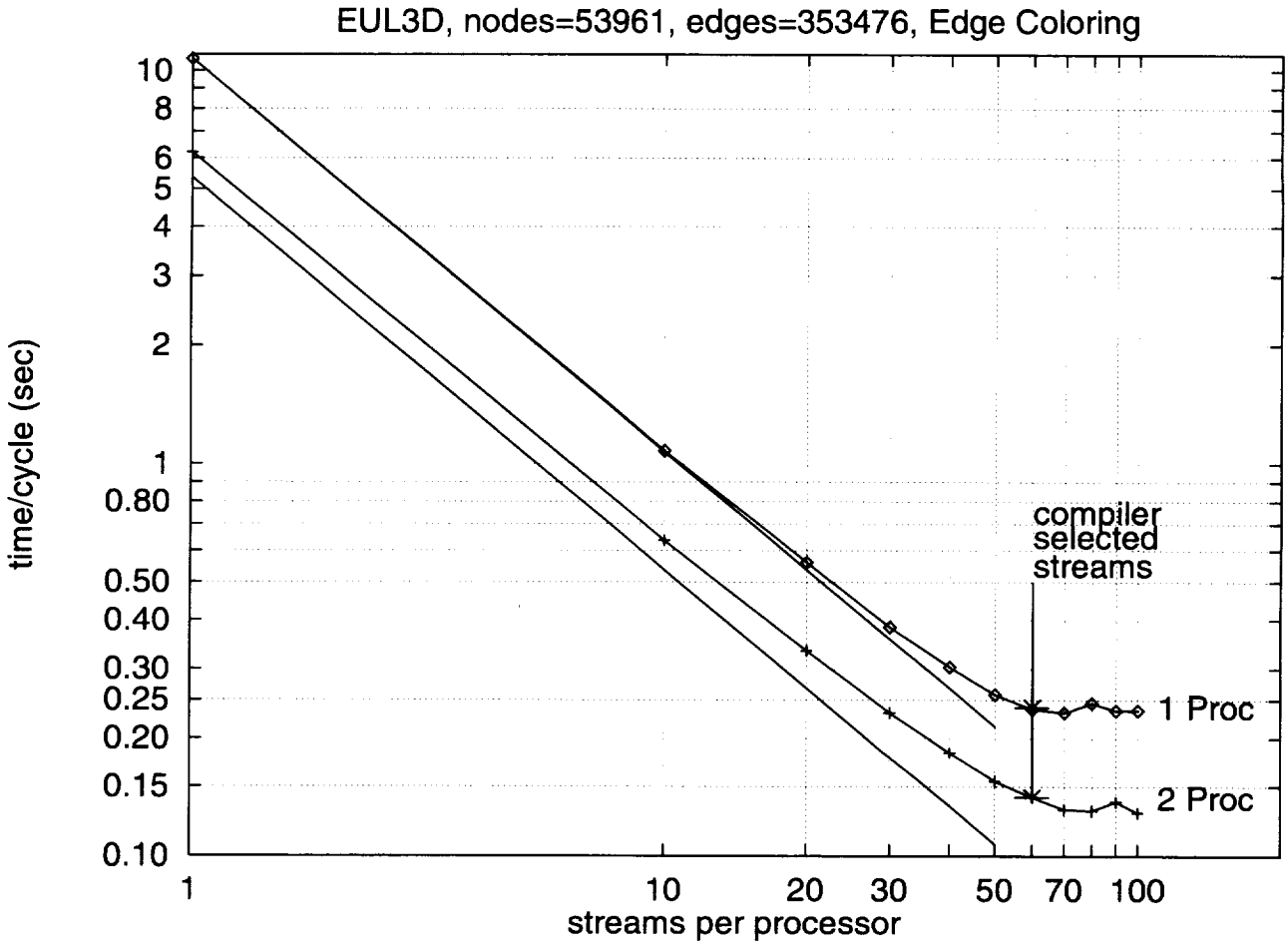


FIG. 4.3. *Performance of colored algorithm*

The plot in Figure 4.3 shows the performance of the colored algorithm as the number of streams is varied. The plot labeled 1 Proc shows the performance of the algorithm on one processor. The time per cycle drops very smoothly from 1 to 30 streams and flattens out at 60 streams. The speedup is about 40. The straight line next to this curve shows ideal speedup.

If we had not controlled the number of streams ourselves but had let the compiler do so, it would have selected 60 streams, a good choice in this case.

The plot labeled 2 Proc shows the performance of this algorithm on two processors. The straight line next to this plot is ideal speedup, based on the one processor-one stream time (the highest data point on the y-axis.) Time per cycle drops smoothly as before but there is a significant difference between the observed and ideal speedups. This is conjectured to be the result of network congestion, in part because the network at SDSC is missing 'wraparound' links.

In the 2 processor case the speedup continues up to 80 streams, showing that it is sometimes useful to override the compiler selected number of streams. We obtain nearly 5% improvement by doing so.

5. The Update Algorithm. The coloring algorithm presented above has two overheads:

1. the time required to actually color the edges and reorganize data (this is a one time cost, assuming the mesh is static), and

2. the overhead of executing the color loop (this includes synchronization overhead at the bottom of the loop).

The full/empty bits of the Tera permit very fine grained synchronization and thus let us eliminate these overheads. The *serial* algorithm can be run in parallel on the Tera, provided the compiler is warned about the sections of codes where it should ensure atomic updates. In this case the preprocessing step of coloring and reorganizing data is not required and the overhead of the color loop and its associated synchronization costs are avoided.

5.1. Using the Full/Empty bits. The behavior of the Tera's full/empty bits may be summarized as follows

- A synchronized write into a variable succeeds only if it is empty, when the write completes, the location is set full.
- A thread attempting a synchronized write into a full location will be suspended (by *hardware*) and will resume only when that location becomes empty.
- A synchronized read from a variable succeeds only if it is full, when the read completes, the location is set empty.
- A thread attempting a synchronized read from an empty location will be suspended (by *hardware*) and will resume only when that location becomes full.

There are several ways of using the full/empty bits, as detailed below.

5.1.1. Synchronized Variables. A variable can be declared synchronized thus:

```
sync real dw(100)
```

In this case, writes and reads to/from `dw()` will follow the full/empty rules given above. This approach requires careful thought and is not recommended for porting existing codes. However it may result in concise and elegant code when a program is written from the ground up with synchronized variables in mind.

5.1.2. Machine generics. Machine language instructions such as `WRITEEF()` ("wait until a variable is empty, then write a value into it, and set the full/empty bit to full") can be invoked from within Fortran or C. Thus, to ensure that the Fortran update

```
dw(i) = dw(i) - xincr
```

is handled properly when several threads are using the same value of `i`, we could use

```
call WRITEEF(dw(i), READFE(dw(i)) - xincr)
```

`WRITEEF, READFE, ...` are not compiled into function or subroutine calls–they become *individual* Tera machine instructions.

This technique is the most flexible and gives full control to the programmer, who has the option of using regular load/stores as well as full/empty bit controlled load/stores on a variable as and where he desires.

The disadvantage in this case is that code starts looking messy.

5.1.3. Compiler directives. Compiler directives can be used to make the compiler use full/empty bits to ensure correct updating. For example, in the following code fragment,

```
C$TERA UPDATE
 dw(i) = dw(i) - xincr
```

the directive instructs the compiler to insert appropriate machine instructions.

This is the cleanest solution as it requires no change to serial code and does not obfuscate the program text. This is the solution we have used. However this approach may not work in all situations.

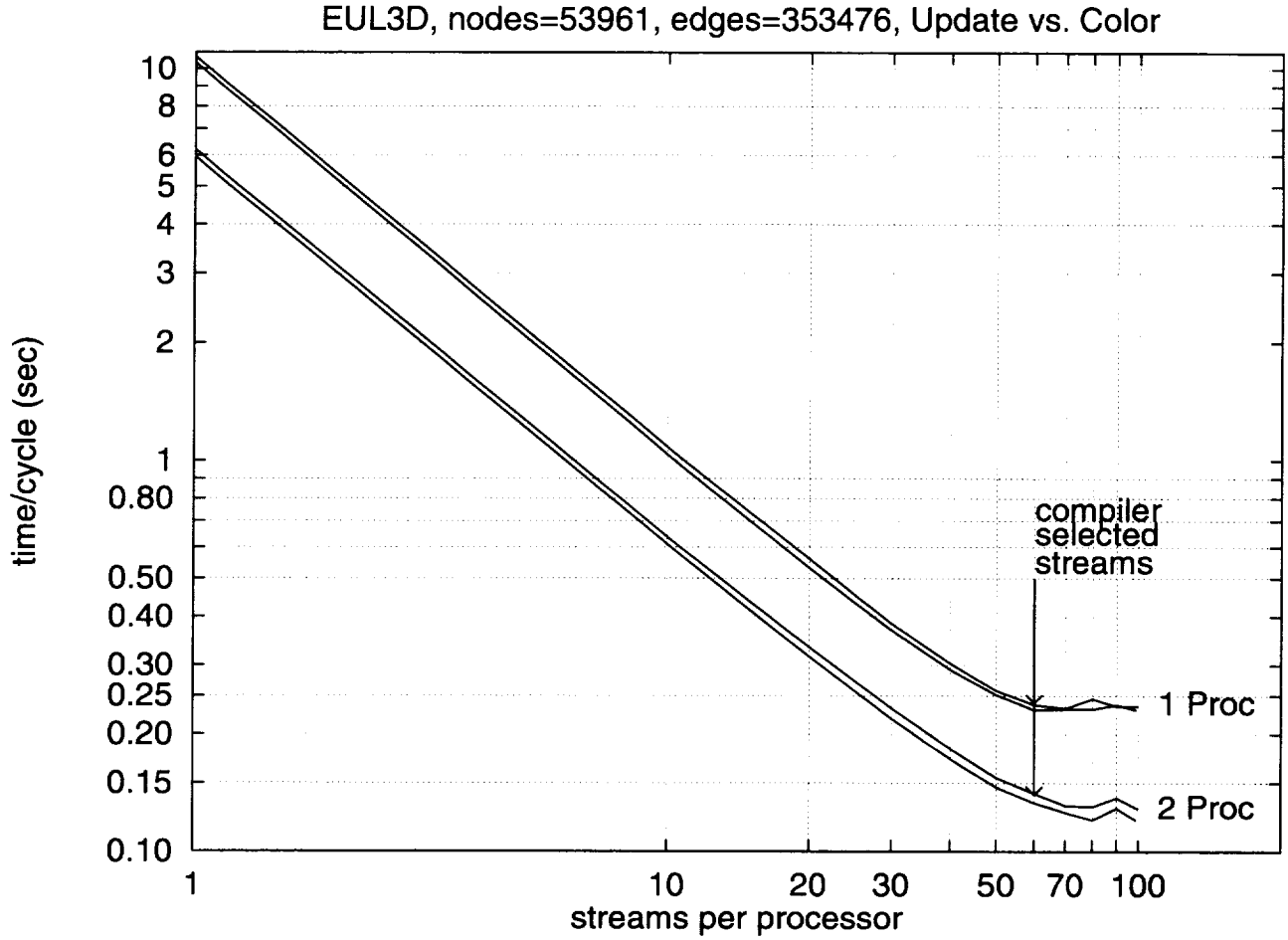


FIG. 5.1. *Update code vs. coloring: Absolute improvement.*

5.1.4. Compiler detection. It is also possible for the compiler to detect program statements where use of full/empty bits would be required and insert the required machine instructions. This is the least intrusive solution but, as in the update approach described above, may not work in all cases. We did not have time to experiment with this approach.

5.2. Performance of the Update Code. The improvement obtained by moving from the traditional edge coloring code to the update code is shown in Figures 5.1 & 5.2. Recall that the update code is just the *serial* code with the addition of a few compiler directives. These directives cause the Tera to use its full/empty bits to ensure correct updating. This eliminates the overhead of the edge color loop and its associated synchronization. Figure 5.1 shows that there is a consistent improvement for both one and two processors. The ratio of the run times for the two programs is shown in Figure 5.2, for both one and two processors. We can see a consistent 4 to 6 % improvement for 2 processors, over the range of 1–60 streams.

5.3. Stream Efficiency. A stream is a piece of hardware. It is interesting to explore how the stream efficiency varies as more and more streams are dedicated to our problem. Figure 5.3 shows our results.

In this figure, the plot labeled "1 Proc" shows

$$\frac{(\text{time for 1 processor, 1 stream})/s}{\text{time for 1 Processor, } s \text{ streams}}$$

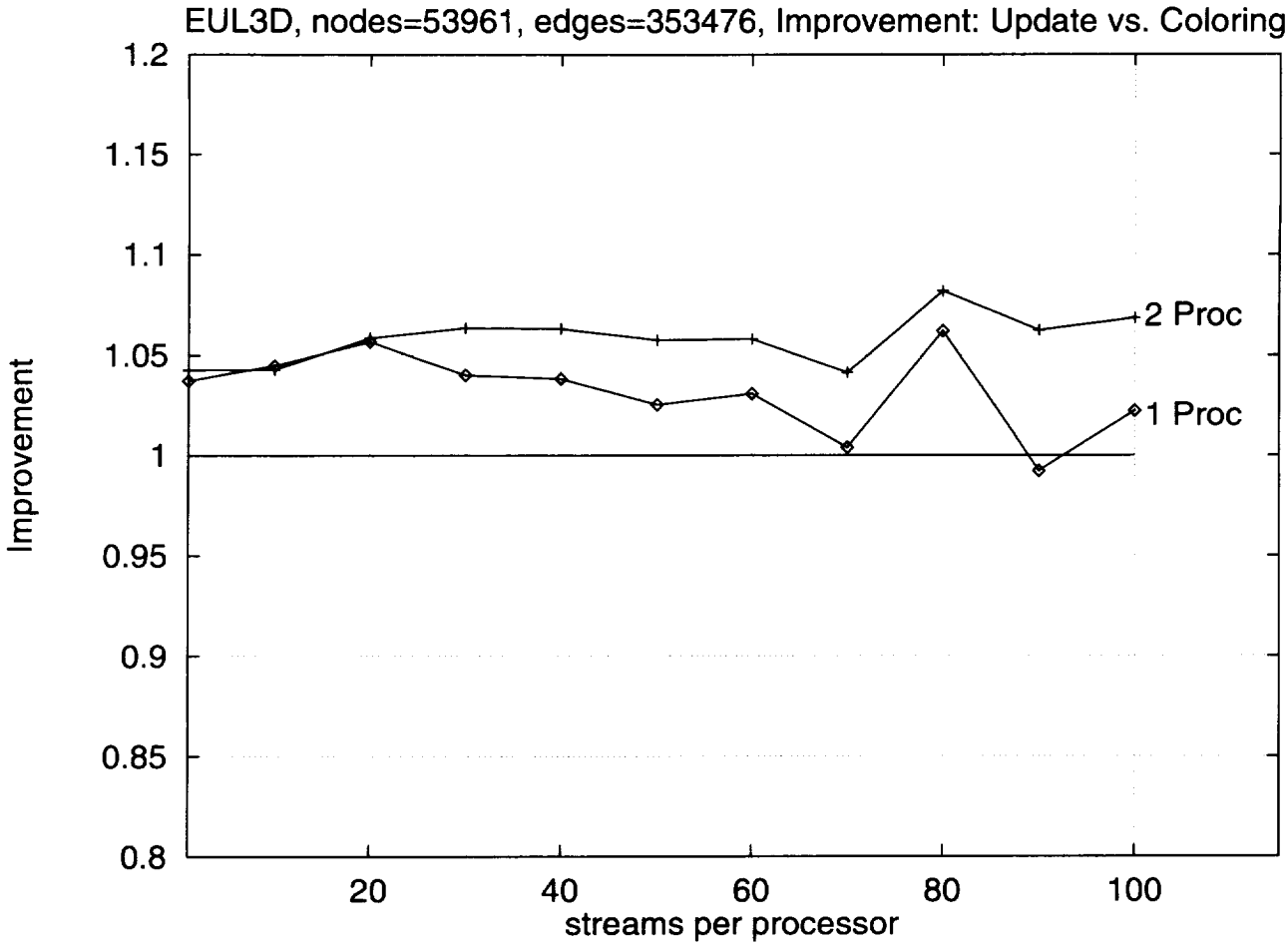


FIG. 5.2. *Update code vs. coloring: Relative improvement.*

The plot labeled "2 Proc" shows

$$\frac{\text{(time for 1 processor, 1 stream)}/2s}{\text{time for 2 Processors, } s \text{ streams per proc}}$$

We can see that stream efficiency is quite good on one processor. It is well above 90% for 1 30 streams and nearly 75% for 60 streams (the compiler selected number of streams for our code).

The gap between the 1 and 2 processor curves is significant and is presumably caused by the limitations of the current network.

6. Experiments with Grain Size. The "edge-based" loop in EUL3D is used in numerous other unstructured mesh problems. Other problems might have grain sizes very different from EUL3D. To get an idea of how performance would vary as the grain size changed, we artificially modified the EUL3D solver[2].

The original solver has 12 variables per node. We modified these to 6 and 22. We also modified the computations in the edge loop to roughly halve or double them.

The results of these experiments are summarized in Figure 6.1 and Table 6.1. The plots show that the speedup curves follow generally the same pattern. The small code, performance saturates somewhat earlier

[2]The experiments in this Section were suggested by David Keyes.

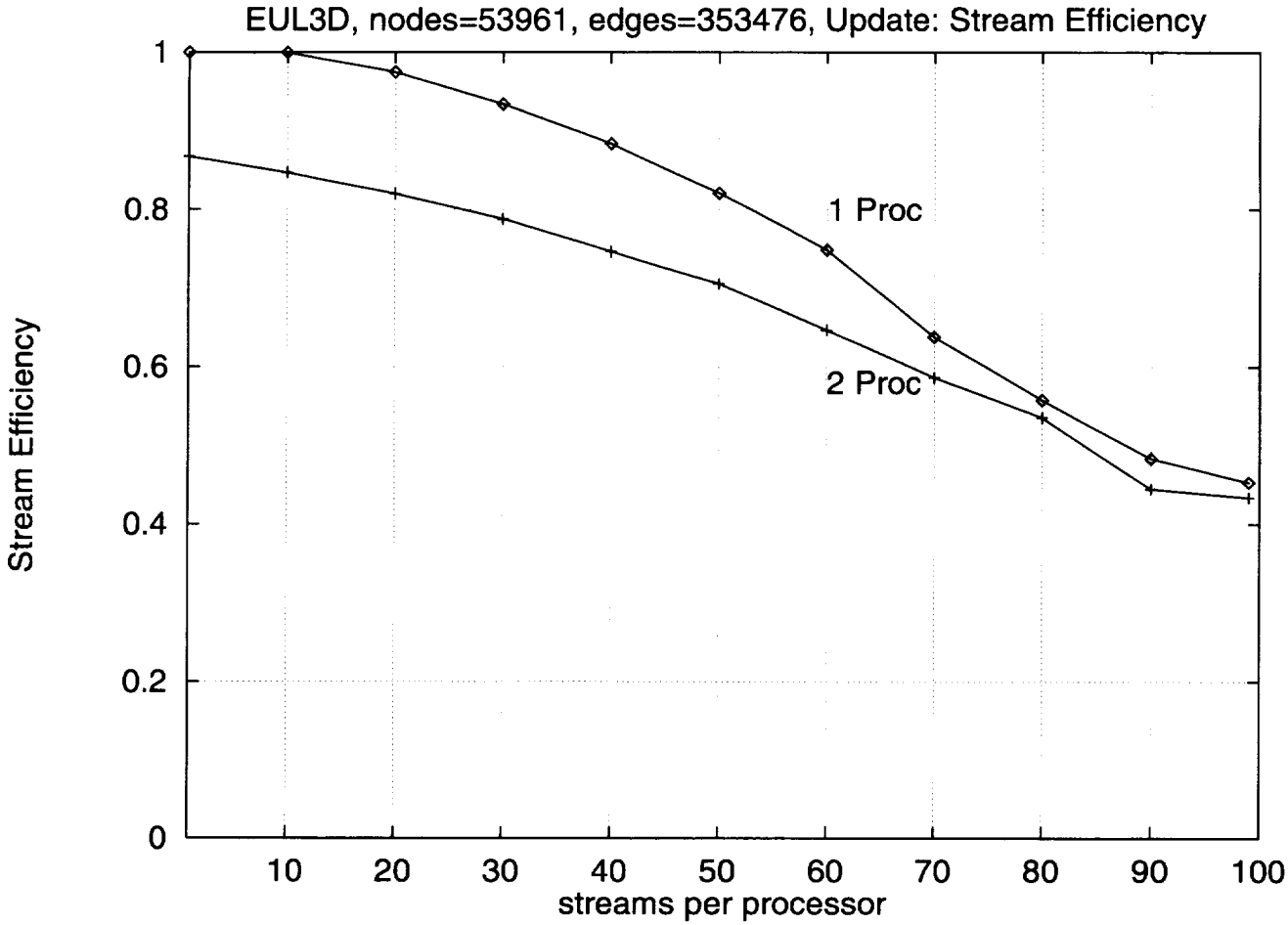


FIG. 5.3. *Stream Efficiency*

TABLE 6.1
Summary of Grain Size Experiments

Code Version	Vars/ node	MFlop/ cycle	MFlop/s (1 Proc)	MFlop/s (2 Proc)
Large	22	74.9	187	325
Original	12	48.7	212	406
Small	6	29.5	211	393
These measurements are at 255MHz.				

than the original code, presumably because there is less work per iteration. The large code's performance is more smooth and, for two processors, the speedup continues to 100 streams. However when looking at the delivered Megaflops per second, we see some dramatic decreases for the large code. These decreases are conjectured to be the result of network congestion or perhaps register overflow and deserve further study once a larger machine becomes available.

7. Conclusions. Our experience with the Tera has generally been positive. We were able to port an existing edge colored parallel code (previously run on the SPP-2000) by changing only the parallelization

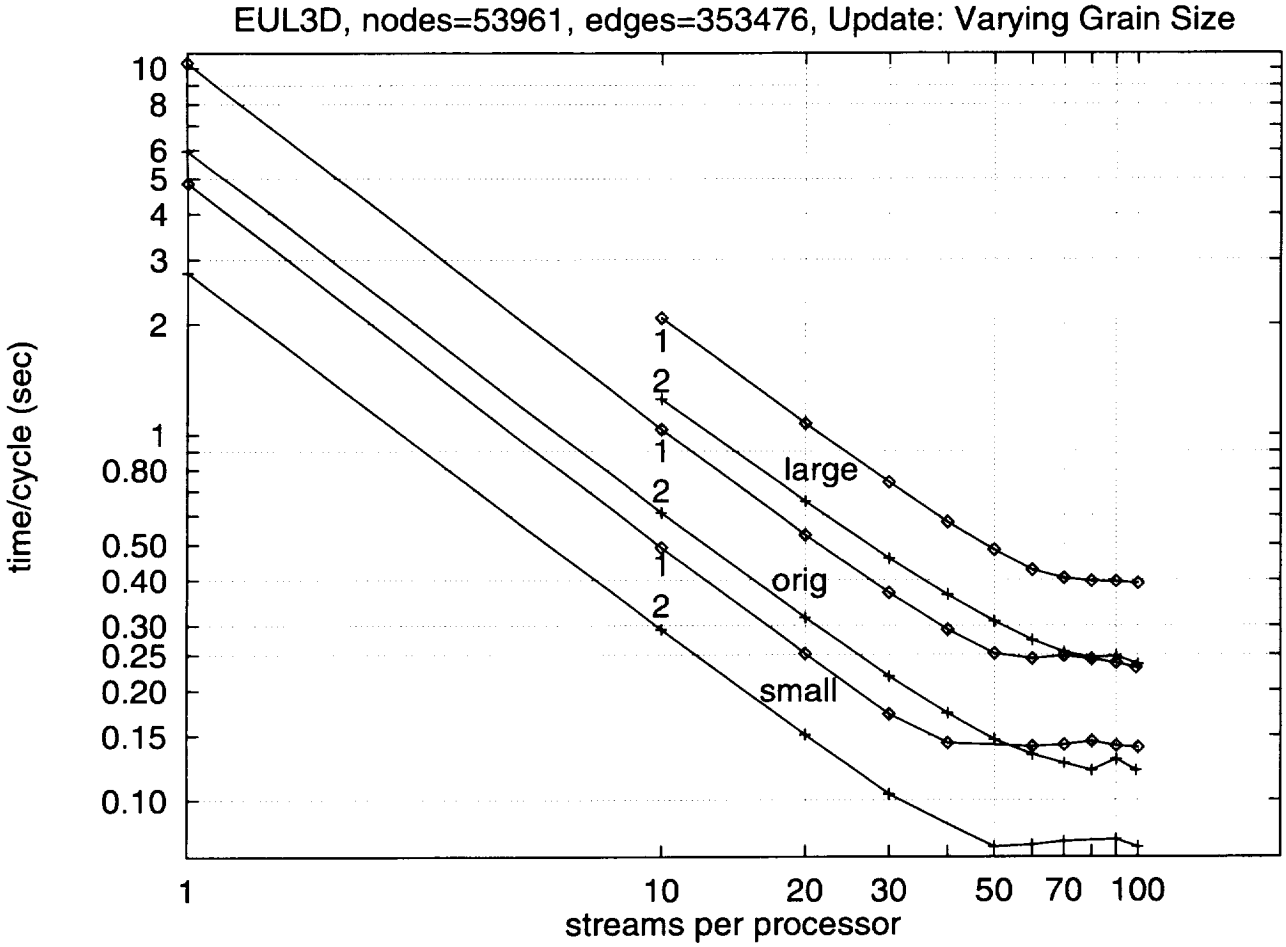


FIG. 6.1. *Effect of Varying Grain*

directives.

We also parallelized an existing serial code (previously run on workstations) on the Tera with the addition of a few compiler directives. In this case we invoked the full/empty mechanism of the machine and thus eliminated the overhead of the edge colored loop.

Both versions of our code were run on 1 and 2 processors. No changes will be required to run on any additional number of processors.

7.1. Problems Encountered. Two main problems were encountered during the course of this research on the Tera. Firstly, since the Tera stores everything in 64 bit words, there was a compatibility problem with the 32 bit integers used in our binary grid file. The obvious solution is to tell the compiler to use 32 bit integers throughout, but this creates a further problem because 32 bit loop indices confuse the compiler. A better solution is to rewrite the binary file so that all variables are 64 bits.

A second, and more aggravating, problem is that there is no way to control the number of streams at run time. For the researcher, such a facility would permit an evaluation of how performance varies with the number of streams. For the practitioner it may often be useful for squeezing out maximum performance for a single, important standalone application by allocating the maximum number of streams to it. At the present time experiments of the type detailed in this report require a recompilation for every change in the

number of streams. This is a tedious and annoying process.

7.2. Future Work. We have demonstrated the parallelization and performance of our code on the existing 2 processor Tera MTA at SDSC. Our primary aim for the future is to run on 4, 8 or 16 processor systems, as they become available, so as to provide a convincing demonstration of sustained Megaflop/s. To provide a more detailed evaluation, we also plan to run larger and smaller meshes.

The edge based loop used in EUL3D is at the heart of many other unstructured mesh algorithms. It will therefore be of interest to port other unstructured mesh problems to the Tera.

Cell based (as opposed to edge based) loops should be similarly easy to parallelize and need to be investigated.

Finally, we plan to port other non-uniform problems, such as multiblock. The Tera's insensitivity to memory access patterns will be a major asset for such problems.

8. Acknowledgments. We are grateful to Manuel Salas, Director ICASE, for his encouragement of this research and to Wayne Pfeiffer for arranging access to the Tera MTA at San Diego Supercomputing Center. John van Rosendale, David Keyes, Piyush Mehrotra and Tom Crockett provided valuable assistance. Allan Snavely, John Feo and Bracy Elton generously shared their knowledge of the MTA with us and lightened our burden considerably.

This research would not have been possible without the hospitality of David Chestnut, Oktay Baysal and Richard Barnwell at the Virginia Consortium of Science & Engineering Universities (VCES).

9. Web Sites of Interest.

`www.tera.com`
`www.sdsc.edu`
`www.icase.edu`

REFERENCES

[1] D.J. MAVRIPLIS, R. DAS, J. SALTZ AND R.E. VERMELAND, *Implementation of a parallel unstructured Euler solver on shared and distributed memory machines*, The Journal of Supercomputing **8**, No. 4 (1995), pp. 329-344.

[2] R. ALVERSON, D. CALLAHAN, D. CUMMINGS, B. KOBLENZ, A. PORTERFIELD AND B. SMITH, *The Tera Computer System*, Supercomputing (1990), pp. 1-6.

[3] G. ALVERSON, R. ALVERSON, D. CALLAHAN, B. KOBLENZ, A. PORTERFIELD, AND B. SMITH, *Exploiting heterogeneous parallelism on a multithreaded multiprocessor*, Supercomputing (1992), pp. 188.

[4] S. BALAY, W.D. GROPP, L.C. MCINNES AND B.F. SMITH, PETSc home page, `www.mcs.anl.gov/petsc`, 1998.

[5] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK AND V. SUNDERAM, PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

[6] *PVM home page,* `www.epm.ornl.gov/pvm/pvm_home.html`

[7] W. GROPP, E. LUSK AND A. SKJELLUM, Using MPI, MIT Press, 1994.

[8] A. SNAVELY, L. CARTER, J. BOISSEAU, A. MAJUMDAR, K.S. GATLIN, N. MITCHELL, J. FEO AND B. KOBLENZ, Multi-processor Performance on the Tera MTA, to be presented at SC98, `www.sdsc.edu/~allans /SC98-MTA/abstract.html`

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	Contractor Report

4. TITLE AND SUBTITLE
The Tera Multithreaded Architecture and Unstructured Meshes

5. FUNDING NUMBERS
C NAS1-97046
WU 505-90-52-01

6. AUTHOR(S)
Shahid H. Bokhari
Dimitri J. Mavriplis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 403, NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Interim Report No. 33

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/CR-1998-208953
ICASE Interim Report No. 33

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified Unlimited
Subject Category 60, 61
Distribution: Nonstandard
Availability: NASA-CASI (301)621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Tera Multithreaded Architecture (MTA) is a new parallel supercomputer currently being installed at San Diego Supercomputing Center (SDSC). This machine has an architecture quite different from contemporary parallel machines. The computational processor is a custom design and the machine uses hardware to support very fine grained multithreading. The main memory is shared, hardware randomized and flat. These features make the machine highly suited to the execution of unstructured mesh problems, which are difficult to parallelize on other architectures.

We report the results of a study carried out during July-August 1998 to evaluate the execution of EUL3D, a code that solves the Euler equations on an unstructured mesh, on the 2 processor Tera MTA at SDSC.

Our investigation shows that parallelization of an unstructured code is extremely easy on the Tera. We were able to get an existing parallel code (designed for a shared memory machine), running on the Tera by changing only the compiler directives. Furthermore, a *serial* version of this code was compiled to run in parallel on the Tera by judicious use of directives to invoke the "full/empty" tag bits of the machine to obtain synchronization. This version achieves 212 and 406 Mflop/s on one and two processors respectively, and requires no attention to partitioning or placement of data—issues that would be of paramount importance in other parallel architectures.

14. SUBJECT TERMS
parallel computing; multiprocessors; supercomputing; multithreaded architectures; Tera computer; unstructured meshes

15. NUMBER OF PAGES
23

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		