Parallel I/O Subsystems in Massively Parallel Supercomputers

Dror G. Feitelson* Peter F. Corbett

Sandra Johnson Baylor Yarsun Hsu

IBM T. J. Watson Research Center

P. O. Box 218

Yorktown Heights, NY 10598

feit@cs.huji.ac.il, {corbett,sandyj,hsu}@watson.ibm.com

Tel: (914) 945-2769, Fax: (914) 945-2141

Abstract

Applications executing on massively parallel processors (MPPs) often require a high aggregate bandwidth of low-latency I/O to secondary storage. In many current MPPs, this requirement has been met by supplying internal parallel I/O subsystems that serve as staging areas for data. Typically, the parallel I/O subsystem is composed of I/O nodes that are linked to the same interconnection network that connects the compute nodes. The I/O nodes each manage their own set of disks. The option of increasing the number of I/O nodes together with the number of compute nodes and with the interconnection network allows for a balanced architecture. We explore the issues motivating the selection of this architecture for secondary storage in MPPs. We survey the designs of some recent and current parallel I/O subsystems, and discuss issues of system configuration, reliability, and file systems.

Keywords: parallel I/O, parallel file system, disk striping, data staging, RAID, scalable performance, hierarchical storage.

*Current address: Institute of Computer Science, Hebrew University, 91904 Jerusalem, Israel

Introduction

Massively parallel processors (MPPs), encompassing from tens to thousands of processors, are emerging as a major architecture for high performance computers. Most major computer vendors are now offering computers with some degree of parallelism, and many smaller vendors specialize in producing MPPs. These machines are targetted for both grand challenge problems and for general purpose computing. As in any computer, it is necessary in MPP architectural design to provide a balance between computation, memory bandwidth and capacity, communication capabilities, and I/O. In the past, most of the design effort was focussed on the basic compute and communications hardware and software. This lead to unbalanced computers, that had relatively poor I/O performance. Recently, much effort has been focussed on the design of hardware and software for I/O subsystems in MPPs. The architecture that has emerged in many MPPs is based on an internal parallel I/O subsystem, encompassing a collection of dedicated I/O nodes, each managing and providing I/O access to a set of disks. The I/O nodes are connected to other nodes in the system by the same switching network as connects the compute nodes to each other.

In this paper, we examine the reasons why many manufacturers of parallel computers have chosen the parallel I/O subsystem architecture. We survey and compare the parallel I/O subsystems in many current and recent MPPs, examining both hardware and software issues, and look at parallel file systems and their user interfaces. We focus on how an internal I/O subsystem can satisfy application requirements, including:

- Supplying the required bandwidth and capacity. The I/O subsystem performance has to be balanced with the computation rate and the communication bandwidth of the computer. Its capacity must accommodate the gigabytes of data often used by applications executing on MPPs. An internal I/O subsystem based on multiple I/O nodes together with a scalable interconnection network provides a scalable architecture that can meet these requirements.

- Minimizing access latency to reduce process wait states. This is achieved by connecting the I/O nodes to the MPP's high-performance network, and eliminating centralized control points. Process wait states are also reduced by the provision of asynchronous I/O at the user interface.

- Providing reliability in the face of faults, and maintaining data availability when some system element is down or being replaced. This is done by employing redundancy across the I/O nodes, or among several disks attached to each I/O node.

- Cost effectiveness. In many cases, adding high-performance I/O hardware to an MPP is expensive. However, the

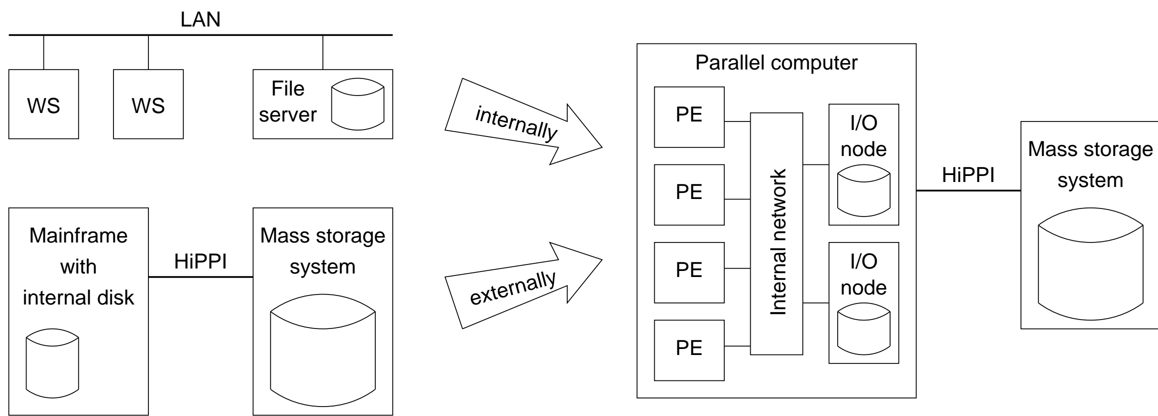

Figure 1: *The I/O systems of MPPs incorporate features of networked workstations internally, and of mainframes and supercomputers externally (WS = workstation, PE = processing element).*

expense may be justified if the parallel I/O subsystem allows the machine as a whole to be used more effectively, especially when considering that the internal I/O subsystem also reduces the requirements placed on external file servers. Development cost is also a major issue to MPP manufacturers. By using a similar node architecture for I/O and compute nodes, development cost can by greatly reduced over the cost of developing a separate external I/O subsystem.

- Supporting complex operations. New file systems promise control over the layout of file data, the ability to partition data among application processes, and support for sharing data among processes. These features all fit well with the programming and data decomposition models being used on MPPs.

The parallel I/O subsystems used in MPPs are a unique architectural feature. The development of such systems therefore seems to contradict the general convergence of cumputing in general to a network-centric structure, where I/O is performed to specialized servers across the network [5]. In contrast, we show that actually I/O systems in MPPs are largely in tune with network-based I/O, but that this is done at two separate levels: first there is the I/O across the MPP's internal network to the dedicated I/O nodes, which is similar to I/O from workstations to file servers on a LAN, and then there is I/O from the MPP as a whole to external mass storage systems, typically across high-bandwidth channels (Fig. 1). Thus the internal parallel I/O system is essentially an adaptation of LAN-connected file servers to a parallel environment. It does not replace the external I/O channels and mass storage systems, but rather augments them with a staging area that provides better service for parallel applications by being more tightly coupled with them. In this sense, an internal parallel I/O subsystem performs the same function as the high-performance I/O devices found

in most mainframes and vector supercomputers.

The systems discussed in this paper are mostly tightly coupled distributed memory MIMD (Multiple-Instruction, Multiple-Data) MPPs. In some cases, we discuss shared-memory and SIMD (Single-Instruction, Multiple-Data) machines as well. For large parts of the discussion, it is convenient to define three node types. *Compute nodes* are optimized to perform floating point and numeric calculations, and have no local disk except perhaps for paging, booting, and operating system software. *I/O nodes* contain the secondary storage in the system, and provide the parallel file system services. *Gateway nodes* provide connectivity to external data servers and mass storage systems. In some cases, individual nodes may serve as more than one type. For example, the I/O and gateway functions are often handled by the same nodes.

We begin by presenting the motivation for using an internal parallel I/O subsystem, and elaborate on possible configurations and on the distinction between internal and external I/O. Subsequent sections deal with reliability and with file systems that exploit the parallel I/O to provide enhanced services to applications.

Parallel I/O

Among commercial MPPs that are currently or recently available, the majority include internal parallel I/O subsystems as an important architectural feature. These I/O subsystems are composed of multiple I/O nodes, each with one or more disks, that are connected to the same interconnection network as the compute nodes. Examples include the scalable disk array in the Connection Machine CM-5 and the I/O partition in the Intel Paragon (see sidebar on architectures with parallel I/O). In this section, we examine why this structure is suitable for parallel applications running on MPPs. We then contrast the services provided by internal parallel I/O subsystem with those provided by external servers.

The main reason for using a parallel I/O system is that it provides parallel access to the data. A typical MPP has its primary memory distributed across many compute nodes. As a consequence, application data is distributed among the nodes of the MPP. The data distribution is application dependent, being designed to enhance locality of reference during computation. Indeed, much recent work on parallel programming has focused on the issue of data decomposition, i.e. how to partition large data structures among multiple processors. This is either done by language directives, as in HPF (High-Performance Fortran) [12], or automatically by compilers that analyze the access patterns [2].

I/O operations involving distributed data structures require communication with all the processors which have parts of the data. This communication can be done in parallel, to use the network bandwidth available to all the compute

nodes. However, it should be noted that the compute nodes are only one side of the I/O operation. The other side is the I/O subsystem. If the I/O subsystem is not itself parallel, it will not be able to utilize the parallel communication channels with the processors. A parallel I/O system, based on multiple I/O nodes, allows data to be transfered in parallel between compute nodes and I/O nodes.

The data distribution causes the characteristics of the I/O required by the compute nodes participating in a parallel application to be very different from what is typical in vector supercomputers. Instead of large sequential accesses, there are many small fragmented accesses. It is true that *in aggregate* the accesses from the different compute nodes often cover the whole data set, but the part accessed by *each* compute node is typically fragmented because of the distribution pattern, rather than being contiguous [17].

The fragmented nature of I/O operations from parallel applications has two consequences: first, it implies that I/O involves the transfer of multiple small pieces of data. This is bad, because I/O operations are much more efficient when large sequential blocks are involved. Second, it implies that there is significant false sharing of file blocks [17]. This means that different compute nodes will likely access disjoint parts of each block, making it harder to cache blocks on compute nodes. A parallel I/O subsystem within the MPP solves these problems, because it is based on the MPP's internal high performance switching network. Such networks allow relatively small messages to be passed efficiently with little latency and high aggregate bandwidth. As a result, it is not so important to cache file blocks at the comute nodes — caching at the I/O nodes is sufficient. These caches, in turn, are also used to coalesce the fragmented accesses from multiple compute nodes into large sequential accesses that can use the disks efficiently. Moreover, the phenomenon of "interprocess locality" allows the the I/O operations from one compute node to cause prefetching of data required by another compute node [10].

We note in passing that a similar idea was implemented by the SSD (solid-state device) storage in the Cray X-MP and Y-MP. Likewise, the nCUBE system allows a fraction of the compute nodes to be used as a staging area, effectively using their memories as an SSD. Such solid-state devices support parallel I/O operations with even less latency than an internal parallel file system, but their cost limits their capacity.

Independent of the parallel nature of I/O requests on MPPs, a parallel I/O subsystem is also required for purely architectural reasons. Microprocessors double their speed every 18 months or so, while the bandwidth of I/O devices grows at a much slower rate. Therefore there is a danger that the supported F/b ratio (this is the ratio of the floating point operation rate of a computer to its bit rate of I/O) will grow larger with time [16]. In MPPs, which employ multiple high-end processors in tandem, the danger is especially great. In the absence of a fundamentally new persistent storage media, the only solution to the problem of scaling both the bandwidth and capacity of MPP persistent storage is

to increase the number of disks accessible by the processors of the MPP. Together with the increased number of disks, there is a need for increased channels to access them. This implies the use of a parallel I/O system. The aggregate bandwidth of such a system is scalable, and can be tailored to specific needs by changing the number of I/O nodes and disks.

Note that parallel I/O systems are already being used for uniprocessors. This is motivated by the ever-increasing gap between CPU performance and disk performance [16]. In most cases, the parallel disks are organized as a RAID (Redundant Array of Inexpensive/Independent Disks) [4], providing an interface similar to that of a single disk, but with higher bandwidth and reliability. While this can be used to provide balanced I/O for a uniprocessor, it is insufficient for an MPP. The RAID controller is often a bottleneck in current RAID designs. Therefore, in parallel machines the interface must be explicitly parallel, along with the underlying disk system.

Finally, another advantage of having a parallel I/O system is the distribution of loads. If I/O operations are scattered across multiple I/O nodes, there is reduced danger of contention at any one device and subsequent performance degradation.

Internal vs. External I/O

Providing an internal I/O subsystem solves the immediate needs of the compute nodes for parallel, high bandwidth, low latency I/O services, but it does not solve the whole I/O problem. It is still necessary to get data into and out of the MPP. In particular, many parallel applications must process large amounts of data that are permanently stored in archival storage systems, often on tape [23]. To handle this, interfaces are required from the MPP to external storage systems. The basic scheme is that data to be processed by one or more applications (often on the order of tens of gigabytes) is preloaded into the internal I/O subsystem and its parallel file system (Fig. 2). Then, one or more applications are run against this data. Any temporary data is stored in internal parallel files as required. Finally, output data is offloaded to an external storage system for archiving, post-processing, or display. The external storage systems are often shared by the MPP and other network connected computers.

There are several advantages to having an internal parallel I/O subsystem as the agent for data migration to and from the MPP. The internal parallel I/O subsystem can handle fragmented requests much more efficiently than an external server. External storage systems provide little or no parallel access to data. The data rates of the devices may be fairly high, on the order of tens of megabytes per second, but the presentation of the data is sequential, and in large blocks. While this is fine for longer term archival and on-line storage, it is unlikely to match the data decomposition

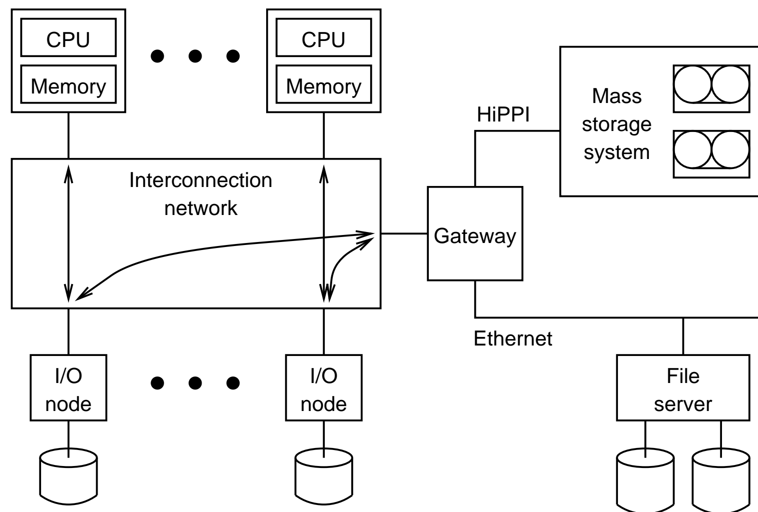

Figure 2: *Internal storage provides a staging area for data from external devices.*

and access patterns generated by parallel applications. Likewise, it is not likely that the boundaries on which the data is divided among compute nodes are aligned with any parameters of the external I/O system, such as the optimal block size for data transfer. Thus transfers of data between the MPP and the external server will usually require a phase of data collection or distribution. This re-organization can be done at the I/O nodes.

It would also be difficult for an external file server to satisfy I/O requests coming directly from compute nodes with a sufficiently low latency. If I/O requests are processed with excessive latency (many seconds for an on-line tape access) then applications will only be able to make I/O requests infrequently, and consequently, in very large blocks. This forces users to manage their own data sets by loading them into compute node memory at program initiation, and offloading them at program completion. On the other hand, an internal I/O subsystem that provides low latency access (in the range of 1ms – 30ms) allows users to program in a more typical style, where I/O requests can be made frequently.

Communication between compute nodes and internal I/O nodes is done through a reliable, low latency, message passing protocol, or through shared memory, rather than the slower network protocols used by LAN connected file servers. This is a consequence of using the MPP's internal interconnection network. Because they serve a more controlled and limited environment, the communication protocols used on MPP networks are optimized for the case of failure free transmission. This also reduces the latency involved in I/O operations.

Data reuse by multiple applications is also possible once the data is loaded into the internal file system. Buffering the data internally for reuse can reduce the bandwidth requirement of the gateway nodes and the external network.

Shared data that is repeatedly updated and reread by a parallel application can be effectively stored in the internal I/O system. Data that is read by many applications can be loaded once into the internal I/O system, and then be readily accessible to all the applications. Data that is produced by one application, and consumed by another can be effectively buffered in the internal I/O system. In effect, the internal I/O system serves as a cache for the external servers.

In the above discussion, we only talk about mass-storage systems as external servers. In addition, there is the option of using smaller file servers, that use the same type of storage devices as I/O nodes, with the same performance characteristics. However, these also cannot replace an internal parallel I/O system. First, external file servers would be connected by a LAN, and require communication protocols with higher latency. Second, the I/O requirements of a large MPP imply that multiple file servers are required. If each server is to be accessible from all nodes, we need a LAN network with enough bandwidth and routing capability to fully interconnect all of them. If the machine is partitioned into domains attached to different file servers, applications will be limited to running on nodes connected to the correct server. An internal parallel I/O system solves such problems by bringing the servers into the system, and connecting them to an internal interconnection network, thus providing adequate bandwidth among all nodes.

Architectural choices

There are several possible architectures for an internal parallel I/O subsystem (Fig. 3). The simplest is to just attach disks to compute nodes ((a) in the figure). Variants of this approach are used in the KSR1, the Meiko CS-2, and the IBM SP1. However, the Meiko CS-2 and IBM SP1 only use the local disks for scratch space and paging, not for persistent files. The reason is the difficulty in collocating applications and the data they need. For example, if a certain file is stored on the disk attached to a specific node, it is best for applications using that file to execute on that node. But that node could already be busy running other applications. Executing the new application on that node would then cause load imbalance, and executing it elsewhere would lead to inter-node communication for the I/O, possibly interfering with the other application's execution, since servicing the I/O requests would require many CPU cycles. Adding a separate I/O processor to the node would reduce interference with the other application's execution, but the I/O traffic could still interfere with its communication, depending on the topology of the network.

If the disks are attached to the network, rather than being attached directly to specific nodes (Fig. 3 (b)), there is no opportunity to collocate the data with a node that requires it. Each disk is equally accessible from all compute nodes in the machine, so each node should be able to access any data, with equal expectations of performance. However, the problem of mutual interference is not solved. Consider two applications writing two different files that reside on

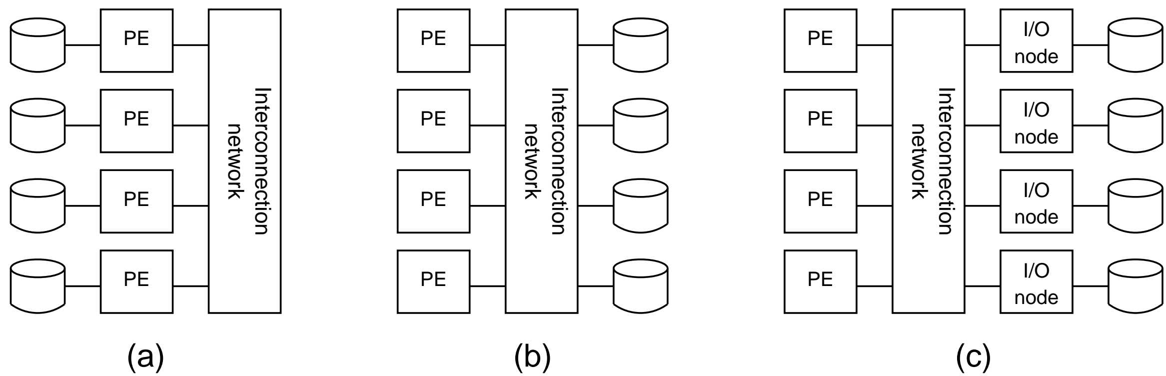

Figure 3: *Possible architectures for an internal parallel I/O subsystem.*

the same disk. The system software on the nodes running the applications must coordinate the block allocation on the disk, so that space is allocated to each individually with no consistency and security breaches. This typically implies some shared data structures with locks, and an access protocol to update them. In addition, there is no obvious place to buffer data of files shared by more than one process.

The internal I/O subsystem architecture selected by most vendors is to use separate I/O nodes, complete with processor, memory, and disks (Fig. 3 (c)). Examples include the CM-5, Intel iPSC and Paragon, nCUBE, Meiko CS-2, IBM SP2, and Tera (see sidebar on Architectures)[1]. Each I/O node is responsible for storing portions of parallel files, and handles all block allocation and buffering of those portions. Each parallel file is distributed in some way across the I/O nodes. Simultaneous data movement between the multiple I/O nodes and the multiple compute nodes of different applications is now possible, without any direct coordination among the compute nodes.

Dedicated I/O Nodes

There are several reasons why so many vendors have made this choice of having dedicated internal I/O nodes. These include the flexibility in locating data and processing, and the ability to provide high bandwidth I/O, as explained above. But dedicated I/O nodes also have drawbacks. In this section, we evaluate the advantages and disadvantages of dedicated I/O nodes, and suggest why this architecture has been chosen by so many MPP manufacturers.

One argument frequently made against providing I/O from I/O nodes across the interconnection network is that the latency of I/O accesses will increase relative to I/O from a local disk. Analysis of network I/O reveals that the dominant portion of access latency for small reads is disk access, which is on the order of tens of milliseconds (writes benefit

[1]The KSR1 can also be configured with dedicated I/O nodes, by preventing user computations from executing on nodes attached to I/O devices.

from buffering which can remove the disk latency from the critical path of the write). Current MPP interconnection networks have latencies of tens of microseconds. Therefore the network itself does not add significant overhead. However, communication and file system software can add milliseconds of latency to large requests if data is copied between software layers. It is therefore crucial to remove unnecessary copying of data, and pipeline large requests through the software layers. Good design can reduce the overall latency of read requests to the point where it is dominated by disk latency. Of course, it is important that the internal network latency not increase dramatically under the load that will be placed on it when I/O is done over the network [3].

The use of dedicated I/O nodes creates an opportunity to reduce access times to below disk access times by exploiting caching and prefetching. It is frequently the case that multiple compute nodes request data that is not contiguous in a file, but that the aggregation of the requests of the compute nodes is contiguous [17]. This occurs, for example, when an application has partioned a file in a strided decomposition among its processes, or when processes are each reading from or writing to the current position of a shared offset into the file. Such applications are said to display interprocess locality. In this case, I/O from shared I/O nodes can result in lower latency than from a local disk because of the benefit gained from prefetching initiated by the requests of other compute nodes. Recognizing such aggregate sequential behavior can also eliminate the small write problem; the aggregate write accesses may completely overwrite file blocks that would otherwise only be partially modified. The need to fetch these blocks before modifying them can be eliminated.

Separation of compute nodes from I/O nodes allows freedom to configure the machine for the application domain. There is a wide disparity in the I/O requirements of applications that run on supercomputers. To keep the compute node CPUs busy, it is necessary to provide enough disks to meet the I/O bandwidth, capacity, and latency requirements. Separating I/O nodes from the compute nodes allows this flexibility, as it allows a computer to be configured with a specific I/O bandwidth and capacity, and reducing latency due to queueing of I/O requests. Parallelism can be configured both at the I/O subsystem interface by changing the number of I/O nodes, and at the disk level, by changing the number of disks.

The distinction between the compute node and the I/O node hardware can simply be that I/O nodes have disks. However, a flexible, general solution allowing for configurations with more or fewer disks as they are required would have to provide additional empty disk slots in each compute node. This would have a great impact on the compute node packaging density. The alternative is to configure compute nodes with either no local disk or a small one only for paging or temporary storage, allowing very compact compute node packaging. On the other hand, multiple high-performance disks can be attached to the I/O nodes. For example, this is done in the IBM SP2 computer, where nodes

are either *wide* (able to have many disks) or *thin* (able to have only a small number of disks).

If disks are attached directly to each compute node, the CPU cycles are divided between computation and serving I/O requests. This not only reduces the cycles available for computation, but also causes unpredictable asynchronous interrupts that could interfere with synchronization among the compute nodes of an application. Parallel applications run only as fast as their slowest processor between each application synchronization point. Therefore, if various compute nodes are required at unpredictable times to service foreign I/O requests, then the performance of applications as a whole can be greatly degraded [22]. Even applications that do not perform any I/O themselves are not immune to such interruptions. In general, it is difficult to guarantee that each application has its data locally (and therefore does not interfere with other applications) except by a static partitioning of the machine. Moving all disk I/O servicing to dedicated nodes ensures that the compute nodes are free to run without unexpected interference from I/O requests. Applications are then only affected by I/O when they make I/O requests themselves.

Finally, system software reliability remains a large issue in MPPs. If the same node doubles as both a compute node and an I/O node, the file system server is vulnerable to node failures caused by application software. If a user writes code that crashes a node, the entire parallel file system may be brought down, at least temporarily. If the user runs in a dedicated partition, on the other hand, the nodes in that partition can be restarted without bringing down the whole system. In particular, there will be no effect on other applications and on nodes like the I/O nodes which only run system code. This point will become insignificant as the system software for parallel computers matures.

Providing internal I/O through dedicated hardware can be viewed as a similar architectural choice to the provision of dedicated I/O coprocessors and channels in mainframe and supercomputers. Examples include the CDC-6600, the IBM 360 and 370, and all Cray computers. In these computers, I/O performance is greatly enhanced by offloading the computations required to perform I/O to the dedicated coprocessors, freeing the main processor to spend more time running user application code.

The fundamental choice between having dedicated I/O nodes vs. incorporating the I/O function into the compute nodes is one of performance vs. cost. The cost for providing hundreds of gigabytes of secondary storage can be very significant. Large installations such as national laboratories typically divide their computing budget into three roughly equal portions: for CPUs, for networking and secondary storage, and for tertiary storage. Adding additional I/O nodes to a computer increases the cost of the computer. It may be cheaper to simply add disks and memory to the compute nodes already present, as additional processors and network ports are not required. However, there will be performance degradations due to the effect on applications processes of asynchronous and unpredictable demands for I/O, particularly if more than one application is running.

The cost of dedicated I/O nodes should be compared with the cost of providing the same level of service via other means. Forgoing an interal I/O subsystem in favor of more or faster external servers is not likely a cost effective solution. In most cases, it turns out that the cost just moves out of the MPP into external servers. One compromise solution is to handle paging and scratch storage on a local disk, and to handle external I/O and I/O to shared or persistent files through a dedicated internal parallel I/O subsystem. This approach is used on the Meiko CS-2 and the IBM SP2.

I/O Node Configuration

In computers that provide a dedicated internal I/O subsystem, the question arises of exactly how to configure it. Following are a few guidelines that show how the configuration can be tailored to meet performance goals. This is not meant to imply that all installations actually follow such guidelines; in many cases, budgetary factors outweigh technical considerations, and the ratio of I/O and compute performance requirements varies widely among installations.

Number of disks per node

Adding disks to a system serves two purposes: it increases the total storage capacity, and it provides additional bandwidth by means of parallel access. The achievable bandwidth of an I/O node is primarily dependent on three parameters: the realizable media transfer bandwidth, the bandwidth of the channel(s) from disk to the network interface, whether directly transferred across the I/O bus or staged through the processor memory, and the bandwidth of the network interface. The goal is to balance these bandwidths, so that all components are fully utilized. If additional capacity is needed, more disks can always be added, even if their total bandwidth is higher than required. Attaching more disks to existing I/O nodes is less expensive than adding new I/O nodes.

Data transfers to disk cannot be sustained at the media transfer rate unless the accesses are purely sequential. The number of disks should therefore be such that their aggregate *average* bandwidth (rather than peak bandwidth) matches that of the I/O bus and network connection to the I/O node.

Let n_d = number of disks per I/O node

B_d = average disk bandwidth

B_c = internal channel bandwidth

B_n = bandwidth of network interface

B_{io} = realized bandwidth of the I/O node

Then the following relation holds:

$$B_{io} \leq \min\{B_d \cdot n_d, B_c, B_n\}$$

Given current technology trends (with network bandwidth higher than disk bandwidth, and also increasing more quickly), this implies multiple disks per I/O node. More than one I/O channel, e.g. multiple SCSI strings, may also be required. This also introduces the option of using a RAID disk array within each I/O node, increasing the effective reliability of the disks.

As an example of the application of this equation, the disk storage nodes in the CM-5 scalable disk array have eight disks each. These disks have a raw transfer rate of about 2MB/s for reads and 1.8MB/s for writes, leading to a total of 16MB/s and 14.4MB/s respectively. These values are slightly lower than the 20MB/s network bandwidth available to each node.

The above equation can be qualified by the expected efficiency of any prefetching and buffering mechanism that is used. We introduce an I/O efficiency factor, ε, that is the ratio of bytes moved over the I/O node network interface to bytes read or written from disk. The resulting equation is

$$B_{io} \leq \min\{\varepsilon \cdot B_d \cdot n_d, \varepsilon \cdot B_c, B_n\}$$

ε is increased if data read from disk is reread by more than one application process before it is evicted from the buffer cache, or if data that is overwritten several times is only written back to disk occasionally. ε is reduced if the block size used for accessing the disk is large compared to the amount of data moved over the network interface to satisfy requests. Similarly, if writes are small relative to the disk block size, and cannot be aggregated to cover complete blocks, then each block that is partially overwritten will be fetched from disk, updated, and then written back to disk. Storing redundant data such as parity can also reduce ε, in the worst case requiring 4 disk block accesses to modify one byte of data.

Number of I/O nodes

The number of I/O nodes should be chosen to balance the system, based on the capabilities of the compute nodes, the realizable bandwidth of the I/O nodes, and the desired F/b ratio that should be supported.

Let n_c = number of compute nodes

n_{io} = number of I/O nodes

F = flops of each compute node

R = desired F/b ratio

Then the following should hold:

$$\frac{n_c \cdot F}{n_{io} \cdot B_{io}} \leq R$$

which leads to

$$n_{io} \geq \frac{n_c \cdot F}{R \cdot B_{io}} \tag{1}$$

Note that $\frac{F}{B_n} \leq R$ must also hold, otherwise it is impossible to satisfy the desired F/b requirement. Also,

$$B_{io} \cdot n_{io} \leq B_n \cdot n_c$$

otherwise, the data produced by the I/O nodes could not be consumed by the compute nodes. Therefore, if $B_{io} = B_n$, which is the best it can be, then

$$n_{io} \leq n_c$$

The problem is deciding what F/b ratio should be supported. A commonly quoted rule of thumb from the mainframe era, attributed to Amdahl, suggests "one Mb/s for one MIPS" [11]. This is sometimes used to advocate a requirement for $F/b = 1$ for supercomputers executing floating-point applications as well. (Note that the bandwidths used to express F/b ratios are expressed in $Mbits/second$.) A recent survey of 8 parallel scientific applications found an average of $F/b \approx 100$ [7]. However, those applications that performed I/O throughout their whole execution (as opposed to just at the beginning and the end) averaged $F/b \approx 50$, and the most I/O-intensive application had $F/b \approx 14$. These numbers do not include the I/O required to load the applications in the first place, and there was no checkpointing. In addition, the four applications that were more I/O intensive were all programmed for the Touchstone Delta, which is known to have limited I/O capability [20]. Therefore the programmers were motivated to minimize their I/O requirements as much as possible, e.g. by recomputing data instead of storing intermediate results. In many cases, such redundant computations can be avoided if the I/O subsystem is fast enough.

A survey on I/O behavior of applications executed on a Cray Y-MP supercomputer yielded different results [21]. While this survey did not focus on F/b explicitly, it did provide application runtimes and the total amount of I/O they performed. Multiplying the runtime by 333 MFlops, the peak performance of a Cray Y-MP processor, leads to an upper bound on the number of floating-point operations executed. For 5 out of the 7 applications surveyed, this results in estimates of F/b in the range of 0.56 to 4.72. Needless to say, the real ratios are actually smaller, as applications never execute at peak performance. The I/O was mainly used for overlaying large data sets in the primary memory, in one case using the Cray's SSD (solid-state device) as the target of I/O. Admittedly, the survey focused on I/O intensive applications, but such applications do in fact exist. Thus the requirement for $F/b \approx 1$ is not overly

aggressive. Measurements from a VAX/Unix environment have shown that an even higher I/O rate might be required [1].

The lower bound on the number of I/O nodes (Eq. 1) can be modified by the inclusion of an I/O efficiency factor, ε_c, that is the ratio of bytes read or written by applications to bytes of I/O across the compute node network interface. ε_c is increased by effective caching of data on the compute node. It is decreased if the amount of data moved across the network to handle small or nonaligned requests is greater than the size of the requests, for example, if a full file block of data is moved to the compute node when a single byte of a file is read. The equation becomes:

$$n_{io} \geq \frac{n_c \cdot F}{\varepsilon_c \cdot R \cdot B_{io}}$$

Amount of memory per node

The amount of memory required on I/O nodes depends on the style of operations supported by the file system and on hardware features such as the ability to transfer data directly between the disk controllers and network adapters, and ranges from small transfer buffers to large caches.

One approach is to optimize the file system to handle extremely large accesses, with no reuse or sharing. The data is piped from the disk controller directly to the network, or streamed through small memory buffers for speed matching, minimizing overheads for buffer management and data copying. This approach is used in sfs on the CM-5, where I/O operations typically involve all the compute nodes, and the file system handles only one such request at a time. Data is striped in 16-byte units — which corresponds to the optimal packet size for the network — and an 8 MB buffer is used to marshal data movement. Intel's PFS on the Paragon also does not perform any caching on I/O nodes. Disk blocks are transferred directly to and from compute nodes, using the "fast path" feature of the underlying OSF/1 AD technology.

A common mode of operation on parallel supercomputers involves all the compute nodes working in parallel on disjoint parts of a large data set (typically in a loosely-synchronous SPMD style). The data set is read from and written to disk, perhaps many times. For example, physical simulations might repeatedly output the whole system state every few time steps, and large data sets require out-of-core computations that iteratively shuffle data between memory and disk. While such I/O operations access large amounts of data in the aggregate, the contribution of each PE can be quite small (or a set of distinct small parts). These small accesses are not necessarily synchronized. This implies strong inter-process locality and sharing of file blocks, and favors the use of a buffer cache on the I/O nodes [17]. Such an arrangement allows a single disk access to service fragmented requests from multiple compute nodes, and saves

the numerous disk accesses that would be required to serve them individually. It also allows disk scheduling to be optimized. This is done in a number of parallel file systems, including Intel CFS, nCUBE, and Vesta (see sidebar).

The size of the buffer cache depends on how long the data needs to be kept in it. Given that we wish to use the disks and network at their full bandwidth, the residence time for data is

$$T = \frac{M_{io}}{n_d \cdot B_d} \qquad \text{or} \qquad T = \frac{M_{io}}{B_{io}}$$

where M_{io} = memory on each I/O node.

T should be large enough to hide asynchrony among the compute nodes, which sets a lower bound on the amount of memory needed. If no synchronization guarantees can be made, data has to be kept for a long time to enable all participating compute nodes to perform their part of the access. This seems to imply that M_{io} tend to infinity. However, it should be noted that the total amount of data handled by all the the compute nodes together is limited by the primary memory available to the application. Therefore an upper bound on the amount of memory needed on I/O nodes is given by

$$n_c \cdot M_c \approx n_{io} \cdot M_{io}$$

where M_c = memory on compute nodes. In practice, this gives a greatly exagerated requirement for I/O node memory because it is rarely true that the entire data set for an application is actively being accessed at any one time. The aggregate behavior of the compute nodes will often exhibit some temporal locality of reference in their accesses to files. Combined with prefetching and interprocess locality, such temporal locality of reference will greatly reduce the average latency of I/O accesses even when the aggregate I/O node memory is much smaller than the aggregate compute node memory. It is also true that the compute node memory is used for more than just data that is either read from or written to the I/O subsystem.

Finally, we note that buffering can be done in any designated memory, not necessarily in the I/O nodes. For example, Cray supercomputers can be configured with SSDs that are up to 8 times the size of primary memory, and serve to effectively buffer data on the way to disk. Similarly, the nCUBE system allows a number of nodes to be designated as a RAM disk rather than being used as compute nodes. MasPar has a large IORAM that buffers between the processor array and the I/O devices.

Placement of I/O nodes

Given a large parallel machine in which nodes are physically housed in multiple racks or are otherwise spatially clustered, the question of I/O node placement, both physical and in the network, should be considered. Physical

placement is defined as the actual physical location where the I/O node is housed. Network placement is defined as the network port location of the I/O node. There are many issues to consider when determining the optimum physical and network placement of I/O nodes within the computer, including the network bandwidth, the switch cycle time, the communication protocols used, and the packaging technology. However, there are two main options to consider when determining both physical and network I/O node placement: disperse the I/O nodes throughout the system, or concentrate them all in one place (e.g. in a single rack for physical placement).

Since I/O messages are typically larger than other messages traversing a network, spreading the nodes also has the potential to reduce network contention relative to a node placement scheme that concentrates the I/O nodes in one place. This is because the larger messages are more likely to cause network congestion and clustering the nodes in one area of the network may compound this problem. If all compute nodes send I/O requests to I/O nodes that are all concentrated in the same area of the network, the resulting performance degradation may be prohibitive. Note that the opposite effect may also occur. Dispersing the I/O nodes may cause a mix of I/O and intercompute node message traffic on the network that degrades compute node performance.

I/O nodes are a shared resource, that might be accessed by multiple programs at the same time. The programs execute on disjoint partitions of compute nodes. Concentrating the I/O nodes in one place in the network then has the possible advantage of reducing interference between programs. Consider the case of a hierarchical switching network, such as a fat tree [18]. If each of several programs runs in its own network partition, then the communication pertaining to each program is localized within its network partition. Only I/O-related communication between the compute nodes and the I/O nodes uses the top levels of the switching network. This is the case in the CM-5.

For physical placement strategies, concentrating the I/O nodes in one place may also allow for more compact packaging of the disks, possibly in specialized cabinets. This approach has been used by some vendors to simplify power, cooling and packaging requirements. It is especially common in architectures where compute nodes do not have the option of local disks. An example is the Thinking Machines' DataVault, used with the CM-2 and CM-5.

Reliability

I/O operations deal with the persistent storage of data. But processors, communication links, and storage devices may sometimes fail. A recent study shows that a failure rate of one per day is likely to be reached with a thousand-node multiprocessor [13]. However, the failures can be masked by other, non-faulty components. The system should be designed so that failures do not cause data to be lost. Furthermore, failures should not even cause data to be unavailable

except for very short intervals. This is important for permanently resident applications such as parallel databases, time-critical programs such as weather forecasting, and also for checkpoint data of long-running applications.

Consider failures in compute nodes and the interconnection network. Such failures have to be handled in any MPP, regardless of its I/O architecture. Procedures for the handling of such failures are well established. For example, transmission errors can be detected using error correction codes such as CRC. Link failures can be overcome by re-routing to circumvent the failed link. Node failures are overcome by reconfiguring the logical view of the machine, removing the failed node. If spares are available, it is possible to reconfigure such that a spare node takes the place of the failed one.

Now consider disk failures. A well known solution is using redundant arrays of independent disks (RAID) [4]. These systems compute the parity of the data stored on several disks, and store the parity itself on another disk. Five schemes for doing so have been defined. The most popular are RAID 3 and 4, in which a set of disks store data and a single additional disk contains the parity of the data on all the other disks, and RAID 5, in which the parity information is distributed across all the disks. If only one device fails at a time, the data that was stored on it can be reconstructed from the data on the other disks and the parity. Thus high availability of the data is maintained. When the device is repaired, its data is reconstructed in order to regain the required redundancy. RAID also potentially increases bandwidth, due to the striping of data across multiple disks.

One way to extend the RAID concept to a parallel I/O system within a massively parallel computer is to calculate the parity across I/O nodes. For example, the CM-5 Scalable File System uses RAID 3 in software to generate parity across all the disks in a partition of the scalable disk array, with one parity disk and one spare disk. The RAID approach can protect against failures of one or more of the I/O nodes. However, it is expensive, both computationally, and in communication cost. All file writes result in an update of data stored in two different nodes, the data node and the parity node for the written data. Also, a protocol is required to commit the newly written data, establishing that it is written and parity protected. This is necessary to establish what data can be reconstructed at recovery time. This protocol may impose some additional overhead on the file system or on applications at runtime. Log structured file systems may reduce some of this write overhead, at the expense of increased overhead for reads and for restructuring files [24]. Logging can also be used to reduce or defer the overhead of parity updates [27].

An alternative is to have a number of disks at each I/O node, and calculate the parity only across those disks. Thus each node has a separate and independent RAID. While this approach does not protect against node failure, it does have several advantages as a means of protecting against disk failure.

- It reduces the load on the interconnection network by eliminating the messages sent between I/O nodes to

calculate the parity.

- The parity calculation and update can be delegated to the RAID controller, thereby relieving the load on the I/O node CPU.

- Finally, independent RAIDs at the I/O nodes contributes to the layered design of the system. File system functions are then built above disks that effectively never fail. This includes both special RAID controllers, and a software RAID implementation in the device driver.

- Using RAIDs implies multiple disks per I/O node, which matches bandwidth requirements as described above, and also reduces the number of network ports and disk controllers that are required to support a given number of disks.

RAID assemblies are becoming widely available for all computing platforms, using standard interfaces such as SCSI. As practically all commercially available MPPs provide I/O nodes with SCSI controllers, it is possible to populate them with RAID devices. Specific examples include the KSR1 system, which creates a RAID 3 of the five disks attached to each node, using four for data and one for parity. The Intel Paragon and IBM SP2 allow multiple RAIDs to be attached to each I/O node. The Meiko CS-2 uses dual-ported RAID subsystems with 5–20 drives on I/O nodes (as opposed to the single SCSI disks used for scratch space on compute nodes).

The main drawback of confining each RAID to the disks connected to a single I/O node is that it reduces the degree of protection against data loss. Calculating parity within the confines of an I/O node protects against disk failure, but does not protect against failure of the I/O node itself. Reconfiguration to replace the failed I/O node is not enough, because then access to the disks attached to the node would be lost. A possible solution is to use twin-tailed RAIDs, and attach each one to *two* I/O nodes: a main one and a backup. Each I/O node serves as the main node for one RAID, and as the backup for another. In addition, a spare I/O node is added (Fig. 4 (a)). If an I/O node fails, the system is reconfigured by switching a whole set of RAIDs to their backups, and removing the failed node. This node can now be repaired off-line. The interconnection network routing is changed accordingly, by changing the mapping of logical I/O node IDs to physical nodes (Fig. 4 (b)). The dual-ported RAIDs in the Meiko CS-2 are connected in this way, but they do not need to re-configure the network because the node mapping is done in software. The IBM SP2 also allows dual-ported RAIDS to be connected to multiple I/O nodes.

In total, this scheme needs only one standby spare node for the whole system. It provides total reliability and availability in the face of a single failure. Multiple disk failures can also be tolerated, as long as they occur in distinct I/O nodes. The down time during which data cannot be accessed when an I/O node fails is just the time required to

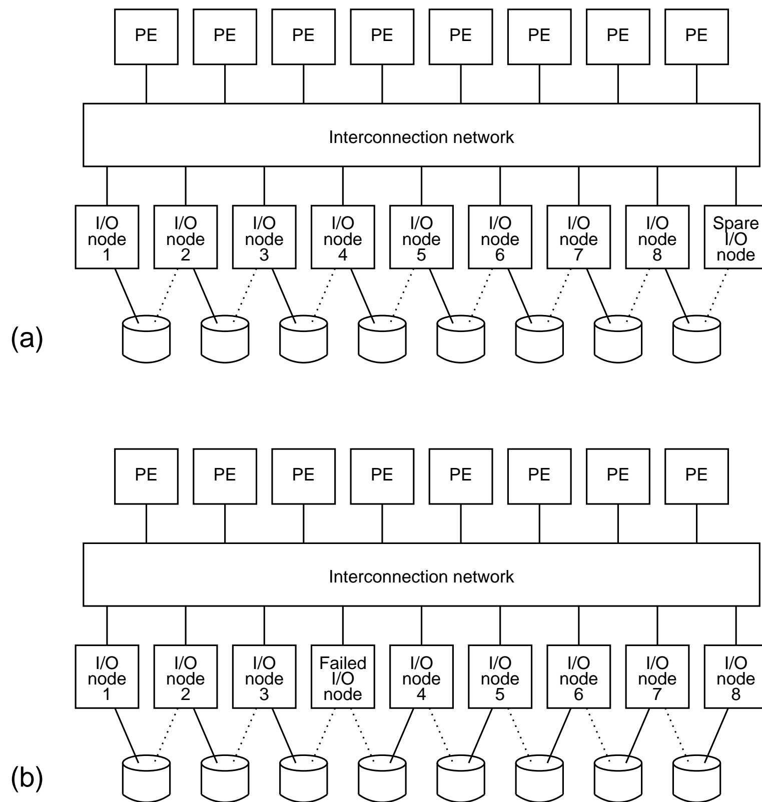

Figure 4: *(a) To provide protection from I/O node failure, every RAID assembly is attached to two I/O nodes: a main (solid line) and a backup (dotted line). (b) When a node fails, a subset of RAIDs is switched over to their backup nodes, and the network routing is changed accordingly.*

reconfigure the network. This is expected to be comparable to the time needed to deal with a compute node failure or a network failure. It is necessary that the file system be reliable across such faults and reconfigurations. This may require support from other operating system components, or the use of reliable hardware components such as nonvolatile memory.

File systems and Interfaces

A result of having an internal parallel I/O subsystem is that it suggests that new semantics and functionality be introduced in the file system, in order to support parallel programs. Currently, parallel file system interfaces is an area of some research and early product activity. No standards have as yet evolved, and there is some controversy about how best to proceed. In this section, we summarize some of the current work in this area [9].

Unix compatibility

Users need high level interfaces to deal with complex I/O subsystems. The most common high-level interfaces are language I/O libraries. In MPPs, these are typically implemented on top of Unix or Unix-like I/O system calls. There is a large amount of existing code for serial computers that requires these libraries or the Unix I/O system calls to be implemented. While these programs do not always run efficiently on MPPs, there are many users who require that this function be preserved while they convert to parallel applications.

At first blush it seems possible to simply use conventional networked Unix file systems, such as NFS or AFS. However, distributed file systems like these are designed to run on workstations connected by local area networks, not on MPPs. As a result, they are not adept at handling parallel applications, and often do not provide correct behavior when used by parallel applications. In NFS, for example, newly written or modified file pages are cached at the client node that performs the write, without immediately updating the master copy of the modified pages stored on the file server [26, 19]. There is no guarantee that the master copy of the file will be updated consistently. Therefore, if multiple client nodes write the same file — which is a normal mode of operation in parallel programs — the file data is likely to be incorrectly updated. AFS provides consistency at the session level, but this too does not solve the problem: the last process to close a file overwrites the updates of all the other processes [14, 19].

To support parallel applications, it is necessary to introduce parallel access functions and semantics into the system, something not provided by Unix. Typically, this is done at the library level, with the underlying system calls not exposed to the user. Most current commercial systems for MPPs provide conventional Unix semantics (i.e. a file is a linear sequence of bytes), and stripe the data transparently across multiple I/O nodes. In most cases, the MPP file system can be cross mounted with conventional Unix-like file systems. Hence direct access from other systems (including uniprocessors) is possible. Examples include Intel's CFS on the iPSC and PFS on the Paragon, the Scalable File System on the CM-5, the Meiko CS-2 parallel filesystem, PIOFS on the IBM SP2, and the KSR1 system.

In order to provide some extended parallel file access semantics without deviating too far from the familiar Unix and language I/O library interfaces, some systems allow files to be placed in certain *access modes*. Such modes define the interactions among I/O operations from different processors, and the way that the data is interleaved. For example, the Express programming system provides the following [25]. `singl` mode means that all the processors synchronize and take part in common I/O operations. For read, the data is read by one node and then broadcast to the other nodes. For write, all nodes must write exactly the same data and only one copy is actually written to the file. All the processors also synchronize for each I/O operation in `multi` mode, but here the data is interleaved according to the processor

IDs. Reads perform a scatter of data to the different compute nodes, while writes perform a gather of data from the compute nodes before writing it in one sequence to the file. A third mode, called `async`, provides uncoordinated access from the various compute nodes, which each have an independent file pointer.

Parallel access modes are not the only deviation from conventional Unix semantics. 64-bit offsets are introduced to support files larger than 2GB, requiring new interfaces or alternatives for system calls such as `seek`. Many systems introduce new system calls, or use the Unix `ioctl` and `fcntl` system calls to set various special parameters and to perform nonstandard functions such as asynchronous I/O, leading to a user interface that is not supported by other machines, and so is not portable.

Parallel I/O systems have an opportunity to match the data layout with the parallel access patterns. For example, it is possible to partition the file such that the data accessed by each process resides on a separate I/O node, leading to reduced communication. This option is lost if the application is restricted to using the Unix interface, because Unix does not provide the tools to describe data layout. This has prompted the design of systems with explicit support for parallel access. Such new designs are not portable. The best long-term solution is to develop a new standard interface for parallel I/O.

Parallel file access

A major problem with the serial semantics of file access occurs in the input or output of partitioned data structures. Many parallel applications use dense and large multi-dimensional arrays, and partition them among the participating compute nodes. This is the basis of the HPF data parallel programming language [12]. For example, a 1000×1000 matrix may be partitioned into 16 blocks of 250×250, with the first compute node working on the first block, the second node on the second block, and so forth. Each compute node stores its block in its local memory. If the whole matrix is then written to file in column-major order (the default for Fortran programs), only the four compute nodes with blocks at the left-hand edge of the matrix participate initially. When these four blocks have been written, the next four nodes pick up. In effect, the serial nature of the file forces the programmer to serialize the I/O operations.

It is possible to avoid the serialization by having each node access the part of the file that it needs, without undue interaction with the other nodes. This approach has been taken by nCUBE and Meiko. These systems allow a file to be partitioned in the same way as a data structure is partitioned. Then the whole data structure can be read or written in parallel. The problem with this approach is that the data may become fragmented into small contiguous portions being accessed by individual compute nodes, leading to many small accesses. Continuing with the example from the previous paragraph, each node would access its block by performing 250 I/O operations of 250 matrix elements each.

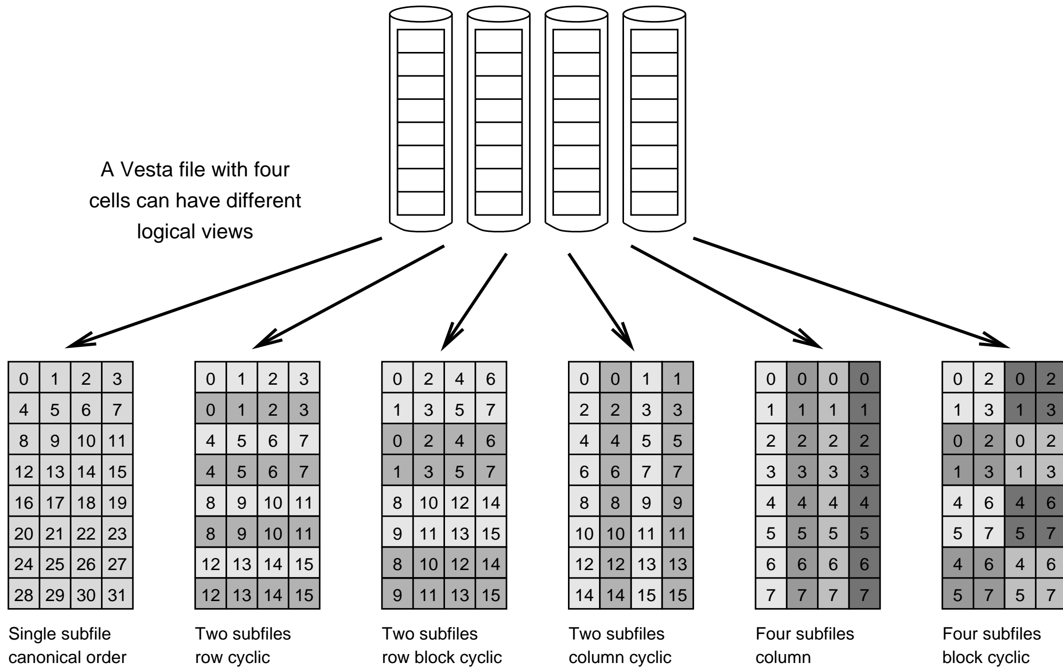

Figure 5: *Examples of logical partitioning in the Vesta file system.*

One solution is to use a two-phase strategy, which decouples the disk access from the partitioning [8]. For reads, the data is first read into memory in large sequential chunks to reduce disk-access overhead, and then message passing is used to redistribute the data as required. For writes, the order is reversed. While this prevents the fragmentation of disk accesses, it requires additional memory buffers at the expense of user memory, and also requires additional node-to-node transfers of data.

Another approach is to change the interface to allow users to exercise more direct control over layout of file data, so it can be matched to access patterns. This allows the use of the storage hierarchy to be integrated into the algorithmic design of applications, leading to better use of I/O capabilities [6, 28]. Especially important is the capability to perform independent I/O operations to the different parallel disks, rather than always doing synchronous parallel I/O operations to all the disks at once.

Tools for control over data layout and independent access are provided in the Vesta parallel file system. In Vesta, a file is not one linear sequence, but is a 2-D structure: a set of *cells*, each of which is a linear sequence, and is stored on a different I/O node. Thus the cells define the degree of parallelism in access to the file. The 2-D file can be *logically partitioned* into subfiles containing rows, columns, or blocks of the file, without actually moving any data (Fig. 5). Subfiles can be opened by different tasks, or they can be shared. In particular, it is possible to define

a sequential view of all the file data, leading to a single subfile that is striped across the I/O nodes and shared by all tasks. As another example, tasks running on different compute nodes can open subfiles corresponding to different cells (columns), resulting in independent access to different I/O nodes. Note that while partitioning is functionally similar to having multiple independent files, it is much more convenient: it allows operations on the whole file (e.g. create, checkpoint, or delete), and allows for many different views (logical partitionings) of the same file data. Vesta also provides a low level interface that allows arbitrary data partitioning and access, without the restrictions of rectilinear decomposition.

Partitioning files is similar to decomposing data structures, and as such is expected to be beneficial for distributed-memory machines. File partitioning is also useful in shared memory machines with non-uniform memory architectures. However, in shared memory machines where all memory is equally accessible, it is not as important to perform the I/O to or from a specific memory module. Therefore such machines have less need for new interfaces. Examples are the KSR1 and Tera machines.

Other interfaces

A major concern with MPPs is the difficulty of programming parallel applications, especially in message passing environments. This difficulty is exacerbated if functions such as asynchronous parallel I/O are used by the application. Therefore, there is some work underway to provide higher level I/O and file system interfaces, particularly in higher level language libraries, and in new high level languages. One possibility is language-level persistent data types. For example, persistent matrices could be introduced into HPF. Other possibilities include C++ libraries that provide persistent parallel objects and that perform operations on them. For example, a C++ class library that provides persistent matrix objects and operators that act on them could be used by application programmers, shielding them from the intricacies of the underlying implementation, and ensuring that the relatively hard task of programming to the parallel interfaces is not repeated by each programmer. As another example, the ELFS system allows the definition of parallel file objects, complete with class-specific access methods, caching, and prefetching [15]. Standardization at this level of interface could allow portability among machines from various manufacturers.

A second major category of interface is the interface for parallel data transfer between MPPs, and the related interface for parallel data transfer between external mass storage systems and MPPs. Some work is going on in this area to produce early drafts of standards, but it will likely be several years before any firm definitions are in place. Issues include what services should be provided, how rate negotiation is accomplished, what level of data fragmentation is supported, and how to coordinate the data moved on each individual channel with the overall data

transfer.

As parallel file systems and mass storage systems both mature, it is likely that the parallel file systems will be integrated as the top layer in the mass storage system hierarchy, and so migration of files or portions of files into and out of the MPP will be done automatically on demand by the mass storage system data movers. For example, Vesta includes some support for such import and export. The RAMA file system is specifically designed as an online cache for data that resides in tertiary storage. Eventually, it may be possible to have an integrated system that migrates multi-gigabyte files from tertiary storage into and out of the MPP's parallel file system on demand in an interactive computing environment. However, file system and storage technology is a long way from achieving this goal. An interim goal is to support offline file migration for batch submitted jobs requiring large amounts of file I/O, and transparent online migration of smaller files for smaller interactive parallel jobs.

Conclusions

An internal parallel I/O subsystem solves many of the I/O problems that occur in massively parallel processors. It provides scalable bandwidth and capacity that satisfy the requirements of applications, and serves as a staging area for data stored in external archival mass storage systems. This has been recognized by many vendors, e.g. Thinking Machines, Intel, Meiko, nCUBE, IBM, MasPar, KSR, and Tera, who all include a parallel I/O subsystem in the architecture of their parallel computers.

The internal parallel I/O subsystems require new parallel or concurrent filesystems. Most vendors try to achieve a high degree of Unix compatibility or appearance in these filesystems. This sometimes means that the parallel features of the system are hidden in special system calls added to the operating system, or above the file system interface in parallel I/O libraries. Moreover, user control over how the data is distributed across the I/O devices is limited. Current research directions include the definition and standardization of new interfaces that allow more efficient use of parallel I/O systems, development of parallel file systems, and integration of these systems with mass storage systems.

Acknowledgements

Thanks to Jim Cownie from Meiko for information about the Meiko CS-2 I/O architecture and its parallel filesystem. Thanks to Brad Rullman from Intel Supercomputer Systems Division for information about the Paragon and PFS. And thanks to Gérard Vichniac from KSR for information regarding the KSR1 I/O subsystem. Any inaccuracies in the

presentation are the responsibility of the authors.

References

[1] J. Akella and D. P. Siewiorek, "*Modeling and measurement of the impact of input/output on system performance*". In 18th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 390–399, May 1991.

[2] J. M. Anderson and M. S. Lam, "*Global optimizations for parallelism and locality on scalable parallel machines*". In *Proc. SIGPLAN Conf. Prog. Lang. Design & Implementation*, pp. 112–125, Jun 1993.

[3] S. J. Baylor, C. Benveniste, and Y. Hsu, "*Performance evaluation of a parallel I/O architecture*". In *Intl. Conf. Supercomputing*, Jul 1995.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "*RAID: high-performance, reliable secondary storage*". *ACM Comput. Surv.* **26(2)**, pp. 145–185, Jun 1994.

[5] S. S. Coleman and R. W. Watson, "*The emerging paradigm shift in storage system architectures*". *Proc. IEEE* **81(4)**, pp. 607–620, Apr 1993.

[6] T. H. Cormen and D. Kotz, "*Integrating theory and practice in parallel file systems*". In *Proc. DAGS Symp. Parallel I/O & Databases*, pp. 64–74, Jun 1993.

[7] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "*Architectural requirements of parallel scientific applications with explicit communication*". In 20th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 2–13, May 1993.

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "*Improved parallel I/O via a two-phase run-time access strategy*". In *Proc. IPPS '93 Workshop on I/O in Parallel Computer Systems*, pp. 56–70, Apr 1993. (Reprinted in *Comput. Arch. News* **21(5)**, pp. 31–38, Dec 1993).

[9] D. G. Feitelson, P. F. corbett, Y. Hsu, and J-P. Prost, "*Parallel I/O systems and interfaces for parallel computers*". In *Multiprocessor Systems — Design and Integration*, C-L. Wu (ed.), World Scientific, 1995.

[10] D. G. Feitelson, P. F. Corbett, and J-P. Prost, "*Performance of the Vesta parallel file system*". In 9th *Intl. Parallel Processing Symp.*, pp. 150–158, Apr 1995.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[12] High Performance Fortran Forum, "*High performance fortran language specification*". May 1993.

[13] R. Horst, "*Massively parallel systems you can trust*". In 39th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 236–241, Feb 1994.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "*Scale and performance in a distributed file system*". *ACM Trans. Comput. Syst.* **6(1)**, pp. 51–81, Feb 1988.

[15] J. F. Karpovich, A. S. Grimshaw, and J. C. French, "*ExtensibLe File Systems (ELFS): an object-oriented approach to high performance file I/O*". In 9th *Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc.*, pp. 191–204, Oct 1994.

[16] R. H. Katz, G. A. Gibson, and D. A. Patterson, "*Disk system architectures for high performance computing*". *Proc. IEEE* **77(12)**, pp. 1842–1858, Dec 1989.

[17] D. Kotz and N. Nieuwejaar, "*File-system workload on a scientific multiprocessor*". *IEEE Parallel & Distributed Technology* **3(1)**, pp. 51–60, Spring 1995.

[18] C. E. Leiserson, "*Fat-trees: universal networks for hardware-efficient supercomputing*". *IEEE Trans. Comput.* **C-34(10)**, pp. 892–901, Oct 1985.

[19] E. Levy and A. Silberschatz, "*Distributed file systems: concepts and examples*". *ACM Comput. Surv.* **22(4)**, pp. 321–374, Dec 1990.

[20] P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.

[21] E. L. Miller and R. H. Katz, "*Input/output behavior of supercomputing applications*". In *Supercomputing '91*, pp. 567–576, Nov 1991.

[22] R. Mraz, "*Reducing the variance of point-to-point transfers for parallel real-time programs*". *IEEE Parallel & Distributed Technology* **2(4)**, pp. 20–31, Winter 1994.

[23] IEEE Technical Committee on Mass Storage Systems and Technology, *Mass Storage System Reference Model: Version 4*. May 1990.

[24] M. Rosenblum and J. K. Ousterhout, "*The design and implementation of a log-structured file system*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 26–52, Feb 1992.

[25] J. Salmon, "*CUBIX: programming hypercubes without programming hosts*". In *Hypercube Multiprocessors 1987*, M. T. Heath (ed.), pp. 3–9, SIAM, 1987.

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "*Design and implementation of the Sun network filesystem*". In *Proc. Summer USENIX Technical Conf.*, pp. 119–130, Jun 1985.

[27] D. Stodolsky, M. Holland, W. V. Courtright, II, and G. A. Gibson, "*Parity-logging disk arrays*". *ACM Trans. Comput. Syst.* **12(3)**, pp. 206–235, Aug 1994.

[28] J. S. Vitter and E. A. M. Shriver, "*Optimal disk I/O with parallel block transfer*". In 22nd *Ann. Symp. Theory of Computing*, pp. 159–169, May 1990.