Preface

Lin Chao
Editor
Intel Technology Journal

This Q1'98 issue of the *Intel Technology Journal* focuses on Intel's tera-scale supercomputer and research on multithreading software libraries for applications.

On June 11, 1997, the Intel supercomputer, containing over 9,200 Pentium® Pro processors, set the world's fastest computing record. Using the industry standard Linpack measurement method, the system calculated 1.34 trillion operations per second (teraflops) making it faster than a speeding bullet. By the time a bullet travels one foot, the computer will have completed 667 million calculations. The supercomputer is used for important scientific simulations such as the effect of a kilometer-wide comet striking the Atlantic Ocean. It will also be used to ensure the safety, reliability and effectiveness of the U.S. nuclear stockpile through computer simulation instead of nuclear testing.

The contract to build this supercomputer was awarded to Intel under the U.S. government's Accelerated Strategic Computing Initiative. It is a joint development of the Department of Energy, Sandia National Labs in New Mexico, and Intel. Besides its 9,200 Pentium Pro processors, it has 573 gigabytes of system memory and 2.25 terabytes of disk storage. It weighs about 44 tons and has 86 cabinets taking up 1,728 square feet.

The Intel TFLOPS supercomputer itself is the subject of four out of the five papers in this Q1'98 issue. The first paper gives an overview that includes a look at system architecture and how 4,536 compute nodes are connected into a single massively parallel supercomputer. The second paper looks at how high performance is achieved with coding and system parallelism. The third paper describes managing and optimizing large-scale parallelism from the point of view of an operating system, and the fourth one looks at the design of the system management environment.

Finally, the fifth paper in this issue describes research in a parallel processing run-time library for applications supporting loop-level parallelism, task-level parallelism and nested parallel threads.

An Overview of the Intel TFLOPS Supercomputer

Timothy G. Mattson, Microcomputer Research Laboratory, Hillsboro, OR, Intel Corp.
Greg Henry, Enterprise Server Group, Beaverton, OR, Intel Corp.

Index words: Supercomputer, MPP, TFLOPS.

Abstract

Computer simulations needed by the U.S. Department of Energy (DOE) greatly exceed the capacity of the world's most powerful supercomputers. To satisfy this need, the DOE created the Accelerated Strategic Computing Initiative (ASCI). This program accelerates the development of new scalable supercomputers and will lead to a supercomputer early in the next century that can run at a rate of 100 trillion floating point operations per second (TFLOPS).

Intel built the first computer in this program, the ASCI Option Red Supercomputer (also known as the Intel TFLOPS supercomputer). This system has over 4500 nodes, 594 Gbytes of RAM, and two independent 1 Tbyte disk systems. Late in the spring of 1997, we set the MP LINPACK world record of 1.34 TFLOPS.

In this paper, we give an overview of the ASCI Option Red Supercomputer. The motivation for building this supercomputer is presented and the hardware and software views of the machine are described in detail. We also briefly discuss what it is like to use the machine.

Introduction

From the beginning of the computer era, scientists and engineers have posed problems that could not be solved on routinely available computer systems. These problems required large amounts of memory and vast numbers of floating point computations. The special computers built to solve these large problems were called *supercomputers*.

Among these problems, certain ones stand out by virtue of the extraordinary demands they place on a supercomputer. For example, the best climate modeling programs solve at each time step models for the ocean, the atmosphere, and the solar radiation. This leads to astronomically huge *multi-physics* simulations that challenge the most powerful supercomputers in the world.

So what is the most powerful supercomputer in the world? To answer this question we must first agree on how to measure a computer's power. One possibility is to measure a system's peak rate for carrying out floating point arithmetic. In practice, however, these rates are only rarely approached. A more realistic approach is to use a common application to measure computer performance. Since computational linear algebra is at the heart of many scientific problems, the de facto standard benchmark has become the linear algebra benchmark, LINPACK [1,7].

The LINPACK benchmark measures the time it takes to solve a dense system of linear equations. Originally, the system size was fixed at 100, and users of the benchmark had to run a specific code. This form of the benchmark, however, tested the quality of compilers, not the relative speeds of computer systems. To make it a better computer performance metric, the LINPACK benchmark was extended to systems with 1000 linear equations, and as long as residual tests were passed, any benchmark implementation, tuning, or assembly coding was allowed. This worked quite well until computer performance increased to a point where even the LINPACK-1000 benchmark took an insignificant amount of time. So, about 15 years ago, the rules for the LINPACK benchmark were modified so any size linear system could be used. This resulted in the MP-LINPACK benchmark.

Using the MP-LINPACK benchmark as our metric, we can revisit our original question: which computer is the most powerful supercomputer in the world? In Table 1, we answer this question showing the MP-LINPACK world record holders in the 1990's.

All the machines in Table 1 are massively parallel processor (MPP) supercomputers. Furthermore, all the machines are based on *Commercial Commodity Off the Shelf* (CCOTS) microprocessors. Finally, all the machines achieve their high performance with scalable interconnection networks that let them use large numbers of processors.

The current record holder is a supercomputer built by Intel for the DOE. In December 1996, this machine, known as the *ASCI Option Red Supercomputer*, ran the MP-LINPACK benchmark at a rate of 1.06 trillion floating point operations per second (TFLOPS). This was the first time the MP-LINPACK benchmark had ever been

Year	System	Number of Processors	MP-LINPACK GFLOPS
1990	Intel iPSC®/860 [2]	128	2.6
1991	Intel DELTA [3]	512	13.9
1992	Thinking Machines CM-5 [4]	1024	59.7
1993	Intel Paragon® [5]	3744	143
1994	Intel Paragon [5]	6768	281
1996	Hitachi CP-PACS [6]	2048	368
1996	Intel ASCI Option Red Supercomputer	7264	1060
1997	Intel ASCI Option Red Supercomputer	9152	1340

Table 1: MP-LINPACK world records in the 1990's. This data was taken from the MP-LINPACK benchmark report [7].

run in excess of 1 TFLOP. In June 1997, when the full machine was installed, we reran the benchmark and achieved a rate of 1.34 TFLOPS.

In Table 2, we briefly summarize the machine's key parameters. The numbers are impressive. It occupies 1,600 sq. ft. of floor-space (not counting supporting network resources, tertiary storage, and other supporting hardware). The system's 9,216 Pentium® Pro processors with 596 Gbytes of RAM are connected through a 38 x 32 x 2 mesh. The system has a peak computation rate of 1.8 TFLOPS and a cross-section bandwidth (measured across the two 32 x 38 planes) of over 51 GB/sec.

Getting so much hardware to work together in a single supercomputer was challenging. Equally challenging was the problem of developing operating systems that can run on such a large scalable system. For the ASCI Option Red Supercomputer, we used different operating systems for different parts of the machine. The nodes involved with computation (compute nodes) run an efficient, small operating system called Cougar. The nodes that support interactive user services (service nodes) and booting services (system nodes) run a distributed UNIX operating system. The two operating systems work together so the user sees the system as a single integrated supercomputer. These operating systems and how they support scalable computation, I/O, and high performance communication are discussed in another paper in this Q1'98 issue of the *Intel Technology Journal* entitled *Achieving Large Scale Parallelism Through Operating System Resource Management on the Intel TFLOPS Supercomputer* [8].

When scaling to so many nodes, even low probability points of failure can become a major problem. To build a robust system with so many nodes, the hardware and software must be explicitly designed for Reliability, Availability, and Serviceability (RAS). All major components are hot-swappable and repairable while the system remains under power. Hence, if several applications are running on the system at one time, only the application using the failed component will shut down. In many cases, other applications continue to run while the failed components are replaced. Of the 4,536 compute nodes and 16 on-line hot spares, for example, all can be replaced without having to cycle the power of any other module. Similarly, system operation can continue if any of the 308 patch service boards (to support RAS functionality), 640 disks, 1540 power supplies, or 616

Compute Nodes	4,536
Service Nodes	32
Disk I/O Nodes	32
System Nodes (Boot)	2
Network Nodes (Ethernet, ATM)	10
System Footprint	1,600 Square Feet
Number of Cabinets	85
System RAM	594 Gbytes
Topology	38x32x2
Node to Node bandwidth - Bi-directional	800 MB/sec
Bi-directional - Cross section Bandwidth	51.6 GB/sec
Total number of Pentium® Pro Processors	9,216
Processor to Memory Bandwidth	533 MB/sec
Compute Node Peak Performance	400 MFLOPS
System Peak Performance	1.8 TFLOPS
RAID I/O Bandwidth (per subsystem)	1.0 Gbytes/sec
RAID Storage (per subsystem)	1 Tbyte

Table 2: System parameters for the ASCI Option Red Supercomputers. The units used in this table and throughout the paper are FLOPS = floating point operations per second with M, G, and T indicating a count in the millions, billions or trillions**.** MB and GB are used for a decimal million or billion of bytes while Mbyte and Gbyte represent a number of bytes equal to the power of two nearest one million (2^{20}) or one billion (2^{30}) respectively.

interconnection facility (ICF) back-planes should fail.

Keeping track of the status of such a large scalable supercomputer and controlling its RAS capabilities is a difficult job. The system responsible for this job is the Scalable Platform Services (SPS). The design and function of SPS are described in another paper in this issue of the *Intel Technology Journal* entitled *Scalable Platform Services on the Intel TFLOPS Supercomputer* [9].

Finally, a supercomputer is only of value if it delivers super performance on real applications. Between MP-LINPACK and production applications running on the machine, significant results have been produced. Some of these results and a detailed discussion of performance issues related to the system are described in another paper in this issue of the *Intel Technology Journal* entitled *The Performance of the Intel TFLOPS Supercomputer* [10].

In this paper, we describe the motivation behind this machine, the system hardware and software, and how the system is used by both programmers and the end-users. The level of detail varies. When a topic is addressed elsewhere, it is discussed only briefly in this paper. For example, we say very little about SPS. When a topic is not discussed elsewhere, we go into great detail. For example, the node boards in this computer are not discussed elsewhere so we go into great detail about them. As a result, the level of detail in this paper is uneven, but, in our opinion, acceptable given the importance of getting the full story about this machine into the literature.

Why Build a TFLOPS Supercomputer?

Even with the benefits of using CCOTS technology throughout the system, a TFLOPS supercomputer is not cheap. In this era of tight government funding, it takes a compelling argument to fund such a machine. The compelling argument in this case was the maintenance of the U.S. nuclear stockpile. It is up to the DOE to maintain this stockpile without nuclear testing; science-based testing will be used instead.

Science-based testing involves a combination of laboratory experiments and computer simulations. The data from experiments plus the results from past nuclear tests will be fed into massive computer simulations that will provide the necessary information to maintain a safe and reliable nuclear stockpile.

DOE scientists have determined that they can only run these simulations if they have 100 TFLOPS computers. Given enough time, the computer industry will create 100 TFLOPS supercomputers. The DOE, however, cannot wait for industry to build these machines; they are needed early in the next century.

In response to this need, the DOE launched a five year, 900 million dollar program in 1995 to accelerate the development of extreme scale, massively parallel supercomputers with the goal of having a 100 TFLOPS computer early in the next century. This program is called the Accelerated Strategic Computing Initiative or ASCI [13]. Scientists from the three major DOE weapons labs: Sandia National Laboratories (SNL), Los Alamos National Laboratory (LANL), and Lawrence Livermore National Laboratory (LLNL) are involved in this program.

The ASCI program will produce a series of machines leading to the 100 TFLOPS machine. The goal is to take advantage of the aggressive evolution of CCOTS technology to build each generation of ASCI supercomputer for roughly the same cost.

The first phase of the ASCI program has two tracks corresponding to the two philosophies of how to build such huge computers: ASCI Red and ASCI Blue. The red track uses densely packaged, highly integrated systems similar to the MPP machines Intel has traditionally built [2,3,5]. The blue track uses clusters of high end, off-the-shelf systems. By considering the successes and failures from each track's initial machines, the direction would be clear for subsequent machines and the path to the 100 TFLOPS goal.

The ASCI red machine was built by Intel and has been in production use at Sandia National Laboratories in Albuquerque, New Mexico since late 1996. Contracts have been awarded for two ASCI blue machines. At Los Alamos National Laboratory, the so-called *ASCI Blue Mountain* [14] machine is being built by Silicon Graphics Inc. This machine will be based on a future version of the Origin 2000 computer and should be operational by mid 1999. At Livermore National Laboratory, the *ASCI Blue Pacific* [15] machine is being built by IBM. This machine will be based on the SP* system with a newer communication switch and a node board with eight PowerPC microprocessors. This machine will be operational in late 1998 or early 1999. The ASCI blue machines will have peak performances in the range of three TFLOPS.

Before leaving this section, we need to address one critical requirement for all of the ASCI machines. According to DOE security regulations, the hard disks used when carrying out classified computing must be permanently and physically isolated from the outside world. In other words, if a disk has *ever* had classified data on it, that disk can *never* be attached to an outside network. Hence, the ASCI supercomputers must have disks that can be switched so the machine can be used for both classified and unclassified computing. Later in this paper, we will address how the ASCI Option Red Supercomputer met this requirement.

The ASCI Option Red Supercomputer: Hardware Overview

The ASCI Option Red Supercomputer is a Massively Parallel Processor (MPP) with a distributed memory Multiple-Instruction, Multiple Data (MIMD) architecture.

All aspects of this system are scalable including the aggregate communication bandwidth, the number of compute nodes, the amount of main memory, disk storage capacity, and I/O bandwidth.

In the following sections, we will discuss the major hardware components used to implement the ASCI Option Red Supercomputer. We will begin with a quick introduction to the Intel Pentium Pro processor. We will follow this with a discussion of the two kinds of node boards used in the system: Eagle and Kestrel. Finally, we will talk about the Interconnection Facility (ICF) used to connect the nodes together and the packaging of all these parts into a full machine.

Figure 1 is a diagram of the ASCI Option Red Supercomputer as it sits at SNL in Albuquerque, New Mexico. The machine is organized into a large pool of

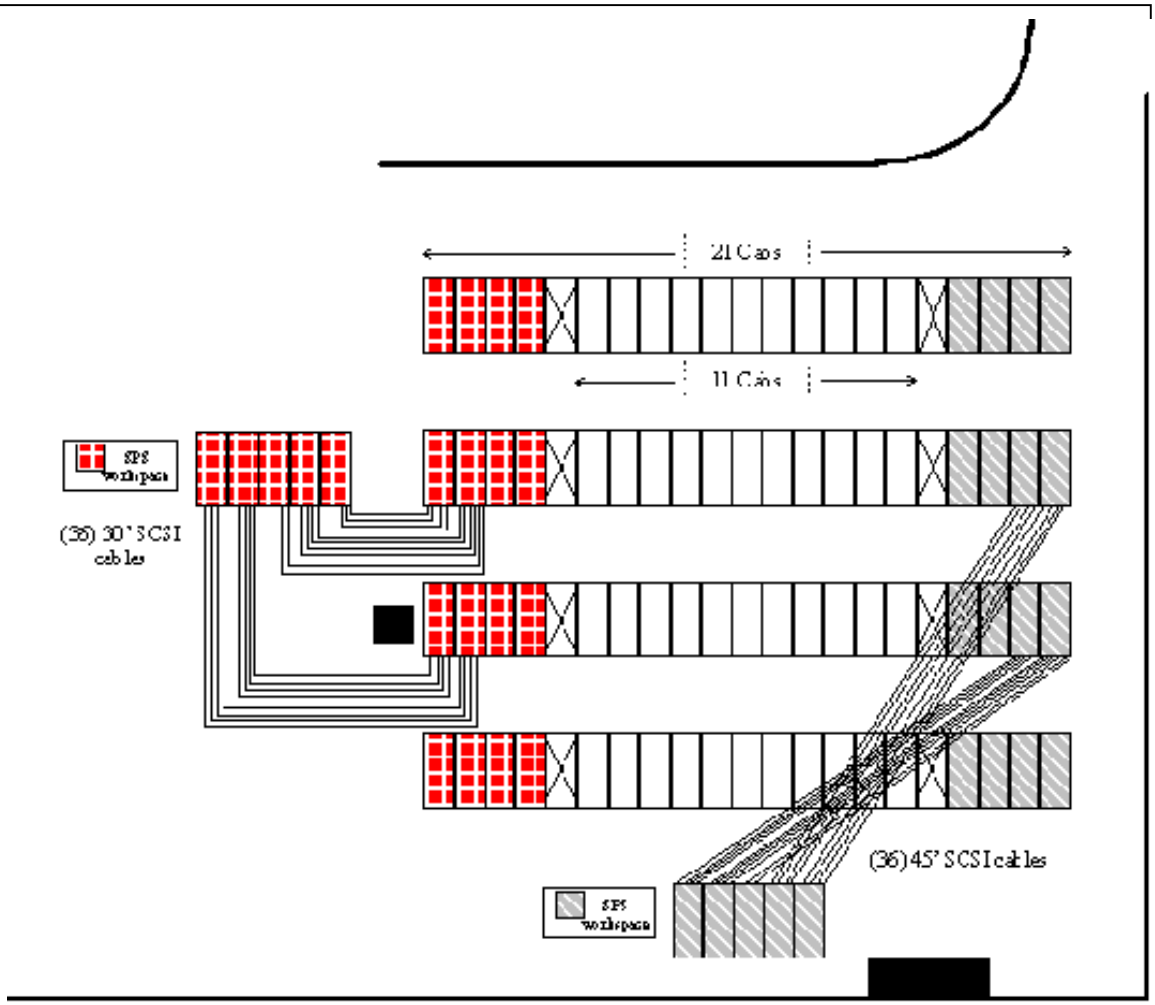


Fig. 1: Schematic diagram of the ASCI Option Red supercomputer as it will be installed at Sandia Nat. Laboratories in Albuquerque NM. The cabinets near each end labeled with an X are the disconnect cabinets used to isolate one end or the other. Each end of the computer has its own I/O subsystem (the group of 5 cabinets at the bottom and the left), and their own SPS station (next to the I/O cabinets). The lines show the SCSI cables connecting the I/O nodes to the I/O cabinets. The curved line at the top of the page show the windowed-wall to the room where the machine operators will sit. The black square in the middle of the room is a support post.

compute nodes in the center, two distinct blocks of nodes at either end, and two separate one-Tbyte disk systems. The end-blocks and their disk systems can be isolated from the rest of the machine by disconnecting the X-mesh cables in the *disconnect cabinets* (marked with an X in Figure 1). This design satisfies DOE security requirements for a physically isolated classified disk system while assuring that both disk systems are always available to users.

Rectangular meshes are needed, hence the number of cabinets set up for isolation must be the same in each row on each end. The most likely configuration would put disconnect cabinets four cabinets over from each end, but this can be varied to meet customer needs. Depending on which types of node boards are used in which slots, this would yield a 400 GFLOPS stand-alone system.

The Pentium Pro Processor

The Intel Pentium Pro processor [11] is used throughout the ASCI Option Red Supercomputer. The instruction set for the Pentium Pro processor is basically the same as the IA-32 instructions used on a Pentium® processor. Unlike the Pentium processor, however, the Pentium Pro processor doesn't directly execute the IA-32 instructions. These complex instructions are broken down at runtime into simpler instructions called micro-operations (or *uops*). The uops execute inside the Pentium Pro processor with the order of execution dictated by the availability of data. This lets the CPU continue with productive work when other uops are waiting for data or functional units.

The Pentium Pro processor can complete up to three uops per cycle of which only one can be a floating-point operation. The floating-point unit requires two cycles per multiply and one cycle per add. The adds can be interleaved with the multiplies so the Pentium Pro processor can have a floating point result ready to retire every cycle. The processors used in the ASCI Option Red Supercomputer run at 200 MHz so the peak floating point rate is 200 MFLOPS.

The Pentium Pro processor has separate on-chip data and instruction L1 caches (each of which is eight KBytes). It also has an L2 cache (256 Kbytes) packaged with the CPU in a single dual-cavity PGA package. All cache lines are 32 bytes wide.

The Pentium Pro processor bus supports memory and cache coherency for up to four Pentium Pro processors. In the ASCI Option Red Supercomputer, however, we used only two processors per node.

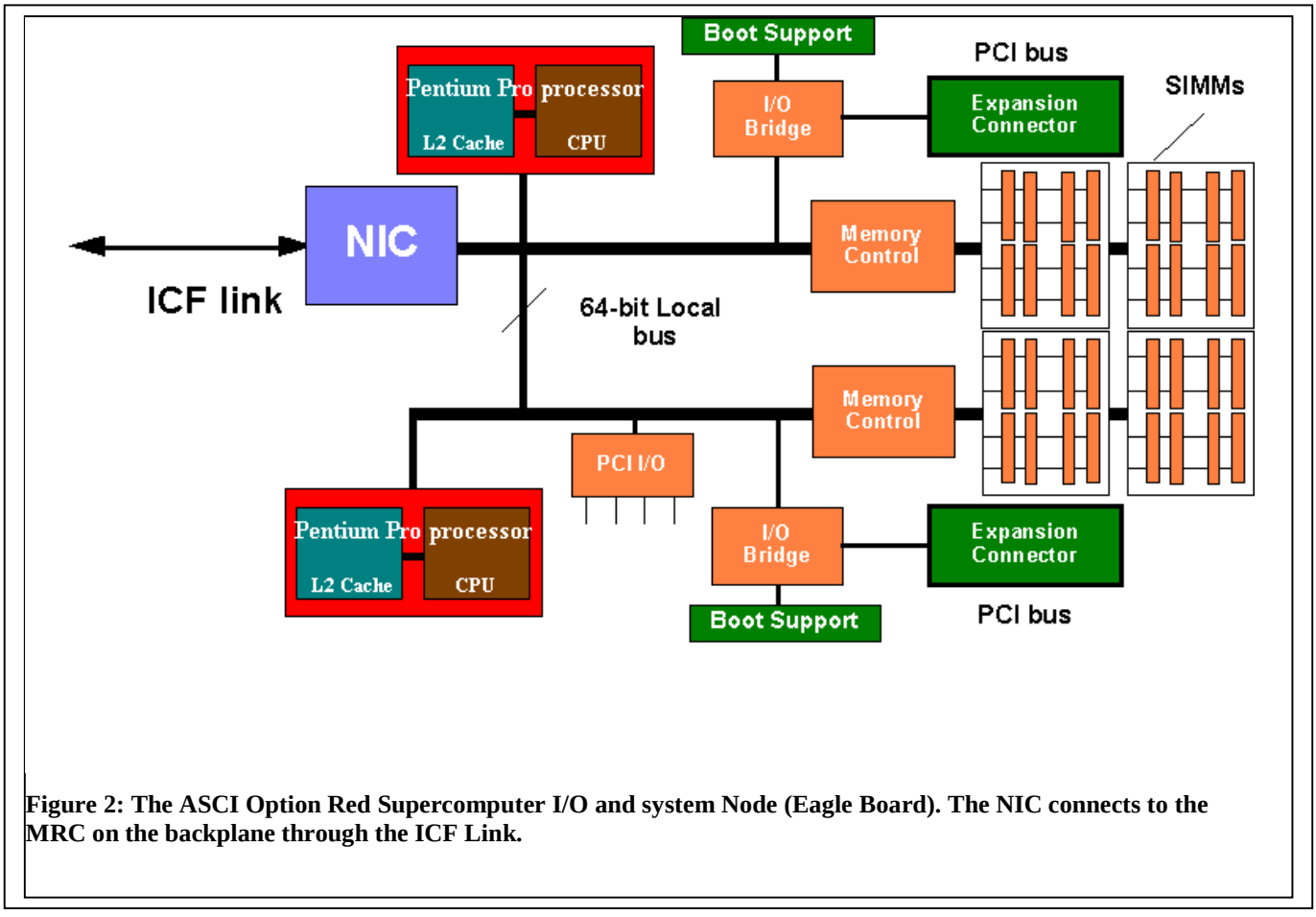


Figure 2: The ASCI Option Red Supercomputer I/O and system Node (Eagle Board). The NIC connects to the MRC on the backplane through the ICF Link.

The Eagle Board

The node boards used in the I/O and system partitions are called *Eagle b*oards. Figure 2 shows a block diagram for an Eagle board. Each node includes two 200 MHz Pentium Pro processors. These two processors support two on-board PCI interfaces that each provide 133 MB/sec I/O bandwidth. One of the two buses can support two PCI cards through the use of a 2-level riser card. Thus, a single Eagle board can be configured with up to three long-form PCI adapter cards. CCOTS PCI adapter boards can be inserted into these interfaces to provide Ultra-SCSI, HiPPI, ATM, FDDI, and numerous other custom and industry-standard I/O capabilities. In addition to add-in card capabilities, there are base I/O features built into every board that are accessible through the front panel. These include RS232, 10 Mbit Ethernet, and differential FW-SCSI.

Each Eagle board provides ample processing capacity and throughput to support a wide variety of high-performance I/O devices. The throughput of each PCI bus is dictated by the type of interface supported by the PCI adapter in use, the driver software, and the higher-level protocols used by the application and the “other end” of the interface. The data rates associated with common I/O devices fit well within the throughput supported by the PCI bus. Ultra-SCSI, for example, provides a hardware rate of 40 MB/sec. This rate can easily be supported by CCOTS PCI adapters.

The memory subsystem is implemented using Intel’s CCOTS Pentium Pro processor support chip-set (82453). It is structured as four rows of four, independently-controlled, sequentially-interleaved banks of DRAM to produce up to 533 MB/sec of data throughput. Each bank of memory is 72 bits wide, allowing for 64 data bits plus 8 bits ECC, which provides single bit error correction and multiple bit error detection. The banks are implemented as two 36-bit SIMMs, so industry standard SIMM modules can be used to provide 64 Mbytes to one Gbytes of memory.

The Kestrel Board

Kestrel boards (see Figure 3) are used in the compute and service partitions. Each Kestrel board holds two compute nodes. The nodes are connected through their Network Interface Chip (NIC) with one of the NICs connecting to a Mesh Router Chip (MRC) on the back-plane. Each node on the Kestrel board includes its own boot support (FLASH ROM and simple I/O devices) through a PCI bridge on its local bus. A connector is provided to allow testing of each node through this PCI bridge. The FLASH ROM contains the Node Confidence Tests, BIOS, plus additional features needed to diagnose board failures and to load a variety of operating systems. Local I/O support includes a serial port, called the "Node Maintenance Port." This port interfaces to the system’s

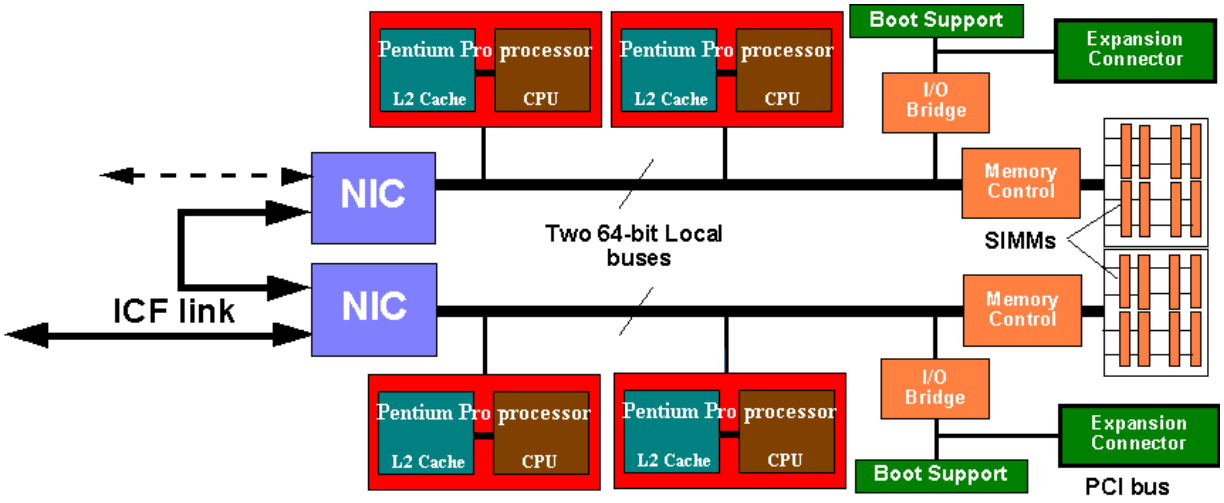


Figure 3: The ASCI Option Red supercomputer Kestrel Board. This board includes two compute nodes chained together through their NIC's. One of the NIC's connects to the MRC on the backplane through the ICF Link.

internal Ethernet through the PSB on each card cage.

The memory subsystem on an individual compute node is implemented using Intel's CCOTS Pentium Pro processor support chip-set (82453). It is structured as two rows of four, independently-controlled, sequentially-interleaved banks of DRAM to produce up to 533 MB/sec of data throughput. Each bank of memory is 72 bits wide, allowing for 64 data bits plus 8 bits ECC, which provides single bit error correction and multiple bit error detection. The banks are implemented as two 36-bit SIMMs, so industry standard SIMM modules may be used. Using commonly available 4 and 8 MByte SIMMs (based on 1Mx4 DRAM chips) and 16 and 32 MByte SIMMs (based on 4Mx4 DRAM chips), 32 MB to 256 MB of memory per node is supported. The system was delivered with 128 Mbytes/node.

Interconnection Facility

The interconnection facility (ICF) is shown in Figure 4. It utilizes a dual plan mesh to provide better aggregate bandwidth and to support routing around mesh failures. It uses two custom components: NIC and MRC. The MRC sits on the system back-plane and routes messages across the machine. It supports bi-directional bandwidths of up to 800 Mbytes/sec over each of six ports (i.e., two directions for each X, Y, and Z port). Each port is composed of four virtual lanes that equally share the total bandwidth. This means that as many as four message streams can pass through an MRC on any given port at any given time. This reduces the impact of communication contention and leads to a more effective use of total system bandwidth.

The NIC resides on each node and provides an interface between the node's memory bus and the MRC. The NIC can be connected to another NIC to support dense packaging on node boards. For example, on the Kestrel board, the NIC on one node, the *outer* node, is connected to the NIC on the other node, the *inner* node, which then connects to the MRC. Contention is minimized in this configuration since the virtual lane capability used on the MRCs was included on the NICs.

Putting the Pieces Together

Boards and an ICF are not enough to build a working supercomputer. In this section we very briefly address how these components are packaged into a full system.

The boards and ICF components are packaged into cabinets and organized into a full system. Each cabinet contains a power supply, four card cages, and a fan unit. The fan unit sits on top of the basic cabinet making the full assembly eight feet tall. A card cage holds any combination of eight Kestrel or Eagle node boards. The card cage also holds two MRC cards with four MRCs per card. The pair of MRC cards implements a cube topology and provides the back and front planes of the split plane mesh and the connection points for the node boards.

Associated with each card cage is the PSB. The PSB monitors the system resources within the card cage and communicates this information back to a system management console over a separate Ethernet network. The cabinets are connected in the X direction with cables between adjacent cabinets. For the Y direction, cables run from the top of one cabinet, down through the floor, and

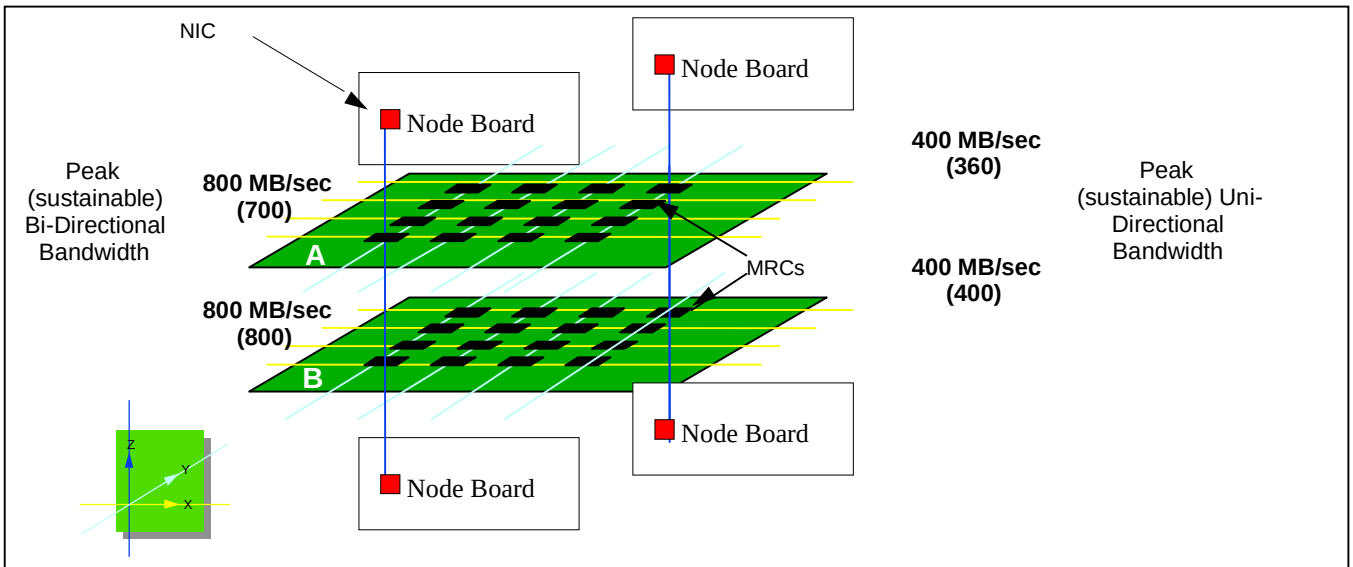


Figure 4: ASCI Option Red Supercomputer 2 Plane Interconnection Facility (ICF). Bandwidth figures are given for NIC-MRC and MRC-MRC communication. Bi-directional bandwidths are given on the left side of the figure whileuni-directional bandwidths are given on the right side. In both cases, sustainable (as opposed to peak) numbers are given in parentheses.

into the bottom of the corresponding cabinet in the next row. This cabling connects the nodes into a 38x32x2 topology.

I/O and network nodes are usually placed near either end of the full machine. As mentioned earlier, the I/O nodes are Eagle nodes since they provide more room for PCI connections. These nodes are connected with standard SCSI cables to the disk RAIDS in separate cabinets.

The User's View of the ASCI Option Red Supercomputer

To the user, the ASCI Option Red Supercomputer has the look and feel of a UNIX-based supercomputer. When users log onto the machine, they get a familiar UNIX prompt. Files are presented to the user and manipulated in the standard UNIX way. Programs are built with *make* and shell scripts provide compiler interfaces that link in the special compilers and libraries needed to build parallel programs. It is only when the user submits a parallel job that the system deviates from a standard workstation environment–but even this deviation is slight.

To submit a parallel job, the user issues a command with a typically cryptic UNIX style name ("yod"). Parameters to "yod" control the runtime factors effecting a computation such as how many and which nodes to use. Once the job is running, the user can monitor the job with a command called "showmesh." This command graphically displays the compute nodes visible to the user and the jobs running on the machine. In principle, it is not that different from the familiar "ps" command used to monitor a job on a UNIX workstation. A standard NQS environment is available to submit batch jobs.

To a programmer, the machine looks like a typical MPP supercomputer. The programs running on each node use standard sequential languages. To move data between nodes, the programs explicitly call routines from a message-passing library (MPI or NX in most cases). When developing software, building programs, or other interactive operations, the computer will have the look and feel of a UNIX workstation.

Fortran77, Fortran90, C, and C++ compilers from Portland Group Incorporated (PGI) are available on the system. In addition, interactive debuggers and performance analysis tools that work with and understand the source code for each of these languages are provided. For data-parallel programming, the HPF compiler from PGI is provided.

While message passing (either explicitly or implicitly through HPF) is used between nodes, shared memory mechanisms are used to exploit parallelism on a node. The user has three multiprocessor options. First, the second processor can be completely ignored. Alternatively, it can be used as a communication co-processor. Finally, a simple threads model can be used to utilize both processors on a single application. The threads model is accessed through compiler-driven parallelism (using the -Mconcur switch) or through an explicit dispatch mechanism referred to as the *COP* interface.

The COP interface lets a programmer execute a routine within a thread running on the other processor. Global variables are visible to both the COP thread and the parent thread. To use COP, the programmer passes COP the address of the routine, the address of a structure holding the routine's input parameters, and an output variable to set when the routine is completed. The COP'ed routine can not make system calls (including calls to the message-passing library).

The debugger on the ASCI Option Red Supercomputer is a major re-implementation of the *IPD* [16] debugger developed for the Paragon XP/S supercomputer. The debugger has been designed to scale up to the full size of the ASCI Option Red Supercomputer. It includes both graphical and command line interfaces. The debugger's command line interface has been designed to mimic the *DBX* interface where ever possible.

The performance analysis tools use the counters included on the Pentium Pro processor and on the NIC. The counters on the Pentium Pro processor let users track a range of operations including floating point operation counts, cache line loads, and data memory references. Each Pentium Pro processor has two counters so only two independent events can be counted at one time. The NIC has ten independent counters.

We anticipate that the applications on this system will run for many hours or even days. Hence, even a system mean time between failure in excess of our target (>50 hours) will not be sufficient. Therefore, a check-point/restart capability will be provided. Automatic check-pointing is exceedingly difficult to implement on systems as large as this one. Hence, applications will need to assist the check-pointing by putting themselves into a clean state prior to explicitly invoking a check-point. (A clean state is one where the communication network does not hold any message-passing state for the application being check-pointed.) The I/O bandwidth will be sufficient to check-point the entire system memory in approximately five minutes.

The ASCI Option Red Supercomputer: Software

The user view of the ASCI Option Red Supercomputer is shown in Figure 5. This view is maintained by the system software which organizes the system into four logical partitions:

- **Service Partition**–provides an integrated, scalable "host" that supports interactive users,

application development, and system administration. An operator can vary the number of service nodes at boot time depending on the demand for interactive services.

- **I/O Partitions**–implement scalable file and network services.
- **Compute Partition**–contains the nodes dedicated to running parallel applications.
- **System Partition**–supports initial system booting. The boot node has its own independent RAID and Ethernet connections. A second Eagle node configured as part of the I/O partition can be cross-connected with the boot RAID to provide an on-line spare to the boot node.

In normal operation, one of the sets of disconnect cabinets will cut the system in two. In this case, each side will see the logical partition model outlined in Figure 5.

In the following sections, we describe the key features of the system software on the ASCI Option Red Supercomputer. We start with the two operating systems used on the system. We then describe the *portals* mechanism used for high performance communication.

The Operating Systems

Each partition in the system places different demands on the operating system. One operating system that met the needs of all of the partitions could have been developed. This would guarantee, however, that the operating system did all jobs adequately but none of them particularly well. We took a different approach and used multiple operating systems tuned to the specific needs of each partition.

The service, I/O, and system partitions are directly visible to interactive users. These partitions need a familiar, UNIX operating system. We used Intel's distributed version of UNIX (POSIX 1003.1 and XPG3, AT\&T System V.3 and 4.3 BSD Reno VFS) developed for the Paragon® XP/S supercomputer. The port of the Paragon OS to the ASCI Option Red Supercomputer resulted in an OS we call the TFLOPS OS. The TFLOPS OS presents a single system image to the user. This means that users see the system as a single UNIX machine despite the fact that the operating system is running on a distributed collection of nodes.

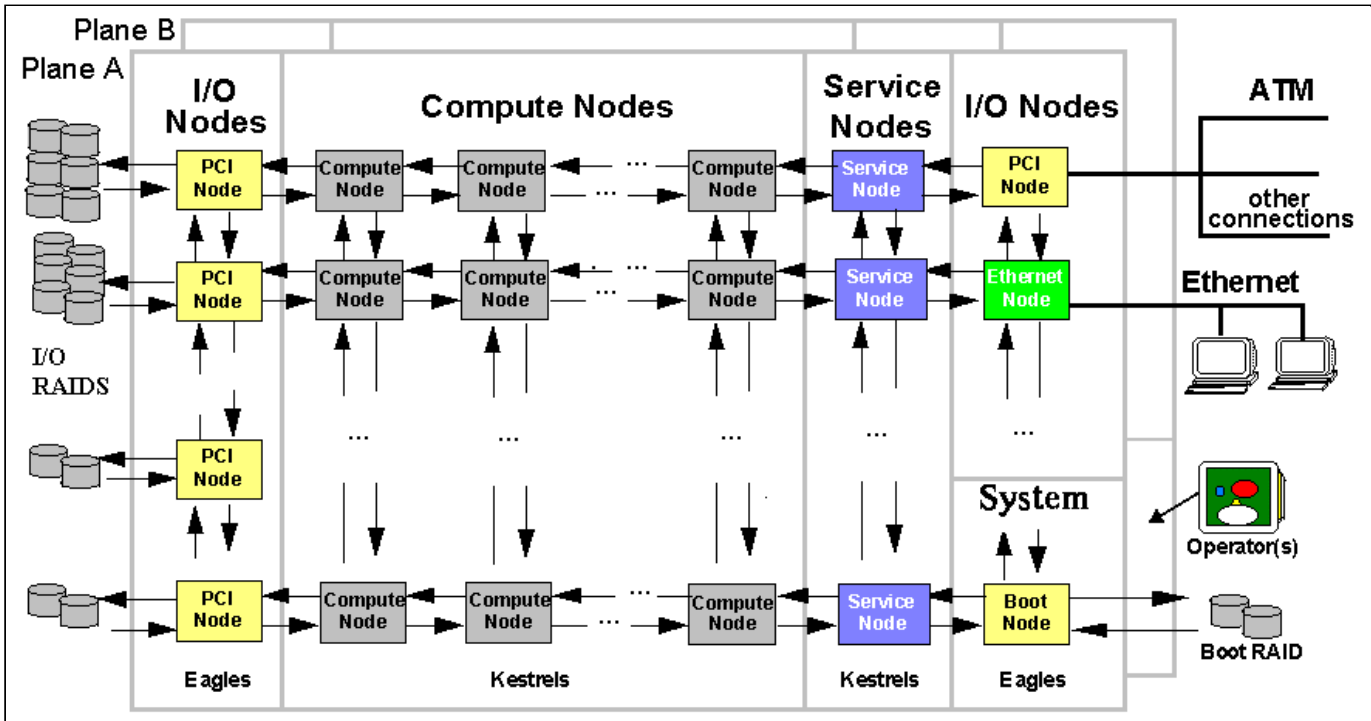


Figure 5: Logical System Block Diagram for the ASCI Option Red Supercomputer. This system uses a split-plane mesh topology and has 4 partitions: System, Service, I/O and Compute. Two different kinds of node boards are used and described in the text: the Eagle node and the Kestrel node. The operators console (the SPS station) is connected to an independent ethernet network that ties together patch support boards on each card cage.

The compute partition has different needs. Users only run parallel applications on these nodes, not general interactive services. Furthermore, these nodes are the most numerous so the aggregate costs of wasted resources (such as memory consumed by an OS) grows rapidly. Therefore, for the compute partition, we wanted an operating system that was small in size, very fast, and provided just those features needed for computation.

On our Paragon XP/S supercomputers, we had great success with SUNMOS [17] –a light-weight operating system from Sandia National Laboratories and the University of New Mexico. For the ASCI Option Red Supercomputer, we decided to work with their next light weight OS (Puma [3]). We call our version of Puma, *Cougar*.

Cougar easily meets our criteria for a compute partition operating system. It is small (less than half a megabyte), of low complexity, and scalable. Cougar can be viewed in terms of four entities:

- host Operating System
- quintessential kernel (Q-Kernel)
- process control thread (PCT)
- application

Since it is a minimal operating system, Cougar depends on a host OS to provide system services and to support interactive users. For the ASCI Option Red Supercomputer, the host OS is the TFLOPS OS running in the service partition. The Q-Kernel is the lowest level component of Cougar. All access to hardware resources comes from the Q-Kernel. Above the Q-Kernel sits the process control thread (PCT). This component runs in user space and manages processes. At the highest level are the users' applications.

Cougar takes a simple view of a user's application. An application is viewed as a collection of processes grouped together and identified by a common group identifier. Within each group, a process is assigned a group rank which ranges from **`0`** to **`(n-1)`** where **`n`** is the number of processes. While the PCT supports priority multi-tasking, it is anticipated that most users will run only one application process per node.

Memory integrity in Cougar is assured by a *hierarchy of trusts* ranging from the Q-Kernel to the PCT to the application. At each level in the hierarchy, lower levels are trusted but higher levels are not trusted. Hence, an application cannot corrupt the PCT or Q-Kernel while a flawed Q-Kernel can corrupt anything.

The Quintessential Kernel or Q-Kernel

There is one Q-Kernel running on each of the nodes in the compute partition. It provides access to the physical resources of a node. For example, only the Q-Kernel can directly access communication hardware or handle interrupts. In addition, any low-level functions that need to be executed in supervisor mode are handled by the Q-Kernel. This includes memory management, context switching. and message dispatch or reception.

The Q-Kernel is accessed through a series of system traps. These are usually invoked by an application or a PCT, but they can also arise from an exception (e.g., a floating point exception) or an interrupt (e.g., timer or communication interrupts).

The Q-Kernel does not set the policy for how a node's resources will be used. Rather, it performs its low-level tasks on behalf of the PCT or a user's application. This design keeps the Q-Kernel small and easy to maintain.

The Process Control Thread

The Process Control Thread (PCT) sits on top of the Q-Kernel and manages process creation, process scheduling, and all other operating system resources. While part of the operating system, the PCT is a user-level process meaning that it has read/write access to the user address space. There will typically be only one PCT per node.

Most of the time, the PCT is not active. It only becomes active when the Q-Kernel receives an application exception (a process blocks itself by a call to the `quit_quantum()` routine) or in response to certain timer interrupts.

When a PCT becomes active, it first checks to see if the most recently suspended process has any outstanding requests for the PCT. Once these requests are resolved, it handles requests from any other PCTs. Finally, it tries to execute the highest priority, run-able application.

ASCI Option Red Message Communication Software: Portals

Low-level communication on the ASCI Option Red Supercomputer uses Cougar portals [12]. A portal is a window into a process's address space. Using portals, a process can write-to or read-from a special address subspace on another process. This address space is user-accessible meaning copying between kernel space and user space is avoided. Since extra copy operations increase communication latencies, portals support low-latency communication. In addition to low latencies, portals provide high-performance asynchronous transfers with buffering in the application's data space.

Application programs will have access to portals, but we expect most applications will use higher level message-passing libraries. A variety of message-passing protocols will be available on top of portals. The preferred library is MPI. This is a full implementation of the MPI 1.1 specification.

To support applications currently running on Paragon supercomputers, we implemented a subset of the NX message-passing library on top of portals. Since both MPI

and NX run on top of portals, latency and bandwidth numbers will be comparable for applications utilizing either library.

System RAS Capabilities

We have set aggressive reliability targets for this system. The key target is that a single application will see a mean time between failure of greater than 50 hours. Some applications (e.g., CTH [5]) have already exceeded this target. In the aggregate, however, we expect far more from our system. Our target is to have the system in continuous operation for greater than four weeks with no less than 97% of the system resources being available. In order to meet these targets, the system includes sophisticated Reliability, Availability, and Serviceability (RAS) capabilities. These capabilities will let the system continue to operate in the face of failures in all major system components.

Three techniques are used to meet our system RAS targets. First, the system includes redundant components so failures can be managed without swapping hardware. For example, the dual plane ICF uses Z-X-Y-Z routing so a bad mesh router chip can be by-passed without causing system failure. In addition, the system will include on-line spare compute-nodes that can be mapped into the system without interruption.

The second RAS technique is to build the system so all major components are hot-swappable and can be repaired while the system continues to run. The compute nodes and the on-line spares, for example, can all be replaced without having to cycle the power of any other module. Similarly, system operation can continue if any of the 640 disks, 1540 power supplies, or 616 ICF backplanes should fail.

Finally, to manage these RAS features and to manage the configuration of such a large system, an active Monitoring and Recovery Subsystem (MRS) is included. At the heart of this system is a PSB–one per 8-board card cage. The PSB board monitors the boards in its card cage and updates a central MRS database using a separate RAS Ethernet network. This database will drive an intelligent diagnostic system and will help manage the replacement of system units. The console for the RAS subsystem is the SPS station. There is one of these connected to the RAS Ethernet network on each side of the machine (see Figure 1). As mentioned earlier, the SPS system is described in more detail in another paper in this issue of the *Intel Technology Journal* [4].

Conclusion

DOE scientists need computers that can deliver sustained TFLOPS in order to get their jobs done. To meet that need, the DOE created the ASCI program, a program that will produce a series of supercomputers ranging from 1 TFLOPS in 1996 to up to 100 TFLOPS early in the next century. The first of these machines is the ASCI Option Red Supercomputer built by Intel.

The machine is a huge massively parallel supercomputer containing over 9200 Pentium Pro processors. It was this supercomputer (actually 3/4 of it) that first broke the 1 TFLOPS MP-LINPACK barrier (1.06 TFLOPS on December 7, 1996). Later, after the full machine was installed at Sandia National Laboratories, we broke our own record and ran the MP-LINPACK benchmark at 1.34 TFLOPS.

Supercomputers, however, aren't built to run benchmarks. A machine is justified by its ability to run applications that solve key problems. On this front, we have succeeded as well. As we built the machine, we used application benchmarks and full application programs to debug the machine. Hence, within hours of installing the machine, it was being used for production calculations. Actually, the first production calculations were carried out while it was still being built in the factory!

It is customary to close a research paper with a discussion of future plans. After we started building this supercomputer, our focus changed from providing MPP systems for end-users, to providing components and expertise to the broader high performance computing market. We learned a lot from our experiences, and this knowledge is being applied throughout Intel. Hence, the core technologies behind the machine will likely show up in future products.

Acknowledgments

The Intel ASCI Option Red Supercomputer is a result of the "blood, sweat and tears" of hundreds of Intel engineers over the course of a decade. It all started *with Intel Scientific Computers* back in the mid 1980's and ended with the *Scalable Systems Division* inside Intel's *Enterprise Server Group* in the late 1990's. This was our last MPP machine, and most of the people behind this computer have already moved on in their careers. We can't list all of you here so we must settle for a collective *thank you*. To these people, we hope you are doing well and that you take as much pride in this remarkable machine as we do.

References

[1] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart, *Linpack Users' Guide*, SIAM, Philadelphia, PA, 1979.

[2] E. Barszcz, "One Year with an iPSC/860," *Proceedings of the COMPCON Spring'91 conference*, p. 213, IEEE Computer Society Press, 1991.

[3] S.L. Lillevik, "The Touchstone 30 Gigaflop DELTA Prototype," *Proceedings of the sixth Distributed memory Computer Conference*, p. 671, IEEE Computer Society Press, 1991.

[4] P.S. Lomdahl, P. Tamayo, N. Gronbech-Jensen, D. M. Beazley, "50 Gflops Molecular Dynamics on the Connection Machine 5", *Proceedings of Supercomputing'93*, p. 520, IEEE Computer Society Press, 1993.

[5] D.R. Mackay, J. Drake, T. Sheehan, B. Shelton, "Experiences in Programming Grand Challenge Applications on the Intel MP Paragon Supercomputer," *Proceedings of the Intel Supercomputer Users Group*, 1995. Available on the web at http://www.cs.sandia.gov/ISUG.

[6] Y. Abei, K. Itakura, I. Boku, H. Nakamura and K. Nakazawa, "Performance Improvement for Matrix Calculation on CP-PACS Node Processor," in *Proceedings of the High Performance computing on the Information Superhighway conference* (HPC_ASIA'97), 1997.

[7] Dongarra, J.J., "Performance of various computers using standard linear equations software in a Fortran environment," Computer Science Technical Report CS-89-85, University of Tennessee, 1989, http://www.netlib.org/benchmark/performance.ps

[8] S. Garg, R. Godley, R. Griffths, A. Pfiffer, T. Prickett, D. Robboy, S. Smith, T. M. Stallcup, and S. Zeisset, "Achieving Large Scale Parallelism Through Operating System Resource Management on the Intel TFLOPS Supercomputer," *Intel Technology Journal*, Q1'98 issue, 1998.

[9] R. Larsen and B. Mitchell, "Scalable Platform Services on the Intel TFLOPS Supercomputer," *Intel Technology Journal*, Q1'98 issue, 1998.

[10] G. Henry, B. Cole, P. Fay, T.G. Mattson, "The Performance of the Intel TFLOPS Supercomputer," *Intel Technology Journal*, Q1'98 issue, 1998.

[11] Pentium Pro Processor technical documents, http://www.intel.com/design/pro/manuals/.

[12]S.R. Wheat, R. Riesen, A.B. Maccabe, D.W. van Dresser, and T. M. Stallcup, "Puma: An Operating System for Massively Parallel Systems," *Proceedings of the 27'th Hawaii International Conference on Systems Sciences Vol II*, p. 56, 1994.

[13] ASCI overview web site, http://www.sandia.gov/ASCI.

[14] ASCI Blue Mountain web site, http://www.lanl.gov/projects/asci/bluemtn.

[15] ASCI Blue Pacific web site, http://www.llnl.gov/asci/platforms/bluepac.

[16] D. Breazeal, K. Callagham, and W.D. Smith, "IPD: A Debugger for Parallel Heterogeneous systems", in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, p. 216, 1991.

[17] B. Traversat, B. Nitzberg and S. Fineberg, "Experience with SUNMOS on the Paragon XP/S-15," in *Proceedings of the Intel Supercomputer User's meeting*, San Diego, 1994.

Authors' Biographies

Timothy G. Mattson has a Ph.D. in chemistry (1985, U.C Santa Cruz) for his research on Quantum Scattering theory. He has been with Intel since 1993 and is currently a research scientist in Intel's Parallel Algorithms Laboratory where he works on technologies to support the expression of parallel algorithms. Tim's life is centered on his family, snow skiing, science and anything that has to do with kayaks. His e-mail is timothy_g_mattson@ccm2.hf.intel.com

Greg Henry received his Ph.D. from Cornell University in Applied Mathematics. He started working at Intel SSD in August 1993. He is now a Computational Scientist for the ASCI Option Red Supercomputer. He tuned MP LINPACK and the BLAS used there-in. Greg has three children and a wonderful wife. He plays roller hockey, soccer, and he enjoys Aikido and writing. His e-mail address is henry@co.intel.com.

The Performance of the Intel TFLOPS Supercomputer

Greg Henry, Enterprise Server Group, Beaverton, OR, Intel Corp.
Pat Fay, Enterprise Server Group, Beaverton, OR, Intel Corp.
Ben Cole, Enterprise Server Group, Beaverton, OR, Intel Corp.
Timothy G. Mattson, Microcomputer Research Laboratory, Hillsboro, OR, Intel Corp.

Index words: Parallel Supercomputer Applications

Abstract

The purpose of building a supercomputer is to provide superior performance on real applications. In this paper, we describe the performance of the Intel TFLOPS Supercomputer starting at the lowest level with a detailed investigation of the Pentium® Pro processor and the supporting memory subsystem. We follow this with a description of the benchmarks used to track the performance of the machine over its development life cycle, which culminated in the first MP LINPACK run to exceed a rate of one trillion floating point operations per second (TFLOPS). Our analysis applies not only to the TFLOPS supercomputer, but also to servers and workstations based on the Intel 32-bit architecture. We conclude with a discussion of the machine's performance on a production application.

Introduction

The Intel TFLOPS Supercomputer, also known as the ASCI Option Red Supercomputer, at Sandia National Laboratories in Albuquerque, NM, is the world's fastest supercomputer. By this we mean that this supercomputer is theoretically capable of doing more floating point operations per second on a given application than any other general purpose supercomputer built to date. With over 9200 Intel Pentium Pro processors each of which is capable of running at 200 million floating point operations per second (MFLOPS), this supercomputer can theoretically run at over 1.8 trillion floating point operations per second (TFLOPS).

An overview of what the supercomputer is and how it is used, the operating systems and parallel I/O running on it, and the scalable platform services that support it are the subjects of other papers in this Q1'98 issue of the Intel Technology Journal. This paper looks at how you achieve high performance with real applications. This improved performance cannot be achieved by adding more or faster nodes since the hardware is fixed. Therefore, we look at algorithmic and coding enhancements to the applications. Furthermore, we investigate what kinds of performance can be reasonably expected, and what can be done to enhance the performance of given applications.

It can be argued that the first barrier to achieving performance on an application is parallelizing it. That is, the data and/or work must be efficiently distributed amongst all the processors in order to achieve optimum performance from the processors working together. How easy, hard, or possible this is depends on the application. It is not our goal here to discuss this difficulty. As mentioned in another paper in this issue, there are around 4500 compute nodes in this supercomputer each having two processors. Let us assume that an application can be at least distributed among these 4500 nodes. Since the funding for this supercomputer comes from the DOE–an organization with vast experience in scalable computing–assuming that the application is parallelized is not entirely unreasonable. And although none of the applications have ever been run on such a large parallel supercomputer, the scientists at Sandia National Laboratories have spent many years achieving parallelism in their data and know how to take advantage of a scalable supercomputer.

Hence, instead of discussing application parallelization, we discuss the efforts required to achieve high performance of existing parallel applications. The total efficiency of the full system cannot be better than the efficiency of a single node. Much of our discussion is focused on a single node or even a single processor. We start with a quick introduction to the Pentium Pro processor followed by our initial performance explorations on a processor and its supporting memory subsystem. We then explore some of the benchmarks used to track system performance and discuss our historic MP LINPACK computation. The paper concludes with a brief discussion of a specific application called CTH.

The Pentium Pro® Processor

The Intel Pentium Pro processor is used on all the nodes in the Intel ASCI Option Red Supercomputer. A full description of the processor is beyond the scope of this paper and is available elsewhere [17]. In this section, we highlight the key features of the processor and emphasize issues that are important when analyzing application performance.

At runtime, an instruction for the Pentium Pro processor is broken down into simpler instructions called micro-operations (or *uops*). Three decode units are available on the processor to carry out this decomposition. One unit (unit 0) can decode complex operations while any of the three units can decode simple operations. The uops execute inside the Pentium Pro processor with the order of execution dictated by the availability of data. This lets the CPU continue with productive work when other uops are waiting for data or functional units. This *out of order execution* is combined with sophisticated *branch prediction* and *register renaming* to provide what Intel calls, *dynamic execution*.

The Pentium Pro processor's core can execute a burst rate of up to five uops per cycle running on five functional units:

- Store Data Unit
- Store Address Unit
- Load Address Unit
- Integer ALU
- Floating Point/Integer Unit

Up to three uops can be retired per cycle of which only one can be a floating-point operation. The floating-point unit requires two cycles per multiply and one cycle per add. The adds can be interleaved with the multiplies so the Pentium Pro processor can have a result ready to retire every cycle. Hence, the peak multiply-add rate is 200 MFLOPS at 200 MHz.

The Pentium Pro processor has separate on-chip data and instruction L1 caches (each of which is eight KBytes). It also has an L2 cache (256 KBytes) packaged with the CPU in a single dual-cavity PGA package. Cache lines are 32 bytes wide. The L1 data cache is dual-ported and non-blocking, supporting one load and one store per cycle for peak bandwidth of 3.2 billion bytes per second (GB/sec) on a 200 MHz CPU. The L2 cache interface runs at the full CPU clock speed and can transfer 64 bits per cycle (1.6 GB/sec on a 200 MHz Pentium Pro processor). The external bus is also 64 bits wide and supports a data transfer every bus-cycle.

The Pentium Pro processor bus offers full support for memory and cache coherency for up to four Pentium Pro processors (though our compute nodes only have two processors). It has 36 bits of address and 64 bits of data. Bus efficiency is enhanced through the following features:

- the capability to defer long transactions
- a bus pipeline with a depth of 8
- bus arbitration on a cycle-by-cycle basis

The bus can support up to eight pending transactions while the Pentium Pro processor and the memory controller can have up to four pending transactions. Memory controllers can be paired-up to match the bus's support for eight pending transactions.

The bus can sustain data on every clock cycle, so at 66 MHz, the peak data rate is 533 million bytes per second. Unlike most CCOTS processor buses, which can only detect data errors by using parity coverage, the Pentium Pro processor data bus is protected by ECC. Address signals are protected by parity.

Memory Movement

The first obstacle to fast parallel performance is the per node performance. Therefore, in this section, we discuss how fast we were able to run code on just one node, which has two 200 MHz Pentium Pro processors. (In many cases, our observations could be extended to the Intel Pentium® II processors, with appropriate adjustments made in the core frequency and cache specifications.)

Scientific applications tend to differ from many commercial applications in that they often have huge data sets, and large amounts of floating point operations are done on these data sets. How quickly data can be moved and manipulated is significant.

Let us start with our definition of memory movement. We use the phrase "memory movement" to refer to the movement of data into the floating-point stack or into the L1_data cache. Frequently, memory movement discussions are restricted to the movement of data from main memory. However, we needed to focus on the larger issue of using data, whether it might already be in cache or not. Therefore, issues such as the instruction decoding sequence and how it affects the rate that data can be pulled from the L1_data cache are included in this discussion of memory movement. In essence, our focus is on anything that impacts memory movement.

Studying memory movement leads to an investigation of the concept of hierarchical memory. There are a finite number of resources available in the hierarchy and any memory reference must eventually traverse the entire hierarchy. As an example, if you do a store of a value in a register to main memory, you need to go to L1 and L2 on your way to main memory, although the write-back nature of the caches may prevent this from happening immediately. The fastest resources (the registers) can work with the smallest amount of data. There are levels of cache to improve data movement efficiency built on top of

the registers. The further one goes from the registers, the longer the wait and the slower the bandwidth for memory movement, as well as the more data it can hold. The base of this pyramid is often the main memory, which in our case is 128 Mbytes (1 Mbyte = 2^{20} bytes), although this hierarchical scheme can easily be extended to include multiple nodes, and finally the parallel I/O subsystem. (As mentioned previously, we choose to focus our attention in this section on the single node model, which includes the registers through the main memory subsystem.)

Despite having 40 internal floating-point registers, the application program can only address the 8 floating point register stack. In many cases, this was a significant bottleneck to overall performance.

The next level of memory hierarchy is the primary on-chip non-blocking L1_data and instruction cache. For the Pentium Pro processor, these are 8 Kbytes each and are two-way set associative, write-back and write-allocate. Reading data from L1 can be done at 1600 million bytes per second or 1526 Mbytes/sec, which amounts to one double per CPU cycle. Writing data can also be done at the same speed. Furthermore, two CPUs can be simultaneously reading and writing data to/from their perspective L1 caches. We prefer, however, to simplify the discussion by temporarily ignoring these issues. More to the point, we claim it is rare that a scientific application will have many reads and writes to the L1_data cache occurring at the same time.

The next level of memory hierarchy is the non-blocking unified off-die L2 cache. In our case, the L2 cache holds 256 Kbytes and can theoretically sustain 1600 million bytes per second reading exclusive or writing. All cache lines are 32-bytes wide.

A compute node has a memory subsystem with 128 Mbytes. The PCI bus operates at 33 MHz. The memory bus runs at 66 MHz., is 64 bits wide, and can support a data transfer every bus cycle. That is, it can do one 8 byte (64 bit) transaction every bus clock. This amounts to a bandwidth of an 8 byte read or write to L2 every 3 cpu clocks or 1600/3 = 533 million bytes/sec (508 Mbytes/sec.)

The first step in our investigation was to find out how fast we could do simple floating-point operations from various levels of memory. First, we looked at how quickly we could load and pop the data (from various places on the hierarchy) onto the floating-point stack. Second, we looked at how we could use this information to build simple kernels like element vector multiply (x(i) = y(i)*z(i), i= 1,...,n) or matrix-matrix multiply (for MP LINPACK).

It is important here to emphasize the specific nature of our study. Other research papers and/or projects tend to look no further than how fast memory can be moved. But for us, since our final goal is ultimately floating point calculations, by necessity our investigation targeted the floating-point stack. IA-32 instructions like REP MOVS (repeat move string long), although interesting for timing memory movement, are not sufficient to meet our final goal. Following is a discussion of those results and observations that bear on the benchmarks and applications discussed later in this paper.

Overheads Due to the Instruction Decoder

There are three instruction decoders on the Intel Pentium Pro processor. These decode complicated IA-32 instructions into simple uops. Only the processor uses the uops: a programmer can not directly code in terms of uops. Only one of the decoders can decode "complex" instructions. The following problems are directly based on overheads due to the decoder. While none of these problems are measurable when moving data from memory, they can be observed when we know the data lies in the L1_data cache.

Reading data from the L1_data cache onto the floating point stack has a theoretical upper bound of one double per clock or 1600 million bytes a second. Using integer touches like "movl 8(%eax), %ebp," as opposed to floating point touches like "fldl 8(%eax)," we have come within a percent of this speed. However, using floating point loads and pops (we say pops instead of stores because the typical case is that we have many loads followed by add-pops or mul-pops and then perhaps only one store), we observed stalls every time an instruction passed over a 16-byte instruction boundary. A random sampling of floating point showed that floating point instructions are typically 5-7 bytes in length. This means that typical compiled floating point codes tend to run at only 80 percent efficiency out of the L1_data cache, encountering a stall every fourth instruction.

Our best code, which tried to avoid this problem, peaked at 1450 Mbytes/sec out of a possible 1526 Mbytes/sec. To achieve this level of performance, we had to use a potentially unrealistic degree of loop unrolling. We also had to keep the offsets small so that the instructions could be decoded in a smaller number of bytes. (A floating point load off of a location in a register can be a two-byte instruction if there is no offset, a 3-byte instruction for offsets up to 128 bytes, and a 6-byte instruction or more for larger offsets.) Another way of viewing this is if you know that you have X data loads from L1_data, instead of taking minimally X cycles, it is likely to take 1.25*X cycles.

These observations pointed out a critical performance issue: codes tend to run faster when the instructions are simpler. Complicated instructions can lead to a stall because only decoder 0 can decode the complicated instructions. Things like multiplying an element from

memory with an element in floating point stack location 0 can be implemented with a single IA-32 instruction, but require several different micro-operations to execute. Codes may often be faster if they are implemented with simpler instructions. For our example, one could first do a load and then in the second instruction do the multiply.

Floating Point Registers and Pipelining

When moving data from L2 or main memory, we often found that we needed to touch several different cache lines in order for the pipeline to be deep enough to obtain the faster bandwidths necessary for scientific calculations. This was one of the cases where having too few floating point registers was immediately apparent. In some cases, we sped up codes by interleaving integer touches of cache-lines before we actually did the explicit floating-point load onto the stack. This was critical to performance tunings of matrix multiply for MP LINPACK, for example.

An important finding was that a single 200 MHz Pentium Pro processor cannot saturate the memory bus bandwidth. That is, out of the 533 million bytes a second or 508 Mbytes/sec, the fastest bandwidth we achieved was around 428 Mbytes/sec or 85 percent of the maximum theoretical bandwidth. Two 200 MHz CPUs, on the other hand, achieved 491 Mbytes/sec. Several tricks were used to achieve this higher performance. We needed to synchronize the CPUs at critical points using a special utility we created. We needed both CPUs to saturate the bus (although higher frequency Pentium II processors are more likely to saturate their bus). We also needed to unroll our floating point calculations sufficiently enough to ensure that there were always several outstanding cache-line requests at once, such as touching doubles X(1), then X(5), then X(9), etc., before attempting to access X(2). This enabled the pipeline to remain busy. Using a second processor to access data from main memory tended to yield two benefits: not only could we then issue instructions fast enough to saturate the bus, but we could also have a second set of eight floating point stack locations by which we could unroll things. Because there were two benefits, this enabled some memory-bound codes to enjoy greater than the 15 percent improvement possible from just saturating the bus. In fact, some memory-bound codes when run on two CPUs actually achieved around a 30 percent benefit. (Naturally, cache-bound codes often achieved a 2x improvement.)

Unfortunately, using two CPUs is sometimes a double-edged sword: accessing different DRAM pages always caused an expensive stall. Although this can and often does happen with a single CPU, it happens far more readily with two CPUs. When using the second processor, the best method is to access alternate cache lines in the same DRAM page. This will allow codes to make effective use of 16 floating point stack locations instead of just 8, and will also prevent the thrashing of memory (page miss penalties) if the different CPUs are working on entirely different DRAM pages. Another big benefit of alternating cache lines is that each processor avoids having to snoop modified cache lines from the other processor's caches.

The difficulty with studying simple kernels is that the critical cases that actually occur in practice are sometimes overlooked. We found our simple examples of moving data from a single vector in L2 or main memory onto the floating-point stack to be insufficient. We therefore launched a study involving the movement of two vectors at the same time, which we found to be simple enough to optimize and realistic enough to capture the major bottlenecks.

Using our new set of kernels, we made several interesting observations. For starters, we found that accessing data from L2 onto the floating point stack tended to run at 800 Mbytes/sec in this somewhat more realistic mode. We found this a little disappointing since it represented only just over half the bandwidth theoretically possible. We also studied the impact of touching one vector from main memory and another from L1. This has a theoretical bandwidth of 16 / ((8/1526)+(8/508)) = 763 Mbytes/sec; however, it was often harder to achieve more than 530 Mbytes/sec. This implied not only that there was no overlap when loading from the two separate places but that the two loads interfered with one another. For touching two vectors from memory, we found that pre-touching at least one of them first with integer touches in small enough chunks to keep the data in L1 allowed performances up to 320 Mbytes/sec out of a possible 508.

We have illustrated how using the registers effectively can improve performance. There are several cases where simply not having enough registers hurts performance. The application CTH, illustrated in greater depth later in this paper, is a case where the same code and same data set tends to produce more loads on IA-32 than other architectures. Another example is an application called MPSalsa, which was a Gordon Bell (fastest real application running on a supercomputer) finalist at IEEE's Supercomputing 1997. At the core kernel, there were a series of matrix vector products from memory. Since the size of the vectors were small (around 24), they should have been able to fit into floating point registers. Instead, there was a significant overhead introduced by interleaving loads from memory with loads from L1 as described above.

Putting It All Together

We concentrated on a single floating point kernel called *element vector multiply* (EVM). This kernel sets X(i) = Y(i)*Z(i) as i goes from one to one million with double-precision vectors X, Y, and Z. Since this involves 24 million bytes of data, it is clearly a problem coming from main memory. Note that the write-back, write-allocate nature of L2 suggests that each operation involves loading X(i), Y(i), and Z(i), and then storing X(i). This is 4 doubles moved from main memory, which theoretically could be done in 4*3 CPU cycles, or one flop done every 12 CPU cycles. On a 200 MHz processor, that would suggest 16 MFLOPS or so. However, observed performance was 4 or 5 MFLOPS. We then set out to determine where the loss of performance was going given what we learned about simple memory movement kernels.

DRAM page misses are just one slowdown. The read of Y(i) followed by the read of Z(i) causes a page miss that halves bandwidth. We also have to read X(i) due to the write-allocate write-back memory mode. This causes another page fault. In write-back mode, data is only written back to memory to make room for something else to be brought into cache. This is eviction: one cannot control when data will be evicted. There is a penalty for intermixing reads and writes to memory. Recall that the 200 MHz Pentium Pro processor cannot keep the memory controller completely busy, thus utilizing 85% of the memory bandwidth.

We then set out to try to improve the speed. We blocked the loop such that we streamed in about 4K of the vector Y (about half of L1_data). This avoids the page faults that we get for alternating reads of X, Y, and Z. Then we loaded the vector Z into the other half of L1 (getting only one page fault for the initial read of Z). We then performed the multiply X(i)=Y(i)*Z(i) over the elements loaded into L1. This results in reading X(i) from memory due to the write-allocate memory model. The result X(i) is written to cache. We repeated this touching Y, touching Z, writing X until about half of L2 was filled with X's modified lines. Now if we were to continue we would start to evict lines of X as we read in Y and Z. This would cause page faults and drop performance back down. Once about half L2 is filled with X, we could do a Cougar instruction called flush_cache (which does the protected assembler WBINVD instruction) to write the modified data in the caches back to memory.

This blocking algorithm looks like:

```
for(sizeof(X/128K))
{
      for(128K of X)
      {
            touch 4K of Y
            touch 4k of Z
            calculate 4k of X=Y*Z
      }
      flush_cache()
}
```

The touch of Y runs at about 425 million bytes/sec. The touch of Z runs at about 390 million bytes/sec due to having to evict the dirty lines of X from L1 to L2. The multiply of 4k of X=Y*Z runs at about 200 million bytes/sec. It was a bit mysterious that this performance was not higher. The flush cache runs at 250 million bytes/sec peak (our own versions of flush cache appeared to run no faster).

The total throughput is then:

```
1/(1/425 + 1/390 + 1/200 + 1/250)
= 71.85 Mbytes/sec
```

written to X. The flops/sec is then 71.85/8 = 8.98 MFLOPS. This is still not close to the 16.6 MFLOPS, but it is 50% better than the naive loop.

The final set of experiments we made was on write-combine caching. Write-combined memory avoids reading the data before you write it. Also, when writing a whole cache-line, the write is "combined" and sent to the memory bus as one request. Note that write-combine doesn't read/write to the caches; rather, data is transferred directly to/from memory so cache coherency issues have to be addressed. Coherency can be handled by flushing cache at the beginning of the EVM routine and, if necessary, at the end of it also. We achieved about 16.8 MFLOPS with stride 1 write-combine EVM.

The write-combine memory model appears to be useful in kernels that involve writing large pieces (greater than L2 size) of contiguous data to memory. Using write-combine on a Pentium Pro processor-based system proved somewhat challenging; however, it is our untested understanding that the methodology is much easier on Pentium II processor-based workstations. If this is the case, then a great deal of our efforts can be applied to Pentium II processor-based platforms, thus enabling many users around the world to take advantage of our work.

Hardware Counters

When running applications on the Intel ASCI Option Red Supercomputer, it is often useful to know what portion of the data is running from what portion of memory. On a Windows NT box, a utility like Intel VTune[5] might find this information. However, in our operating system environment, more closely resembling UNIX, this was not an option, especially on applications

too large to fit on a single workstation. Therefore, we accessed the hardware counters directly. The ones we found most useful for studying memory movement were PP_DATA_MEM_REFS (0x43), PP_L2_LINES_IN (0x24), and PP_DCU_LINES_IN (0x45). Assuming that the number of references per element in every cache line accessed was the same, reads and writes rarely overlapped, and that the vast majority of data references were all double precision loads to the floating point stack, we generated the following observations:

The fraction of data from Memory, FracM, is

$$FracM = \frac{MIN(PP\_L2\_LINES\_IN * 4, PP\_DATA\_MEM\_REFS)}{PP\_DATA\_MEM\_REFS}$$

The number of L2 and L1 hits is

PP_DATA_MEM_REFS-PP_L2_LINES_IN*4,

and the number of just L2 hits is
MIN(ABS(PP_DCU_LINES_IN - PP_L2_LINES_IN)*4, PP_DATA_MEM_REFS)

The fraction of data from L2 is then

$$FracL2 = \frac{(Number\_of\_L2\_hits) * (1.0 - FracM)}{(Number\_of\_L2\_hits - Number\_of\_L1\_hits)}$$

The fraction of data from L1 is then
FracL1 = 1.0 - FracM - FracL2.

While the foregoing assumptions are simple and do not always apply, they gave us a useful estimate to work with. We could then apply this estimate to our observations about the overheads incurred when accessing data from the various levels of memory resulting in overall performance estimates for an application. For example, we discussed earlier that five cycles are typically used to access four elements from L1, so that the minimum number of cycles for accessing the data that was in L1 might be 1.25 * FracL1. Similarly, fudge factors of 1.59 existed for FracL2, and 3.57 (or 3.13 if dual processor) for FracM. An example of a situation where we used this is shown in the discussion on the application CTH.

Performance Tracking

Most of the early applications work on the Intel ASCI Option Red Supercomputer was designed to validate the soundness of the system design and its ability to scale to thousands of nodes. This work was quite successful with several applications (including some full production applications) running on up to 4500 nodes.

While it is important that the ASCI Option Red Supercomputer functions correctly, it is equally important that the system delivers the expected performance. To track system performance, we created a performance benchmark suite. The goal of this suite was to produce a handful of numbers to assess system performance. The

System	si238	si58	babyflop
Software Release	WW34a_1	WW45	1.2 WW39
Date tests were ran	9/17/96	12/31/96	12/4/97
Livermore Loops			
AM MFLOPS	33.9	42.6	48.3
GM MFLOPS	29.6	33.9	38.9
HM MFLOPS	24.3	25.9	29.1
Minimum MFLOPS	5.9	5.7	6.0
Maximum MFLOPS	61.4	111.8	118.4
Standard Deviation MFLOPS	16	28.4	29.4
Comtest			
Bandwidth - MBytes/sec	272.4	302	302
CSEND Latency - υsecs	12	10	9
Stream Test			
Copy MBytes/sec	85.9	109.1	114.9
Scale MBytes/sec	107.2	108.6	108.7
Add MBytes/sec	128.7	129	130.0
Triad MBytes/sec	128.5	129.3	129.4
Matrix Multiply			
F77, per-node MFLOPS	62.4	54.6	55.3
libkmath, per-node MFLOPS	119.4	111.1	112.3

Table 1: Results from the performance tracking benchmark suite. The tests are not strongly dependent on the number of nodes. These particular tests used four nodes. None of these tests used the second processor for computation. The System names refer to internal systems at Intel.

performance tracking suite includes the following codes:

Livermore Loops: A measure of the performance of the Fortran77 compiler with loops typical to scientific computing. The arithmetic (AM), geometric (GM), and harmonic means (HM) are reported as well as the range and standard deviation in the MFLOPS.

Comtest: Measures the bandwidth, latency, and standard deviation for a pair-wise, nearest neighbor ping-pong test.

McCalpin Stream: Measures performance of memory intensive applications [9]. Specific tests are vector copy, element-wise scale and add, and the triad (i.e., a(i)=a(i)+b(i)*c(i)).

Parallel Matrix Multiply: Measures performance of a parallel matrix multiply. The performance per node is reported in MFLOPS for a 4-node multiplication of order 300 matrices.

The performance levels are reported in Table 1 for several dates spread out over the course of the project. The numbers have largely stabilized, and significant additional improvements are not anticipated. The Livermore Loop and Stream test numbers are in the same ballpark as those from other high-end workstations. The communication numbers are among the best ever reported for an MPP system. Finally, the matrix multiplication numbers provide a measure of compiler performance by comparing MFLOPS rates for compiled and assembly-coded multiplications. The compiled code is a factor of two slower than the assembly code, which is not unusual compared to Fortran compilers on other high-end workstations.

These tests provide a good relative measure of the system performance. They are not very good, however, at detecting systematic errors in the system's performance. To resolve this issue, we needed a benchmark for which we have an analytic performance target. If we match this target, then we know our system is performing as it should.

An application well suited to this type of analysis is MP Quest [13], an ab initio quantum chemistry program developed at the Sandia National Laboratories. In an earlier study[10], we analyzed the nboxcd() kernel from MP Quest. This kernel resembles a modified dense matrix multiply operation. Our analysis showed that this kernel should run somewhere between 110 MFLOPS to 130 MFLOPS (depending on the state of the L2 cache prior to the kernel's operation).

We created a stand-alone benchmark program based on this kernel. Table 2 compares results for these tests built with the PGI compiler and the Intel C/C++ Compiler for Win32*systems. Three different releases of the PGI compilers are included: 9/96 (release 1.1), 12/96 (release 1.2-5), and the 12/97 (release 1.6-3). The Intel C/C++ compiler (9/96 release) is the Pentium Pro processor reference compiler developed by Intel. These single node computations were carried out on a 200 MHz -based node. These tests used two forms for the benchmark: one with the original code and the other with the loops unrolled. The expected optimum performance ranges are from 110-120 MFLOPS.

The Intel C/C++ compiler hits the target performance. This compiler is highly optimized for the Pentium Pro processor so its high performance is not surprising. The PGI compilers are well short of the target performance. (PGI is still working on the compiler, however, and future releases will hopefully close the gap.)

MP LINPACK Performance

MP LINPACK is a well known benchmark for high performance computing. The benchmark measures the time it takes to solve a real double precision (64 bits) linear system of equations with a single right-hand side. On December 4, 1996, we set a new world record for MP LINPACK by running the benchmark in excess of one TFLOPS. At that time, the Intel ASCI Option Red Supercomputer was only 80% complete, but that was more than enough to break the MP LINPACK TFLOPS barrier. Actually, the previous record was 368 GFLOPS so we did not just break the record, we shattered it!

While the rules for the LINPACK benchmark require use of the standard benchmark code, MP LINPACK lets you rewrite the program as long as certain ground rules are followed [6]. Our MP LINPACK code used a two-dimensional block scattered data decomposition with a block size 64 [9]. The algorithm is a variant of the right looking LU factorization with row pivoting and is done in

Code	PGI 9/96 (Rel 1.1)	PGI 12/96 (Rel 1.2-5)	PGI 12/97 (Rel 1.6-3)	Intel C/C++ Compiler
Original Kernel	26	56	67	83
Kernel with unrolled loops	30	75	87	120

Table 2: Performance in MFLOPS for the NBOXCD() Kernel from MP Quest.

accordance with LAPACK [1]. The parallel implementation [4,8,15] used a two-dimensional processor mesh and did a block wrapped mapping of the matrix. Columns of processors cooperated synchronously to compute a block of pivots that were then passed asynchronously across the rows. A look ahead pivot was used to keep pivoting out of the critical latency path. We report timings for real floating point operations and not "macho" FLOPS obtained by using Strassen [14] (or Winograd [16]) multiplication. The code explicitly computed all the relevant norms and did several rigorous residual checks to guarantee accuracy. The matrix generation was identical to ScaLAPACK version 1.00 Beta, which is a standard MPP package for Linear Algebra [2].

The benchmark results are maintained in the LINPACK Performance Report: "Performance of Various Computers Using Standard Linear Equations Software" by Dr. Jack Dongarra at the University of Tennessee [6]. He has accepted our TFLOPS entry into his 12/16/96 report, which is available on the web [6], e-mail, and ftp. RMAX was 1.068 TFLOP, NMAX or N was 215000, and N1/2 was 53400. N1/2 is the minimum problem size (to the nearest 100) such that half the RMAX performance was achieved. That is, over half a TFLOP was achieved on this machine using a problem size of 53400. The RMAX was found on 12/4/96, and N1/2 was found on 12/6/96. The number of floating point operations done is roughly (2N^3)/3 for a problem of size N.

The MP LINPACK 1.3 TFLOPS run (on 6/9/97) was run on 9152 Pentium Pro (TM) 200 MHz processors. RMAX was 1.338 TFLOPS. NMAX or N was 235000. N1/2 was 63000. Both runs used MPI.

The code for the 1.06 TFLOPS MP LINPACK record was derived from programs used to set earlier MP LINPACK records on Intel's Paragon supercomputers. The initial implementation was based on work by Robert van de Geijn [15]. The Delta code was modified to run on the Intel MP Paragon and it used hand-tuned Intel i860 processor assembly code kernels. For the TFLOPS benchmark, these kernels were written in x86 assembly code. For a detailed description of the techniques and algorithms used in this code, see the paper by Bolen et. al. [4]. Our past work with MP LINPACK has shown that for very large problems, at least 93% of the runtime is consumed by the BLAS-3 matrix multiplication code, DGEMM (which computes C=C-A*B). The dual processor code for large DGEMM problems ran at 345 MFLOPS.

Increasing Parallel Efficiency

We employ many techniques to increase parallel efficiency once a code has already been initially scaled. For MP LINPACK, we used the lookahead pivot technique described above. We also used a common optimization technique based on the observation that memory-to-memory copies tend to run at very slow speeds like 80 Mbytes/sec (see our previous results on element vector multiply) but the communication bandwidth of the machine is closer to 400 Mbytes/sec. This means that if an incoming message needs to be copied from an operating system into a user buffer, this takes more time than sending the message. When one posts a message ahead of time, and sends a "handshake" to tell a node it is ready to receive the message (a delicate process since we don't wish to introduce bottlenecks), the communication overheads go down, which enables a code to scale to more nodes. Sometimes it is even faster for a node to send a message to itself, than to call memcpy().

In some cases, we have to reduce I/O to assist in scalability. This is beyond the scope of this paper.

Matrix-Matrix Multiplication

The matrix-matrix multiplication behind MP LINPACK is an upper product update of the form C = C - A*B where C is large with usually slightly more rows than columns, and the number of columns of A (and rows of B) is typically small (in our case 64). Disregarding notations contained elsewhere, suppose C is MxN, A is MxK, and B is KxN, where M>=N>>K.

DGEMM has 2*M*N*K flops and at least 2*M*N + M*K + K*N memory references. If K is sufficiently large, cache re-use will be higher, and the loading and storing of C will be amortized. We typically block A into chunks that fit into L1, and B into chunks that fit into L2, and then complete the relevant portion of C before proceeding to the next chunk of A or B. Because L1_data is 8 Kbytes and 2-way set associative, we have found that it is unwise to use more than 4K of data for A in any one given time. For K=64, solving for 8 rows of C by copying 8 rows of A into a scratch space and doing the multiply is ideal for several reasons[8]. First, this means that 8*64*8 = 4096 bytes of A will hopefully remain L1_data cache resident. Since there is no convenient instruction for accessing across a row, and we would prefer to avoid continually updating the integer registers, copying A into a contiguous space helps side-step this problem because we can change the storage format of A. A DGEMM implementation on top of this is also beneficial because the issue of A or A transpose (another DGEMM option) becomes irrelevant since we always assume a copy of A[7]. Furthermore, we would prefer to process a number of rows that are a multiple of the cache line size to avoid additional cache movements when the initial arrays are aligned on cache-line boundaries.

An outer-level blocking outside the row blocking is done on the columns of B and C so that B always remains in L2. For unfavorable leading dimensions of B, another copy can be done on B. However, this can be avoided

within the context of a careful MP LINPACK implementation.

Due to a limitation of the number of floating point registers, the actual inner DGEMM kernel can only access a column of B or C at a time. Furthermore, even though it may be working on eight rows of C, it can only do so with four rows at a time (a cache line size). Each row can be thought of as an independent dot product, requiring a floating point stack location. Accesses to A are repeated each time a multiply is necessary, but fortunately we block A to be L1_data resident. Accesses to B, however, can be amortized over the four different dot products. Furthermore, to reduce the overhead of latency to L2, we typically keep two B's around, which in effect pre-touches the next B needed several cycles before it is first used. This effectively uses seven of the eight available floating point stack locations (four for the four dot products eventually going into C, two for B, and one for A loads). The whole length of all four dot products are unrolled to minimize overheads.

An unexpected benefit (about a five percent improvement) was observed by including the "fxch" floating point exchange instruction in selected points within our assembly DGEMM. Ironically, the fxch instructions were inserted in locations that did not impact the final result. That is, just before adding stack location 0 to 1, we would occasionally exchange stack location 0 and 1 first, thus adding stack location 1 to 0. Commutativity ensures these are the same, but apparently internal registers allocated to the tasks by the micro-operations treated the two situations somewhat differently. At one point we believed that the unnecessary fxchs were throttling the rate of the retiring instructions, bringing them in sync with the decoding instructions. But we also found that the spurious fxchs were only beneficial when data was running from cache. Around memory movements, taking some of the fxchs out again improved performance further. (This makes sense since something is more likely to occur around the large latency of a memory touch.) Although the fxch is supposed to be a "free" instruction, it takes up space in the reservation pool which has a limited capacity of 40 micro-operations. Exceeding this capacity leads to a stall.

We also used integer touches to pre-fetch C before it was needed. In effect, we would touch a cache line of C, do the 4 dot products, and then load C in to add it to the results.

Finally, when things were optimized on one processor, we split the matrix multiply up on two CPUs to maximize single node performance. Recall that a certain number of columns were blocked off of B and C to keep a strip of B in L2. The resulting matrix multiply was further stripped into groups of rows such that the relevant portion of A would remain inside the L1_data cache. We simply had one CPU take the odd group of rows and the other take the even so that both CPUs would be working on distinct, but close, portions of memory. Since B is not written to, having both CPUs share chunks of B in their respective L2 cache is not a problem.

Pieces of the DGEMM created for MP LINPACK were ported into the Intel Math Kernel Library currently available for Windows NT [5,7].

Other BLAS

MP LINPACK also relies partially on a matrix triangular solve with many right-hand sides. The upper triangular matrix is small (64x64), but the right-hand sides are large. We found that assembly tuning pieces of the upper triangular solve, interleaved with calls to DGEMM, yielded very high performances. The right-hand sides were split between the processors.

We are currently involved in providing UNIX-gnu-based optimized BLAS (and FFTs) for the Intel ASCI Option Red Supercomputer. But we also have efforts underway to provide extended precision math kernels. IA-32 naturally does work in 80-bit arithmetic. If we make an effort to directly support computation done in this framework (special IA-32 instructions exist for loading and storing 80-bit quantities to get around the 64-bit conversions), then some iterative codes might run faster. It is unlikely that the MFLOP rate will go up since doing 80-bit memory transactions is slower than their 64-bit counterparts (bus widths are usually 64-bits). However, the increased accuracy could enable less work to be done to ensure a final acceptance criterion, which would mean getting the answer faster. We are also looking into software-extended formats such as 160-bit arithmetic for this machine.

An Application Example

CTH is an Eulerian-Lagrangian code used at Sandia National Laboratories for shock physics studies. It contains approximately 440K lines of Fortran code, spread among ~1600 files. A parallel version of this code was developed for the Intel Paragon supercomputer at Sandia prior to the installation of the ASCI Option Red Supercomputer. Studies of this code showed that it scales nearly linearly with the number of computational nodes employed, suggesting that this code is appropriately balanced from the “massively parallel point of view.” This code is in continuous use on the Intel ASCI Option Red Supercomputer and has been run for extended periods (~150 hours) on a sizable number of nodes (2048), as well as having had a few limited runs on 4500 nodes, using over 100 Mbytes of the 128 Mbytes available per node. (It should be noted that the limiting factor on the duration of the full-machine runs was the machine schedule, rather than any hardware or software problems.) This is clearly a real application that can take advantage of the full system, and one where getting the optimal performance

has a real payoff. For example, a ten percent improvement would cut a 150 hour run down to 135 hours, shaving off over half a day, which is useful since management is often waiting on the answers and the queues for future runs are usually full.

Initial discussions with the CTH group revealed that the serial version of CTH, which runs on a wide variety of platforms, does not have well-defined kernels that could be tuned to provide significant speedup to the full code. Nevertheless, we examined a few of the most significant routines in order to spot repeated patterns that might be improved wholesale. The 3D EFP problem was chosen as a representative data-set for work on the CTH code. Profiling indicated that one of the most significant routines was ELSG, which is around 3700 lines of code. Using the ideas presented earlier in this paper, we investigated the performance of this routine.

We used the performance counters to gain an understanding of the memory characteristics of this code. This showed that the program's performance was bounded by the costs of memory movement, a surprising result given that the data appeared to be in the caches. More specifically, we found that the fraction of data from L1 was .85, the fraction from L2 was .12, and the fraction from memory was .03. Given the number of floating point operations based on the PP_FLOPS counter, this implied the maximum achievable performance for the particular data set was 27 MFLOPS. The actual performance was around 17 MFLOPS. Additional losses were due to branch misprediction, speculative execution, floating point dependencies, and other trouble spots.

We then took the major routine and created two instances of it, one for each CPU. Each CPU then did half the work with appropriate synchronizations being added to ensure correctness. The modified code ran slightly slower than the original code. Normalizing the original code's time to 1.0, the modified code ran at 1.13. The dual processor code ran at 0.61. Perfect speed-up was not possible because not every computation could be parallelized.

There were several other significant observations. We tried to avoid latency stalls associated with computing logical values by precomputing them, combining them, or removing them when possible. We used reciprocals when appropriate in order to minimize divides. We interlaced independent calculations to avoid floating point calculation stalls. We used the monitors to see where the data movement was going, and ended up achieving about 60-70 percent of the peak observable based on the memory movement.

One of the important functions of a supercomputer is the ability to run extremely large problems on an extremely large number of nodes reliably. CTH is an example of an application that has done just that. It has not only run on the full machine, but it has done so for a large number of uninterrupted hours.

Conclusions

The Intel ASCI Option Red Supercomputer is in routine production use. The machine is successfully addressing the problems that motivated the DOE to purchase it. One feature of the machine that we haven't talked about is its ability to rapidly switch between classified and unclassified operating modes [12]. While this isn't a performance issue, it does make the machine more broadly usable and therefore impacts the application programmer directly.

Performance on such a complex machine means many things. It means understanding single node performance, and knowing where the memory bottlenecks lie. In this paper, we have briefly discussed some of our more important findings in that area. It means understanding where the cycles are going for applications like CTH using tools such as the hardware counters. It means taking the care to do specialized tunings like asynchronous message passing and lookahead pivots to make codes like MP LINPACK parallelize well across a large number of nodes. It means experimenting with techniques like write combine memory to see when this is most beneficial. It means creating a performance suite to ensure that the compiler and the operating system are always running at optimal speeds.

Our performance and optimization studies are an ongoing effort. In this paper we have highlighted some of the major efforts and discoveries. Our final goal is to obtain correct codes running as fast as possible. We have demonstrated high theoretical peaks for important benchmarks like MP LINPACK. Application codes have been running on the machine for over a year now, even though we completed this supercomputer in June 1996.

Acknowledgements

Many people have worked on the MP LINPACK benchmark over the years. In addition to Greg Henry's work on the program, valuable contributions were made by Robert van de Geijn (University of Texas in Austin), Bob Norin (Intel Corp.) Brent Leback (Axian Corp.), Stuart Hawkinson (Axian Corp.), and Satya Gupta (Intel Corp.).

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., *LAPACK Users' Guide*, SIAM Publications, Philadelphia, PA, 1992.

[2] Blackford, S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S.,

Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C., *ScaLAPACK Users' Guide*, 1997, SIAM Publications, Philadelphia, PA 19104-2688, ISBN 0-89871-397-8.

[3] "Computers–Design Issues and Performance." Technical Paper in Supercomputing 1996, Proceedings of Supercomputing '96, Pittsburgh, Pennsylvania, http://www.supercomp.org/sc96/proceedings.

[4] Jerry Bolen, Arlin Davis, Bill Dazey, Satya Gupta, Greg Henry, David Robboy, Guy Schiffler, David Scott, Mack Stallcup, Amir Taraghi, Stephen Wheat, LeeAnn Fisk, Gabi Istrail, Chu Jong, Rolf Riesen, Lance Shuler, "Massively Parallel Distributed Computing: World's First 281 Gigaflop Supercomputer," Proceedings of the Intel Supercomputer Users Group 1995, http://www.cs.utk.edu/~ghenry/isug.ps.

[5] The Intel Performance Library suite, http://developer.intel.com/design/perftool/perflibst/

[6] Dongarra, J.J., "Performance of various computers using standard linear equations software in a Fortran environment," Computer Science Technical Report CS-89-85, University of Tennessee, 1989, http://www.netlib.org/benchmark/performance.ps

[7] Greer, B., Henry, G.,"High Performance Software on Intel Pentium Pro Processors or Micro-ops to TeraFlops," Proceedings for Supercomputing 1997, San Jose, CA.

[8] Gupta, S., Hawkinson, S., Henry, G., "Performance Of Matrix Matrix Multiply (DGEMM) For MP-LINPACK On Pentium Pro Processors," An Intel Internal Whitepaper, January 1996.

[9] Hendrickson, B.A., Womble, D.E., "The torus-wrap mapping for dense matrix calculations on massively parallel computers," SIAM J. Sci. Stat. Comput., 1994, http://www.cs.sandia.gov/~bahendr/torus.ps.Z

[10] S. Gupta and T.G. Mattson, "Optimization of MP QUEST for the ASCI Option Red System," Intel TFLOPS Project Research Report, 1996.

[11] Intel optimization manuals, http://developer.intel.com/design/pro/manuals/242816.htm and 242690.htm

[12] Mattson, T.G., and Henry, G, "The ASCI Option Red Supercomputer," Proceedings for ISUG 1997, Albuquerque, NM.

[13] Sears, M., *MP Quest User Guide*, Documentation distributed with the MP Quest program.

[14] Strassen, V., "Gaussian Elimination is not Optimal," Numer. Math. Vol. 13, 1969, pp. 354—356.

[15] van de Geijn, R.A., "Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC(R)/860 Systems," 1991 Annual Users' Conference Proceedings. Intel Supercomputer Users' Group, Dallas, TX, 10/91.

[16] Winograd, S., "A new algorithm for inner product," IEEE Trans. Comp., Vol. C-37, 1968, pp. 693—694.

[17] Pentium Pro Processor technical documents, http://www.intel.com/design/pro/manuals/.

Authors' Biographies

Greg Henry received his Ph.D. from Cornell University in Applied Mathematics. He started working at Intel SSD in August 1993. He is now a Computational Scientist for the ASCI Option Red Supercomputer. He tuned MP LINPACK and the BLAS used there-in. Greg has three children and a wonderful wife. He plays roller hockey, soccer, and he enjoys Aikido and writing. His e-mail is henry@co.intel.com

Pat Fay is presently an Intel computational scientist. He is responsible for assisting the Los Alamos National Laboratory scientists in using the Intel ASCI Option Red Supercomputer. He received his Ph.D. in Physics from Clemson University in 1993 and a Masters of International Business from the University of South Carolina in 1987. His e-mail is pfay@co.intel.com

Ben Cole is the Intel computational scientist on-site at Sandia National Laboratories. For his Ph.D. thesis, he studied transport processes in particle accelerators, comparing experimental results to a numerical model implemented on a parallel architecture. He has a second career as a father to an energetic three-year-old. His e-mail is cole@co.intel.com

Timothy G. Mattson has a Ph.D. in chemistry (1985, U.C Santa Cruz) for his research on Quantum Scattering theory. He has been with Intel since 1993 and is currently a research scientist in Intel's Parallel Algorithms Laboratory where he works on technologies to support the expression of parallel algorithms . Tim's life is centered on his family, snow skiing, science and anything that has to do with kayaks. His e-mail is timothy_g_mattson@ccm2.hf.intel.com.

Achieving Large Scale Parallelism Through Operating System Resource Management on the Intel TFLOPS Supercomputer

Sharad Garg, Server Architecture Lab, Beaverton, OR, Intel Corp.
Robert Godley, Intel Supercomputers, Beaverton, OR, Intel Corp.
Richard Griffiths, Intel Supercomputers, Beaverton, OR, Intel Corp.
Andrew Pfiffer, Microprocessor Products Group, Beaverton, OR, Intel Corp.
Terry Prickett, Intel Supercomputers, Beaverton, OR, Intel Corp.
David Robboy, Enterprise Server Group, Beaverton, OR, Intel Corp.
Stan Smith, Microprocessor Group, Beaverton, OR, Intel Corp.
T. Mack Stallcup, Intel Supercomputers, Beaverton, OR, Intel Corp.
Stephan Zeisset, Microprocessor Group, Beaverton, OR, Intel Corp.

Index words: operating system, TFLOPS, scalable, parallel, GB/sec, TB.

Abstract

From the point of view of an operating system, a computer is managed and optimized in terms of the application programming model and the management of system resources. For the TFLOPS system, the problem is to manage and optimize large scale parallelism.

This paper looks at the management in terms of three key topics: memory management, communication, and input/output. For memory management, we discuss some of the design decisions made including the appropriate use of demand paged virtual memory in the system. For communication, we describe the software protocols and interactions that permit a system of 4500 nodes to approach the maximum hardware performance. For I/O, we look at the problem of funneling data from many computation nodes to a small number of external devices.

Introduction

Providing high performance computing is the overriding goal of the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) program. Some of the performance requirements of the system Intel built for the DOE are as follows:

- a minimum one TeraFLOPS (TFLOPS) sustained floating point performance on MPLINPACK (a parallel benchmark)
- a minimum one Gigabyte (GB) per second of sustained disk I/O to a one Terabyte (TB) file system for a given application
- a reliable file system that can tolerate the failure of one disk without any loss of data

This paper looks at the design tradeoffs and decisions regarding the operating system that were made in order to meet the above requirements. We examine the interrelationships between three components of the operating system:

- memory management
- communication between nodes
- access to the file system

One component we discuss is the Parallel File System (PFS) that makes a sustained throughput of one GB/sec possible. There are performance problems inherent in a file system that is not local to the compute nodes. There also are scalability issues involved with I/O on a system containing 4500 compute nodes with only 18 nodes providing file I/O. We will show the motivation for the system architecture that gives rise to such issues in the first place and discuss the solutions. Inter-node

communication is critical to I/O performance in a system where remote nodes handle I/O requests. The memory management design, in turn, is critical to communication performance. Design decisions in all of these areas were closely interrelated and we will discuss these also.

Each node in the TFLOPS system can be thought of as an enhanced, dual-processor PC with additional communications capability. A node has two 200MHz Pentium® Pro microprocessors and 128MB of memory. It has the capability for symmetric multi-processing between the two processors. On the I/O nodes, a 32-bit PCI bus is added as well as an additional 128MB of memory. This allows off-the-shelf PCI cards to be used for I/O. The nodes are connected via an 800MB/sec bi-directional communications network.

The programming model for this system views an application as a set of cooperating, autonomous processes. Many applications run on all the compute nodes for tens or even hundreds of hours at a time. Each process in the application is independent. If a process exits, all other processes in the application can continue to run. Explicit message passing is used to exchange information between the processes in an application. The UNIX* programming environment is provided for applications. However, providing this environment does not require UNIX to actually run on a compute node. It just requires that UNIX library calls be supported by the OS on the compute nodes. Almost all of the standard UNIX chapter two and three library calls are available to the programmer. Additional entry points have been added for message passing and asynchronous I/O operations.

The architecture of the system is Multiple Instruction Multiple Data (MIMD) with distributed memory. The machine runs two separate operating systems, each specialized for the tasks performed by the two sections of the machine (see Figure 1). One section of the machine contains the service and I/O partitions and runs the TFLOPS Operating System (TOS) [1]. The other section of the machine contains the compute partition that runs the Cougar Operating System [2]. Parallel applications run only under Cougar in the compute partition. The distinction between the two sections of the system is only in the software. The nodes and communications network in both sections are identical. Each section of the machine is scalable. A *scalable* operating system means the number of nodes in the machine can be increased without modifications to the OS. When nodes are added, the performance of a scalable system increases approximately in proportion to the number of additional nodes. This capability is extremely important to achieving the required performance in this system and will be discussed in the memory management section of this paper.

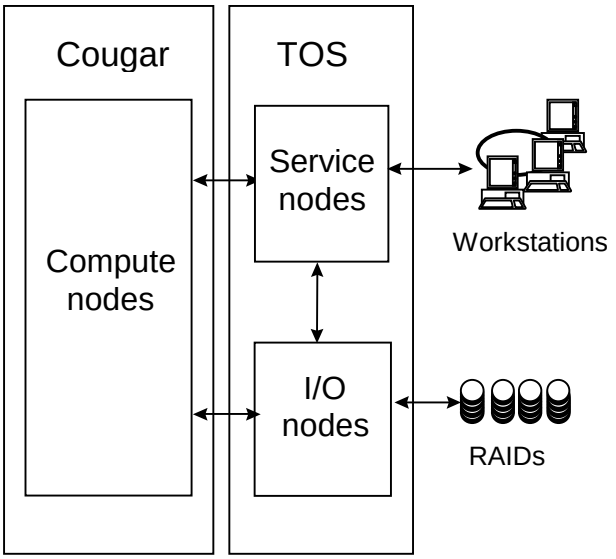


Figure 1: Overview of the TFLOPS system architecture

The two operating systems must communicate in order for applications to run. (In this paper, references to applications always mean parallel applications running on the compute nodes unless TOS processes are explicitly mentioned.) All communications between TOS and Cougar are done exclusively via message passing. While TOS has several communications' protocols, a single protocol is used for communications between TOS and Cougar.

One important interaction between the two operating systems occurs when an application is invoked. All applications are invoked on the service partition, but run on the compute nodes. A TOS process is started by the user and it causes the application to be loaded on the desired compute nodes. This loader process sends messages to one compute node that fans out the messages to the other compute nodes. Once the application is loaded on all the compute nodes, the application is then started by the loader process.

TOS is derived from the Paragon Operating System. It provides UNIX services to the users. All user interaction occurs within the TOS section of the machine. TOS is scalable, yet provides the users with a "single system image." For example, if there are 30 people on the machine, each person may be running processes on any one of the 17 TOS nodes in the service partition. However, to each user it will look like one large computer. This allows access to all components of the system, such as disks and networks, no matter which node in the machine a user is logged into.

The scalability of the TOS section allows a number of UNIX processes to run concurrently, increasing job

throughput and system response for users. A multi-node service partition also increases the aggregate communications bandwidth between the TOS processes that control the applications and the applications running on the compute nodes.

Cougar is derived from the Puma technology developed by Sandia National Laboratories and the University of New Mexico. It is a small OS (<1MB) designed to manage node resources and processes, provide protection, and facilitate message passing. One of the design goals for Cougar was to be small and deterministic. (*Deterministic* means the system will produce consistent execution time for applications from run to run.) Programmers want a deterministic system in order to accurately predict the performance of their applications. This is particularly important for TFLOPS applications where the binaries can be 6-8MB in size, or even larger, and run for hundreds of hours. Cougar must also be scalable, as the final machine requires over 4500 compute nodes to achieve the desired computational requirements.

All I/O requests on the compute nodes are handled by the TOS section of the machine. Each request causes a message to be sent to the appropriate TOS process which then sends a reply, if one is required. This mechanism is used for both standard I/O (standard in/out/error) and file I/O.

This separation of computation and I/O had a profound effect on the design of the system. The remainder of this paper will discuss some of the problems that were caused by this separation. As the solutions to these problems involve both memory management and communications, we first discuss computational aspects of the system, concentrating on memory management. Then we discuss communications' problems and solutions, followed by a detailed discussion of I/O in the TFLOPS system, including a description of PFS.

Memory Management on the Compute Nodes

This section discusses how performance requirements motivated certain design decisions in compute node memory management. It shows how the memory management design is tied to message passing and I/O performance.

The compute nodes run the Cougar operating system, which is optimized for scientific applications with some specialized requirements. In general these applications require the following:

- intensive computation
- massive amounts of data in memory
- fast communication between nodes
- a high performance file system

A key requirement that influences design decisions is that application performance must scale in proportion to the number of nodes. If the program is too slow or requires more memory, the user can run it on more nodes. In principle, it is possible to add more nodes to the system itself to increase its performance. Adding nodes can improve the performance in the following ways:

- More processors are available for higher overall computation speed.
- More memory is available to the application.
- There is higher aggregate memory bandwidth across all the nodes.

Of course, adding nodes does not in itself guarantee better application performance. The application design must also be scalable to take advantage of the additional nodes.

Operating System Design Requirements

An operating system allocates and controls system resources for applications. The compute nodes of the TFLOPS system do not have I/O devices, so the primary resources that must be controlled are memory, processor cycles, and the communications network. The operating system should not get in the way of the paramount goals of application performance and scalability. This dictates several requirements of the system.

Although the Cougar operating system is capable of multi-tasking, we have optimized it for a single process per node. Our model in general is *space sharing* rather than time sharing. In other words, multiple users can allocate disjoint subsets of the 4500 nodes on the system, and run applications on them concurrently, with a set of nodes dedicated to each application. This frees us from having multiple users on any one node and permits some design decisions that will be explained below.

With only one process per node, each process has access to almost all of the physical memory on the node. If users need more memory, they can get it by allocating more nodes.

Design Decisions

A key requirement of the system is message passing performance, which is covered in another section of this paper, but this requirement influences memory management. The Cougar operating system uses the hardware memory protection mechanism of the processor to protect the operating system from users and to protect user processes from each other. Cougar gives each process access to a *virtual memory space*, which is a set of addresses that the process can use. The virtual memory

space is mapped by the processor to the underlying physical memory of the computer. By virtual memory, we do not mean demand paged virtual memory; rather, we mean a mapping of virtual addresses to physical memory. The memory for a user process is divided into several *regions*, which are contiguous expanses of virtual memory. Typical regions for a process are its code, data, heap, and stack.

The IA-32 architecture provides options for three memory page sizes: 4KB, 2MB, and 4MB. Cougar uses the 2MB page size, which permits better message performance than 4KB pages. When moving a large message into or out of memory, the message can be moved in large blocks without breaking it up into packets the size of a 4KB memory page. That improves the message bandwidth. For large messages, the bandwidth is almost asymptotic to the hardware bandwidth, and thus Cougar attains approximately twice the bandwidth of TOS (see the next section on Communication for the numbers). Using large pages also slightly improves the computation performance, as it increases the number of Translation Lookaside Buffer (TLB) hits.

Another optimization for message passing is to keep the user's regions of memory physically contiguous. Although separate regions may be far apart in the virtual memory space, each region is a contiguous expanse of virtual memory. However, at the hardware level, these regions consist of *pages* of memory that are mapped to individual pages of physical memory. The pages of physical memory could be scattered. In our implementation, each contiguous region of virtual memory is mapped to an underlying contiguous region of physical memory.

Although physically contiguous memory regions are less important to performance than using the large page size, they still benefit performance. When the operating system kernel is going to use a buffer in user space to send or receive a message, the kernel must *validate* the memory; that is, it must make sure the entire buffer is within memory the user has permission to use. If user memory is physically contiguous, then the entire buffer can be validated at once rather than validating each individual page of the buffer.

In a multi-tasking system, physically contiguous regions can cause fragmentation of physical memory. However, since we are optimizing for a single process per node, fragmentation is not a problem. Also, if we needed demand-paged virtual memory, then we could not use contiguous physical regions.

Demand Paging vis-a-vis Message Passing and I/O Performance

Since the applications on this system need a massive amount of memory, the question is often asked as to why demand paging isn't used? This subsection addresses that question, because the discussion illuminates some issues that tie in to message passing performance and I/O performance.

For massively parallel scientific applications, we discovered that demand paging causes several performance problems. The foremost problem is that demand paging is too slow because the backing store is not local to the nodes. The data must be passed as messages to a file server on another node. Furthermore, there are many compute nodes and few file servers (the so-called *many-to-one problem*). Under conditions of heavy paging, the file servers can become backlogged with requests, and the compute nodes experience long latencies waiting for a page.

Depending on your point of view, we have either described a disadvantage of demand paging or a shortcoming of the system architecture. Why is the backing store not local in the first place? We want to emphasize that there are good reasons for this architecture and they include

- the need for reliable, redundant I/O devices,
- security of classified data, and
- the cost, heat, and space that would be consumed by disks on 4500 individual nodes.

These issues are further discussed in other sections of this paper.

Demand paging was originally devised in order to optimize the throughput on large time-sharing systems. When a process has a page fault on that type of system, other processes can be scheduled to keep the processor busy. This results in the greatest possible parallelism between I/O and CPU cycles. On our system, with only one process per node, typically nothing can happen while a process waits for a page, so the latency to handle a page fault is dead time.

Another problem is that non-present pages add latency to message passing. On a demand-paged system, before receiving a message, the operating system must make sure the memory for the message is physically present. If a page is not present, an operating system can allocate a page and map it in. This requires some processing time with a small increase in latency. If no free page is available, then a page must be flushed to the backing store to make memory available, which increases the latency. If

an incoming message does not completely fill a page, then the page must be fetched from the backing store before it can be written. This also adds latency. Using large pages increases the probability that a message will not completely fill a page. Cougar assumes that all pages are always present, so it only has to validate the memory addresses before receiving a message.

A more subtle problem is the propagation of latency to other nodes. For example, suppose node A is doing computation, but is stalled waiting for a not-present page to arrive. Then suppose Node B needs data in a message from node A before it can proceed. The paging latency on node A propagates to node B. With a large number of nodes, this can have a crippling effect on the system.

Another motivation for using demand paging is that most applications have a relatively small working set. Intuitively speaking, this assumes over a short period of time, most programs use only a small amount of the memory available to them. Over a longer period of time they re-use the same memory. This limits the amount of demand paging required. Scientific applications do not conform to this assumption. Typically, these applications fill all available memory with large arrays of data such as floating point numbers, and they traverse the arrays. This can result in a large working set.

Finally, programmers want deterministic performance; that is, the same behavior from one run to another, so that they can predict the performance of their programs. This is best achieved with resident physical memory.

Communication

In the TFLOPS system, the two operating systems have specific requirements for message passing. The main design goal for Cougar is to provide high performance message passing while perturbing the running application as little as possible. With respect to message passing, high performance means high bandwidth and low latency. For TOS, these characteristics are also desirable, but there are additional constraints.

The TFLOPS network connects the individual nodes of the system. Several features of the network increase the efficiency of message passing; namely:

- restricted access network (unauthorized agents do not have access)
- reliable network (messages are always received unless there are hardware failures)
- two Pentium Pro microprocessors per node

In most networks, authentication is an important part of sending and receiving messages. On the TFLOPS network, security checks are still necessary. However, the checks are not as complicated as those required on a unrestricted network, such as an Ethernet. This reduces the amount of work required to send a message, thereby increasing the efficiency of message passing. Because the network is reliable, the operating systems can assume messages are not lost. This also decreases the amount of work required to send a message, increasing the speed of message passing.

The second processor can be used in several different ways according to application requirements. For example, if an application is message-passing bound, this processor can be used as a message-passing engine to reduce the latency of sending messages. This quite often improves application performance. (Other uses of the second processor are beyond the scope of this paper.)

The network hardware is also designed to provide very high performance. The specifications for the network include a bi-directional bandwidth of 800MB/sec and a latency of 2.5μsec. The remainder of this section will discuss some of the specific methods used in the operating systems that take advantage of these characteristics and provide high bandwidth and low latency message passing. We will also highlight differences between the two operating systems.

Maximum Bandwidth

There are two main factors that influence the bandwidth a communication protocol can achieve. One is the number of times data is copied when a message is sent. The other is the overhead associated with breaking the data up into packets. If a message is greater than a given size, the network hardware requires the message to be sent as separate packets.

Cougar eliminates the need for copying data by leaving the management of communication buffers up to the user. On the sender side, the application passes a pointer to the message to the communications library and guarantees that it will not modify the message until the message has been transmitted. This allows the operating system to directly transfer the message from the application's address space to the network.

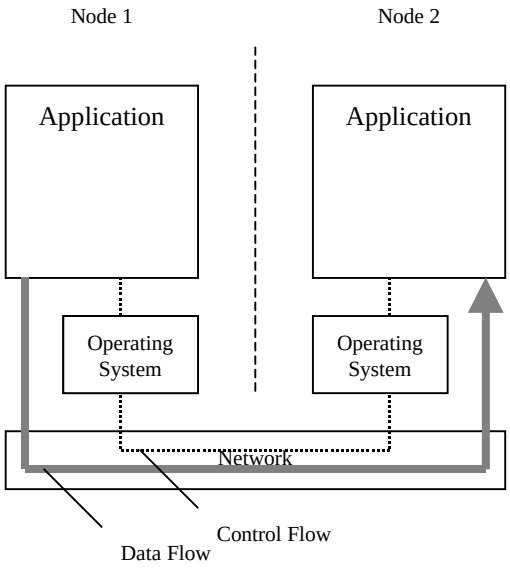


Figure 2: Transfer of an application message between compute nodes

On the receiving side, if the operating system has a pointer to the user's buffer before the message arrives, then the message is said to be *pre-posted.* Pre-posting a message allows the operating system to directly transfer a message from the network to the application's address space (see Figure 2). If a message is not pre-posted, the operating system must deposit the incoming message into a system buffer when it arrives from the network. Then, when the application requests the message, the message must be copied from this system buffer into the application's address space. The Pentium Pro chip set can copy data from memory to memory at a rate of about 80MB/sec. This is an upper bound on message bandwidth if messages are not pre-posted. However, using pre-posted messages, Cougar can attain a bandwidth of 380MB/sec., almost a factor of 5 times faster than if the data is copied.

In contrast, under TOS, a more complex message passing protocol is supported. With this protocol, a message buffer may be modified after it has been passed to the library. Also, the receiver of a message does not pre-post receive buffers. In order to avoid making copies of communication data, we have made extensive use of memory management facilities. On the sender side, transmitted data is marked as copy-on-write in user address space. Transmission of the data is delayed until the corresponding receive operation is posted on the receiver side. As long as the data is not modified by the sender, it is not copied into a separate send buffer. If the sending process never writes to this memory, then the data is transmitted without making any copies. This protocol provides an excellent means of flow control in many-to-one communication scenarios, as almost no memory is needed on the receiver side until the message is actually consumed.

It is important to note that while the use of memory management facilities in TOS allows data transmission with zero copies, it places a fixed cost on each page of data transmitted. This cost is associated with the protection of the data on the sender side and re-mapping of it on the receiver side. Cougar does not have this overhead. The use of physically contiguous, virtual address space in Cougar also allows the use of large packet sizes, up to the hardware limit of 1 MB. This lowers communications overhead because most messages can be sent in one packet. Even for large messages, the total number of packets is kept to a minimum. In contrast, TOS is limited to a packet size smaller or equal to the virtual page size of 8KB. (TOS uses an 8KB virtual page size instead of the 2MB pages that Cougar uses.) This restriction on packet size is imposed because TOS can not guarantee two virtual pages are physically adjacent to each other.

The bandwidth achieved with the TOS message passing protocol is about 190MB/sec for message sizes of 256KB and larger. For Cougar, the bandwidth for the same size message is 380MB/sec.

Minimum Latency

The latency a communication protocol can achieve for small messages is determined by the amount of code executed when a message is sent between two nodes. This means lightweight protocols ensure the best use of a very low latency network.

Cougar uses a lightweight protocol for message passing. It allows a process on one node to open up a portion of its address space to a process on another node. The operating system on the sending and receiving nodes can then transfer data directly from user address space on the sending node, via the network, to the desired memory location on the receiving node. Once this opening into the user's address space is established, many separate messages can be sent without any further overhead.

In contrast, the message passing protocol on the TOS side supports very rich semantics, with type conversions for every element of a message. This severely limits the minimum latency that can be achieved. The protocols to provide this capability are much more complicated than those used in Cougar, requiring more code to be executed.

Another factor that can increase message latency is the flow control mechanism used to guarantee the delivery of messages. Cougar has no flow control at the OS level. Instead, it is left up to the application to make sure

adequate buffers are available for message reception. The responsibility for flow control can be left to the application because the protocol is designed to run on a reliable network. The application only has to deal with buffer management, not the unexpected loss of messages.

On the TOS side, the OS provides flow control which guarantees that no data will be lost regardless of the amount and timing of messages. When a message is sent between TOS nodes, initially, only a small informational message is sent to the receiver. The receiver is thus notified a message is available. The data can then be requested from the sender when the receiver is ready for the message. This is known as a pull model of flow control. To reduce the overhead for very small messages (up to about 120 bytes), the message data can be sent along with the notification message. This data is then buffered on the receiver side as long as buffer space is available. If no space is available, a more complex model is used.

The latency of the TOS message passing protocol is about 90μsec for small messages and about 130μsec for larger messages where the pull model of flow control is used. For Cougar the latency for the minimum size message is 16μsec.

The tradeoff here is that programs running on Cougar can achieve latency improvements over TOS by a factor of 5 to 8. However, the programmer must analyze the message-passing patterns of the application and provide flow control at the application level and also provide sufficient buffer space to handle all incoming messages that are not pre-posted. The programmers writing the ASCI codes are willing to pay this price for performance.

Scalable I/O for Thousands of Nodes

The TFLOPS system is required to sustain a transfer rate of one GB/sec between a set of compute nodes and the file system. This challenge involved solving problems in several interrelated areas. The TFLOPS hardware has a set of devices and I/O buses that is capable of both transferring data at an aggregate rate of one GB/sec and meeting the reliability and security requirements of the system. In order to meet the performance requirement, the file system software must be able to exploit the full bandwidth of the hardware. This is done by TFLOPS Parallel File System (PFS) and the I/O service processes.

PFS stripes user data files across the TFLOPS disk devices, which allows the transfer of data between the compute nodes and the I/O nodes to occur in parallel. For a large enough I/O request, each I/O node can sustain the maximum possible bandwidth of its storage device.

To handle the I/O requests from many compute nodes in parallel and balance the load, a set of I/O service processes is required. However, to minimize copying, the user data moves directly between user space on the compute nodes and the operating system space on the I/O nodes, bypassing the I/O service process. A flow control mechanism is required to fan the communication in from many compute nodes to a few I/O nodes without loss of data or loss of performance. The following section describes the architecture of the I/O subsystem.

I/O Architecture

There were two main approaches considered for providing I/O services to the many compute nodes in the TFLOPS system. One was to attach a disk to each compute node. The other was to concentrate the file system on a small set of specialized nodes that process I/O requests. For a number of reasons we chose the latter option, using Redundant Array of Independent Disks (RAIDs) for secondary storage. The more important reasons for our decision were as follows:

- Reliability

The system must survive any single disk drive failure. A design with a disk per node would add complexity to both the system hardware and software. A RAID is designed to handle the failure of a single disk drive. A RAID stores parity information that allows it to reconstruct user data in the event of a single drive failure. (This operation is described in Appendix A: Single Drive Failure Recovery.)

- Security

The customer decided to configure the TFLOPS system into three sections: a classified section, containing compute, service, and I/O nodes; a non-classified section that also contains compute, service, and I/O nodes; and a "floating" compute section that contains only compute nodes. The floating compute section is attached to either the classified section or to the non-classified section.

For security reasons, once a disk drive is connected to the classified section, it must be "scrubbed" in a precisely defined manner before it can be removed from the classified system. By physically decoupling disk hardware from the compute nodes, reconfiguring the system is greatly simplified. Disks do not move between the classified and the non-classified sections.

- Hardware design issues

A disk per node would increase the per-node power requirements, complicating the system's design and cooling requirements.

- Leveraging existing hardware

The TFLOPS communications network is designed to move data at 800MB/sec. This transfer rate is much higher than the I/O rate to a single disk or RAID device. Utilizing this high speed network hardware simplified the design of the I/O system.

The RAID hardware is described in Appendix B: RAID Subsystem.

TFLOPS File system

The TFLOPS File System is composed of two parts: the UNIX File System (UFS) and the Parallel File System (PFS). PFS is built on one or more UNIX file systems. A PFS file is striped over multiple UFS files in a round-robin fashion. Each of these files is referred to as a stripe file (see Figure 3).

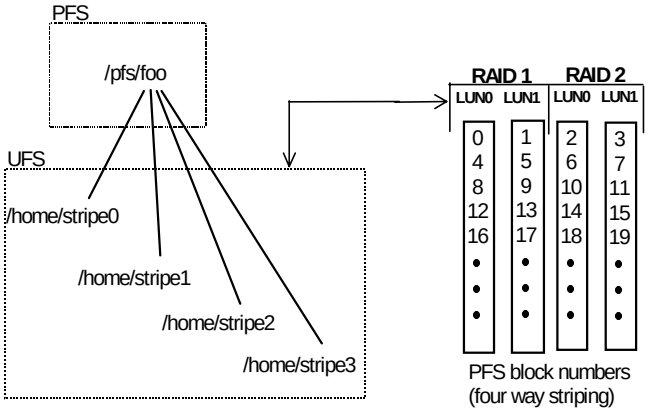


Figure 3: PFS stripe file mapping

The PFS component of the TFLOPS File System provides application programs with the following critical functionality:

- High speed transfer rate

There is no such thing as a disk device capable of transferring data at 1 GB/sec. PFS issues I/O operations to each UFS stripe file in parallel. This allows the aggregate PFS transfer rate to equal the sum of the I/O bandwidths to the individual UFS stripe files.

- Large File Size

  A PFS file may span all available space in each of the stripe file systems. It is therefore possible for a single PFS file to be over a terabyte in size.

The TFLOPS system is comprised of over 4500 compute nodes that have no physical connection to an I/O device other than the high speed communications network. All application I/O is performed via Remote Procedure Calls (RPC) from the compute node to a small number of service nodes. The inherent many-to-one communication problems are handled by I/O service processes coupled with inter-node flow control mechanisms.

The notion of specialized nodes for specific functions permeates the TFLOPS design. Applications run solely in the compute node partition. Standard UNIX programs and the application loader process all execute in the service node partition under the control of TOS. The RAIDs in the system are attached to the I/O nodes. All CPU cycles on the I/O nodes are dedicated to I/O; no other processes run there.

An application is presented a UNIX I/O programming interface through a set of runtime libraries. This interface was enhanced with asynchronous (i.e., non-blocking) read and write operations. The ability to overlap computation and I/O can dramatically improve the per node computational performance.

When an application starts, each compute node is assigned to an I/O service process. Each I/O service process provides service to potentially many compute nodes–by default 256 compute nodes per I/O service process. All communications between an application process and the I/O service process are conducted via RPC's over the high-speed communications network. The I/O service process translates the RPC into a TOS file system request (see Figure 4).

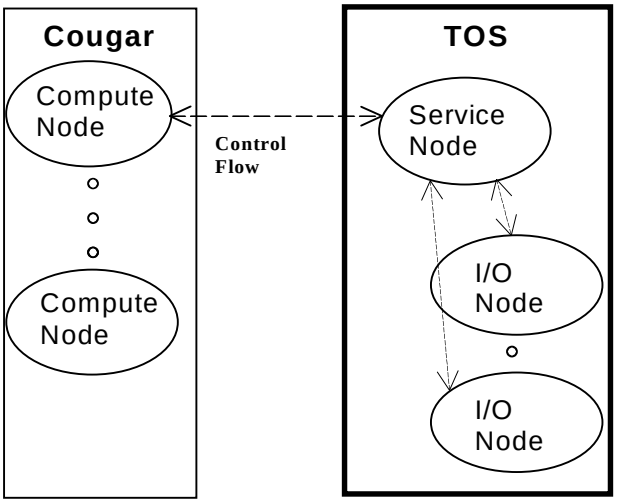


Figure 4: I/O service process control flow

PFS Write/Read Operations

When an application process writes a block of data to a PFS file, an RPC containing the address of the buffer and the length of the data is sent its I/O service process. The data itself is not sent with the RPC, only the control

information is sent. The I/O service process determines which I/O nodes contain the portion of the file being written and sends RPCs to each of those nodes. These I/O nodes transfer the data directly from compute node memory to the stripe files. Data from the application buffer fans out across all affected I/O nodes in parallel thus achieving a high aggregate data transfer rate (see Figure 5).

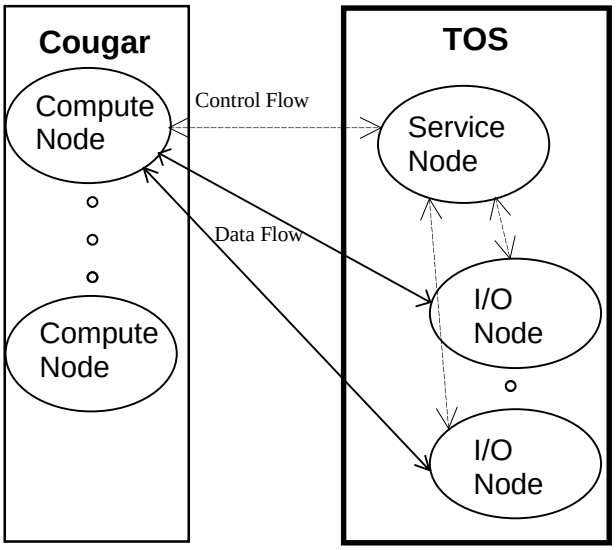


Figure 5: I/O service process control and data flow

When an application process reads a block of data from a PFS file, the processing and parallelism are similar to a write operation. I/O nodes fan in the file data directly from the stripe files to the application buffer.

I/O Flow Control

The I/O service processes receive I/O operation requests from the set of compute nodes mapped to them. TFLOPS file system requests are generated on behalf of the requesting compute nodes. To achieve maximum parallel performance, asynchronous read and write file system operations were implemented. These operations allow the application and the I/O service process to issue an I/O request and then continue processing. An I/O service process I/O request may affect multiple I/O nodes depending on the stripe factor of the PFS file.

Each I/O node receives and processes PFS stripe file requests from an I/O service process. The stripe file requests contain only control information. The actual application data transfers directly from the application buffer to RAID device buffers at the I/O node, thus eliminating expensive data copies.

A single I/O node supports concurrent data transfers to multiple compute nodes while disallowing concurrent transfers to the same compute node. However, different I/O nodes are capable of concurrently transferring data from the same compute node.

When an I/O node transfers application data from a compute node, TOS is responsible for the flow control. The ability of the I/O service process to issue asynchronous read and write requests necessitated the addition of flow control code which limits the number of asynchronous I/O requests issued. Without flow control, I/O node bandwidth and stability degrade due to memory starvation caused by the buffering of I/O requests. Given the small number of I/O nodes and inter-node transfer policies, the I/O node flow control issues reduce to a set of manageable problems that do not stand in the way of achieving maximum I/O bandwidth.

I/O Results

The TFLOPS system was shipped with 18 RAID units on the classified section and 18 RAID units on the non-classified section. Each RAID can store approximately 64 GB of file system data, hence the total storage capacity of the TFLOPS system is approximately 2.25 TB, or about 1.125 TB per section.

There are a number of factors that affect the aggregate I/O transfer rate. Some of the more important factors are as follows:

- the number of compute nodes executing the user's application
- do all compute nodes access the same file, or does each compute node access a different file
- the number of I/O service nodes
- the size of the I/O request
- the PFS stripe unit size
- the PFS stripe factor

The requirement of sustaining an I/O bandwidth of one GB/sec, for both read and write operations, was demonstrated at the factory after most of the compute node hardware was already installed at Sandia. The test system was configured with 18 RAIDs, 12 I/O service processes, and 432 compute nodes. The TFLOPS file system was configured as 18 separate PFS file systems, with each PFS striped across a RAID's two logical disk devices. All the compute nodes were simultaneously performing asynchronous I/O to their own file using a request size of 8 MB.

After all of the TFLOPS hardware was installed at the customer site, the I/O performance tests were replicated using the same hardware configuration describe above. Soon after this demonstration, the customer decided to change the configuration of the system so only nine

RAIDs were used for PFS. Consequently, in the following discussion of I/O performance we do not have data for using all 18 RAIDs for PFS.

Figure 6 shows the read and write performance when nine RAIDs were configured into one PFS file system, with the PFS stripe factor varying between 2 and 18. In this test:

- The number of compute nodes was set to 16 times the stripe factor (i.e., the ratio between the number of compute nodes and the PFS stripe factor was fixed at 16:1).
- The number of I/O service processes was fixed at 8.
- The request size was fixed at 2 MB.
- The PFS stripe factor and stripe unit size were both fixed at 1 MB.
- The process on each compute node accessed its own file.

The figure shows that adding additional RAIDs to PFS results in near linear increase in read and write bandwidth.

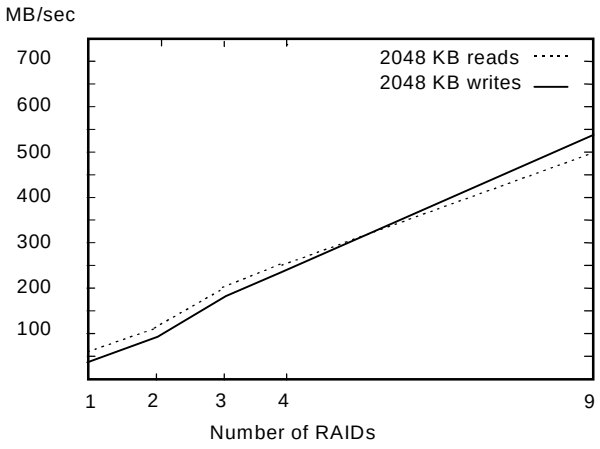


Figure 6: PFS throughput scalability

Figure 7 shows the read and write performance from 3584 compute nodes as a function of the I/O request size. The TFLOPS file system is configured as nine PFS file systems, each striped two ways across a single RAID. In this test:

- The RAIDs were configured as nine separate PFS file systems.
- The number of I/O service processes was set to 32.
- The PFS stripe factor and stripe unit size were set to 1 MB.
- Each compute node process was accessing its own file.

The figure shows that an application running on a large number of compute nodes can achieve an I/O transfer rate of 0.5 GB/sec on nine RAIDs.

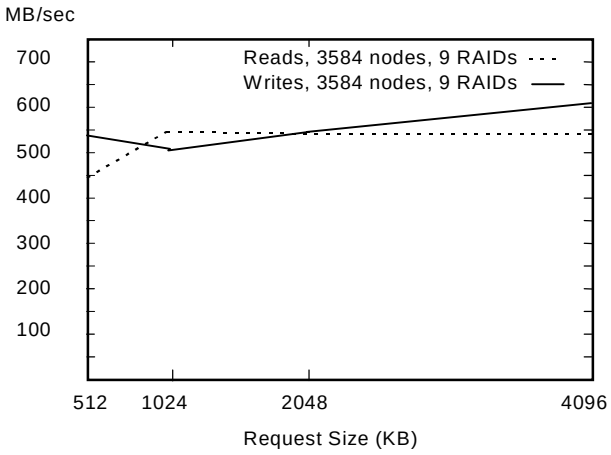


Figure 7: PFS read and write bandwidth

The TFLOPS file system has met both of its primary requirements: it provides storage space for at least one TB and it can sustain an aggregate transfer rate of one GB/sec. The file system also scales with the number of RAIDs attached to the system.

Conclusion

The Intel TFLOPS Supercomputer has accomplished its performance requirements of one TFLOPS sustained floating point performance and one GB/sec sustained I/O to the file system. It also met the system reliability and security requirements. This paper discussed some of the design tradeoffs in terms of memory management, communication, and file I/O. The decisions are inter-related.

This computer is running today at Sandia National Laboratories and doing production work on ASCI applications. No one in the world has yet matched the performance of one TFLOPS, nor of one GB/sec of sustained I/O. A few months after system delivery, scientists at Sandia commented that the system had already done more work than their previous system had done in the last three years. They have run physics simulations with larger problem sizes and finer resolutions than have ever been run before. In the development of the TFLOPS system, we have demonstrated Intel architecture processors are capable of spanning the range from desktops to teraflops.

Appendix A: Single Drive Failure Recovery

With a potential TB of data at stake, a single drive failure could be catastrophic. The Symbios RM20 RAID subsystem's design gracefully handles single drive failure. When a drive fails, the RM20 controller detects it and takes the following steps:

- It marks the drive as failed and sets both audio and visual alarms.
- It activates one of the Global Hot Spares and begins reconstructing the data of the failed drive from the parity data stripped across the remaining data drives. (Note that reconstructing is time-sliced with normal disk requests so the RAID remains in service. The time-slice algorithm is a dynamically tunable parameter.)
- When an operator replaces the failed drive, the RM20 detects the replacement, reconstructs the new drive, and returns the Hot Spare to availability.

A software daemon running on TOS polls the RM20s periodically reporting any failures to both the console and system logs.

Appendix B: RAID Subsystem

The Symbios RM20 has two bays of ten drives each and two controllers. The controllers can be set active/active (each controller controlling one group of drives) or active/passive (one controller controlling all drives and the other controller configured as a spare). The disk drives are Seagate 4GB Barracudas with a 3.5" form-factor. All the drives share a common internal SCSI bus regardless of their LUN assignment to allow for global hot sparing.

We configured the RM20 with dual active controllers, two LUNs of 9 drives each (the equivalent of eight for data and one for parity) and two global hot spare drives. The RAID controllers present each LUN as a single, logical disk device to the host. The RM20's two controllers are each connected to a Symbios 875 PCI SCSI host adapter.

On the host side, there are two PCI buses on an I/O node, each bus supporting a single 875 adapter card. A node configured for I/O has one RM20 attached, representing two logical disk devices.

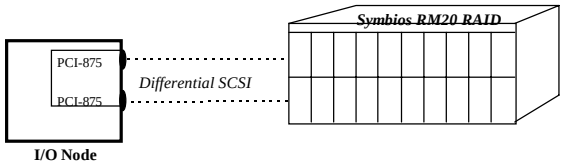


Figure 8: Symbois RAID connection to TFLOPS system

The obvious configuration of each rank of drives assigned to one LUN did not yield the performance we required to reach a gigabyte a second. We worked closely with Symbios for several months to tune and optimize the RM20 to obtain the maximum, raw bandwidth. The final configuration has drives from both ranks assigned to each LUN–something of a sawtooth pattern. This helped balance the contention for the internal bus shared between the two LUNs. The RM20 also has dozens of inter-related tunable parameters that by trial and error we were able to fine tune to reach our performance goals.

Acknowledgment

We would like to acknowledge the entire team that designed and built the Intel TFLOPS Supercomputer.

References

[1] Zajcew, R., Roy, P., Black, D., et.al., "An OSF/1 UNIX for Massively Parallel Multicomputers." *Proceedings of the USENIX conference*, January 1993.

[2] Wheat, S., Riesen, R., Maccabe, A., van Dresser, D. and Stallcup, T., "Puma: An Operating System for Massively Parallel Systems." *Proceedings of the 27th Hawaii International Conference on Systems Sciences*, Vol II, 1994, p.56.

Authors' Biographies

Sharad Garg received a B.Sc. in computer Science from the University of Allahabad in 1982, an M.Sc in Computer Science from the University of Connecticut, USA in 1988, and a Ph.D in Computer Science from the University of Connecticut, USA in 1992. He taught as an Assistant Professor in the Computer Science Department at the University of Delaware from 1992-94. He is currently working as a Server I/O Architect in Server Architecture Lab in ESG. Technical interests include broad category of parallel processing, especially in parallel I/O performance. His e-mail address is sharad@co.intel.com.

Robert Godley graduated with an MA in mathematics from San Diego State University in 1974. He joined Intel in 1988 and has worked for Intel Supercomputer, ESG,

since 1992. He has supported file system code on both the Paragon and TFLOPS operating systems. His e-mail address is rlg@co.intel.com.

Richard Griffiths is a Senior Software Engineer with Intel Supercomputers, Enterprise Servers Group. He is a member of a small team of engineers sustaining the TFLOPS software. Richard is a self-taught engineer with no formal degrees. He is a part-time jazz musician and artist with an interest in MIDI and real audio. Richard's e-mail address is richardg@co.intel.com.

Andrew Pfiffer received a B.Sc in computer science from SUNY Oswego in Oswego, NY in 1985. He first worked with supercomputing at the NSF Cornell Theory Center, and later worked for Topologix, Inc. and Cogent Research. Andrew joined Intel in 1991 to work on the Intel Paragon and is currently working on Merced™ validation for SPD in Santa Clara. His e-mail address is andyp@co.intel.com.

Terry Prickett received a B.Sc. in Computer Science from Oregon State University, Corvallis, Oregon in 1975. After spending 17 year in the supercomputer business, he joined Intel's Supercomputer Systems Division in 1992 and is currently at Intel Supercomputer, ESG. His e-mail address is terry@co.intel.com.

David Robboy has been a software engineer at Intel for over 14 years, working on various flavors of the UNIX operating system. He is now sustaining the TFLOPS operating system. He has a B.A. in mathematics from Reed College. His e-mail address is robboy@co.intel.com.

Stan C. Smith works on Merced architecture validation in the Microprocessor Products Group. He received a BSCS from the University of Oregon in 1976. His technical interests include distributed operating systems, multicomputers, high-speed communications networks, and robotics. His e-mail address is stans@co.intel.com.

T. Mack Stallcup has worked at Intel for seven years. He worked in Factory Automation at Fab 7 and as an on-site Parallel Systems Engineer at Sandia National Laboratories. He has been a Senior Software Engineer for Intel Supercomputer, ESG since 1995. He received a B.Sc. in Chemistry from the New Mexico Institute of Mining and Technology and an M.Sc. in Computer Science from the University of New Mexico. His e-mail address is tmstall@co.intel.com.

Stephan Zeisset received a Master's degree in Computer Science from the Munich University of Technology in 1994. Since then he has been working on the operating system of the TFLOPS system and its predecessors, with a specialization on distributed memory management. Stephan currently works for MPG, porting the Mach kernel to Merced for validation purposes. His e-mail address is: sz@co.intel.com.

Scalable Platform Services on the Intel TFLOPS Supercomputer

Bradley Mitchell, Server Software Technology, Beaverton, OR, Intel Corporation

Index words: DMI, management, scalability

Abstract

This paper describes Scalable Platform Services (SPS)—a collection of software providing the manageability solution for Intel's latest parallel processing supercomputer.

Compared to previous generations of supercomputer management environments, such as that of the Intel Paragon™ Supercomputer, the SPS makes significant strides in feature offerings and overall usability. The SPS consists of distributed, low-level hardware monitoring, and control functions networked to a centralized management station which are in turn exported to administrators through command-line and graphic user interfaces. This software system demonstrates successful application of off-the-shelf standard components, chiefly the Desktop Management Interface (DMI) supported by Intel and the Desktop Management Task Force.

Together with specialized management hardware, the SPS offers a platform management architecture designed for scalability, availability, usability, and high performance.

Introduction

Supercomputers installed at customer sites require good manageability characteristics. In general, high demand exists for machine cycles, and frequent system downtime proves very costly. For the Intel TFLOPS supercomputer in particular, the sheer quantity of densely packaged hardware makes the task of identifying and repairing failed components difficult. (For an overview of the Intel TFLOPS supercomputer, please refer to the paper entitled *An Overview of the Intel TFLOPS Supercomputer* also in this Q1'98 issue of the Intel Technology Journal.)

The manufacturing and system integration of supercomputers also demands fairly comprehensive management support. Assembling the Intel TFLOPS supercomputer involved rigorous hardware configuration and testing activities that could not be completed on schedule without sufficient automation. Facilities for hardware resource sharing became crucial. Even boot and shutdown procedures for the TFLOPS operating systems needed abstraction.

SPS, a distributed software system, addresses both manufacturing and field requirements as outlined above. The SPS architecture focuses on the unique characteristics of the TFLOPS platform yet succeeds in leveraging off-the-shelf software products to meet its delivery time constraints and quality objectives.

One can compare managing the Intel TFLOPS supercomputer with the task of managing a collection of several thousand distributed PC desktops. (This analogy will be revisited later in the paper.) Although both environments contain roughly the same number of "nodes," the Intel TFLOPS supercomputer poses a number of additional challenges from a manageability perspective. These arise from the machine's scale, use of custom hardware components, usage model, and so on.

A significant goal for management environments according to Intel's Wired for Management (WfM) initiative is to reduce the administration cost associated with an installed hardware base. In the PC desktop realm, reducing this cost involves a combination of hardware support, software support, and standards efforts. Although WfM does not address the supercomputer realm explicitly, the principles of WfM remain applicable. By utilizing built-in hardware support, leveraging industry standards such as the Desktop Management Interface, and by providing new examples of management software components, the SPS for TFLOPS serves as a fine example of WfM principles in action.

TFLOPS Platform Management

The SPS focuses on aspects of system management that relate closely to the unique hardware architecture of the

Intel TFLOPS supercomputer. In particular, the TFLOPS machine includes a Monitoring and Recovery Subsystem consisting of hundreds of boards dedicated to hardware instrumentation and low-level control functions. Private Ethernet, as well as a mesh of serial line connections, network these boards together. A considerable fraction of the SPS software exists to provide communication and co-ordination services for utilizing this "platform within the platform."

Scalable Platform Services

The SPS provides the following features to a TFLOPS supercomputer system administrator:

Scripted booting/shutdown. Individually-executing scripts co-ordinate the booting or shutdown of the two distinct operating systems available on the machine.

Fault management. A software agent detects hardware faults as they occur, isolating affected components from the running system when possible, reporting fault information, and initiating automatic recovery operations in some scenarios. This same agent receives notifications of software faults from the running operating systems and takes appropriate corrective actions.

Configuration management. A software agent maintains up-to-date hardware inventory data, supplying this data to clients on demand. This same agent accepts client requests to partition the available hardware resources into independent sub-machines.

Repair services. Individually-executing scripts support board repair, power control, firmware upgrade, and hardware reset operations.

Field diagnostics. Scripts encapsulate diagnostic test scenarios covering many platform hardware components.

Operating system console access. A gateway service provides access to operating system node consoles from remote clients.

Architecture

Based on a centralized Desktop Management Interface

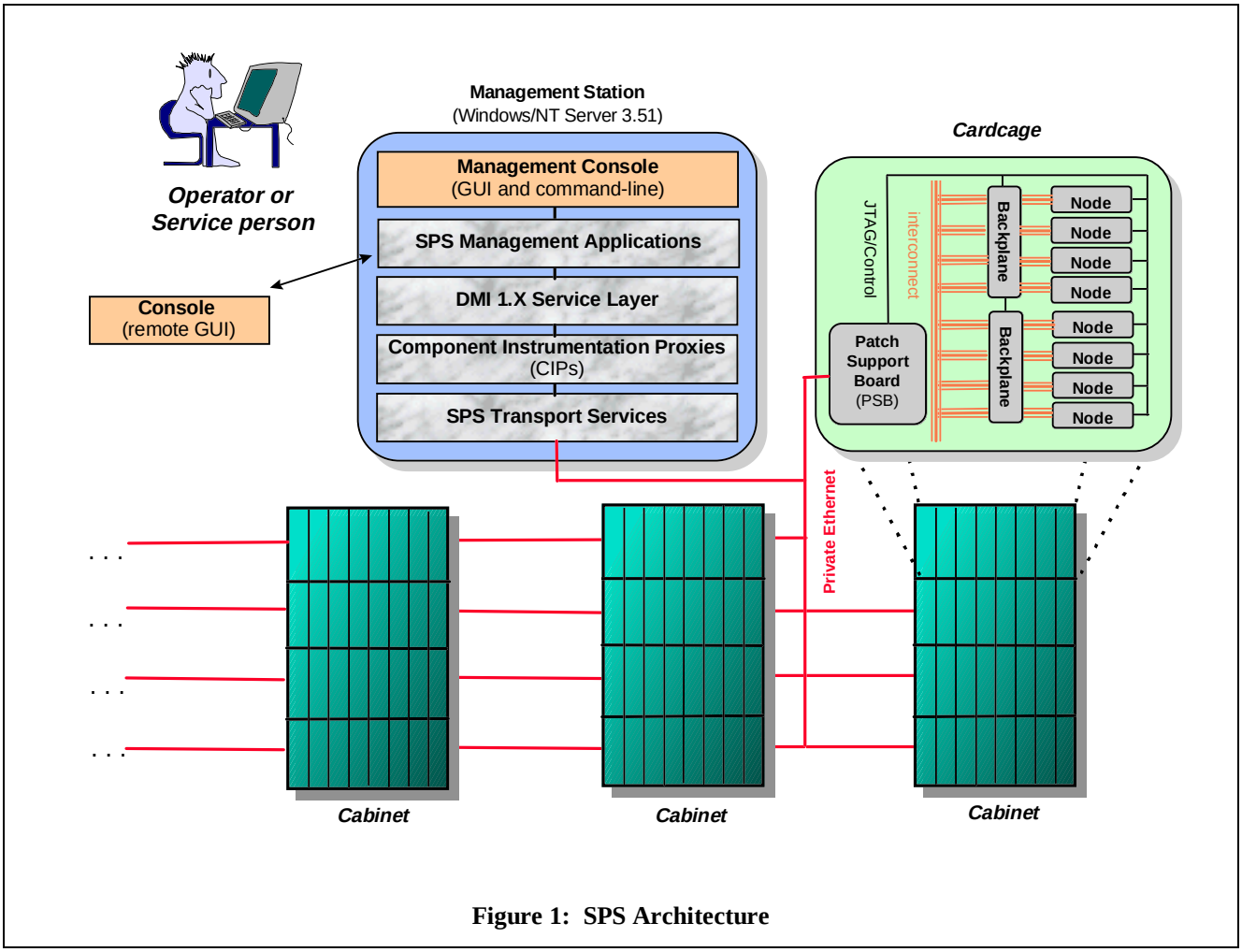


Figure 1: SPS Architecture

(DMI) database, the SPS includes a co-operative set of local management applications (MAs) as well as remote component instrumentation. Five MAs exist corresponding at a high level to the management features supported: *Booting*, *Configuration*, *Diagnostics*, *Fault*, and *Repair*. Unlike many DMI MAs that have a dedicated human interface, the SPS GUI integrates all five MAs into a single coherent presentation. The DMI database and MAs install onto the dedicated "management station" that serves as a central point of administration for the platform

The platform hardware includes Patch Support Boards (PSBs) that oversee each group of eight computational nodes in much the same way the Intel Server Monitor Module oversees a single server system. A fully-populated TFLOPS machine includes roughly 300 PSBs. Each PSB includes an i386™ processor and runs the Wind River Systems VxWorks* real-time operating system. Software agents implemented above the OS, coupled with hardware instrumentation capability built into each PSB, provide a full range of monitoring and control functions for the nodes, backplanes, and other platform components.

Via a transport layer on each end, DMI component instrumentation proxies on the management station communicate with PSB agents to collect instrumentation data for and deliver management requests from a central location. On the station, five such proxies exist broadly corresponding to the types of hardware present in the machine: *Node*, *Backplane*, *PSB*, *Clock*, and *Cabinet*.

An administrator interacts with the SPS through a graphic user interface (GUI) and related command-line interfaces. The GUI implements direct manipulation interfaces for specifying hardware components, individually or in groups, as the target of management operations. It launches management scripts that encapsulate specific booting, diagnostic, or repair operations. Its network map includes an event-driven interface that color-codes individual hardware components based on their real-time state. Finally, it displays in real-time the OS event log entries made by the management applications.

Figure 1 illustrates the SPS software architecture in relation to the hardware platform. Subsequent sections describe the software layers in more detail.

Patch Support Board Software

The PSB software consists of the VxWorks kernel and a set of runtime-loaded SPS modules.

The base VxWorks distribution from Wind River required several source extensions and modifications for operation on a PSB. For example, the SPS team re-wrote its network initialization module—delaying the network initialization process within the OS until platform configuration discovery within SPS modules completes. Additionally, the team added support for reading the on-board EEPROM needed to access, among other things, the PSBs MAC address.

Each SPS object module implements an agent or, alternatively, a library of services shared among agents. Each agent provides management services either for one type of Field Replaceable Unit (FRU)[1] or for one type of communication interface (serial or packet). Libraries implemented include a JTAG[2] scan library providing a standards-based interface to instrumented hardware devices.

In general, the PSB agents support both monitoring and control functions. They receive hardware interrupts and collect instrumentation data on behalf of the management station. As needed, the agents transmit information to the station over the private Ethernet network. The agents also accept requests such as those for power or reset control from the management station and execute the required processing and notification procedures.

Finally, this software implements a communication and keep-alive scheme among the PSBs, taking advantage of the serial line network fabric connecting all PSBs. The management hardware for the Intel TFLOPS supercomputer thus includes two management networks: a secondary network assists in managing the hardware critical to the machine's computational mission, and a tertiary network assists in managing the management hardware itself. Both of these networks exist "out-of-band" from the machine's primary node interconnect.

Node Maintenance Port

One of the two standard serial ports (COM2) on the node board serves a special function as the Node Maintenance Port (NMP). This port connects to the PSB and is used in two ways: the message mode supports reliable datagrams, and the raw mode supports an unstructured, not necessarily reliable, character stream.

The PSB supports multiplexing of data in the two modes and also works as a gateway for the NMP communication

[1] SPS supports the following FRU types: node boards, backplanes, clock boards, power supplies, blower units, and the PSB itself.

[2] JTAG refers to the Joint Test Action Group and IEEE Standard 1149.1 for implementing boundary-scan functionality in hardware devices.

between its eight nodes and the management station. The SPS management applications use the NMP message mode to communicate reliably with the extended node BIOS and with downloaded, off-line node diagnostics. The NMP raw mode supports an administration and debug console to the operating system on the nodes. On the management station, NMP libraries communicate in message or raw mode transparently to any node in the system by hiding the communication to the gateway service on the appropriate PSB.

Management Station Software

All software components resident on the management station operate in a 32-bit Windows NT* environment. Figure 2 illustrates these components, and they are discussed in the following sections.

PSB Transport Service

The PSB Transport runs as a Windows NT* service on the management station. It provides transport services for sending and receiving reliable datagrams between the station and one or more PSBs, multicast group definition, PSB enumeration, and notification when PSBs enter or drop off the network. The inter-process communications mechanism between the transport and its users incorporates local RPC. PSB Transport interface libraries hide the RPC initialization and tear-down details.

Component Instrumentation Proxies

The SPS utilizes DMI as the primary management interface. This choice allows the TFLOPS system to use standards-based management and helps facilitate a single operational view through a single management interface.

The DMI standard defines a *Component Instrumentation* interface that performs the required low-level management operations. However, the actual low-level operation takes place on the PSB. To bridge this gap Component Instrumentation Proxies or CIPs were implemented.

As mentioned earlier, the five proxies provided are *Node*, *Backplane*, *PSB*, *Clock*, and *Cabinet*. Each CIP runs as a Windows NT* service that communicates with a corresponding agent on the PSB to collect instrumentation data and to deliver management

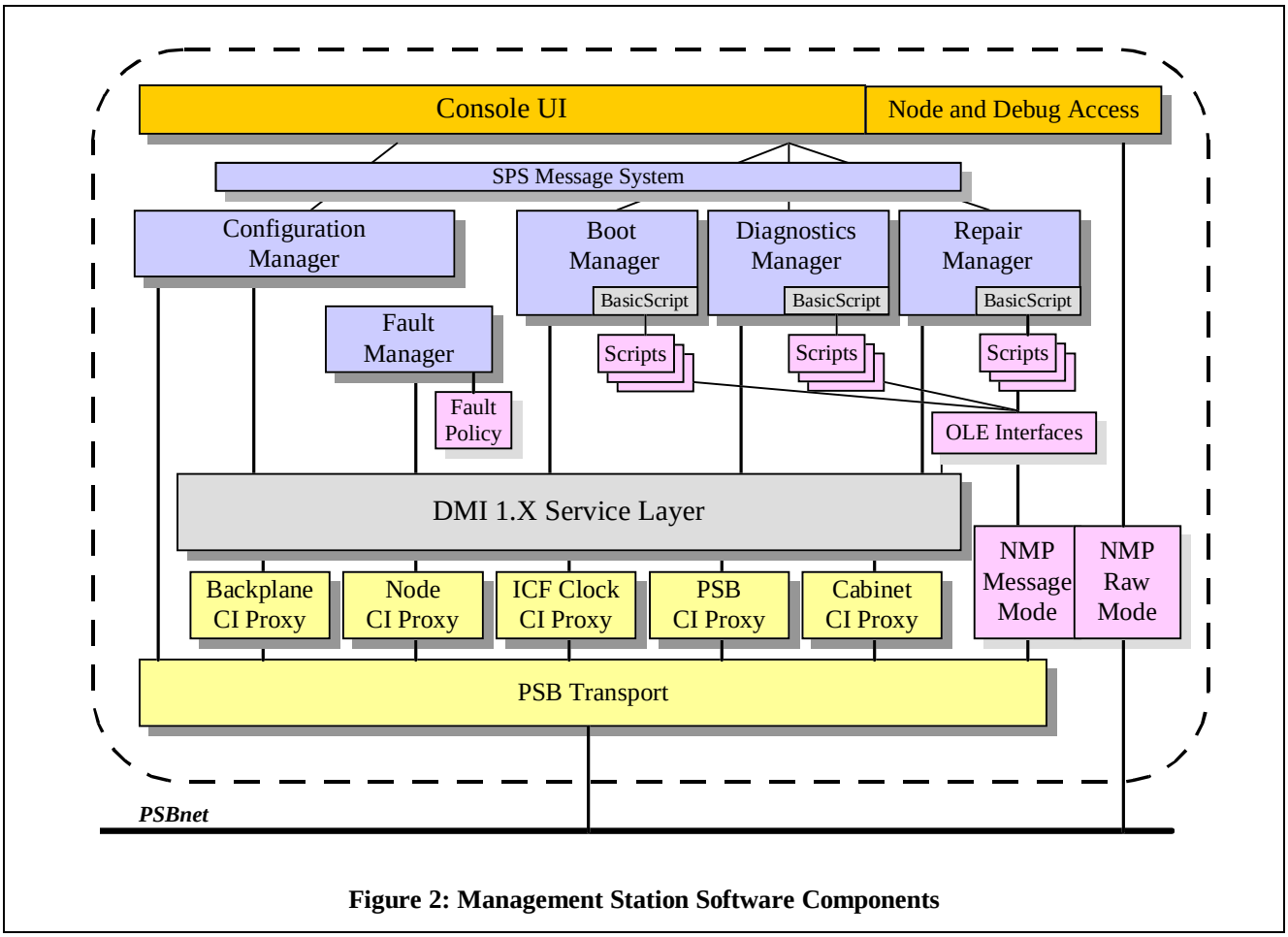


Figure 2: Management Station Software Components

requests. The two primary interfaces for a CIP are with the PSB Transport and the DMI Service Layer.

DMI Service Layer

The SPS utilizes version 1.X of the Desktop Management Interface (DMI). DMI includes a specification written and maintained by the Desktop Management Task Force (DMTF), an industry consortium chartered with the development, support, and maintenance of management standards for PC systems and products. As part of this specification, the DMI service layer provides a standards-based interface between the SPS management agents and component instrumentation responsible for low-level management tasks. It is this interface boundary that SPS takes advantage of to provide the illusion of a single unified system through a single DMI database.

A DMI database generated for the Intel TFLOPS supercomputer contains more than 3,600 component entries. This results in more than 21,500 individual DMI groups and more than 99,000 DMI attributes. Prior to the SPS project, Intel's existing DMI service layer implementation could scale to only 254 component entries. However, Intel made the necessary enhancements to accommodate this SPS scaling requirement.

In addition to scaling the number of components supported, Intel's DMI implementation required an enormous reduction in the time required to generate a large component database. The Intel team reduced this time from two calendar days initially to less than fifteen minutes.

Management Applications

The five SPS management applications are *Boot*, *Configuration*, *Diagnostic*, *Fault* and *Repair*. Each MA runs as a Windows NT* service on the management station.

The Boot, Diagnostic, and Repair Managers advertise a list of available control operations. Each entry in this list corresponds to an executable script. The team implemented these scripts with the BasicScript* environment from Summit* Software, so chosen because the LANDesk* Server Manager product also uses it.

When invoked, each script executes in a separate thread on the management station under control of the BasicScript run-time environment. Multiple scripts may execute concurrently. To allow scripts to access the DMI service layer, the SPS team developed a simple OLE[3] interface library to DMI. (The team prepared a similar OLE library for interface to the NMP.) Most scripts interact with an operator as described in the next section.

The Configuration Manager exists primarily as a server-side agent to the client user interfaces. It uses the SPS message system to converse with instances of the SPS GUI and command-line utilities. It initializes the PSB private network.

The Fault Manager monitors events reported by the PSBs and processes all fault events. It includes a custom inference engine that synthesizes and correlates fault indications from across the platform and initiates automatic corrective action where possible. The Fault Manager includes a `lex`/`yacc` fault grammar.

SPS Message System

The SPS team implemented a simple network transport supporting reliable datagram communications between SPS Managers and (potentially) remote clients. Its main purpose is to support the SPS GUI running on a machine other than the management station.

The Message System uses secure RPC interfaces to support authentication. Additionally, a built-in authorization scheme restricts operator access to management functions according to their membership in Windows NT* security groups.

The Message System provides an API matching that of the Intel LANDesk* Server Manager Message System (server-side) and the Network Transport Server (client-side). Because SPS originally used the LDSM transport, matching the LDSM API allowed the SPS team to replace those components in isolation.

Graphic User Interface and Scripting Environment

Figure 3 illustrates the SPS graphic user interface. This interface runs either on the management station or on a remote Windows NT* console. The GUI utilizes the SPS message system to interface to the SPS managers. The GUI implementation relies heavily on the Microsoft Foundation Class (MFC) library for Win32.

The SPS GUI employs a network map metaphor. The topmost portion of the main window contains a thumbnail sketch representing all cabinets in the machine, rendered in rows and columns just as the actual hardware is installed on the floor. Beneath this sketch, the middle portion of the main window displays a zoom-able view of one or more cabinets within the machine.

As shown in Figure 3, objects on the screen include color codes to match the current state of the corresponding

[3] OLE refers to Microsoft's Object Linking and Embedding technology.

piece of hardware. The color red corresponds to a faulted FRU, green to a healthy FRU, and so on.

An operator launches SPS scripts from the tree included in a dock-able window at the bottom of the GUI main window. The GUI also presents a list of existing partitions in this window. To create partitions, the operator rubber-bands selected cabinets. Likewise, the rubber-banding mechanism allows an operator to zoom the main view in or out.

Figure 3 illustrates a typical configuration for the TFLOPS machine. While the full system includes four rows of 19 cabinets each, operators typically de-couple the rightmost 4x4 cabinets for customer site security purposes. When de-coupled, the SPS GUI will not render these cabinets, although it will account for them in its numbering scheme. When the system transitions to a non-secure operating mode, operators typically re-cable the rightmost 4x4 block and correspondingly de-couple the leftmost 4x4. SPS allows the operators to construct hardware/software "partitions" of the hardware for this and other reasons such as unobtrusive diagnostic testing or hardware resource sharing.

Challenges

The most pervasive challenge faced by the SPS team lies in integrating the diverse collection of off-the-shelf software components.

Another significant challenge, one faced in many high-end platform software development projects, is a lack of available target hardware. The partitioning feature in the SPS mitigated this problem to some degree in the development labs. Yet, the limitation applies particularly now in the sustaining phase of the project when nearly all existing hardware has been shipped to the customer.

A variety of scaling challenges was also encountered. Utilization of the PSB network required careful planning to avoid performance bottlenecks. The DMI implementations required modification to support the large number of installed components. And the design of the SPS GUI required particular emphasis on ease of navigation and clear display of fault notifications.

Another serious development challenge grew out of the need for SPS to support manufacturing test needs. Lacking any other tool support, the manufacturing team required access to incomplete versions of SPS for booting, diagnostic, power control, and other base features. Combined with each of the other challenges mentioned above, this forced the development team to regularly make trade-offs between feature availability, stability, and progress toward the final product. Nor did the feature set in the field always match exactly the functionality needed internally. In this environment of rapid change, it proved very difficult to re-train users of the SPS as features were enhanced over time and the need for special "work-around" procedures was removed.

In a system as large as the Intel TFLOPS supercomputer, the secondary management problem–managing the support hardware and software–can become as difficult as some people's primary management problem. SPS attempted to deal with both problems. However, details

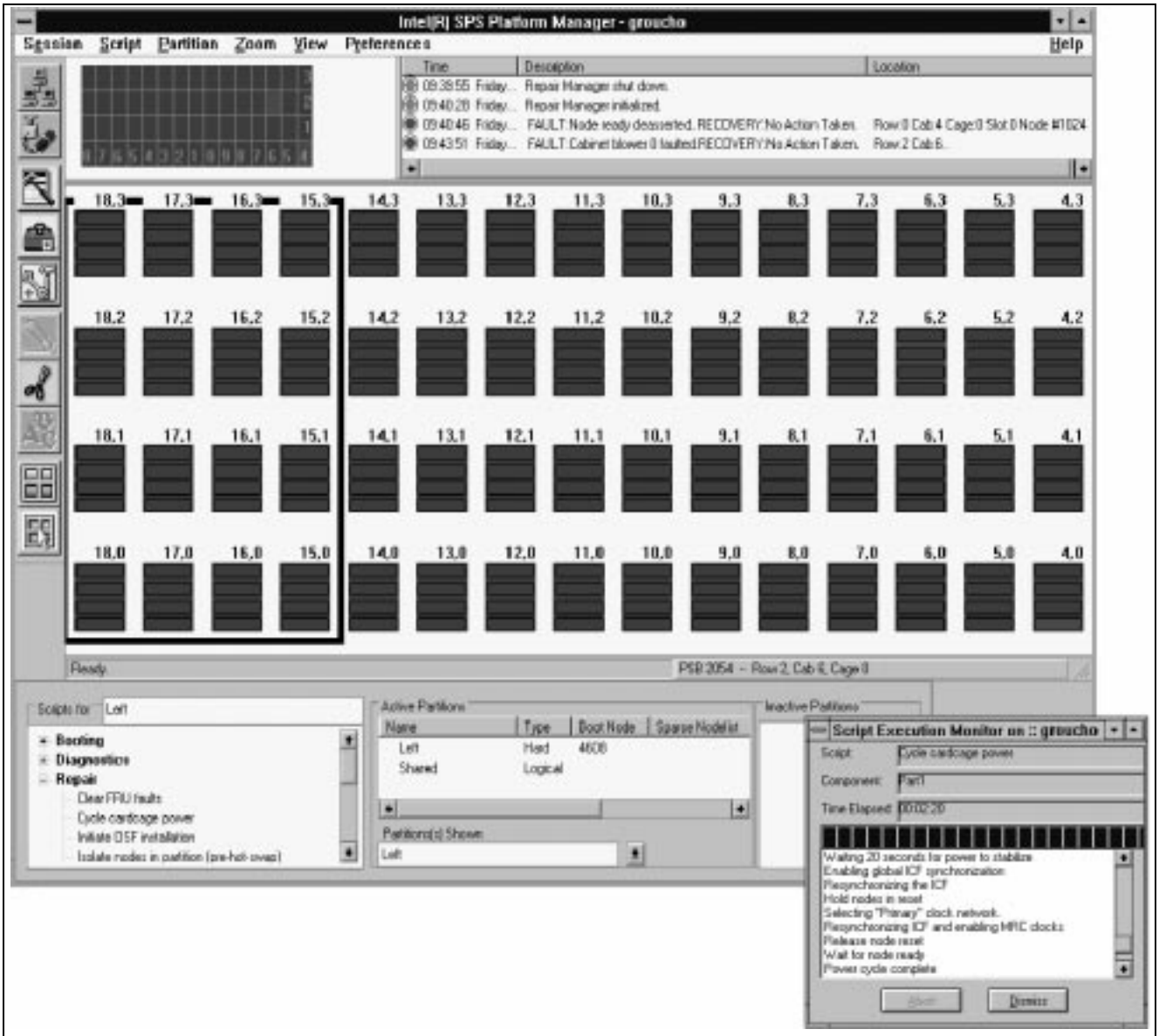


Figure 3: SPS Graphic User Interface

such as software/firmware installation and upgrade of PSBs, or static IP address assignment for the PSB network, proved non-trivial to debug and overcome in practice.

Finally, the SPS overcame a few cultural barriers to acceptance. Traditional supercomputer management environments consist of UNIX* workstations exclusively, and for SPS to be appealing to certain classes of users, UNIX-like command interfaces to particular SPS features were required. By using the TelnetD `telnet(1)` server for Windows NT* from Pragma Systems, Inc., and by tasking Pragma to tailor the product for better interoperability with the TFLOPS OS debugger, these concerns were largely ameliorated.

Results

The SPS played an important role in enabling Intel to win the one teraflop performance race, to meet its contractual commitments, and to raise the bar for supercomputer management software.

In general, the team's experiences in integrating off-the-shelf software components were positive and met or exceeded expectations. In particular, DMI performed nicely and proved to be the single biggest "win" for the project. In most cases, including the DMI implementation, the VxWorks kernel, BasicScript, and the RATP specification, enhancements were required or software defects were encountered that required source modifications. Yet, in every case, these alterations did not derail the project, in part because the changes were small and in part because they were anticipated as part of the development effort.

In only one case, that of the LANDesk Server Manager, did the team replace components of the product in mid-stream. This was due not to any particular technical shortcoming of LDSM, but rather to the fact that the feature offset between it and SPS was too great. Specifically, where LDSM targets heterogeneous workgroups that include servers, SPS targets one instance of one unique type of Intel-architecture based server.

As in many software development projects, the team found room for improvement. In general, more tool support could have been provided for managing SPS itself, including installation, upgrade, and fault recovery features. Features may have been phased into the product in an order better suited to the manufacturing bring-up effort. The customer environment, and the in-house manufacturing environment for that matter, was not so well understood by the SPS development team, causing a few client features to be implemented that proved unnecessary in practice. Finally, the SPS architecture could not accommodate management of the TFLOPS I/O subsystem nor could it completely account for management of the hardware cabling.

Discussion

While there are similarities between managing the TFLOPS system and managing a network of distributed PC desktops, they are certainly not the same. One aspect of the TFLOPS supercomputer that simplifies the management task comparatively is its fixed size and network topology.

Another simplifying factor is its homogeneity: most nodes share identical hardware, firmware, and system software. Unlike distributed PCs, these nodes do not have individual owners who may install arbitrary software packages or otherwise customize the local environment. Only one or a few different parallel applications run across all the machine's nodes at a given time.

Whereas PC network administrators may come from a wide variety of backgrounds and receive varying levels of training, TFLOPS administrators are literally hand-picked and acquire an in-depth knowledge of the machine. This even further simplifies the management burden.

On the other hand, some aspects of system administration become more difficult in the TFLOPS domain. The tightly-integrated, custom hardware design of the Intel TFLOPS supercomputer naturally tends toward more frequent system outages. The system software deployed is not so mainstream as that typically used on large-scale PC networks, leading to even less system stability. TFLOPS nodes lack their own monitors, keyboards, or mice, limiting visibility into the machine's operation. Furthermore, self-administering the management hardware and management software required effort on a scale not typically found in commercial networks.

Nonetheless, many aspects of administration remain clearly similar in both domains. Hardware faults must be detected and reported to any remote location the administrator requires. Firmware upgrade, emergency power-down, and remote reset capabilities are likewise required. The ability to trouble-shoot any single node is extremely useful, and the ability to isolate a node from the overall network is also valuable. Many of the features exported by the SPS live in this "common ground" between supercomputer and PC-based server management.

Conclusion

Compared to previous-generation management environments for Intel supercomputers, such as those for the Intel Paragon™ Supercomputer, SPS makes significant strides in feature offerings and overall usability.

The SPS usage model, based on an intimate knowledge of the server platform's internal hardware constructing and networking topology, has appeal in the scientific supercomputer market where the user base tends toward an "expert" knowledge of underlying implementation details of the system. More mainstream management products, typically built for heterogeneous, distributed commercial environments, often cannot know these details or choose to hide them from the users. Applications such as LANDesk Server Manager employ browsing metaphors in cases where the SPS environment requires more direct interfaces.

In most cases, leveraging vendor products such as the DMI Service Layer afforded clear gains for the project schedule, stability, and/or available features However, some off-the-shelf technologies proved a less-than-ideal fit for the SPS. The process of identifying potential sources of leverage and making build-or-buy decisions can have significant downstream effects on the project schedule and product quality.

It is reasonable to expect that some of the lessons learned from implementing management support for an ultra high-end platform can have cross-over benefits to the more mainstream Intel server product offerings. On one level, the SPS product should prove interesting to those developing DMI-based management software products. On another level, the SPS experience could prove useful to those interested in software systems that leverage off-the-shelf third-party components. The code base of SPS, being too closely tied to the Intel TFLOPS supercomputer's unique hardware characteristics, likely has little direct application in more mainstream environments. However, perhaps some of the SPS design principles, such as scalable communication, user interaction, and configuration management services, can be successfully applied.

Acknowledgments

The author thanks the following members of the development team for their support throughout the project: Ray Anderson, Johannes Bauer, Roy Larsen, Mouli Narayanan, Don Neuhengen, and Rajesh Sankaran.

A special mention goes to Linda Ernst who formulated the original SPS concept and succeeded in convincing people of its feasibility.

References

[1] DMI Specification., http://www.dmtf.org.

[2] VxWorks and Wind River Product Information, http://www.wrs.com/html/vxwks52.html. http://www.wrs.com/html/productindex.html.

[3] Joint Test Action Group IEEE 1149.1 Standard, http://ada.computer.org/tab/tttc/standard/s1149-1/home.html.

[4] Reliable Asynchronous Transfer Protocol RFC 916, http://globecom.net//ietf/rfc/rfc916.shtml.

[5] BasicScript and Summit Software Product Information, http://www.summsoft.com/html/products.htm.

[6] LANDesk Server Manager Product Information, http://www.intel.com/network/server/index.htm.

[7] Pragma Systems and TelnetD Product Information, http://www.pragmasys.com/TelnetD/

Author's Biography

Bradley Mitchell received an S.B. in Mathematics with Computer Science from M.I.T. and an M.C.S. in Computer Science from the University of Illinois, Urbana-Champaign. Before joining Intel in 1993, he was employed by General Dynamics Corporation. Bradley currently works in Oregon as a Senior Software Engineer in Server Software Technology. His e-mail address is bradley_mitchell@ccm.co.intel.com.

Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems

Milind Girkar, Microcomputer Research Labs, Santa Clara, Intel Corporation
Mohammad R. Haghighat, Microcomputer Research Labs, Santa Clara, Intel Corporation
Paul Grey, Microcomputer Research Labs, Santa Clara, Intel Corporation
Hideki Saito, CSRD, University of Illinois
Nicholas J. Stavrakos, CSRD, University of Illinois
Constantine D. Polychronopoulos, CSRD, University of Illinois

Index words: parallelism, Windows NT*, fibers, compilers, multithreading.

Abstract

Powerful desktop multiprocessor systems based on the Intel Architecture (iA) offer a formidable alternative to traditional scientific/engineering workstations for commercial application developers at an attractive cost-performance ratio. However, the lack of adequate compiler and runtime library support for multithreading and parallel processing on Windows NT* makes it difficult or impossible to fully exploit the performance advantage of these multiprocessor systems. In this paper we describe the design, development, and initial performance results of the Illinois-Intel Multithreading Library (IML), which aims at providing an efficient and powerful (in terms of types of parallelism it supports) API for multithreaded application developers. IML implements a parallel execution environment, which creates, enqueues, dequeues, binds, and schedules user-level threads on Windows NT* threads and fibers. One of the unique and novel features of IML is its support for both loop-level (data) parallelism and task-level (functional) parallelism, as well as nested parallel threads. Although loop-level parallelism is most useful in scientific and engineering applications, functional parallelism is often the norm in multimedia, Internet, and Java* applications. IML upgrades the multithreading support available on the iA-based Windows NT* platforms to levels comparable or superior to those found on high-end parallel systems and supercomputers. Multithreading a number of diverse benchmarks (ranging from POV-Ray to SPECfp95 and the BLAS routines) using IML resulted in significant speedups on a 4-way SMP Pentium® Pro processor based system. Future releases of IML will support several loop scheduling schemes as well as controlled thread migration for the purpose of dynamic load balancing. The programmer or the compiler would thus be able to customize scheduling on a per loop basis taking into consideration performance-sensitive characteristics such as branches inside loops and data locality. The Intel C/C++ and FORTRAN compilers and the Parafrase-2 experimental parallelizing compiler are being enhanced in order to automatically generate the IML API, thereby facilitating the development of multithreaded application codes that fully exploit the performance potential of iA-based multiprocessor servers and desktops.

Introduction

Parallel processing is rapidly becoming mainstream technology influencing architecture and software design from the home PC market (in the form of instruction-level parallelism (ILP), Intel MMX™ technology, and multiple processors on PC boards) to the business field where Symmetric Multi-Processing (SMP) servers have become increasingly popular. While Intel compilers provide intrinsics to generate Intel MMX instructions so that independent software vendors (ISVs) can easily incorporate this technology into their products, there has been little support for programmers to make use of iA-based SMP systems for parallel processing. In fact, multiprocessing may be the most significant enabling factor for moving large-scale engineering and business applications to iA-platforms for the first time, thereby opening new opportunities in the discriminating high-end market.

The Illinois-Intel Multithreading Library (IML) upgrades the multiprocessing support on iA-based Windows NT* to levels comparable to or higher than the multiprocessing libraries provided for high-end multiprocessor servers and scalable parallel processor systems.

IML, unlike other previous or current runtime systems, supports functional parallelism [2][3][4] where task execution conditions are expressed by a directed acyclic graph (DAG), in addition to the more conventional *loop* parallelism and the single-level cobegin/coend functional parallelism. IML is also capable of exploiting *arbitrarily nested parallelism*, which has not been available in any of the commercial multiprocessing libraries, including high-end multiprocessors from Sun Microsystems and SGI, as well as supercomputer systems from Cray, Fujitsu, NEC, and IBM. SPMD-style nested parallelism is an optional feature of the OpenMP standard [6].

IML has been used at the Center for Supercomputing Research and Development (CSRD) and Intel for in-house software development. Application programs written in FORTRAN, C, and C++ for numerical computing, database, and 3D graphics have been successfully ported to the IML library. The Parafrase-2 automatic parallelizing compiler [7] developed at the University of Illinois has been modified in order to generate calls to IML automatically, thus exploiting loop and functional parallelism exposed by the compiler. The IML binaries and the documentation are now available to the public through the IML home page on the web [5]. The Intel compilers are being modified to automatically parallelize programs and generate calls to the IML library.

The rest of the paper describes the design and the implementation of IML and the results from our initial performance study. Our measurements indicate that the performance of IML matches or exceeds highly-tuned commercial libraries for existing multiprocessors for many common single-level DOALL loops while adding support for more general parallelism. Conventional libraries provide multithreading support for simple, singly-nested parallel loops, which allows these libraries to be simpler in design and to incur lower overhead costs. IML implements a queue-based multithreading environment, which supports general loop and functional parallelism and allows arbitrary nesting of parallel loops and unstructured parallel constructs such as nested cobegin/coend.

Design

Basic Design Alternatives: Single Parallel-Task Descriptor vs. Pool of Parallel Tasks

Existing commercial and experimental multiprocessing libraries allow only one parallel loop to be executed at a time. If a second parallel loop is encountered during the execution of a parallel loop (as is the case with nested parallel loops), the second loop is treated as sequential. The same can be said for nested cobegin/coend constructs, also referred to as functional parallelism. Such an execution environment can be supported by a single task descriptor specifying the loop body and the number of iterations. However, supporting the execution of multiple parallel loops (which may be nested or disjointed in arbitrary control flow patterns) or functional parallelism (where precedence requirements are specified by a DAG) necessitates a pool of ready parallel tasks from which assignments to user threads are made (Figure 1). In IML, a collection of task queues[1] is used to implement such a pool of parallel tasks. User-level threads such as one or more iterations of a parallel loop or a function call are then bound to ready-to-execute tasks and are scheduled for execution. In Figure 1, threads execute the task scheduling loop *fetch-execute-enqueue* until the program terminates.

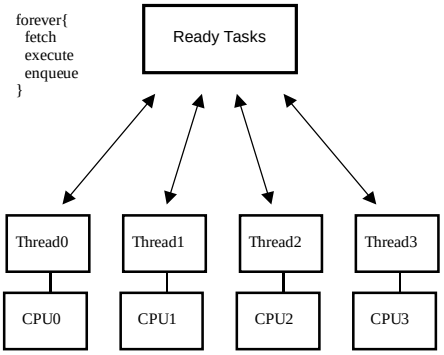


Figure 1: Pool of parallel tasks

Queue-Based System Design Alternatives: Centralized Queue, Distributed Queue, or Global-Local Queue

A pool of parallel tasks can be implemented by a single shared queue. Two major drawbacks of this approach are contention and locality. In order to exploit maximum parallelism, thread parallelism should be exploited at the finest possible granularity that amortizes the overhead of task management and scheduling. However, small task sizes lead to more scheduling events, consequently increasing the contention on a shared queue. Moreover, in cache-coherent systems, this is likely to lead to poor cache hit rates as a result of multiple processors updating a single queue structure. Alternatively, multiple queues can

[1] Throughout the paper, the term queue is used in a broad sense, a list that is subject to insertions and deletions.

be distributed among threads, for example, one queue for each thread. In this configuration, if a thread cannot find more work on its local queue, it can access remote queues, which facilitates load balancing. When workload is well distributed among tasks, contention is minimized which, in turn, promotes cache locality of the task queues. An extreme drawback to this approach can arise when an idle thread accesses a large number of remote queues before finding a task to execute[2]. The shortcoming of the two approaches discussed above can be eliminated by introducing a global queue, in addition to local queues. IML employs a distributed queue configuration because the primary target architecture consists of a small number of processors. Future releases may provide generalized support for large-scale parallel or distributed computing systems.

Implementation Basic API Functions

Application programmers can write parallel programs with the following functions. Support for the OpenMP standard [6] is currently being implemented.

- **`iml_DOALL()`** enqueues a parallel loop task. This function returns when the loop task is completed. The parameters of this function specify the pointer to the function representing the loop body, the number of iterations, the policy for loop scheduling,[3] the minimum chunk size, the number of parameters to the loop body, and the actual parameters to the loop body.
- **`iml_COBEGIN()`** enqueues a set of functionally parallel tasks. This function returns when all the tasks are completed. The parameters of this function specify the number of tasks, the pointers to the functions representing the tasks, the number of parameters to the tasks, and the actual parameters to the tasks.
- **`iml_EnQ()`** enqueues a task that is not a parallel loop. This function returns immediately after enqueueing the task, and thus does not wait for the completion of the task. The parameters of this function specify the pointer to the function representing a task, the number of parameters to the task, and the actual parameters to the task. This interface allows programmers to implement arbitrary functional parallelism.
- **`iml_DecAndFetch()`** performs user-level synchronization. This function atomically decrements the counter and returns the value after the decrement. Combined with **`iml_DOALL()`**, this can be used to implement a DOACROSS (partially parallel) loop. Combined with **`iml_EnQ()`**, this can be used during the scheduling of DAG-parallel tasks. This function takes the pointer to a counter as its argument.

Extended API Functions

Extended API functions are provided for experienced application programmers. The extended API can also be used by a compiler for automatic generation of calls to IML (such is the case with Parafrase-2 and Intel parallelizing compilers). Simple examples of the basic and the extended API functions are given in the Appendix.

- **`iml_ReInitMultiThread()`** changes the number of active threads used by IML.
- **`iml_GetThreadID()`** returns the thread ID of the current thread.

The full performance potential of the above API can only be exploited with appropriate support from the OS kernel. In particular, an application can add/release threads as it goes through different phases of its execution in a way that accurately reflects the parallelism in the underlying computation. This results in better utilization of processors and memory and translates not only to lower execution times, but also to improved average workload turnaround time in a multi-user environment. OS support at the level of allocating and reclaiming resources from user processes would be necessary in order to exploit this capability of IML.[4] However, this is not the case with Windows NT* at present.

Windows NT*: Threads and Fibers

A thread is a unit of computation scheduled by the operating system to run on a processor. A fiber is a unit of computation that runs on a thread and is scheduled by a user[10]. Some of the important characteristics of fibers are as follows:

- Fiber switching is measured to take 50-60 cycles on a 200 MHz Intel Pentium® Pro processor. On the other hand, the cost of suspending a thread is orders of magnitude greater.
- There is an order of magnitude difference in the cost of creation and deletion between fibers and threads.
- Similar to threads, fibers provide a user-level context that includes a program counter, registers, and a stack.

[2] Effective scheduling algorithms and thread migration schemes minimize the occurrence of such extremes.

[3] IML implements various scheduling algorithms from which the programmer can select on a loop-by-loop basis.

[4] Hybrid implementations are possible but cumbersome and may conflict with software compatibility.

- Scheduling of fibers is controlled by the user, while scheduling of threads is controlled by the operating system.
- The relationship of fibers to threads is analogous to that of threads to processors. An active fiber is bound to a thread, just as an active thread is bound to a processor. A thread can have at most one active fiber; similarly, a processor can have only one active thread. *Inactive* (or unbound) threads and fibers do not receive any computational resources.

Figure 2 illustrates the relationship between threads and fibers. Without fibers, a context switch, even within a process, is performed by the operating system. In Figure 2(a), bold lines represent the bindings between threads and processors. A context switch corresponds to a reconnection of a bold line from one thread to another thread. This operation has two drawbacks. It is expensive and not controllable by the user. These drawbacks can be overcome with the introduction of fibers. Fibers detach execution contexts from threads, allowing their scheduling to be explicitly controlled by the user. Multiple threads are still needed to maintain multiple active fibers. In Figure 2(b), regular lines represent the bindings between fibers and threads. A user-level context switch corresponds to a reconnection of a regular line from one fiber to another fiber, while a kernel-level context switch is still represented by a reconnection of a bold line. Since fibers are lightweight, easy to manage, and can be explicitly scheduled by the user, they are used in the

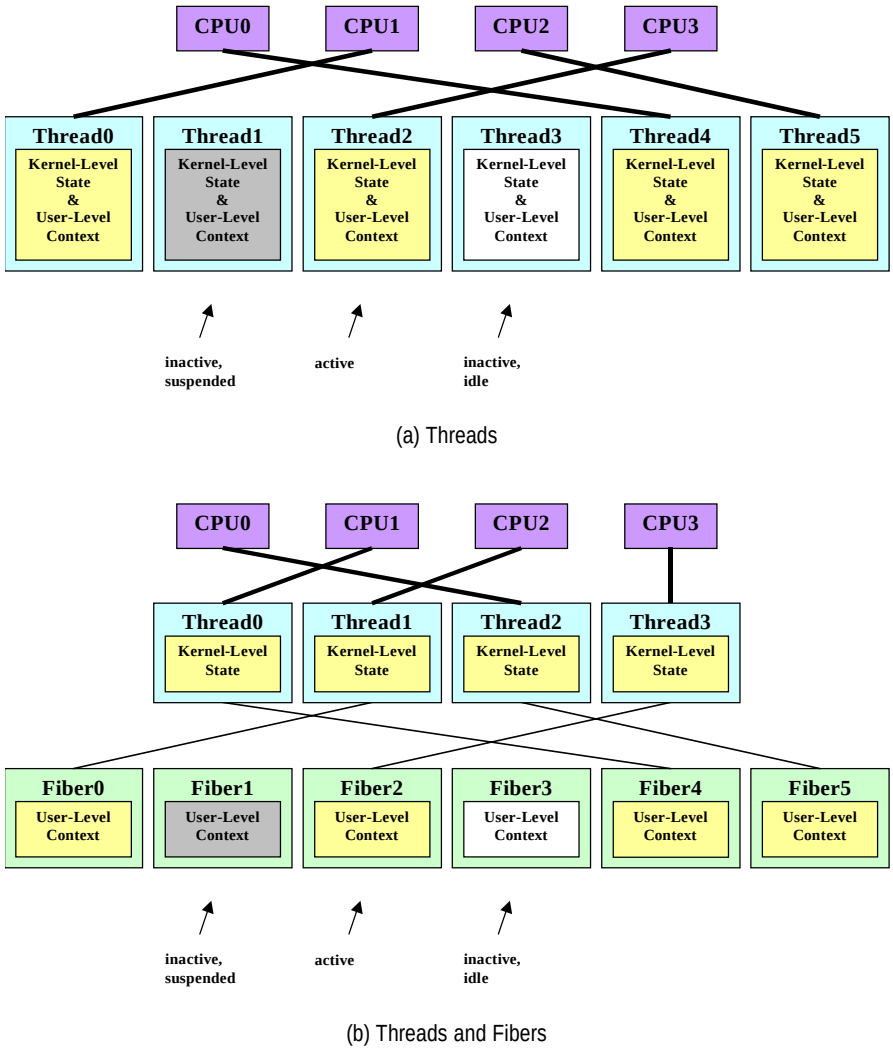


Figure 2: Relationship between threads and fibers

implementation of IML.

Threads or Tasks

In IML, tasks are represented by task descriptor blocks (TDB). A TDB contains a function pointer, a list of arguments to that function, and for parallel loops, a starting index, a dispatch counter, a minimum chunk size, a loop scheduling policy, and a pointer to a loop descriptor block (LDB). A LDB contains a completion counter and a pointer to the parent context of the loop. Each task is represented by a single TDB except for a parallel loop, which can be divided into one or more TDBs that share a single LDB.

Figure 3(a) is an example of a parallel loop, whose body is represented by the function `add_vectors_body`(). Figure 3(b) illustrates the relationships between the TDBs and the LDB for this parallel loop. The number of iterations `n` is assumed to have the value 1000. In this example, the parallel loop is divided into four TDBs of 250 iterations each (the initial value of the dispatch counter). The completion counter in the LDB is initialized to 1000, the number of iterations in the parallel loop. The parent context in the LDB is set to the address of the fiber executing the function call `iml_DOALL()`. The TDBs are enqueued into the task queues, where they wait to be scheduled for execution. Finally, the fiber executing `iml_DOALL()` yields to another fiber to participate in task execution.

Distributed Shared Queue (DSQ) and Load Balancing

In order to avoid contention on a centralized queue, IML uses multiple task queues distributed across multiple threads. Each thread owns a (local) queue, and can also access (remote) queues owned by other threads in order to achieve balanced load distribution. For systems with many processors, a hierarchical DSQ implementation may be preferable to a flat implementation. However, since the current target of IML is a four-way SMP Pentium Pro processor based system, IML employs a flat DSQ implementation. The current scheduler accesses remote queues in a round-robin fashion after the local queue becomes empty. This scheduling policy enhances cache locality of local queue accesses when threads continue to schedule tasks from their local queues.

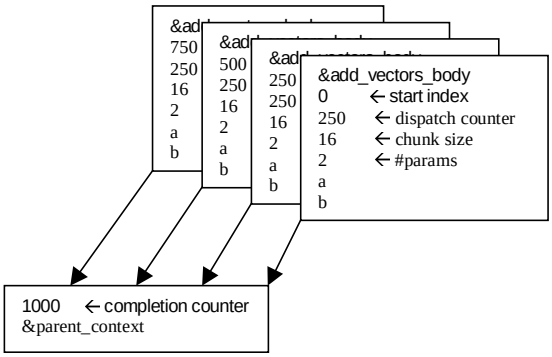
Dynamic load balancing is achieved when threads with empty local queues acquire tasks from remote queues. If a remote task is a non-loop task, the thread dequeues the task and executes it. If the task is a parallel loop, the thread splits the task in half [8], places one of the tasks in its local queue, and begins executing it. (This policy is chosen to maintain locality while reducing the cost of load balancing. User-specified minimum chunk size is honored in any loop-scheduling events.) Figure 4 illustrates the configuration of the DSQ. Threads and their local queues are connected by the bold lines. The regular lines represent the connections between threads and remote queues. IML allows users and external libraries to create multiple threads, and for each of these threads (multiple instances of Thread 0 in IML) to take advantage of IML. This enables rapid porting of existing threaded applications to IML.

Each task queue in IML is implemented as a stack to facilitate support for nested parallelism. When a thread encounters the first (outermost) level of parallelism, the newly created tasks are pushed onto the appropriate

```
void add_vectors(double *a, double *b, int n){
  int chunk = 16;
  int params = 2;
  iml_DOALL(&add_vectors_body, &n,
            &static_schedule, &chunk,
            &params, a, b);
  return;
}

void add_vectors_body(int *start, int *iters,
                      double *a, double *b) {
  int i;
  for(i = *start; i < (*start + *iters); i++){
    a[i] = a[i] + b[i];
  }
}
```

(a) A Parallel Loop



(b) TDB and LDB for (a)

Figure 3: A parallel loop, task descriptor blocks, and a loop descriptor block

stacks. Inner parallel tasks are pushed onto the local queue, hence increasing locality of task queue operations, as well as locality between inner parallel tasks.

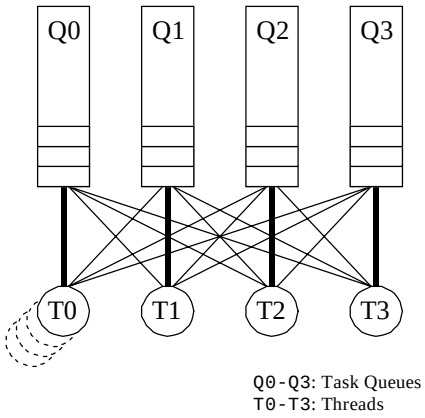


Figure 4: DSQ and threads

For example, in the case of doubly-nested parallel loops, the outer loop is distributed across multiple threads, while each inner loop is enqueued to the local task queue. Each thread continues executing iterations of the inner and outer loops from its local queue, until all the tasks in the local queue are completed. At this point, threads acquire tasks from remote queues making it possible for them to participate in the execution of inner parallel loops from other threads. By enqueueing all the inner loop iterations to the local queue, locality among these iterations is exploited.

Lock-Free Stack

The task queue stack is implemented without software locks [9] by using the iA instruction `lock CMPXCHG8B`, which performs an atomic compare and exchange operation. Figure 5(a) and (b) illustrate enqueue and dequeue operations of the pointer P2, respectively. In the enqueue operation (Figure 5(a)), the `lock CMPXCHG8B` instruction compares the pair "Empty-Top" (brown) against the stack top (green), and if the comparison succeeds, the stack top is replaced by the pair "Top-P2" (blue). When the comparison fails, the operation must be repeated with the new stack top. In the dequeue operation (Figure 5(b)), the top of the stack, which is the value to be dequeued, is used to construct the pair "Top-P2." The `lock CMPXCHG8B` instruction compares the pair "Top-P2" and the stack top, and replaces the stack top with "Empty-Top" if the comparison succeeds.

Unfortunately, the lock-free implementation allows only one access point to a task queue. Therefore, a thread obtains a remote task from the top of a remote task queue, even though outermost parallel tasks are found at the bottom of the task queue.

Process Stack Management

Exploitation of parallelism requires multiple execution contexts to be active simultaneously. In IML, each of

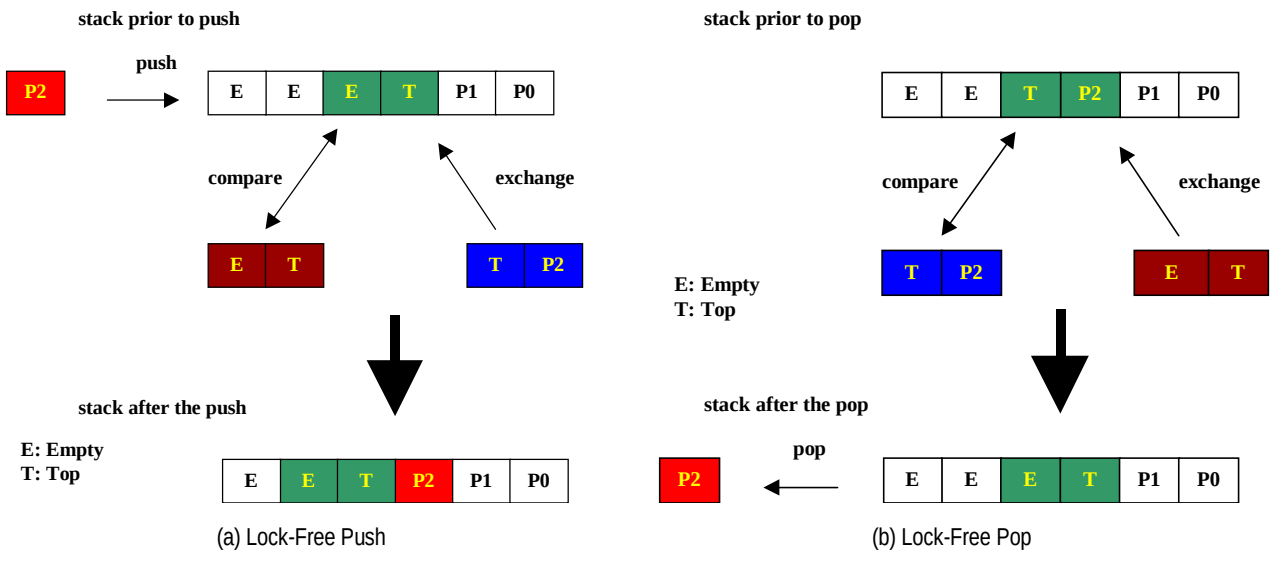


Figure 5: Lock-free push/pop operations

these contexts corresponds to an active Windows NT* fiber. The collection of stacks from active and suspended fibers resembles a cactus stack. Figure 6 illustrates the structure of the execution contexts for a parallel loop nested inside a COBEGIN section. The difference between this structure and a true cactus stack is that all the variables in parent contexts that are needed by child contexts are passed via parameters, instead of being accessed by a static linkage pointer.

In IML, when a new level of parallelism is invoked, the parent context is immediately suspended, and child contexts are initiated from a pre-allocated and recycled fiber pool. Upon completion of the parallel section, one of the active children contexts resumes the parent context [1].

Compiler-Generated Parallel Code

Parallel programming, compared to sequential programming, is a difficult and error prone process. Methods to automate or semi-automate this process are of great value to programmers. Automatic tools, such as automatic parallelizing compilers, are the ultimate tools that programmers can use to parallelize programs. Ideally, these compilers relieve the programmer of all the concerns of parallelization. However, two decades of research in parallel optimization has shown that optimal parallelization is often not achieved solely through automatic methods. In fact, semi-automatic parallelization techniques allow the user to guide the pre-processor or compiler in parallelizing the code. Fully and semi-automatic parallelizing methods are discussed in the following paragraphs.

Before discussing the two methods of parallelizing code, however, we first need to discuss how to transform code in order to interface with the IML. The transformation for parallel loops is described here, but a similar

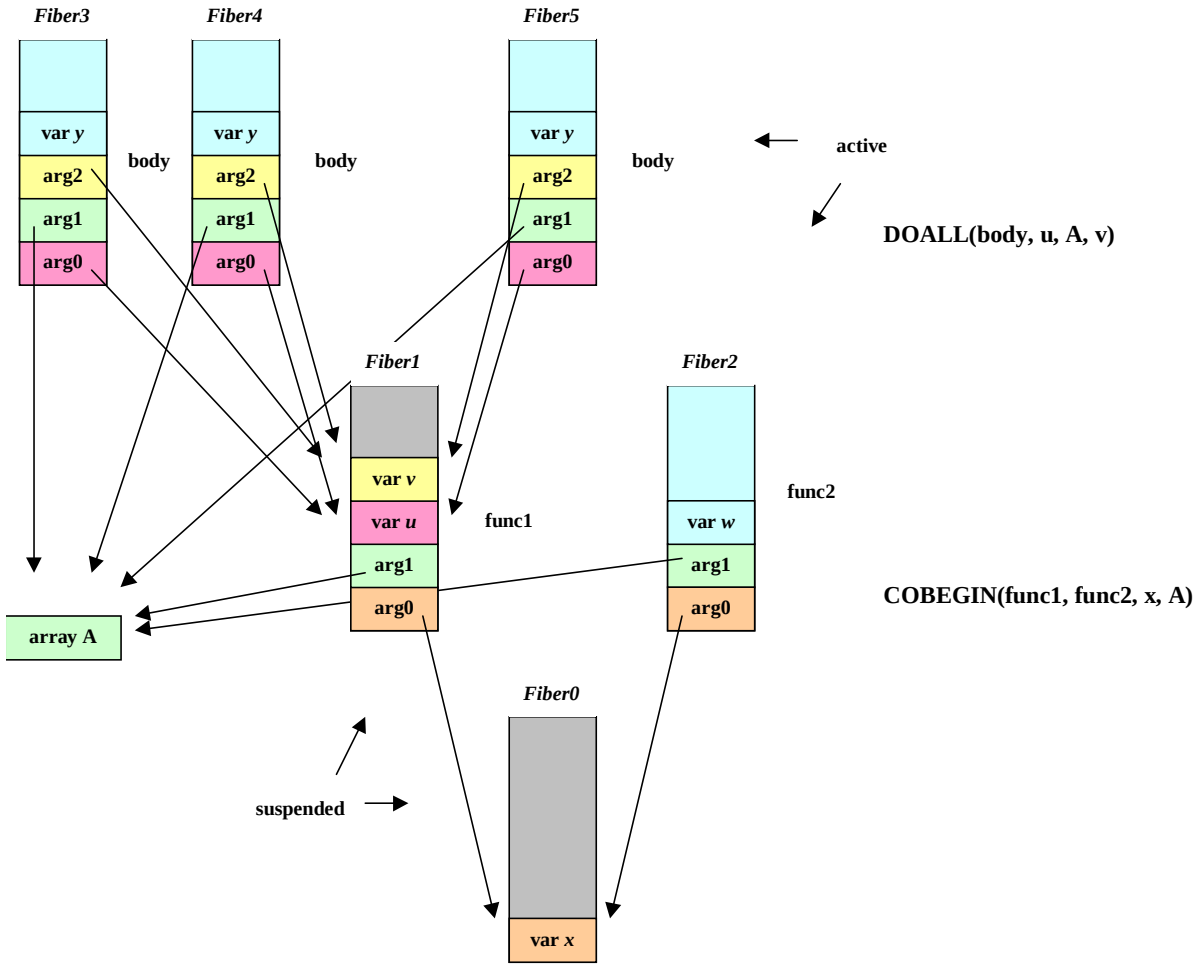


Figure 6: Stack configuration for a DOALL loop nested in a COBEGIN section

transformation is needed for parallel tasks. For each parallel loop, a new function is created that contains the loop body and the necessary support code. The shared and private variables of the loop are determined. Private variables (i.e., those with no cross-iteration dependencies) are redefined as local variables in the new function. The loop iteration variable is also declared as a local variable. All other variables are classified as shared and are declared as formal parameters of the new function. At the original site of the parallel loop, the body of the loop is replaced by a call to the IML entry point, `iml_DOALL`. All the information needed to execute the loop in parallel is passed to this entry point. This consists of the number of iterations, a pointer to the newly created function, the list of shared variables, the loop scheduling type, and the minimum chunk size. Some needed support code is also inserted around the call site.

Fully Automatic Parallelization

As mentioned above, the most convenient, but not necessarily the optimum, way to construct parallel programs is to utilize fully automatic tools such as parallelizing pre-processors or compilers that handle both the discovery of parallelism and the translation of parallel constructs. Two compilation systems, based on IML, were developed to facilitate fully automatic parallelism.

The Parafrase-2 parallelizing compiler, developed at the University of Illinois, was enhanced to output a transformed source code file with calls to the IML. The solid line path in the left side of Figure 7 shows this completely automatic path, which relinquishes the programmer from any parallelization effort. Parafrase-2 inserts all the necessary source code to manage the detected parallelism.

At Intel Microcomputer Research Labs (MRL), a parallel optimization module was added to the Intel compilers. This module accepts as input a standard intermediate representation of the source code produced by the front ends. Control and data flow analysis is performed on the intermediate form, and data dependence analysis is done on its loop constructs to discover loops with no loop carried dependencies (i.e., DOALL loops). These loops are then marked for a translator to convert them to the form required by the IML interface. This path is shown in solid lines in the right side of Figure 7.

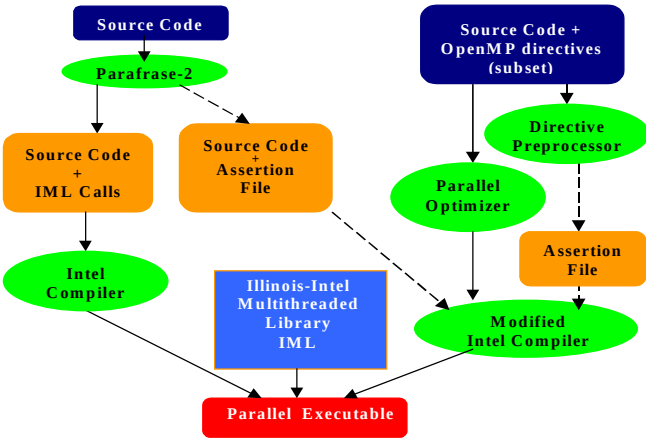


Figure 7: Parallel code generation

Semi-Automatic Parallelization

As stated earlier, fully automatic parallelization has its limitations. Fortunately, these limitations can be overcome by providing supplemental information about the program to the compiler or pre-processor, including information that cannot be determined at compile time. This information, which can be represented in many forms, such as directives, external assertion files, or interactive questioning by the compiler during compilation, is critical in the generation of efficient code.

Two semi-automatic methods have been implemented, corresponding to the diagonal path and the rightmost path (dashed lines) of Figure 7. Both methods encode parallel information in an assertion file, which the Intel C/C++ and FORTRAN compilers have been extended to access.

To automate the assertion file generation process, Parafrase-2 has been extended to generate an assertion file (along with a source code file). This process is fully automatic when Parafrase-2 generates efficient parallel code. However, when the code generated by Parafrase-2 does not perform adequately, the information in the assertion file can be augmented by the programmer to increase the performance of the parallel executable.

Another method is to encode parallelism in the source code via OpenMP directives explicitly. The augmented source code is then passed through a directive preprocessor, which generates an assertion file from the directives. The assertion file and the source code are then given to the modified Intel compilers to generate the parallel executable.

These semi-automatic methods detach the identification and the exploitation of parallelism. Parafrase-2 or the programmer identifies the parallelism in the program, while the modified Intel compilers transform the program to exploit this parallelism. Compared to the scheme where IML calls are inserted by Parafrase-2, the configurations using the assertion file increase the accuracy of the analysis performed by the modified Intel compilers.

Performance

Several experiments were performed to measure and evaluate the benefits of IML. These experiments were performed on a system with the following configuration:

- Intel System: Four 200MHz Pentium Pro processors, each with 256KB L2 cache; 4 way interleaved memory 512MB (60ns Fast Page Mode), Matrox MGA Millenium graphics card with 4MB VRAM
- Microsoft Windows NT* Server version 4.0
- Intel C/C++/FORTRAN Compiler version 2.4 (for compilation of IML and application programs)
- Microsoft Macro Assembler version 6.11d (for compilation of IML)
- Illinois-Intel Multithreading Library version 1.1

Intel System Memory Subsystem Performance

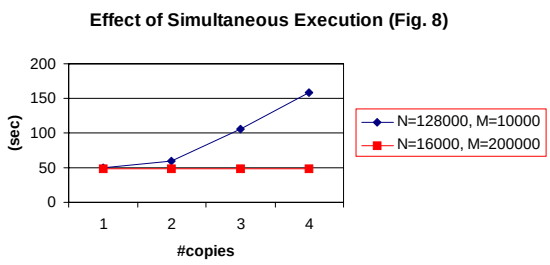
Before proceeding with the experiments that present the results with IML, a simple experiment was conducted to determine the impact of the memory subsystem performance on the results.

The effect of main memory bandwidth was evaluated using the code segment in Figure 8. When the array `block` fits into the L2 cache, almost perfect cache locality is achieved, resulting in very few main memory accesses. On the other hand, a large number of cache misses on the L2 cache were observed for larger sizes of the array `block`.

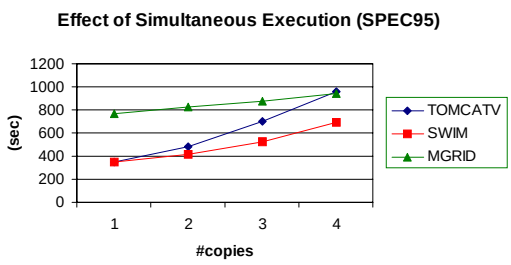
```
double a=0, block[N];
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
    a += block[i];
  }
}
```

Figure 8: Code segment for memory subsystem test

Figure 9(a) illustrates the performance degradation of this code when multiple copies of this program are executed simultaneously. By running multiple independent processes of the same program, the experiment creates increased requirements on the bandwidth to main memory. For small sizes of array `block`, four copies of the program are executed without any performance degradation. When the size of array `block` is larger than the L2 cache (and thus each process now initiates more main memory accesses than the case of small array sizes), a performance degradation of approximately 220% is observed for four copies. This behavior is not limited to the test case. For example, three SPECfp95 benchmarks, MGRID, SWIM, and TOMCATV show 22%, 97%, and 174% slowdown, respectively, when four copies are concurrently executed (Figure 9(b)). The performance degradation observed in Figure 9 is attributable to the bandwidth between secondary cache and main memory. Therefore, it can be improved by performing cache locality optimizations. In the following experiments, no manual cache locality optimizations were performed.



(a) Execution Time of Test Code in Fig. 8



(b) Execution Time of SPECfp95 Codes

Figure 9: Effect of simultaneous execution

BLAS3

BLAS3 is a library package for matrix-vector operations. Several complex BLAS3 library functions from a preliminary version of the Intel Math Kernel Library (MKL) were ported to IML. MKL uses a conventional multiprocessing library, which exploits only a single level of parallelism. The computational kernels of these functions are written in FORTRAN and iA assembly with cache locality optimizations.

The speedup curve of one of these library functions, CGEMM, is presented in Figure 10 and can be seen to scale linearly. As the problem size increases, a moderate increase in the speedup is observed. The figure also illustrates that there is no significant difference in performance between IML and MKL. Unlike MKL which is a non-queue-based, singly nested, loop-only library and hence highly tuned, IML is a queue-based, runtime system that supports any mix of arbitrarily nested loop and functional threads, and hence is better suited for a larger class of application codes. Thus, one would expect that the additional functionality and general-purpose nature of IML would increase the overhead cost. Due to efficient implementation, this is not the case as is clear from Figure 10, and IML incurs thread management overhead comparable to fine-tuned libraries that support single loop only parallelism.

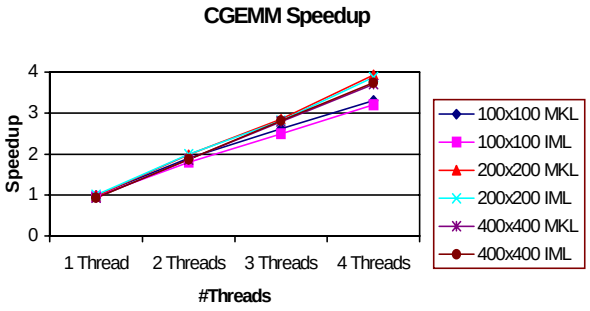


Figure 10: CGEMM speedup

SPECfp95

Three of the SPEC95 floating-point benchmarks, MGRID, SWIM, and TOMCATV were parallelized by the Parafrase-2 compiler. The benchmarks are numeric intensive and highly parallel. The solid lines of Figure 11 show the speedup curves for these benchmarks as measured on the actual system. Automatic parallelization with the Intel compiler produced similar results. As expected, the poor scaling is the result of the limited memory bandwidth. Extrapolating from these benchmark results and the performance of the memory subsystem for each benchmark from Figure 9(b), projected speedups, shown as the dotted lines in Figure 11, are obtained. These speedup curves correspond to a hypothetical system with sufficient main memory bandwidth.

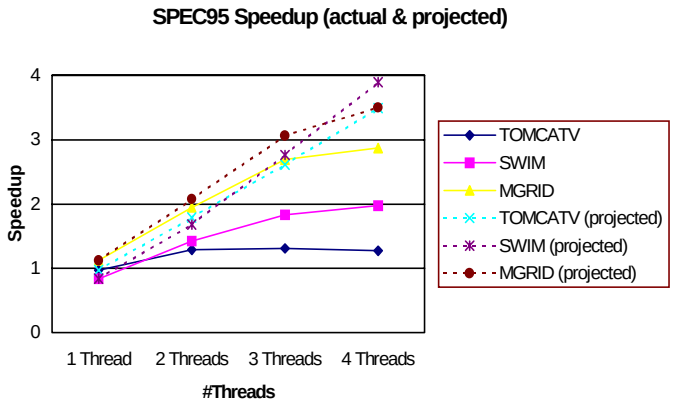
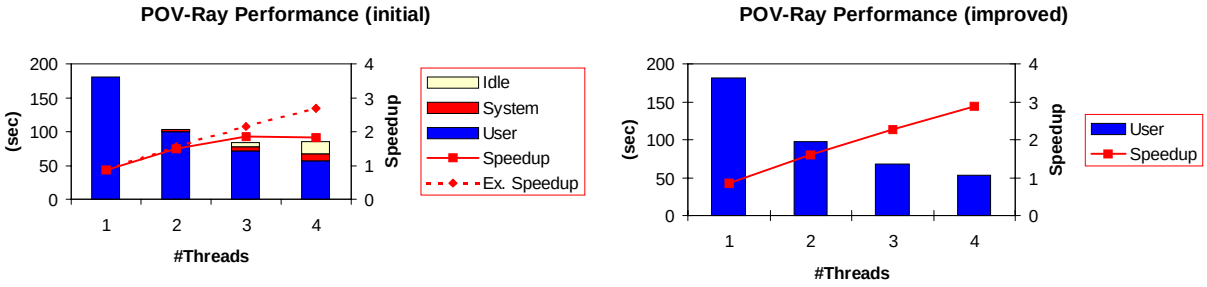


Figure 11: SPEC95 Speedup (actual & projected)

POV-Ray for Windows*

POV-Ray is a ray-tracing software package available to the public. This application is highly parallel since every pixel can be processed independently. In this experiment, however, only the parallelism between horizontal scan lines was exploited. The performance of the initial port is shown in Figure 12(a).



(a) Initial Performance

(b) Improved Performance

Figure 12: POV-Ray for Windows* performance

Execution times are represented by the bar graph. The colors blue, red, and white correspond to user, system, and idle time of the parallelized POV-Ray, respectively. The solid line represents the speedup over the original source distribution. The system and idle times, depicted in the bar graph, are due to the mutual exclusion inside `malloc ()`. The dashed line represents the projected speedup, computed only from the user time.

To eliminate this system level overhead, a second port enclosed the `malloc ()` function calls with user-level mutual exclusions, resulting in the performance shown in Figure 12(b). The system level overhead was eliminated, and linear speedup was obtained. Although the second port performs better than the initial port, it still suffers from serialization in the `malloc()` routine. A truly parallel implementation of `malloc()` would allow for even greater performance gains.

Conclusions

In this paper we have described IML, the Illinois-Intel Multithreading Library designed to support various types of parallelism efficiently. IML extends substantially the degree of available support for multithreading (found in other experimental or commercial systems) by providing the capability to express nested loop and cobegin/coend parallelism. Users can benefit from IML in terms of a reduction in development time by expressing parallelism in the IML API. To further assist the application developer, the Parafrase-2 compiler at the University of Illinois and the Intel FORTRAN Compiler have been modified to analyze programs to detect parallelism (automatically and with directives) and to generate calls to IML. Performance of automatically generated parallel code for SPECfp95 applications with IML is the same as hand-coded parallel programs.

Acknowledgments

This work was supported in part by a grant from Intel Corporation, in part by DARPA under grant MDA 904-96-C-1472, and in part by ONR under grant N00014-94-1-0234.

References

[1] Chow, J-H. and Harrison, L., Microtasking Recursive, Parallel Programs, *In Proc. of Int'l Conf. on Parallel Processing Vol. 2: Software*, 1990

[2] Girkar, M. Functional Parallelism: Theoretical Foundations and Implementation, *Ph.D. Thesis*, University of Illinois, 1992.

[3] Girkar, M. and Polychronopoulos C., Automatic Extraction of Functional Parallelism from Ordinary Programs, *IEEE Trans. on Parallel and Distributed Systems* Vol. 3, No. 2, 1992.

[4] Girkar, M. and Polychronopoulos, C. Extraction of Task-Level Parallelism, *ACM Trans. on Programming Languages and Systems* Vol. 17, No. 4, 1995.

[5] IML Home Page. http://www.csrd.uiuc.edu/IML.

[6] OpenMP Home Page. http://www.openmp.org.

[7] Polychronopoulos, C. *et.al.*, PARAFRASE-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors, *Int'l J. of High Speed Computing* Vol. 1, No. 1, pp. 45-72.

[8] E. D. Polychronopoulos, "Scheduling Heuristics for Multiprocessors," PhD Thesis in progress, Laboratory for High-Performance Computing, University of Patras, 1997.

[9]Valois, J. Implementing Lock-Free Queues, *Proc. of Int'l Conf. on Parallel and Distributed Computing Systems*, 1994, pp. 64-69.

[10] Win32 Fiber APIs. Microsoft Corporation.

Authors' Biographies

Milind Girkar received a B.Tech. from the Indian Institute of Technology, Mumbai, an M.Sc. from Vanderbilt University and a Ph.D. from the University of Illinois at Urbana-Champaign, all in computer science. Currently, he is a software engineer in Intel's Microcomputer Research Labs where he works on parallelizing compilers and Java Just-In-Time compilers. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail address is mgirkar@gomez.sc.intel.com.

Mohammad R. Haghighat is a software engineer at Intel's Microcomputer Research Labs where he works on parallelizing compilers and Java Just-In-Time compilers. He holds a B.Sc. in Computer Science and Engineering from Shiraz University, and an M.Sc. and a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. He is the author of a book on symbolic analysis for parallelizing compilers. His e-mail address is mhaghigh@gomez.sc.intel.com.

Paul Grey did his B.Sc. in Applied Physics at the University of the West Indies and his M.Sc. in Computer Engineering at the University of Southern California.

Currently he is a staff software engineer at Intel's Microcomputer Research Labs, researching compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., Sun Microsystems, and Silicon Graphics. His research interests include optimizing compilers, parallel computer architectures and 3D computer graphics. His e-mail address is pgrey@gomez.sc.intel.com

Hideki Saito received his B.E. degree in information science from Kyoto University in 1993. Currently, he is a Ph.D. candidate in the Department of Computer Science, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. At CSRD, he participates in the PROMIS parallelizing compiler project. His research interests include computer architectures and program optimizations for parallel processing. His e-mail address is saito@csrd.uiuc.edu.

Nicholas Stavrakos received his B.Sc. degree in computer engineering from the University of Illinois at Urbana-Champaign in 1994. Currently, he is a Ph.D. candidate in the Department of Electrical and Computer Engineering, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois. At CSRD, he participates in the Parafrase-2 and PROMIS parallelizing compiler projects. His research interests include parallelizing compilers, symbolic analysis, and multithreaded code generation. His e-mail address is stavrako@csrd.uiuc.edu.

Constantine D. Polychronopoulos is a Professor in the Department of Electrical and Computer Engineering and the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He received his Ph.D. from the University of Illinois at Urbana-Champaign in 1986, his M.Sc. from Vanderbilt University in 1982, and his B.Sc. from the University of Athens in 1980.

His research interests are on compilers and architectures for high-performance computer systems, multithreading, and multiprocessor operating systems. He has published extensively on parallelizing compilers and scheduling, and has been leading the Parafrase-2 and PROMIS projects at CSRD. He was the recipient of a 1989 NSF Presidential Young Investigator award, and is a Fulbright Scholar. Some of the results of his research team have been implemented in several commercial systems by DEC, Cray Research, Convex, Alliant, SGI and others. His research work has been funded by NSF, DARPA, ONR, and industry. His e-mail address is cdp@csrd.uiuc.edu.

Appendix: Examples for IML API

This appendix gives the examples for IML API functions. The code segments presented in this paper are simplified for explanatory purposes. Further details on the usage of IML can be found in the IML Home Page [5].

Basic API Functions

In this section, the usage of basic API functions is demonstrated using the original code shown in Fig. A-1. In this example, all three loops (i, j, and k) are parallel, and the two outermost loops (j and k) can be executed simultaneously.

```
double A[M][N], B(N), c;
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
    A[i][j] = i * j;
  }
}
for(k=0;k<N;k++){
  B[k] = k;
}
c = foo(A, B, N, M);
```

Figure A-1: Code Segment for Fig. A-2 to A-5.

Converting the j- and k-loops to DOALL results in the code shown in Fig. A-2. The k-loop is executed after the j-loop is completed.

```
    double A[M][N], B(N), c;

    iml_DOALL(&jloop, &M,
              &static_schedule,
              &chunk, &params,
              A, N);
    iml_DOALL(&kloop, &N,
              &simple_schedule,
              &chunk, &params, B);
    c = foo(A, B, N, M);



  void jloop(int *start,
             int *iters,
             double *A, int *N){
    for(j=*start;
        j<*start+*iters;j++){
      for(i=0;i<N;i++){
        A[i][j] = i * j;
      }
    }
  }
  void kloop(int *start,
             int *iters,
             double *B){
    for(k=*start;
        k<*start+*iters;k++){
        B[k] = k;
      }
    }
  }
```

Figure A-2: Using `iml_DOALL()` for Outer Loops

The i-loop can also be converted to DOALL as in Fig. A-3. Unlike conventional libraries that would internally execute the i-loop described in this fashion as a sequential loop, IML can actually execute it in parallel.

```
  double A[M][N], B(N), c;

  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
  c = foo(A, B, N, M);



void jloop(int *start,  int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self_schedule, A, j);
  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-3: Using `iml_DOALL()` for all loops

Furthermore, the j- and k-loops can be executed simultaneously, using `iml_COBEGIN()` (Fig. A-4) or `iml_EnQ()` (Fig. A-5).

Extended API Functions

The usage of the extended API functions is demonstrated using the original code shown in Fig. A-6. The reduction operation can be performed in parallel, where each thread reduces to its own variable, and global reduction across the result of the thread-wise reduction is performed afterwards (Fig A-7). If a sequential section of the program persists for a period of time, the programmer or the compiler can use `iml_ReInitMultiThread()` to reduce the number of active threads (Fig. A-8).

```
  double A[M][N], B(N), c;

  iml_COBEGIN(&tasks, &jloop_0,
              &kloop_0,
              &params, A, B, N, M);
  c = foo(A, B, N, M);


void jloop_0(double *A, double *B,
             int N, int M){
  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
}
void kloop_0(double *A, double *B,
             int N, int M){
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
}
void jloop(int *start, int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self-schedule, A, j);

  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-4: Using `iml_COBEGIN()`

```
  double A[M][N], B(N), c;

  iml_EnQ(&jloop_0, A, B,
          N, M, &cnt, &c);
  iml_EnQ(&kloop_0, A, B,
          N, M, &cnt, &c);



void jloop_0(double *A, double *B,
             int N, int M,
             int *cnt, double *c){
  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
  if (iml_DecAndFetch(cnt)==0){
    iml_EnQ(&foo_0, A, B, N, M, c);
  }
}
void kloop_0(double *A, double *B,
             int N, int M,
             int *cnt, double *c){
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
  if (iml_DecAndFetch(cnt)==0){
    iml_enQ(&foo_0, A, B, N, M, c);
  }
}
void foo_0(double *A, double *B,
           int N, int M, double *c){
    *c = foo(A, B, N, M);
}
void jloop(int *start, int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self-schedule, A, j);

  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-5: Using `iml_EnQ()`

```
double a, B(N);
for(i=0;i<N;i++){
  a += B{i];
}
```

Figure A-6: Code Segment for Fig. A-7 and A-8

```
  double a, A(NCPU), B(N);

  iml_DOALL(&iloop, &N,
            &static_scheduling,
            &chunk, &params,
            A, B);
  for(i=0;i<NCPU;i++){
    a += A[i];
  }


void iloop(int *start, int *iters,
           double *A, double *B){
  ID = iml_GetThreadID();
  for(i=*start;
      i<*start+*iters;i++){
    A[ID] += B{i];
  }
}
```

Figure A-7: Using `iml_GetThreadID()`.

```
  double a, A(NCPU), B(N);

  iml_DOALL(&iloop, &N
            &static_scheduling,
            &chunk, &params,
            A, B);
  iml_ReInitMultiThread(1);
  // suspend all other threads
  for(i=0;i<NCPU;i++){
    a += A[i];
  }


void iloop(int *start, int *iters,
           double *A, double *B){
  ID = iml_GetThreadID();
  for(i=*start;
      i<*start+*iters;i++){
    A[ID] += B{i];
  }
}
```

Figure A-8: Using `iml_ReInitMultiThread()`.