

SANDIA REPORT

SAND97-0134
Unlimited Release
Printed January 1997

Final Report for the Tera Computer TTI CRADA

George S. Davidson, Constantine Pavlakos, Claudio Silva

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND97-0134
Unlimited Release
Printed January 1997

Final Report for the Tera Computer TTI CRADA

George S. Davidson
Constantine Pavlakos
Claudio Silva
Computer Architectures Department

Sandia National Laboratories
Albuquerque, NM 87185-0318

Abstract

Tera Computer and Sandia National Laboratories have completed a CRADA, which examined the Tera Multi-Threaded Architecture (MTA) for use with large codes of importance to industry and DOE. The MTA is an innovative architecture that uses parallelism to mask latency between memories and processors. The physical implementation is a parallel computer with high cross-section bandwidth and GaAs processors designed by Tera, which support many small computation threads and fast, lightweight context switches between them. When any thread blocks while waiting for memory accesses to complete, another thread immediately begins execution so that high CPU utilization is maintained. The Tera MTA parallel computer has a single, global address space, which is appealing when porting existing applications to a parallel computer. This ease of porting is further enabled by compiler technology that helps break computations into parallel threads.

DOE and Sandia National Laboratories were interested in working with Tera to further develop this computing concept. While Tera Computer would continue the hardware development and compiler research, Sandia National Laboratories would work with Tera to ensure that their compilers worked well with important Sandia codes, most particularly CTH, a shock physics code used for weapon safety computations. In addition to that important code, Sandia National Laboratories would complete research on a robotic path planning code, SANDROS, which is important in manufacturing applications, and would evaluate the MTA performance on this code. Finally, Sandia would work directly with Tera to develop 3D visualization codes, which would be appropriate for use with the MTA. Each of these tasks has been completed to the extent possible, given that Tera has just completed the MTA hardware. All of the CRADA work had to be done on simulators. Nevertheless, the codes developed at Sandia National Laboratories helped Tera improve the performance of their compilers, and helped identify and fix problems associated with compiling real, large industrial codes. At the same time, Sandia has carefully examined the MTA machine for graphics applications and has determined that it is very promising in this area. Jointly, Sandia and Tera expect that the actual hardware will perform very well on 3D graphics applications.

Introduction

This report describes research in high performance computing jointly performed by Tera Computer Corporation and Sandia National Laboratories, which was funded by a Technology Transfer Initiative (TTI) Cooperative Research and Development Agreement (CRADA). Tera Computer developed a new computer architecture and Sandia National Laboratories developed and modified existing large scale codes for evaluation with the new architecture. This report will describe the Tera architecture, discuss the Sandia applications, and include a detailed discussion of the potential for using the Tera computer for graphics applications.

The Tera Computer Architecture

Hardware

The Tera computer system is a shared memory multiprocessor. In the shared memory programming model, the performance of the system does not depend on the placement of data in memory. Tera implements such a shared memory programming model, given sufficient parallelism in the tasks being run.

The system can accommodate up to 256 processors. The sizes to be offered for sale will be 16, 32, 64, 128 and 256 processor systems, see Table 1. Availability of a limited number of systems is expected in the second half of 1996.

The system runs stand-alone and requires no front end. Network connection to work stations and other computer systems is accomplished via 32 or 64 bit HIPPI channels. All data path widths are 64 bits including the processor-network interface.

Table 1, System characteristics of the Tera multi-threaded computer.

Processors	16	16	256
Peak Gflops	16	64	256
Memory, Gbyte	16-32	64-128	256-512
HIPPI channels	32	128	512
I/O Gbyte/s	6.2	25	102

Tera processors are multi-threaded. Each processor switches context every 3 ns cycle among as many as 128 distinct instruction streams (hardware threads), thereby hiding up to 128 cycles (384 ns) of memory latency. In addition, each stream can issue as many as eight memory references without waiting for earlier ones to finish, further augmenting the memory latency tolerance of the processor.

A stream implements a load/store architecture with three addressing modes and 31 general-purpose 64-bit registers. The instructions are 64 bits wide and can contain three operations: a memory reference operation (M-unit operation or simply M-op for short), an arithmetic or logical operation (A-op), and a branch or simple arithmetic or logical operation (C-op).

The clock speed is nominally 333 Mhz, giving each processor a data path bandwidth of a billion 64-bit results per second and a peak performance of one gigaflops. The peak memory bandwidth is 2.67 gigabytes per second, and the processor sustains well over 95% of that rate. See Table 2 for performance estimates on various common kernels.

The processors implement IEEE Standard 754 arithmetic using the 64-bit double basic format. Hardware support for infinity arithmetic and denormalized operands is provided. Addition, subtraction, multiplication and division and conversion to and from both signed and unsigned integer formats are supported directly. The type of integer rounding can be selected independent of the rounding mode. There are floating point multiply-add, multiply-subtract and multiply-subtract reverse operations. These operations round only once. Division and square root are accomplished with the help of Newton's method iterations but nevertheless always yield the required 1/2-ulp accuracy. Floating point maximum and minimum operations are also provided.

Support for fast, 128-bit "doubled precision" arithmetic is incorporated. In doubled precision two floating point numbers are used to represent a single value, yielding approximately twice the precision of ordinary floating point. There are instructions that help compute the doubled precision sum, difference, product quotient, and square root in a few instructions.

Every processor has a clock register that is synchronized exactly with its counterparts in the other processors and counts up once per cycle. In addition, the processor counts the total number of unused instruction issue slots (measuring the degree of under-utilization of the processor) and the time-integral of the number of instruction streams ready to issue (measuring the degree of over-utilization of the processor). All three counters are user-readable in a single unprivileged operation.

Eight counters are implemented in each of the protection domains of the processor. All are user-readable in a single unprivileged operation. Four of these counters accumulate the number of instructions issued, the number of memory references, the time-integral of the number of instruction streams and time-integral of the number of messages in the network. These counters are also used for job accounting. The other four counters are configurable to accumulate events from any four of a large set of additional sources within the processor including memory operations, jumps, traps and so on.

Table 2, Performance estimates for typical kernels on a 64 processor machine.

Kernel	Estimated time
Matrix multiply, 1K x 1K	50 ms
3D FFT, 256 x 256 x 256	63 ms
Sparse matrix-vector, 400M NZ	50 ms
Integer sort, 100M	36 ms

Interconnection Network

The interconnection network is a three-dimensional packet switched network nominally containing $p^{3/2}$ nodes, where p is the number of processors. These nodes are toroidally connected in three dimensions to form a $p^{1/2}$ -ary three-cube, and processor and memory resources are attached to some of the nodes. The latency of a node is three cycles: a message spends two cycles in the node logic proper and one on the wire that connects the node to its neighbors. A p -processor system has worst-case one-way latency of $4.5p^{1/2}$ cycles and an average latency of half that. A node has not six ports but four (five if a resource is attached), so x and y connections are alternately omitted in the cube depending on the parity of the z coordinate.

Each port simultaneously transmits and receives an entire 164-bit packet every 3 ns clock cycle. Of the 164 bits, 64 are data, so the data bandwidth per port is 2.67 GB/s in each direction. The network bisection bandwidth is p times this value. The network routing nodes contain no buffers other than those required for the pipeline. Instead, all messages are immediately routed to an output port. Messages are assigned random priorities and then routed in priority order. Under heavy load, some messages are derouted by this process. The randomization at each node ensures that each packet eventually reaches its destination.

The network has both parity checking and SECDED error correction. It will automatically correct any one bit error in the network, even a transient error, and will pinpoint the particular wire responsible for the error even if an intermittent connection is the cause. The interconnection network can be configured to bypass any portion of itself on a port-by-port basis, including the attached processor and memory resources.

Memory

The memory system is implemented as either $2p$ or $4p$ memory units distributed around the network, where (once again) p is the number of processors. Memory is implemented using 16 megabit DRAM chips. The memory units are interleaved 64 ways. Memory references by the processors are randomly scattered among all of the banks of all of the memory units except for instruction fetches that access two nearby memory units via a dedicated data path. Memory latency is independent of stride thanks to this randomization of memory addresses by the hardware.

The memory units can be addressed by 8-bit byte, 16-bit quarterword, 32-bit halfword or 64-bit word. A fetch-and-add operation is provided on words. In addition, every word has associated with it four additional bits of access state which, among other things, help implement lightweight synchronization operations. Waiting for synchronization is initially busy, but turns into non-busy waiting after a programmable number of attempts. At this time a trap occurs and the stream state is saved and enqueued at the target memory word. Subsequent references to that word are made to trap by another access state bit. Traps do not change instruction stream privilege in the processor.

Memory units are equipped with a single error correcting, double error detecting code. The access state is protected with its own error correcting code.

Input/Output

The maximum bandwidth in a p processor system is $200p$ megabytes per second in each direction via p duplex HIPPI channels. This does not represent a significant load on the interconnection network. The latency tolerant I/O processors that attach the HIPPI channels to the network can access data anywhere in the system, but typically data are transferred to or from operating system buffers located in memory which are then mapped into the address space of the user.

Maximum Strategy Gen5-XL RAID's are used, with a sustained bandwidth of about 130 megabytes per second each. At least $p/16$ disk arrays must be configured in a p processor system. The maximum capacity per disk array is about 360 gigabytes, so system disk capacity can approach $300p$ gigabytes.

The file system has the ability to distribute a single file across as many disk arrays as are attached to the system. Transfers take place between the disk arrays and buffers in I/O memory, and these buffers are then directly mapped into the address space of user application programs. In this way, copying is avoided. Since the buffers are held in shared memory, every processor can easily access them. Concurrent I/O arising from within parallel loops or elsewhere will be automatically parallelized by the Tera compilers and libraries by letting each thread access the buffers for data.

The Tera MTA supports HIPPI-PH, -FP, -LE, and -SC, and provides a complete set of HIPPI-based TCP/IP capabilities including sockets, telnet, the r-commands, and NFS. Other networking and remote file access capabilities such as DCE and archival storage facilities such as tape libraries and NSL-Unitree will be implemented as customer needs dictate.

Software

Operating System

The Tera operating system is a fully symmetric, distributed parallel version of the UNIX operating system, based on Berkeley sources and modified to use a highly concurrent microkernel developed by Tera. Checkpointing is provided and conforms with Posix P1003.1a draft 12. Each processor has multiple protection domains, allowing it to support as many as 15 distinct address spaces (and 15 tasks). Thus the whole system can run as many as $15p$ tasks truly in parallel. Each protection domain can contain a dynamically varying number of instruction streams, from one up to the maximum permitted on the processor (about 100). An instruction stream acquires more addressability as its privilege increases within a protection domain.

An instruction stream is dynamically created or destroyed using a single instruction, generated either by compiled code or by the run-time environment. If a protection domain needs more instruction streams it can acquire them from the processor in competition with other protection domains. The operating system is not involved at all in instruction stream acquisition or release.

The run-time environment normally multiplexes instruction streams among an unbounded number of threads as described below. If the run-time environment encounters insufficient parallelism to fully occupy its processor resources, it will surrender some of its protection domains to the operating system for allocation to other tasks. Conversely, as the run-time parallelism increases, additional protection domains can be acquired.

The Tera operating system is a first-class multi-user implementation of Unix in all respects. There is no front-end processor, and even modest versions of the hardware will service hundreds of users at one time. Large and small tasks will be run concurrently without needing to partition the system or manually intervene in any way. A two-tier scheduler is incorporated into the Tera microkernel; it provides better resource allocation to large tasks (those currently running on more than a single processor) via a bin-packing scheme, and schedules the smaller tasks using a traditional Unix approach.

Programming Model

Tera provides a thread-based programming model that permits a mixture of implicit and explicit parallelism in the same program. The virtual machine has an unbounded number of processors with uniform access to all memory locations. The programmer may specify an unbounded number of threads interacting via shared data structures. The runtime system will acquire physical resources from the operating system and use these resources to implement the virtual machine.

Besides providing a high level of classical single-thread optimization, Tera's compilers perform automatic parallelization of Fortran, C, and C++ source programs. Loop nests are restructured to enhance parallelism, with scalar expansion and array privatization used where beneficial. Whole program analysis and optimization are performed, including inline expansion and parallelization of loops containing function calls and I/O. A broad spectrum of linear recurrences and reductions are parallelized automatically, including even those reductions with unknown dependencies (histogramming causes dependencies of this kind).

The programmer can insert pragmas and directives into the program to help the compiler in automatic parallelization. The parallelism that the compiler discovers and exploits supplements whatever is explicitly generated by the user. Inner-loop parallelism is easy for the compilers to discover and manage, whereas outer-loop parallelism may only be evident to the programmer.

Tera's shared memory model is release consistent, which means the memory behaves the same as a sequentially consistent memory model, as long as data access is properly synchronized (the program is data-race-free) and the synchronization is performed only using full/empty bits or volatile variables.

The overhead involved in instruction stream creation and termination as generated by the compilers to exploit parallel loops is around four instructions plus two per created thread, not counting additional instructions to load the new thread's registers with useful data. Stack space is shared by these threads. Explicit user threads incur higher creation and termination overhead, 50 to 100 instructions, but allow unrestricted procedure calls that precludes stack sharing. Any of these varieties of thread can synchronize and can pass data as often as once per instruction.

Languages and Compilers

Fortran 77 is supported, with a high degree of Cray compatibility. The MIL-STD intrinsics and a set of parallel directives are also provided. C is ANSI compliant, and except for templates and exceptions, C++ conforms to the ARM. Both C and C++ include pragmas for parallelism. All three languages may call each other freely. The compilers are implemented with a front-end for each language driving a common back-end.

Tera's language implementations incorporate extensions to allow parallel programming and access to the hardware full/empty bits. A new statement type is added as an extension that creates a new thread of control and binds it to a block of statements. The runtime environment allows an unbounded number of threads and performs all necessary resource management. Two type-qualifiers (similar to "volatile") are added to describe how a particular memory access interacts with the full/empty bit. These provide complete access to the hardware's synchronization capabilities.

A macro-assembler is provided with built-in support for high-level language calling conventions and data types. The assembler uses a Scheme interpreter (Scheme is a dialect of Lisp) as its core and the macro language is essentially Scheme extended with primitives typical of two-pass assemblers.

Development Tools

Tera's debugger, tdb, can manipulate multiple threads of control. Traditional source-level thread-by-thread debugging similar to gdb is provided; tdb also has the ability to organize threads into groups and apply scheduling operations to those groups. The debugger provides an unlimited number of very inexpensive data breakpoints (watchpoints) as well as the traditional program breakpoints. Conditional breakpoints are evaluated by the thread which triggers the break, and other threads are not affected unless the program is stopped. The `duel` command in tdb allows the selective display of data including nonlinear data structures and data distributed across stack frames of multiple threads, whether blocked, runnable, or running.

Tera provides a trace-based performance analysis system. The compilers can automatically provide instrumentation points, e.g. at function boundaries, and the programmer can insert additional ones. The time, the processor utilization and the average dynamic concurrency level are all immediately available from the hardware counters.

Like the compilers, the system development tools execute on the Tera system. Both "make" and the compiler are parallel applications. The compiler automatically collects interprocedural summary information and performs data dependence analysis to implement incremental compilation and linkage based on the semantic content of program changes rather than mere timestamps. The edit-compile-debug development cycle is accelerated by these features. The interprocedural dependence data are stored in a database; Tera plans to make this information available to the programmer via a source level browser at a future date.

Essentially all machine operations are available via intrinsic functions which are compiled into single machine operations and scheduled with the rest of the generated code. These functions, and the high level of compiler optimization, all but eliminate the need for assembly language programming.

Facilities and Maintenance

The system power dissipation will be approximately 6KW per processor. Disk arrays will add another 4 KW each. The system is water cooled, with one cooling distribution unit (CDU) for every 32 processors.

A 16 or 32 processor system will occupy a 6' by 5' footprint, and a 64 processor system will occupy a 6' by 10' footprint. CDUs are 3' by 7' and can be placed along a wall. The Uninterruptible Power unit footprint is 3' by 6' and the RAID footprint is approximately 3' by 3'.20

The interconnection network hardware allows reconfiguration to bypass faulty components. This capability can be used to allow scheduled hardware maintenance at the convenience of the customer. The system is completely scannable, i.e. every logic flip-flop is readable and writable. Test vectors for the system will allow rapid problem identification and correction. In the field, the scan system and its test vectors will support a level of automatic diagnosis that will let the system be reconfigured and brought back into operation with a minimum of delay, which will provide exceptional availability.

Software Research

Sandia identified three areas where cooperative research with Tera could benefit our programs. Three tasks were specified covering the work Sandia National Laboratories would undertake. Each of these tasks required collaboration with Tera for completion. The first task was to transform CTH, a shock physics code used at Sandia for weapon safety calculations, to perform optimally on the Tera architecture. The second task was to develop a manufacturing application that required the kind of performance the Tera machine would offer. This application was a robot path planning code, which would need to be further developed and then parallelized for the Tera computer. The third task was to jointly explore how well the Tera computer could provide high performance scientific visualizations. A common thread between all of these tasks was to provide the compiler writers at Tera early access to significant application codes, which could be used to test the robustness of the compilers.

Much of the software work could be started on workstations at Sandia. However, actual performance could only be evaluated on the prototype parallel computer at Tera. Because the actual hardware was just being completed as this CRADA ended, the tasks could not be fully completed. However, good progress was made by using simulators and by extrapolating performance measured on other parallel computers.

CTH

The main objective of the research involving our shock physics code, CTH, and the Tera computer was to evaluate how well the native compilers could parallelize the code, without hand tuning. We wished to compare that performance to what we were able to do on our Paragon computer using considerable hand tuning. We intended to aggressively tune the code for parallel performance on the Tera MTA. Again, this particular aspect of the research depended on access to MTA hardware, and was thus never completed (CTH is so large that it was useless to attempt to use the simulator instead of waiting for real hardware). However, several benefits came from this research, even if the main objective was never achieved.

Tera has licensed the CTH code from SNL and used it to tune the Tera Fortran compiler for high performance. This was important to Tera because CTH is typical of several important hydrocodes in use at the national laboratories. In the course of this compiler work, several problems were discovered and fixed, which resulted in a more reliable and robust compiler.

SANDROS

The serial version of the robot path planning code was completed in the first year. In the second year it was parallelized for a shared memory multiprocessor. That implementation on a four processor SGI Skywriter did not precisely duplicate the conditions of the Tera MTA computer, however it showed that SANDROS could be parallelized effectively using a shared memory paradigm. From this preliminary parallelization, we were able to postulate that it would successfully parallelize on the Tera computer, and would have significantly better performance than the SGI version.

Execution profiles of SANDROS execution indicated that the pairwise distance calculation accounted for about 80% of the cpu time in a typical problem. This routine is particularly amenable to parallelization, since each pair of objects can be assigned to a processor. Although a domain decomposition of this type is not particularly scalable, it works well for small multi-processor machines. Additional parallelism is expected to be available on the MTA by taking advantage of its ability to parallelize inner loops (the SGI can parallelize outer loops only).

The SGI implementation used parallelizing functions provided by the SGI C compiler to parallelize the pairwise distance calculations. On the 4 processor Skywriter and a problem with 16 pairs of objects, we were able to achieve a parallel efficiency of 92%. This efficiency should increase for larger problems. The efficiency should also be better using the faster MTA interconnection network on the Tera computer.

In addition to the SGI implementation, we also developed a version of the code for a network of workstations. This PVM based implementation did not work particularly well on our network. Apparently, the overhead associated with passing large amounts of data over a thin-wire ethernet is too great for a tightly-coupled application such as SANDROS. Using 8 processors on a problem with 16 pairs of objects produced only a 10% speed increase.

Further work on SANDROS was deferred until actual MTA hardware was available, so final performance results could not be measured in the period of this CRADA.

Volume Rendering

In the first year we identified and evaluated visualization codes suitable for the Tera machine. We were especially interested in parallel volume visualization codes because no existing machine had sufficient performance to meet our need for real-time volume renderings. We decided to work closely with the researchers who had helped us develop a Parallel Volume Rendering (PVR) code for the Paragon. Claudio Silva modified the PVR code that was developed for the Paragon and ran several tests using the MTA simulator supplied by Tera. In the course of these tests several compiler problems were corrected and we learned a great deal about generating parallel code for the MTA. Additionally, Tera personnel got involved to help produce meaningful results from the MTA simulator runs. Claudio's report, together with the results produced by Tera personnel, constitute the next and final section of this CRADA report.

Evaluation of the MTA for Parallel Volume Rendering

The overall goal of our work was to study the suitability of the Tera supercomputer architecture for real-time graphics applications. Our primary interest lay in volume rendering applications. In volume rendering, a scalar field, such as the densities of an MR scan of a human head, or the pressure of a computational fluid dynamics simulation are rendered by mapping the properties of the field to optical properties of clouds. Volume rendering consists of rendering these clouds by using simplifications of the equations that describe light scattering inside and on the surface of the cloud.

The choice of initially concentrating our efforts on volume rendering, instead of surface graphics, was based on several factors. First, the geometry of volumes makes them very large, on the order of several megabytes to gigabytes. Second, even with the simplifications, the light scattering simulations are very expensive, and the color mapping of volumes is a time consuming and nontrivial process that needs to be performed interactively to be effective. Third, several datasets of interest have "soft boundaries", with no abrupt changes in density that could be used to create a "hard surface" representation (e.g., basically an iso-surface). In these cases, effective visualization can only be performed with the use of motion in a highly interactive process. Finally, surface graphics hardware acceleration is well developed, with several effective commercial products, and can be performed in real-time for relatively large polygonal models. Hardware-accelerated volume rendering is still in its infancy with no suitable commercial hardware currently available.

In developing the Cube4 hardware architecture for volume rendering, Pfister and Kaufman [Cube4 '96] point out the necessary computational requirements of a real-time volume rendering machine. They emphasize the need for balance between computational power and memory bandwidth in order to achieve high rendering rates. They point out that the raw memory bandwidth is a major bottleneck, because each voxel needs to be fetched, classified and shaded at several frames per second. Note that traditional architectures, which use large caches, are not well suited for volume rendering because high bandwidth is only achieved between the CPU and the cache, but in general each voxel is only touched once (or at most a few times with high temporal locality), thus minimizing cache efficiency. With its multiple pipelined memory modules and multi-threaded architecture (specially designed to hide the memory latency), the Tera machine seems like the ideal computer for volume rendering.

Overview of the Conducted Research and Accomplishments

Our initial plan was to port a pre-existing volume rendering code to the Tera machine and study its performance, possibly optimizing the code to achieve optimal performance.

The original schedule consisted of first porting the code to the Tera simulator (called "Zebra") at Sandia while the final assembly of the actual machine was being performed. Then, we would perform a detailed performance analysis on the actual machine.

We ported the volume rendering code to the Tera simulator, instrumented it for performance analysis, and hand parallelized it with the addition of "future" variables and statements in the code. Several runs on Zebra have been performed. Unfortunately, our resource and time constraints, combined with the unavailability of an actual system, made it very difficult for us to make a complete assessment of the actual performance of our code on the Tera architecture. A variety of problems were encountered in trying to use the MTA simulator, including very slow turnaround on simulator jobs, somewhat due to inexperience with how best to make use of the simulator. This inexperience, together with unknown simulator quirks pertaining to what systems it can or cannot run on properly, resulted in a large number of ineffective simulator runs (as discussed below), before truly meaningful results were produced.

Problems Encountered

Documentation

One of the major problems throughout our work has been the extremely precarious documentation on the simulator system. Tera was notified that the Zebra simulator had no documentation, in the software distribution (even though the distribution takes over 100M of disk space).

The available documentation consists of a small set of conference papers and draft internal papers, and a couple of unfinished manuals. With the direct help of TERA's research and development personnel, we were able to proceed.

Compiler bug

Our volume rendering code (the first real application we tried to run on the simulator) made a compiler bug surface. The compiler gave no warning or sign of trouble during compilation, but the Zebra simulator would stop the execution of the renderer after just a few minutes.

The problem turned out to be the lack of initialization of a *static int double* array inside a function. Without proper initialization, garbage was placed in a floating point register. Moving the array definition outside the function solved the problem.

Debugger availability

In trying to discover the source of the simulation crash described above, the issue of the availability of a debugger surfaced. The technical staff at Tera was extremely helpful and tried to send us their debugger, which is based on the public domain GNU debugger. Unfortunately, our CRADA contract says nothing about the debugger; so concerns by the Tera lawyers prevented sharing their version of gdb with Sandia.

Speed and accuracy of the Zebra simulations

Zebra is an instruction-level simulator. It is supposed to accurately simulate all the facets of the Tera machine, including the multi-threading schedule, memory latencies, multiple processors, etc. It should be the ideal development environment; unfortunately, instruction-level simulation is extremely slow.

Because Zebra is so slow, only very small renderings can be done. This in turn generates a technical problem: because Zebra tries to accurately simulate the scheduling overhead (which is supposed to be around 200 instructions), and our problem size is so small, this overhead actually seems to interfere with our problem, making it impossible to generate accurate timings. This was ultimately overcome with help from Tera personnel who helped isolate certain sections of our code for analysis.

Another problem is that Zebra only runs on the Sparc. The fastest Sparc at our disposal was a Sun Ultra-Sparc1, which is fast enough for most purposes, but it took over a week to render our small 32-by-32-by-32 sphere at 64-by-64 resolution, with 64 streams in Zebra's "fast" mode. Again, we were able to improve on this turnaround in the latter stages of our work by isolating our analysis to smaller sections of our code, per guidance from Tera.

Another problem with the slow speed of the simulations was that the long run times exposed our simulations to possible interruptions (machine reboots and/or crashes).

Finally, the usual compile, run, analyze, improve programming cycle was severely handicapped by the slow Zebra speed.

Tera simulator code portability

The Tera simulator code only works on Sparcs, and seems to be selective about which version and which operating system it can use. At Sandia and Stony Brook, we have access to a large number of multiprocessor Sparcs (4-processor Sparc20's and Sparc1000), but we were unable to run the simulator on most of these. The only machine we were able to use consistently for running the simulator was an Ultra-Sparc1, running at 167MHz (with Solaris 2.5). Unfortunately, this Solaris machine would not produce anything but erroneous results. It was only last ditch efforts to run on a SunOS machine at Stony Brook which finally produced meaningful results which corroborated results.

Code development and modification

Overall, only minor code development and modification had to be performed. One was the introduction of instrumentation function calls at select locations, for example we placed the Tera runtime “read counters” routines in the rendering code.

In order to test the Tera parallel execution and the use of “futures”, we changed our rendering code. The low-level rendering functions consist of several nested loops, which perform ray casting along rays emanating from each pixel on the screen. Each ray can be handled independently, but the overhead in initiating a thread for each ray seemed too high. Instead, a thread was created for each scan-line. As far as the output images are concerned, the output of the new code with futures is exactly the same as the old, sequential code.

Some other minor modifications, like using a zero-order interpolation for improving the simulator run times, were also performed.

In the latest stage of our efforts, Preston Briggs of Tera modified the code slightly, by reducing the computations to a single frame and isolating the timing computations to the actual rendering portion of the code, all in order to improve simulation turnaround. This version of the code was ultimately used to produce our most meaningful results.

Initial Results

All the performance numbers presented here are for rendering a 32-by-32-by-32 volumetric sphere at 64-by-64 resolution. This problem size was too small to be realistic, but was at the extreme limit of what we could simulate with Zebra. Initial results were obtained using Zebra release 12, running on an Ultra-Sparc1 with Solaris.

Although only our latest sets of results were meaningful, we include all results here for completeness, to document the entire process.

Three different versions of the volume rendering code were used. One version was a plain sequential version (i.e., the parallelization is completely left to the Tera compiler -- see Table 3.). The second version was hand parallelized (using “futures”) to make a different stream for every scan-line (see Table 4.). The third version was the version produced by Preston Briggs of Tera, which was a perturbation of the second version.

Initial timing tables generated for the sequential and hand parallelized versions are shown below:

Streams	1	4	16	64
Time	8.5M	23.5M	91.5M	366M
Phantoms	2.8M	10896	11903	14744
Ready	0	60M	1.3B	22B
Total Time	45M	120M	481M	1.9B

Table 3. Run times, using Zebra release 12, for one, four, sixteen and sixty-four streams with no hand tuning, using only the parallelism found by the compiler.

Streams	1	4	16	64
Time	9M	16M	56.5M	220M
Phantoms	3.2M	11337	11903	14744
Ready	0	41M	820M	14B
Total Time	47M	85.7M	306M	1.1B

Table 4. Run times, using Zebra release 12, for one, four, sixteen and sixty-four streams with hand coded “futures” for each image scan line and including parallelism found by the compiler.

For these simulations, the following details should be noted:

- 1) Zebra was forced to use a different number of streams by using the option “streams X”.
- 2) The code calculates 5 images, from different angles, but, since the dataset is a sphere, all the images are the same and each takes approximately the same amount of time. In the tables, the Time, Phantoms and Ready columns contain the instruction counts for each image. The Total Time column shows the time for the complete run, including reading the dataset, etc.

The codes were also run with Zebra’s “auto-growth”. The run times were very close to those obtained using “-streams 1”.

More “Accurate” Results

The performance numbers in the previous section were sent to Tera for consultation. We received several suggestions from their technical staff. In particular, it was pointed out that in “fast” mode, Zebra might not output realistic scaling performance. Instead, runs in “accurate” mode are necessary. Unfortunately, each run in accurate mode could take several weeks to complete, because the simulator runs much slower in accurate mode (for comparison, the runs presented in the previous section took two weeks to complete). However, we were able to obtain some “accurate” simulation results in the final weeks of the CRADA. The “accurate” mode TERA simulation results using 1, 8, and 16 streams in accurate mode for the hand parallelized code are presented in the following table.

Accurate mode for hand parallelized code

Streams	1	8	16
Time	115M	115M	125M
Phantoms	110M	100M	80M
Ready	14M	18M	34M
Total Time	640M	683M	662M

Table 5. This table shows a constant rendering time, independent of the number of streams simulated. These simulations used hand tuned code and automatically detected parallelism as found by the compiler.

All of these results (initial and more accurate), unfortunately, indicated poor performance. Following these results, an effort was undertaken at Tera to understand why we were getting such strange numbers.

A test was made to see what parallelism the compiler could find automatically (using the compiler option “-trace:PAR”). However, it was decided that this approach was not worth pursuing.

Our code was slightly reorganized and timing calls were moved around to isolate the rendering code in order to enable faster simulations. Using the command line “zebra - AllowModeSwitch -fast -r a.out”, the following results were produced:

Streams	Ticks	Speedup	Phantoms	Utilization
1	113,362,463	1.0	110,238,489	3%
2	57,303,771	2.0	54,187,424	5%
4	28,977,071	3.9	25,859,866	11%
8	14,905,571	7.6	11,786,579	21%
16	7,651,708	14.8	4,528,369	41%
32	4,635,393	24.5	1,503,090	68%
64	3,699,487	30.6	545,545	85%

Table 6. Results produced at Tera for a slightly-modified, hand-parallelized version of the volume rendering code. “Ticks” indicates the total number of clock ticks to render one frame. “Phantoms” indicates the number of wasted ticks, when the processor is idle.

These results are still for a single processor, using a different number of streams. Note the steady improvement in processor utilization as streams are added, with an overall 30 times improvement in performance when using 64 streams (versus 1). This indicates that, for this code, the system is able to do a very effective job of hiding latencies and maximizing use of processor resources.

Once these results were generated, Claudio turned to trying to reproduce the results. Strangely enough, even after getting a fresh copy of the simulator and using the identical code, which was run at Tera, Claudio continued to see erroneous results (much to our consternation). It was only then that a final attempt was made to run the simulator on a SunOS machine, which gave the following results:

Streams	Ticks	Speedup	Phantoms	Utilization
1	57,946,985	1.0	55,606,905	4%
2	27,635,238	2.0	25,493,064	8%
4	13,936,191	4.2	11,793,147	16%
8	7,139,218	8.1	4,994,410	29%
16	3,609,639	16.0	1,460,248	59%
32	2,567,374	22.6	408,732	84%
64	2,410,404	24.14	230,122	90%

Table 7. Results produced at Stony Brook using Zebra Release 15 on a SunOS Sparc system, with the slightly modified volume rendering code produced by Tera.

While these results vary some from the results produced at Tera, the trend is similar, and the overall processor utilization at 64 streams is better -- indeed, the results on the whole are a little better. It has been hypothesized that the differences are due to the fact that Claudio's final runs were generated with an even newer release of Tera's compilers, etc., than had been used by Preston Briggs at Tera.

A major portion of the proposed work has been completed. The code has been ported and hand optimized, and several runs have been performed. In this process, considerable knowledge about the TERA architecture has been acquired, and, more than ever, we believe its performance on real-time graphics applications should be superb.

However, we have been unable to "prove" the architecture performs well, due to the fact that an actual system was still unavailable to us when this work was completed.

Conclusion

We have completed the period of work covered by the TTI CRADA and have addressed each of the tasks specified in that CRADA. The joint research has helped Tera have a more robust compiler and has given confidence that the compiler will correctly handle large codes of interest to Sandia National Laboratories, other national laboratories and similar engineering organizations in industry and at universities. The most current and complete research indicates that the Tera machine may be a superb real-time graphics computer when it is manufactured and is available for real applications.

Acknowledgments

We have received major help in all the phases of the project from Gail Alverson, Simon Kahan, and Preston Briggs (from Tera). Help from Gary Montry, of Southwest Software, is also greatly appreciated.

DISTRIBUTION:

2 Tera Computer Company
Attn: Simon Kahan
Attn: C. L. Schlaff
2815 Eastlake Avenue East
Seattle WA 98102

1 MS-0188 C. E. Meyers, 4523
1 MS-0318 G. S. Davidson, 9215
1 MS-1110 C. Pavlakos, 9215
1 MS-1110 C. Silva, 9215
1 MS-1380 D. Lambert, 4213
1 MS-9018 Central Technical Files, 8940-2
5 MS-0899 Technical Library, 4414
2 MS-0619 Review & Approval Desk, 12690
For DOE/OSTI