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.

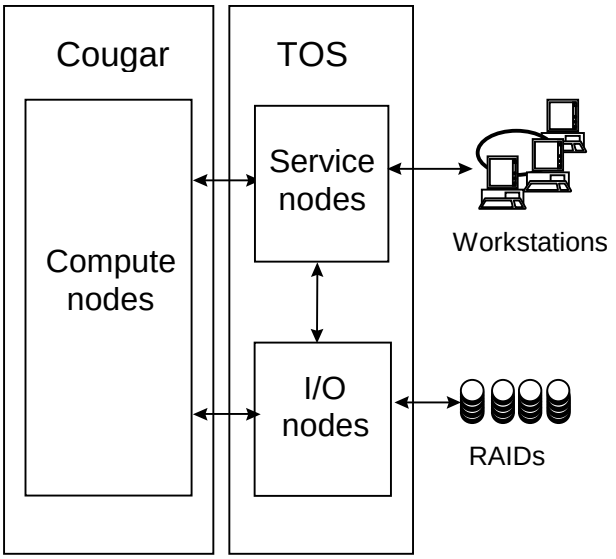

Figure 1: Overview of the TFLOPS system architecture

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.

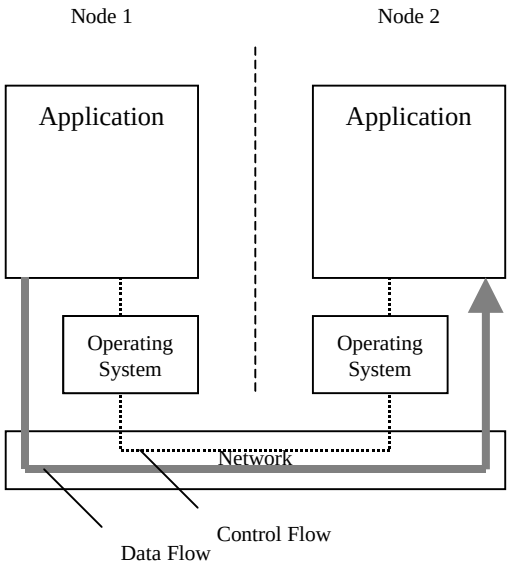

Figure 2: Transfer of an application message between compute nodes

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).

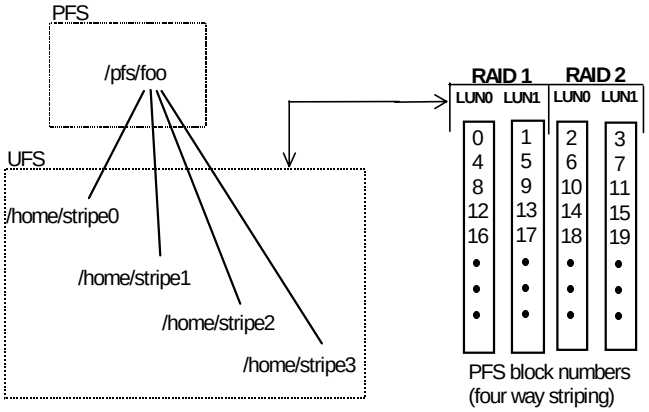

Figure 3: PFS stripe file mapping

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).

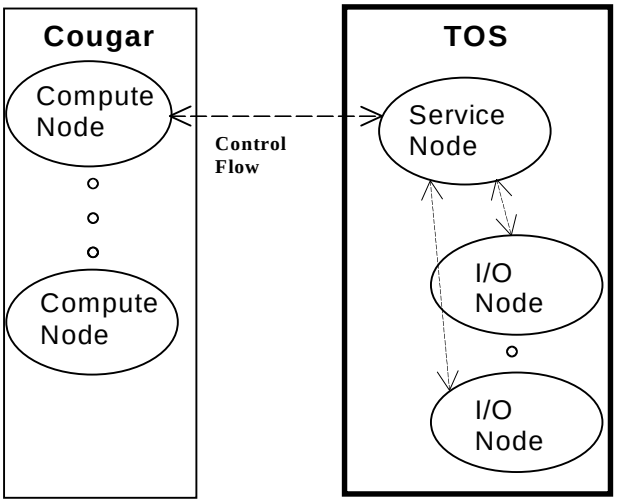

Figure 4: I/O service process control flow

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).

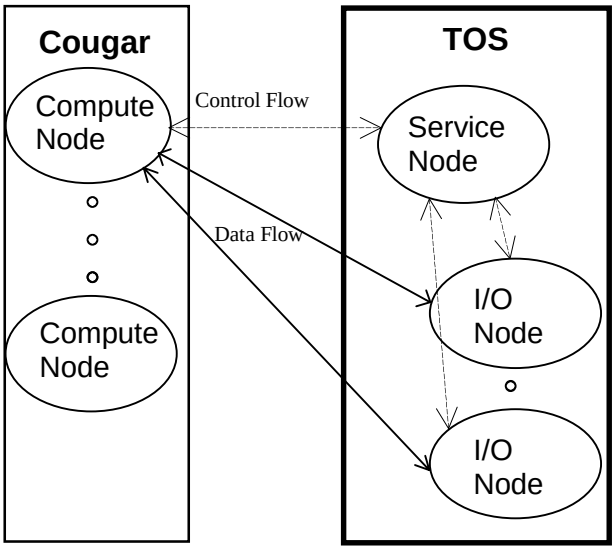

Figure 5: I/O service process control and data flow

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.

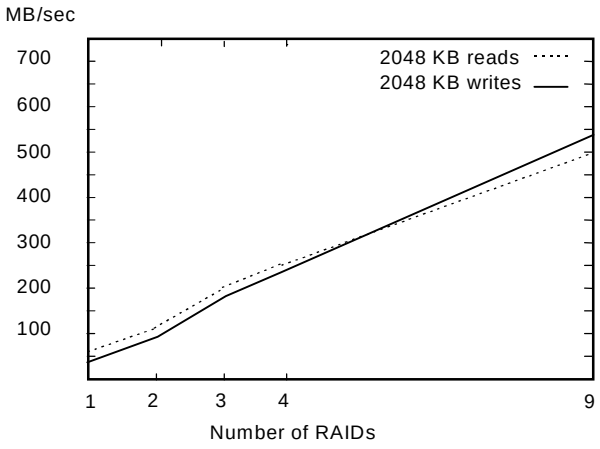

Figure 6: PFS throughput scalability

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.

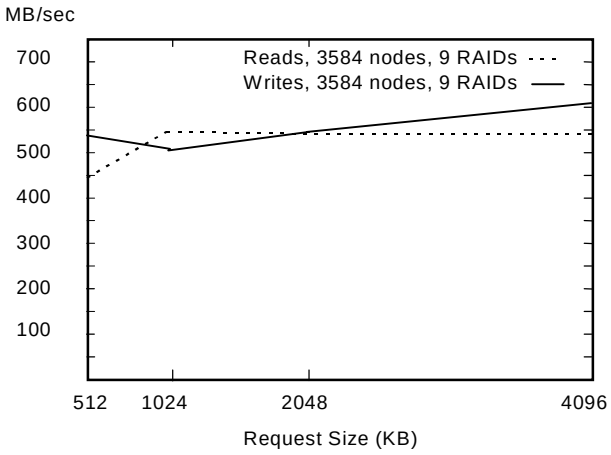

Figure 7: PFS read and write bandwidth

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.

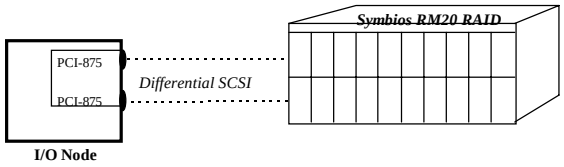

Figure 8: Symbois RAID connection to TFLOPS system

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.