To appear (possibly revised) in
Multiprocessor Systems — Design and Integration
C.L. Wu (Ed.), World Scientific Publication Co., 1995

Parallel I/O Systems and Interfaces for Parallel Computers

Dror G. Feitelson, Peter F. Corbett, Yarsun Hsu, Jean-Pierre Prost
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

1 Introduction

Continued improvements in processor performance have exposed I/O subsystems as a significant bottleneck, which prevents applications from achieving full system utilization [33, 54]. This problem is exacerbated in massively parallel processors (MPPs), where multiple processors are used together. As a result, I/O subsystems have become the focus of much research, leading to the design of parallel I/O hardware and matching system software.

The requirement driving the work on I/O subsystems is the desire to achieve a balanced system [8]. The degree to which a system is balanced is typically expressed by the F/b ratio, which is defined as the ratio of the rate of executing floating point operations (F) to the rate of performing I/O, in bits per second (b). A widely accepted rule of thumb, attributed to Amdahl, calls for $F/b \approx 1$. While this was originally expressed in instructions rather than floating point operations, there is evidence that this requirement holds for computationally intensive numerical codes as well [20].

Given the high rate of increase in performance of processors, and the lower improvement rate of disks, $F/b \approx 1$ leads to the use of multiple disks in parallel. This has the advantage of being able to use multiple heads at once, increasing throughput, but introduces reliability problems. The common solution is to encode the data with some level of redundancy, so that if one disk fails the data can be reconstructed from the others [33]. The resulting organization is called a RAID, for Redundant Array of Independent Disks. The encoding typically involves calculating the parity of data striped across a set of disks, and storing the parity itself on another disk. This approach is now widely accepted in industry [9].

It should be noted that RAID defines how data is stored and protected, but not the interface for data access. Most systems use a conventional serial interface, where the whole RAID operates as a single device that just happens to have a larger capacity and higher bandwidth. In parallel systems such an interface is often limiting, because the data needs to be accessed in parallel by multiple processors. Therefore parallel

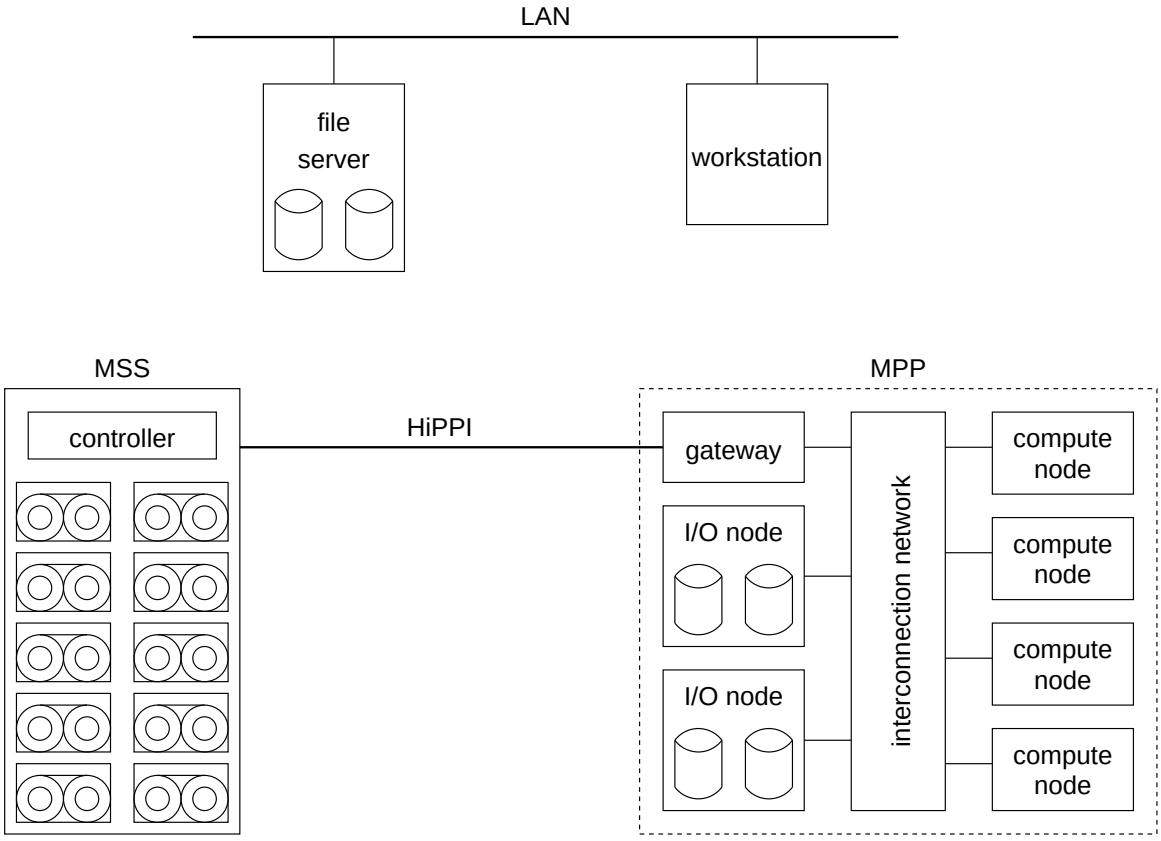


Figure 1: The I/O architectures of MPPs resembles that of LAN-connected workstations at two levels: first, there is internal I/O from compute nodes across the MPP's interconnection network to I/O nodes, and then there is I/O from the MPP as a whole across high-bandwidth channels to external mass storage systems.

interfaces have been developed for MPPs. The idea of redundant encoding is often used under the parallel interface, e.g. as a number of independent RAIDs that are accessed in parallel.

The general trend in recent years is toward network-connected I/O devices [32]. In a typical office environment, this means diskless workstations served by dedicated file servers. In an MPP, this structure can be seen at two levels (Fig. 1). First, there is internal I/O from compute nodes to I/O nodes. Compute nodes are the computation engine of the MPP, and are used to run parallel user applications. In terms of hardware, they often use the same components as workstations. I/O is then performed over the MPP's internal high-performance interconnection network. Such I/O operations are serviced by dedicated I/O nodes with disks. These nodes do not run user applications: rather, they are the MPP's internal dedicated I/O servers, just like the dedicated file servers found on LANs.

The second level is external I/O to mass storage systems (MSS) that are used

for archival storage. At this level, the MPP as a whole uses some high bandwidth link (typically HiPPI) to transfer data to and from the MSS. This is often mediated by special gateway nodes, that are specially configured to support the required high-bandwidth transfers. The MSS acts as a dedicated I/O server for the MPP, and possibly for other systems as well.

This chapter is about internal parallel I/O systems in MPPs. Section 2 deals with the architecture of such parallel I/O subsystems. Section 3 discusses the semantics of parallel I/O operations, and reviews the interfaces used to express different semantics. Section 4 is about implementation issues and their performance implications. Finally, section 5 presents the conclusions.

2 Parallel I/O Architectures

While some parallel machines are dedicated to a single application, most support multiprogramming. This means that multiple user jobs can execute at once, using space-slicing, time-slicing, or a combination of both [19]. In such an environment, I/O devices become a shared resource. Consequently it is undesirable to couple the I/O resources with any specific application. Rather, I/O devices should be independent and equally accessible by all. This approach has the added advantage that one job will not be perturbed by I/O operations of another job, as would be the case if the I/O devices were tightly coupled to the first job in some way.

Based on such arguments, most vendors of parallel machines elect to have dedicated I/O nodes act as an internal shared I/O server. These I/O nodes are used for storage of persistent data, i.e. data that is supposed to outlive any single instance of an application's execution. Examples include the Connection Machine CM-5 from Thinking Machines Corp. [65, 40], the nCUBE hypercube [26], the Intel iPSC hypercubes [56] and Paragon mesh, the Meiko Computing Surface CS-2, and the IBM Scalable POWERparallel system SP2. Even the MasPar SIMD array processor has an internal parallel I/O system. While this is based on a large dedicated memory buffer that interfaces the computational array to the disk arrays (rather than on I/O nodes), it is accessible in parallel via the router network [49]. The only major MPPs that do not have internal I/O nodes are the Cray T3D and Fujitsu VPP500. The Cray T3D currently uses a Cray Y-MP front-end to service I/O. However, it can have a number of I/O gateway nodes connected to the front-end, and plans call for the gateway nodes to connect directly to I/O controllers and through them to devices [34]. The Fujitsu VPP500 also uses a front end, which is also connected via a number of control processors [48].

In addition to persistent storage, there may be need for temporary storage used only during a single execution. Examples include swap space for virtual memory, or temporary storage for explicit overlays and out-of-core computations [64]. This space can be supplied as part of the shared space on the dedicated I/O nodes. Alternatively, additional I/O devices can be connected to the compute nodes, reducing the load and

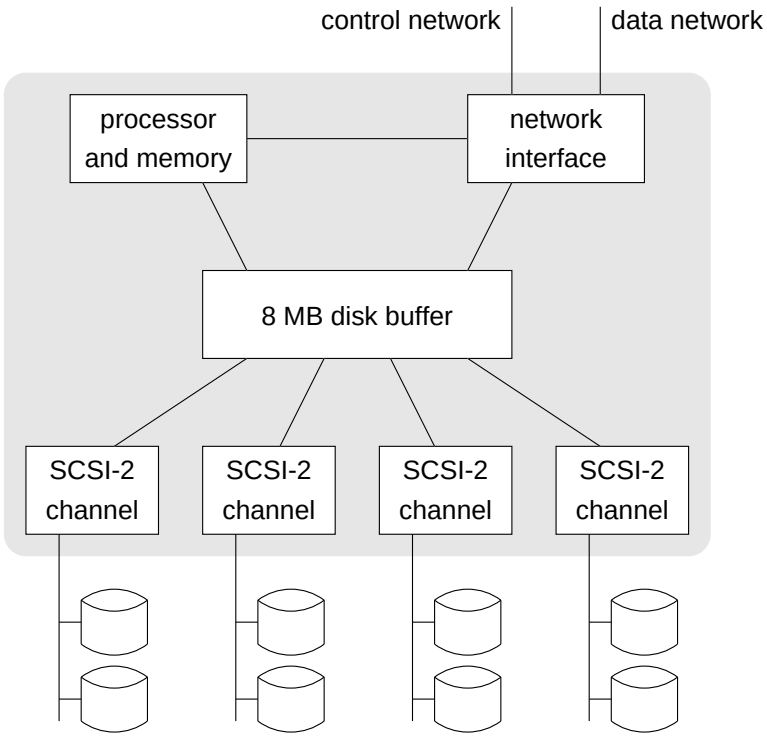


Figure 2: A CM-5 I/O node (called a Disk Storage Node in CM-5 terminology) [44].

congestion on the shared resource. Such additional I/O devices are available on the IBM SP2 and the Meiko CS-2. I/O devices can also be connected to all nodes of the KSR1 machine, which can double as both compute nodes and I/O nodes [22].

2.1 Example: the CM-5 Scalable Disk Array

The Connection Machine CM-5 is the last model out of Thinking Machines Corporation, and has been available since 1992 [65]. The compute nodes in this machine are based on SuperSPARC microprocessors with 4 optional vector units. The interconnection network has a fat-tree topology, implemented as a multi-stage network. The machine can be partitioned into partitions that correspond to sub-trees in the network. Each partition also has a control workstation.

I/O nodes — called "disk storage nodes" (DSN) in CM-5 terminology — are also SPARC-based. Each DSN has 8 disks, each with a sustained bandwidth of about 1.5 MB/s (Fig. 2). The aggregate bandwidth is therefore a close match to the 20 MB/s bandwidth provided by the data network. An 8 MB buffer is used to stream data between the disks and the network. DSNs are usually packed in groups of 3. The whole set of DSNs taken together form the I/O partition, which is called the "scalable disk array" (SDA). This is a partition of the machine just like other partitions, and also corresponds to a subtree of the network (see Fig. 3).

The system comes in a number of scales, which measure the number of stages in its

system size	*network ports*	*compute-heavy*			*balanced*			*I/O-heavy*		
		CN	*ION*	*F/b*	*CN*	*ION*	*F/b*	*CN*	*ION*	*F/b*
scale 3	64				32	2	29.8			
scale 4	256	128	6	37.9	64	9	12.6	32	12	4.7
scale 5	1024	512	12	78.6	256	24	19.4	128	48	4.7

Table I: Suggested configurations for the CM-5E. Figures are from Thinking Machines Corp. sales brochures. CN is the number of compute nodes, ION is the number of I/O nodes, and F/b is computed as the ratio of advertised peak flops to data transfer rate.

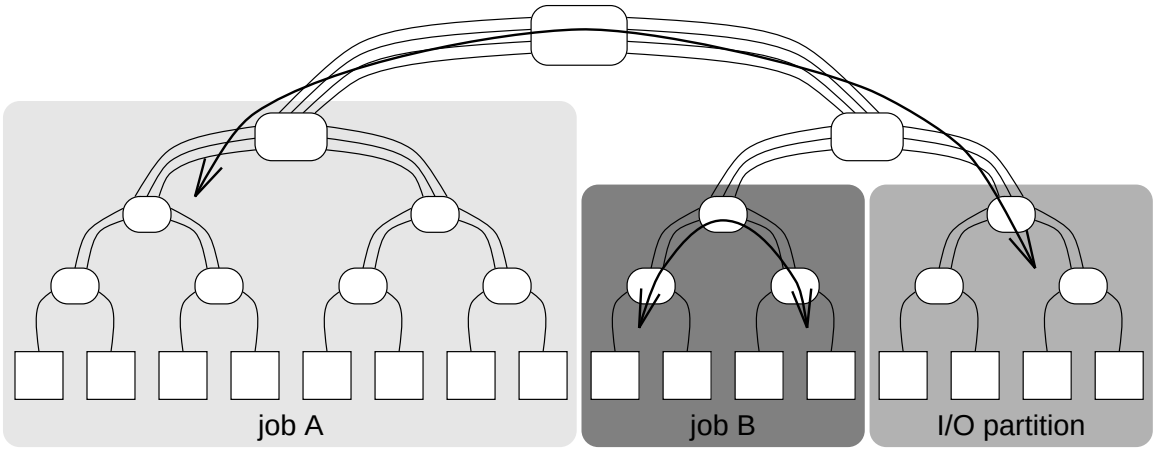
network. Each scale has a certain number of network ports. These ports can be used to connect compute nodes, I/O nodes, or other I/O devices (for example, a HiPPI gateway requires 8 network ports, so as to match the high bandwidth of the HiPPI channel). By using different numbers of compute nodes and I/O nodes, it is possible to create different configurations for compute-heavy or I/O-heavy installations. Specific configurations suggested by Thinking Machines are compared in Table I. Note that in all cases the provided F/b ratio is larger than 1.

2.2 I/O Node Placement

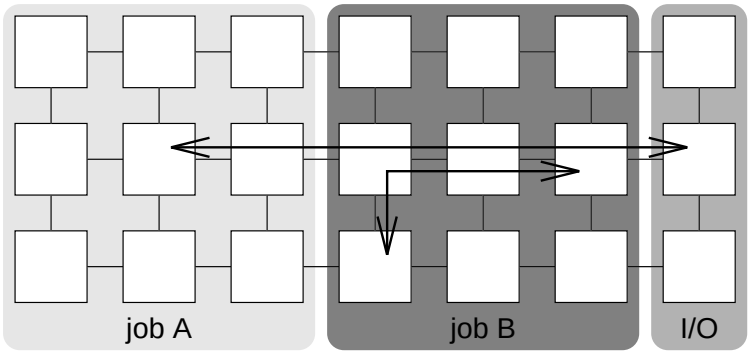
While using dedicated I/O nodes prevents I/O operations from directly influencing other jobs, the separation is not always complete. Obviously, if multiple jobs perform I/O operations at the same time, these operations will cause some conflicts at the shared I/O nodes. But there is also a danger of conflicts in the interconnection network. I/O is necessarily implemented by messages sent from the compute nodes to the I/O nodes and vice versa. These messages can interfere with other messages in the network, thus degrading application performance. Whether or not this happens depends on the network design (Fig. 3). For example, the CM-5 data network is designed so that each application executes in a separate partition of compute nodes, with a dedicated part of the network [40]. The I/O nodes also form a separate partition. In addition, interpartition traffic (such as I/O traffic from an application partition to the I/O partition) uses another part of the network, that does not belong to any partition. Therefore I/O traffic does not have any effect on jobs that are not performing I/O themselves.

The Intel Paragon design, on the other hand, does not have this feature. While applications still execute on partitions of compute nodes, messages can sometimes use links that are external to the partition. In addition, messages from one partition to another (such as I/O traffic) can pass through partitions that are in the way, using the same links that are used by the application running in that partition. This can cause noticeable congestion and degradation in communication performance for such applications [42].

Other systems, such as the IBM SP2 and Meiko CS-2, do not necessarily concentrate all the I/O nodes into an I/O partition. These machines are based on a



CM-5 design:
I/O traffic from job A does not interfere with job B



Intel Paragon design:
I/O traffic from job A
passes through job B

Figure 3: I/O from one job may cause interference with the communication of another job, depending on the network design.

multi-stage interconnection network. The network is not partitioned as it is in the CM-5, so message passing traffic and I/O traffic from different jobs can interfere with each other. As any node in the system can be designated as a compute node or an I/O node, there is an opportunity to decrease the interference with I/O by a judicious choice of network ports for the I/O nodes. However, it is not yet clear how to do so.

One approach that has been suggested is to spread the I/O devices evenly across the machine, to provide some degree of locality between compute nodes and I/O devices [57, 61]. However, this is questionable for two reasons. First, it imposes restrictions on the compute nodes used to run applications, because these must match I/O nodes that already contain persistent data to be used by the application. Second, given that interconnection networks in MPPs are much faster than disks, the correct model is that all the persistent storage is equally distant from all compute nodes [2]. The physical placement is actually immaterial.

However, the physical placement does affect network contention, and specifically, interference between message passing traffic belonging to applications and that belonging to I/O. If either of these patterns creates a significantly higher load than the other, concentrating the involved nodes in one place will cause contention and degraded performance [4]. It is then best to distribute the nodes throughout the machine, thereby also distributing the communication volume throughout the network.

On the other hand, distributing the I/O nodes throughout the system implies that they come between adjacent compute nodes. This could impair message passing performance within an application. For example, an IBM SP2 frame can contain up to 16 nodes. A 16-process parallel job can therefore be loaded onto a single frame, and use that frame's high-performance switch exclusively. But if some of the frame's nodes are I/O nodes, the 16-process job must span two frames, and use the inter-frame links. In some configurations (where frames are connected directly to each other, rather than using an additional switching stage), these have lower aggregate bandwidth and are therefore more susceptible to contention [63]. Such scenarios are avoided if I/O nodes are concentrated in a separate frame of their own.

2.3 Sustained vs. Peak Bandwidth

As with computation capabilities, I/O sometimes also exhibits a wide gap between advertised peak bandwidth and the bandwidth that is sustained in practice by applications. For example, applications running on the Touchstone Delta typically achieved only 4 MB/s aggregate bandwidth using CFS (Intel's Concurrent File System [56]), even though the system had 32 I/O nodes with a disk bandwidth of about 1 MB/s each [47].

Various factors can lead to performance problems with parallel I/O. In many cases, these are related to the fact that multiple processes are performing I/O operations at the same time, to the same files. Examples include:

- Network contention in accessing the I/O nodes. If there are many more compute nodes than I/O nodes, the I/O nodes may become network hot spots.
- Lack of buffer space, leading to thrashing [51]. This happens when the buffer caches on the I/O nodes are insufficient for the combined traffic from all the compute nodes.
- Inefficient disk scheduling due to interleaved requests for data at different offsets. This can be offset by appropriate software mechanisms. It is discussed at length in Section 4.

Some of the problems encountered in various systems can be attributed to evolving complex software systems, that never quite catch up with hardware developments. But in many cases it appears that the problem lies with system configuration. Many installations use the flexibility provided by parallel systems to beef-up the compute

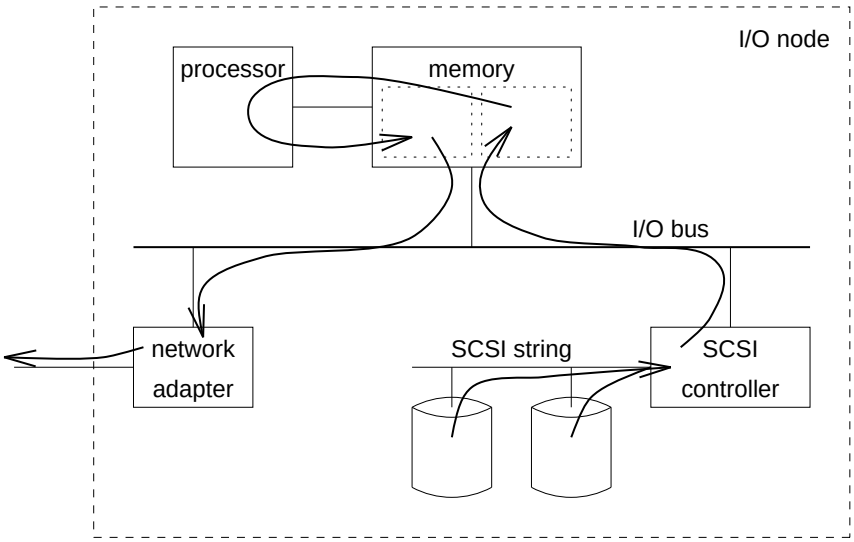


Figure 4: The bandwidth obtained from an I/O node depends on the component with the lowest bandwidth.

power and advertised peak Flops of their machine, and do not invest commensurate resources in I/O. As a result, the systems are unbalanced to begin with. The I/O nodes are underpowered relative to the compute nodes, and as a result they are overwhelmed by I/O requests and performance deteriorates. Thus it is doubly important to install sufficient I/O capabilities in parallel systems.

It should also be noted that in calculating the peak bandwidth, many components need to be taken into account. The component with the lowest bandwidth is the bottleneck that limits the bandwidth of the whole system. An example is given in Fig. 4. Data is read from two disks attached to the same SCSI string, and stored in memory using DMA from the SCSI controller. The processor then performs a memory-to-memory copy, e.g. to pack the data for transfer to the requesting compute node. Then the network adapter reads the data from the memory, again using DMA, and injects it into the network.

In many systems, the disks themselves are the slowest component, so the peak bandwidth is simply the sum of the bandwidths of the disks. But if multiple disks are used, their aggregate bandwidth can be large enough to saturate some other component in the transfer. For example, the SCSI bandwidth limits the number of disks that can profitably be connected to it. Likewise, the SCSI controller cannot handle an unlimited number of strings or disks. The I/O bus can be a limiting factor, especially considering that the data must traverse it twice: first in the DMA from the SCSI controller to the memory, and then in the DMA from the memory to the network adapter. Therefore the I/O bus bandwidth must be at least twice the peak disk bandwidth. The memory system may also be a bottleneck. To sustain peak performance, it must be able to support concurrent DMAs and processor access,

to allow copying from one buffer to another. Finally, the network adapter and the network itself must have adequate bandwidths.

2.4 RAID Configurations

While not much data is available on disk lifetimes, it is reasonable to model them as exponential [24, 25]. This means that the probability that a disk's lifetime is longer than t is given by $e^{-t/MTTF}$, where $MTTF$ is the mean time to failure, or in other words, the mean lifetime of such disks. To give some intuition about this expression, we note that the probability that a disk survives for half of the MTTF is 0.61, for exactly the MTTF is 0.37, and for twice the MTTF is 0.14. A well-known characteristic of the exponential distribution is that it is memoryless, so this model implies that disks have a constant failure rate. As a result, the MTTF of an array of disks is inversely proportional to the number of disks[1]. For a single disk it has been estimated that the MTTF is in the ballpark of 50,000 hours, or 5.7 years. For 100 disks, the MTTF drops to 500 hours, or about 3 weeks, which is unacceptable.

The solution to the problem of short MTTF in disk arrays is to encode data redundantly [33, 24, 9]. The simplest form of redundant data encoding is mirroring, that is keeping two copies of all data on separate disks. The obvious drawback of this solution is that half of the space is wasted. A more efficient encoding is obtained by computing the parity of the data, and storing it on an additional disk. If the original data was stored on d disks, the space overhead is reduced from $\frac{1}{2}$ to $\frac{1}{d+1}$. Luckily disk failures are self-identifying, so the parity information is sufficient for reconstructing lost data (as opposed to the situation in memory or data transmission, where simple parity can identify the existence of an error but not its location, and therefore cannot correct it). The reconstruction is done by computing the parity of the surviving data and the original parity information. Such schemes form the basis of RAID systems [33, 24, 9].

The price of parity protection is increased overheads. When new data is written, its parity has to be computed and stored. When a disk fails, all the other disks have to be read in order to reconstruct its data. Most of the research on RAID involves the reduction of this overhead and its even distribution among the different disks [39, 45, 28, 62, 60]. For example, the RAID 5 scheme improves on the RAID 3 and 4 schemes by distributing the parity information among all the disks, thereby avoiding a bottleneck for parity updates (Fig. 5).

In parallel systems, RAID is used in two main ways: hardware RAID boxes attached to I/O nodes, or software implementations in a parallel file system. An example of the RAID box approach is the IBM SP2 [29]. The server nodes in this machine, which are called "wide" nodes, support a large variety of external connections and peripherals. One of these is the IBM 7135 RAID device, which is accessible via a simple SCSI interface. This device can operate in RAID 1 mode (mirroring), RAID

[1]Strictly speaking, this is based on the assumption that failures of different disks are independent.

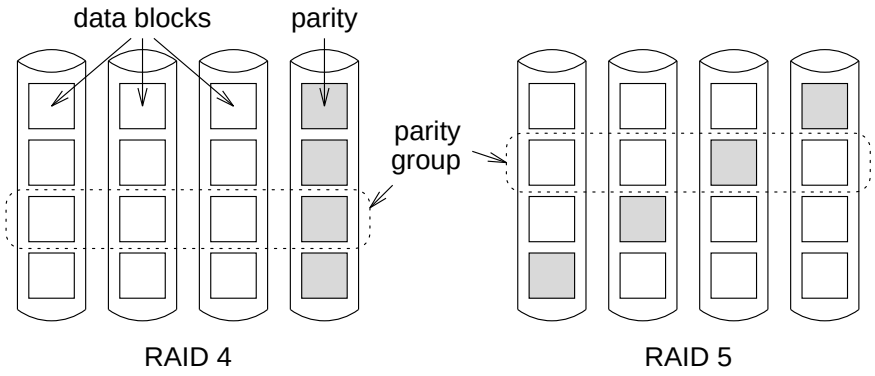


Figure 5: Examples of RAID 4 and RAID 5 organizations. RAID 3 is similar to RAID 4, except for striping by single bytes rather than full blocks.

3 mode, or RAID 5 mode. The disks are arranged in banks of 5 disks each. In the RAID 3 and 5 modes, one of these is used for parity, so the overhead is 20%. For additional data protection, the 7135 contains two redundant controllers, and can be configured in a twin-tailed scheme connected to two distinct wide nodes. This ensures data availability even in the face of failures in support components such as the controller, SCSI channel, or node.

An example of the software approach is the scalable file system (sfs) on the CM-5 [44]. sfs stripes data across the disks of the SDA in units of 16 bytes (which matches the CM-5 data network packet size). One disk in the whole SDA is used for parity, and one as a hot spare. Thus the system creates a RAID 3 configuration in software, spanning multiple DSNs and multiple disks in each one. Computation of parity and reconstruction of data are done by the system software, based on conventional SCSI disks with no hardware protection.

3 Semantics of Parallel I/O Operations

In conventional (Unix) systems, files are nearly never shared at the same time by more than one process (at least not for writing) [3]. In parallel systems such sharing is the norm, including extensive sharing at the block level [38]. Thus there is an urgent need to define what happens when multiple processes open and access the same file. For example, if p processes write "hello world!" to a file, what should happen? Options are that p copies be written, that only a single copy be written, or that the multiple copies be interleaved in some way character by character. All these options are justifiable and may be useful under certain circumstances. Therefore we need a mechanism to allow the programmer to specify which one is desired.

The mechanism used by most commercial systems so far is to place each open file in a certain *mode*. File modes are an addition to the normal Unix-like file system interface. When a file is in one of the parallel modes, `read` and `write` operations in the application processes become synchronization points. When such operations

mode	*description*	*examples*
broadcast reduce	all processes collectively access the same data	Express singl PFS global mode CMMD sync-broadcast
scatter gather	all processes collectively access a sequence of data blocks, in rank order	Express multi CFS modes 2 and 3 PFS sync and record modes CMMD sync-sequential
shared offset	processes operate independently but share a common file pointer	CFS mode 1 PFS log mode
independent	allows programmer complete freedom	Express async CFS mode 0 PFS Unix mode CMMD local and independent

Table II: File modes used in various parallel I/O systems.

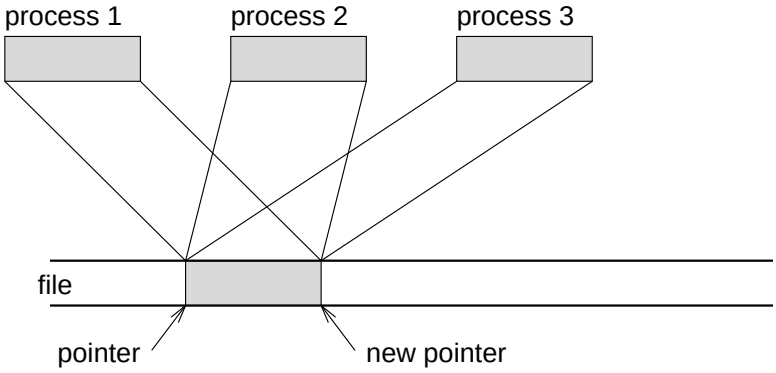


Figure 6: Data access pattern for the broadcast/reduce file mode.

are issued, all the processes synchronize, and perform a collective I/O operation to the file. This allows for the definition of crisp semantics for I/O operations that are performed in parallel by multiple processes to the same file.

3.1 File Modes

The most common modes and the systems that use them are summarized in Table II. We first describe the different modes in detail, and then review systems that provide different selections of such modes.

In the *broadcast/reduce* mode all processes access the same data (Fig. 6). If the access is a `read`, the same data is broadcast to all the processes. This is useful for reading headers with information that is needed by all processes, or the whole input if it is parameterized and does not need to be partitioned among the processes. When

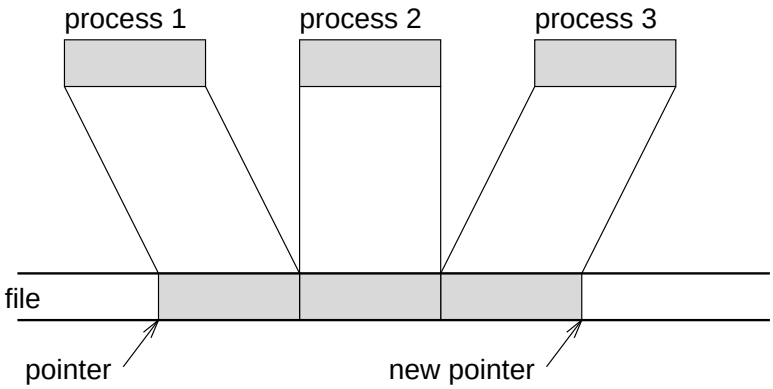


Figure 7: Data access pattern for the scatter/gather file mode.

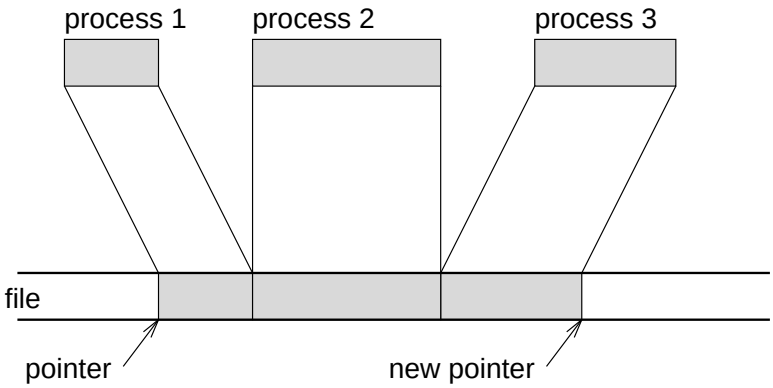


Figure 8: Data access pattern for the scatter/gather file mode, with variable block sizes.

writing, this mode causes only one copy of the data to be written to the file. This is useful for writing results that represent the whole computation. The written data can either come from a selected process, or from an arbitrary one. Some systems also provide the service of checking whether all processes write identical data or not.

In the *scatter/gather* mode, processes access contiguous chunks of data according to their serial numbers. Two variants of this mode have been suggested: either all chunks are of the same size (Fig. 7), or they can be different (Fig. 8). This mode is useful for partitioning data among the processes in simple patterns, both for reading and for writing. More complicated partitioning patterns are considered below.

With a *shared pointer*, I/O operations are not collective. Thus this is not a barrier synchronization point, but rather a mutual exclusion synchronization to access and update the shared pointer. When the file is in this mode processes also access contiguous chunks of data, but the order is not predefined. Rather, the order is determined on the fly by the order in which the processes perform the I/O operations. The number of operations from different processes can be different. This mode is useful for writing a log or for self scheduled reading and processing. Fig. 9 shows an

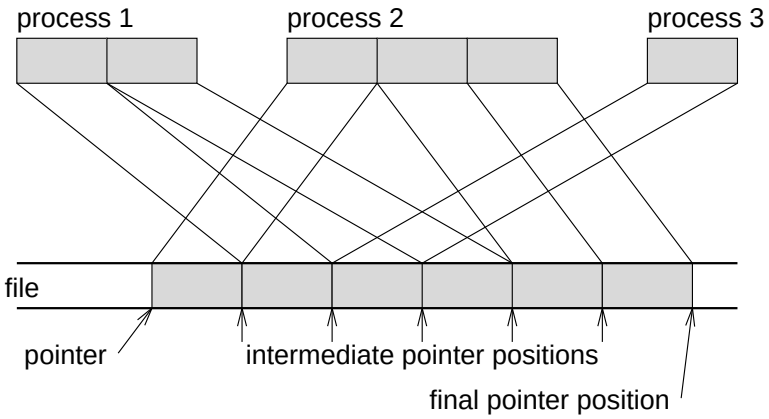


Figure 9: Data access pattern for the shared pointer file mode.

example where the order of accesses happens to be process 2 first, then process 1, then 3, 1, 2, and 2.

All systems also provide a mode with unrestricted *independent* access. This can be used in case the other modes do not match the programmer's requirements. In this mode each process has its own private file pointer, and it can `seek` and perform I/O operations anywhere in the file. There are no implied interactions with other processes. The burden of ensuring that the resulting pattern makes sense is left to the programmer.

Various systems provide different combinations of the above modes, sometimes with additional variants designed to improve efficiency. File modes were originally introduced as part of the Cubix environment for programming hypercubes at Caltech [59]. This system was later commercialized as the ParaSoft Express environment [53], and includes a library of message passing functions usable in SPMD or MIMD programs. The parallel tasks in Express have access to files using the system calls of the base system. To define the semantics of parallel access by multiple tasks, files can be placed in one of three access modes: `singl` means that all processes synchronize and take part in common I/O operations, with only one copy of the data in the file itself; `multi` means that all processes synchronize and their data is interleaved according to the processor IDs; and `async` grants uncoordinated access by the different processes.

Intel CFS is a commercial file system used on Intel iPSC machines, the Touchstone Delta, and the Paragon [56, 30]. File data is striped across multiple disks in 4 KB blocks. Four access modes are provided: `mode 0` provides no coordination, and all accesses are independent; `mode 1` provides a shared seek pointer, which is useful for things like writing a log file asynchronously; `mode 2` requires synchronous access, and interleaves the data according to the process IDs, and `mode 3` is the same as `mode 2` with the additional requirement that all accesses be of the same size. The reason for `mode 3` is that scatter/gather with fixed sizes can be implemented more efficiently

than with variable sizes, because the offset accessed by each process can be computed directly. If variable sizes are allowed, as in `mode 2`, the implementation requires a prefix computation to find the offset accessed by each process.

Intel PFS is a new implementation targeted for the Paragon, and providing essentially the same interface [31]. Improvements include control over the striping parameters (at the file system level, not for each file) and new access modes. Modes `Unix`, `log`, and `sync` correspond to CFS modes `0`, `1`, and `2`, respectively. Mode `record` is similar to CFS mode `3`, but uses the fact that access sizes are known in advance to allow asynchronous access. Finally, `global` mode provides synchronized access with only one copy of the data in the file (like Express `singl`). PFS is mountable in the system-wide directory hierarchy, and is compatible with other types of file systems. Thus it can be used to stripe data across the Paragon's I/O nodes, but also to stripe across NFS mount-points in a cluster environment.

Thinking Machines sfs (scalable file system) is a Unix-compatible file system for the CM-5's scalable disk array [44]. Data is interleaved in 16-byte units to create a RAID-3 configuration in software. CMMD is a library layered on top of whatever file systems exist on the compute partition's control processor, including sfs [5]. Normal Unix I/O is supported, with four file access modes: `local`, where accesses from different processes are completely decoupled from each other; `independent`, which is logically like local, but the processes share all the file-descriptor state except for the seek pointer, in order to reduce the load on the servers; `synchronous sequential`, in which accesses are interleaved according to the process IDs, and `synchronous broadcast`, which is logically equivalent to one process doing the I/O for all of them.

The MasPar I/O system provides a special version of the scatter/gather mode that is suitable for SIMD computations. The main difference from the other systems is that the participation of each processor is qualified by an "enabled" bit [49]. Processors that are not enabled for the I/O operation (the enabled bit is 0) do not participate. Parallel read operations distribute data only to the enabled processors, and parallel write operations collect data only from the enabled processors. These operations have two versions. In one the enabled processors access successive data elements in rank order, and in the other they can each specify an arbitrary offset.

3.2 File Partitioning via Scatter/Gather

An important function of file modes is that they define what part of the data is accessed by what process. In the broadcast/reduce mode, each process accesses all the data. In the shared offset mode, the data accessed is determined by the order in which the different processes perform I/O operations. In the scatter/gather mode, the data is partitioned according to the serial numbers of the processes.

Partitioning file data among the processes is a very useful feature. In many applications, the input dataset is partitioned among the processes, and then each process operates on its part of the data. For example, this is typical when the data represents a physical domain, and the parallelization is done by domain decomposition.

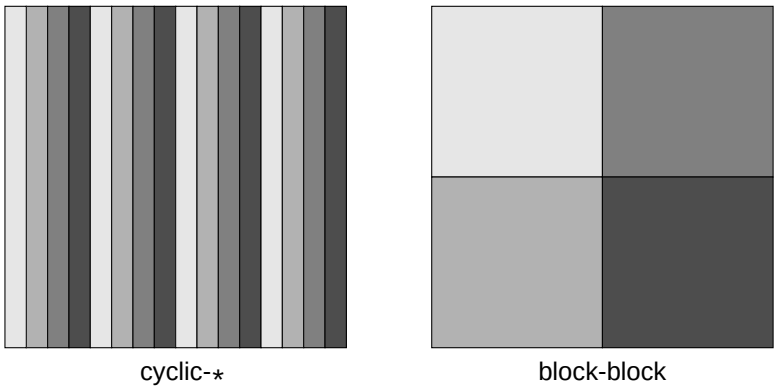


Figure 10: Two example patterns of partitioning a 2-D matrix.

Consider a weather code operating on a 3-D grid [1, sect. 2.1.1]. The input to such computations typically includes atmospheric measurements for all the grid points. As the grid is partitioned across the processes, so is the input. Likewise, the output is often a concatenation of data produced by different processes. For example, the output of CFD calculations is typically a pressure surface across the physical domain, with each part being contributed by the process that owns that part of the domain.

The scatter/gather file mode can express some data decompositions, but not others. To keep things simple, we shall use a 2-D matrix as a running example. Assume the matrix is stored in column-major order in a file. Each element of the file is of type `double`. The number of processes (and processors) is p, and the matrix size is $n \times n$, where n is a multiple of p.

Partitioning the file in a column-cyclic manner means that the first process accesses the first column, the second process accesses the second column, and so on. After p columns, the pattern is repeated. In general, process i accesses all those columns whose numbers are equal to i modulo p (Fig. 10 left). This can be expressed using the scatter/gather file mode by accessing a full column each time. At each access, all processes synchronize, and then the first process accesses the first chunk of data, that happens to correspond exactly to a single column, the second accesses the second chunk of data, and so on. Thus the first set of one access from each process covers the first p columns. The next set of accesses covers the next p columns, and this continues until the whole matrix is finished.

Practically all other common access patterns cannot be expressed via the scatter/gather file mode. We use the block-block decomposition as an example. In this scheme, the matrix is partitioned into p square blocks, which are assigned to the different processes (Fig. 10 right). This implies that only $\sqrt{p}$ of the processes have data in the first $n/\sqrt{p}$ columns of the matrix[2]. To access the first column, these $\sqrt{p}$ processes can each access $n/\sqrt{p}$ data elements, while the other processes access 0

[2] We assume that p is a square.

system	*advantages*	*disadvantages*
nCUBE [16]	simple partitioning based on bit permutations	all sizes must be powers of 2
array partitioning library [7, 6, 23]	supports common array partitioning patterns, high level of abstraction	must access full array in one operation
nested strided [50]	supports the common multidimensional access patterns	user needs to compute offsets and strides in all dimensions, must access a full multidimensional structure
Vesta [13]	supports all rectilinear decompositions, simple parameterized interface, express mapping to hardware	only 2-D partitioning, one dimension related to hardware more than to data
MPI-IO [10]	provides highest level of expressiveness, partitions can overlap, can also distribute data noncontiguously into memory	user has to construct MPI derived datatypes to describe the partitioning

Table III: Comparison of proposed interfaces for expressing file partitioning.

elements each. Note that the other processes have to participate (even though they are not accessing any data) because the scatter/gather file mode implies that all I/O operations are collective. This is then repeated $n/\sqrt{p}$ times. After the first $n/\sqrt{p}$ columns are accessed, the next set of $\sqrt{p}$ processes takes over. The net effect is that the access is serialized and involves a lot of redundant synchronization.

The problems in implementing the desired access patterns based on a scatter/gather file mode has led most programmers to use the independent mode instead [38]. In this mode, each process can seek to a different offset in the file, and access the desired data irrespective of what the other processes are doing. For example, each set of $\sqrt{p}$ processes can seek ahead to their part of the matrix in the block-block decomposition, rather than participating in irrelevant I/O operations with other processes. This eliminates the redundant synchronization, at the expense of a heavier burden on programmers. However, it may also cause inefficiencies in disk access, as explained in Section 4. To eliminate these problems, as well as to provide convenient target interfaces for parallel compilers, such as High Performance Fortran (HPF) [43, 27], new interfaces have been proposed recently. These are summarized in Table III and described in the following subsections.

3.3 The nCUBE Partitioning Scheme

An alternative to using the scatter/gather file mode is to define an interface that allows partitioning to be expressed directly. This approach has been taken in the

Vesta parallel file system, in the nCUBE system, and in a few libraries. We start with the nCUBE system.

The nCUBE design is based on the notion of address bit permutations [16, 17]. The bits of the address of each data byte are permuted and divided into two groups. One of the resulting sets of bits gives the ID of the process that will access this byte. The other set gives the offset into that process's buffer. Given that any subset of address bits can be used to generate the process ID, this provides for flexible partitioning patterns.

While conceptually elegant, the nCUBE scheme suffers from one major deficiency: all sizes must be powers of two. This includes the array dimensions, the partition dimensions, and the number of partitions. It is a direct result of using bit positions to define the partitioning. As a result, this scheme has not gained wide acceptance.

3.4 Partitioning Induced by Array Decomposition

While there are a number of competing ideas about how to partition file data (as witnessed by the subsections in this section), there is relative agreement about how to partition multidimensional arrays. This agreement is captured in the partitioning directives of HPF [43, 27]. Essentially, this is done by a list of directives, one for each dimension of the array. There are 3 options: `BLOCK` divides the array into equal size blocks and assigns them to successive processes, `CYCLIC` assigns successive array elements to processes in round-robin manner, and * means that this dimension should not be distributed. The example in Fig. 10 uses this terminology.

Given that files are often used to store array data, the same partitioning scheme can be used. In effect, the distribution of the array data among the processes induces a partitioning of the file segment that stores the array. This has been suggested in a number of libraries, especially in the context of providing I/O for HPF [7, 6, 23]. Naturally, it allows all the common partitioning patterns to be expressed.

The interface supported by these libraries is a high-level interface suitable for direct use by programmers, and using the same abstraction (i.e. partitioned arrays). An analogous low-level interface has also been proposed recently. It is based on viewing the array data as it is in the file, namely a sequence in some canonical order. Access to a subarray is then expressed as a set of nested strided accesses [50]. This interface requires its user (a programmer or a compiler) to determine the offsets and strides that should be used by the different processes.

3.5 The Vesta Partitioning Scheme

All the schemes described so far are based on partitioning the data based on a logical structure as perceived by the program. The Vesta parallel file system from IBM Research partitions the data as it is laid out on disks instead [11, 13]. This is done in two steps. First, structural parameters are defined when the file is created, and used to map the file to I/O nodes. Then, partitioning parameters are defined when the file is opened. These are expressed in terms of the structural parameters.

The structural parameters are the *basic striping unit* (BSU) and the number of *cells*. The choice of each number is arbitrary up to system limits. The number of cells specifies the maximum parallelism of the file in terms of I/O nodes used. Each cell looks much like a Unix file, i.e. it is a byte addressable one dimensional array of data with a defined end point. The cells of the file are maximally distributed in a round robin fashion among all the I/O nodes, beginning with a randomly chosen I/O node. Thus, if the number of cells is chosen to equal the number of I/O nodes, one cell will be placed on each I/O node, and the maximum level of parallel access is achieved.

The BSU size is the atom of data that is used when defining the striping across cells within the file. It is also used as the basis for the definition of multiple parameterized decompositions of the file into disjoint subfiles. The bytes within a BSU will always be part of the same subfile, but different BSUs of data can be grouped into different subfiles in a large number of ways, both within cells and across cells. The number of cells of a file and the size of its BSU are fixed at creation time for the life of the file.

Vesta files are explicitly two dimensional, and this concept of two dimensional data is preserved at the Vesta user interface. We consider the dimension across cells to be horizontal, and the dimension within cells to be vertical. At the user interface, access is made to subfiles of a file, not to the file itself, or to its cells. To access file data, a process first opens a subfile. The subfile is specified by five parameters of the open call. Four of these specify a partitioning of the file: Vbs specifies the number of contiguous BSUs to be grouped into a subfile from within each cell. Vn specifies the number of subfiles to be interleaved within each cell. Hbs specifies the number of adjacent cells from which BSUs with the same relative position in each cell will belong to the same subfile. Hn specifies the number of subfiles to interleave across the cells of the file.

The product $Vbs \times Hbs$ defines a basic block of data mapped onto the two dimensional array of cells and BSUs within cells. This block is repeated over the file Vn times vertically and Hn times horizontally to define a template of data decomposition into different subfiles (see Fig. 11). This template is repeated as many times as necessary horizontally over the cells of the file, and vertically until all cells have been exhausted of data. Each repetition of the template contributes one block of data to each of the subfiles, except in edge cases where the template extends beyond the boundaries of the file.

The fifth parameter of the open call specifies which subfile is to be accessed by the task, given the partitioning specified by the other four parameters. Subfiles are numbered from 0 to $Hn \times Vn - 1$ in row major order of blocks within the template. In most applications, all processes open a file with the same partitioning parameters, but each specifying a different subfile to access. The program can then proceed with each process issuing the same file I/O calls, but working against its own disjoint portion of the file. Thus it is guaranteed that the accesses are non-conflicting.

Returning to the 2-D matrix example, it is easy to see that Vesta supports all the common rectilinear decompositions: data can be accessed in rows of BSUs, columns of

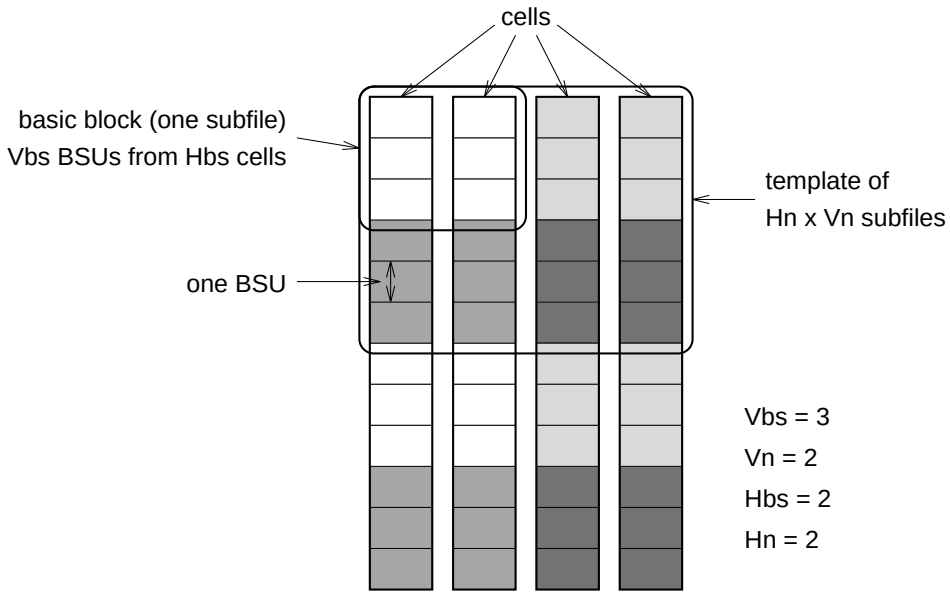


Figure 11: Example of Vesta file structure and partitioning. Subfiles are identified by different shades of gray.

BSUs, or blocks of BSUs. These are the same decompositions as those supported by HPF [43]. The specific example used above is obtained by the parameter values $Vbs = n/\sqrt{p}$, $Vn = \sqrt{p}$, $Hbs = n/\sqrt{p}$, and $Hn = \sqrt{p}$. Each process can then access its block with no required coordination or synchronization with other processes. However, this requires that cells correspond to columns of the matrix, and BSUs to elements of the matrix. While this is possible, it might lead to suboptimal performance due to an excessively large number of cells (if the matrix is large). The alternative is to map a number of matrix columns to each cell (e.g. $n/\sqrt{p}$ columns). The exact layout of data should then be done to match the number of columns or rows that are to be accessed at once. For example, in order to optimize for partitioning into blocks, it would be better to organize the data in row-major order within each cell, rather than using column-major order.

The main difference between partitioning via the scatter/gather file mode and Vesta partitioning is that in Vesta the partitioning is defined in advance, rather than being linked to a specific I/O operation. An important by-product of this distinction is that a single I/O operation can then access multiple disjoint chunks of data: it is enough that they are contiguous *in the partition*, and they do not have to be contiguous in the file. In some cases this can reduce message passing overhead considerably, by combining a number of small chunks of data into a single message. An example is given in Fig. 12. Two processes access alternate data items from a single disk (a cyclic partitioning pattern). Using file modes, a loop accessing one item

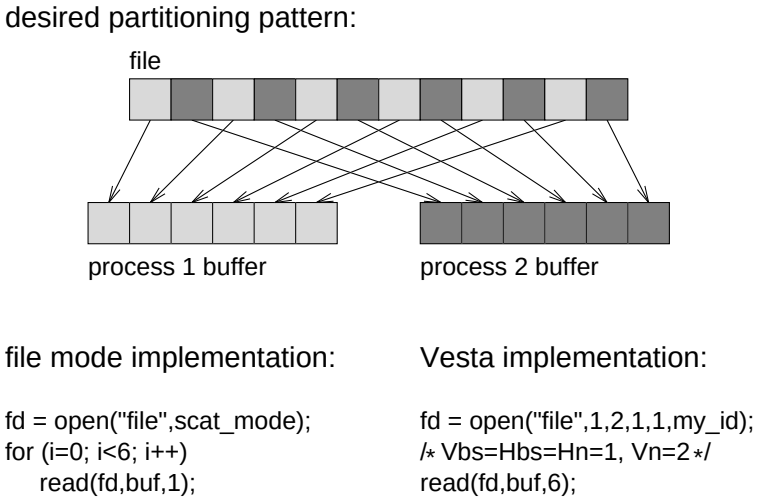


Figure 12: Predefined partitioning, as done in Vesta, can reduce the number of I/O operations (and messages used) relative to partitioning based on the scatter/gather file mode.

at a time is required. With Vesta partitioning, all the data can be accessed in a single I/O operation.

3.6 Partitioning Using MPI Datatypes

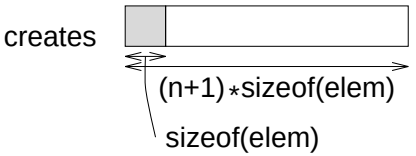
While Vesta partitioning is a significant improvement over the scatter/gather file mode, it still may not fulfill all user requirements. For example, it does not directly support the partitioning of 3-D structures. Partitioning based on rectilinear array decomposition as mentioned above solves this particular problem, but still cannot express partitions such as diagonals in a matrix. Diagonals and other partitioning patterns can be expressed by another recent proposal, the MPI-IO interface [10].

In the MPI-IO proposal, partitioning is expressed by using MPI derived datatypes. MPI derived datatypes are a mechanism for creating complex structures out of simpler components [46]. For example, it is possible to create a vector where a certain basic element is repeated a certain number of times with a given stride. If the stride is larger than the element size, this leaves holes between successive elements. Partitioning is expressed by conceptually tiling the file with such a derived datatype, called the *filetype*. The process then gains access to those parts of the file that correspond to the basic elements in the filetype, skipping those parts that fall under holes.

As a simple example, consider the implementation of the broadcast/reduce pattern and the scatter/gather pattern. Broadcast/reduce is achieved when all the processes use exactly the same filetype. Note that this is a generalization of the conventional broadcast/reduce file mode, because this filetype may have holes in it. Scatter/gather is achieved by using complementary filetypes. This means that the present elements in the filetype used by one process correspond to holes in the filetypes of all other

step 1: one element and n holes

```
blklens = { 1, 1 }
disps = { 0, (n+1)*sizeof(elem) }
types = { elem_t, MPI_UB }
MPI_Type_struct( 2, blklens, disps, types, &one_elem_t )
```



step 2: create the filetype for process i

```
blklens = { n-i, i }
disps = { i* sizeof(elem), (n-i)* n *sizeof(elem) }
types = { one_elem_t, one_elem_t }
MPI_Type_struct( 2, blklens, disps, types, &diag_t )
```

for n=6 and i=2, this creates

when the data is viewed as a 6x6 matrix
in column-major order, this becomes

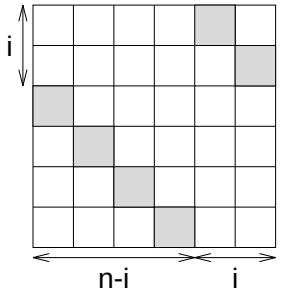


Figure 13: Using MPI derived datatypes to partition a matrix into diagonals, in the proposed MPI-IO interface.

processes. Again this is a generalization of the conventional scatter/gather mode, because some data may be left untouched, and the partitioning is not necessarily in order of process IDs.

Partitioning to access diagonals of a matrix is a special case of this generalized scatter/gather. The implementation is shown in Fig. 13. It is assumed that the matrix is stored in column-major order. Each process starts by creating a derived datatype that describes a single element followed by n holes, so that the next element is the next one in the same diagonal. This is then used as a building block for the filetype that extracts the i^{th} diagonal from the matrix.

The MPI-IO proposal is unique in that it also uses MPI derived datatypes to express the way data is laid out in memory. Thus the interface allows a non-contiguous data set in a file to be transferred to or from a non-contiguous dataset in memory. This is useful when the data in memory is also a part of a larger data structure, e.g. the internal part of a domain that is surrounded by vectors that are shared with neighboring processes. By using MPI derived datatypes, the extraction of data for I/O is expressed by the same mechanism as the extraction of data for inter-process message passing.

3.7 Relationship to the Physical Location of Data

Of all the partitioning schemes described above, only the Vesta scheme expresses the partitioning in terms that can be mapped directly to actual I/O devices. All the others express the partitioning at an abstract level of data structuring, and do not provide any control over the actual layout. This unique feature of Vesta is both an advantage and a disadvantage.

The importance of allowing programmers control over data mapping is that such control is required in order to achieve optimal performance. If users do not know how the data is mapped, they cannot guarantee that the minimal number of disk accesses are performed. Such guarantees are important in the context of high-performance computing, because disk accesses are orders of magnitude slower than floating point operations and even message passing. Performing an extra disk operation may be equivalent to performing many thousands of extra computations. Indeed, some researchers have developed algorithms for out-of-core computations where the complexity is measured in disk operations rather than in computational steps [52, 66, 14]. Vesta is the only system to date that provides the required control, at least to some degree.

On the other hand, the coupling between the partitioning scheme and the physical layout comes at the expense of a clean abstraction of the data structures. Thus the two dimensions of the vesta partitioning scheme are not equivalent. The horizontal dimension (across cells) reflects the parallelism in the data storage, while the vertical dimension (BSUs in cells) reflects the data. Typically, the hardware extent will be much smaller than the extent of the data structures. For example, a system with 10 I/O nodes may be called upon to handle matrices of 10000×10000 elements.

Programmers will therefore have to contend with mapping large data structures into a smaller number of cells. This burden can be eased by using higher level libraries, such as MPI-IO, that will be implemented above Vesta.

The issue of how data is actually arranged on disk may be different for persistent and transient files. Persistent files may be accessed by programs other than the one that created them. In particular, it might be advantageous to access such files using a conventional sequential program, e.g. for debugging or visualization of scientific results [15, 23]. This implies that the file should be stored in a manner that is compatible with conventional systems, and their sequential view of files. Transient files, on the other hand, can be stored in the most convenient manner for parallel access, sacrificing the compatibility with sequential systems [7, 23].

4 Implementation Issues

The common patterns of partitioning file data among a set of processes imply that data transfer is broken into small components [38, 35]. First, the data accessed by any given process is distributed across multiple I/O nodes, so only a fraction of the transfer is handled by each one. Second, within each I/O node, the data is interleaved with data being accessed by other processes, so it is not contiguous. Actually implementing an I/O operation as a set of small accesses like this would result in significant performance penalties, because of startup and latency costs that would be associated with each component [18, 35]. Therefore it is imperative that parallel I/O operations be implemented in a way that exploits the fact that in aggregate all the processes together are performing a large structured I/O operation.

With interfaces that allow partitioning to be expressed, like those described in the previous section, it is possible to access non-contiguous data in the file with a single operation. If a number of these access components are stored on the same I/O node, then they can be transferred together in a single message. This reduces the total message passing overhead, and thus improves performance [21, 12].

Additional improvements can be obtained by coordinating the disk accesses that serve multiple processes. This typically implies that collective I/O operations should be used. Such operations are performed by all the participating processes at the same time, and usually include a barrier synchronization point. For example, when files are in the broadcast/reduce and scatter/gather access modes, `read` and `write` operations are performed collectively.

4.1 Performance Benefits of Collective I/O

Disk scheduling is known to have a crucial impact on I/O performance. When multiple processes running on distinct compute nodes share access to data on a single disk, their request streams must be coordinated in order to prevent detrimental effects on the disk scheduling. This is true even if the processes are accessing disjoint data sets which are interleaved on the same disk.

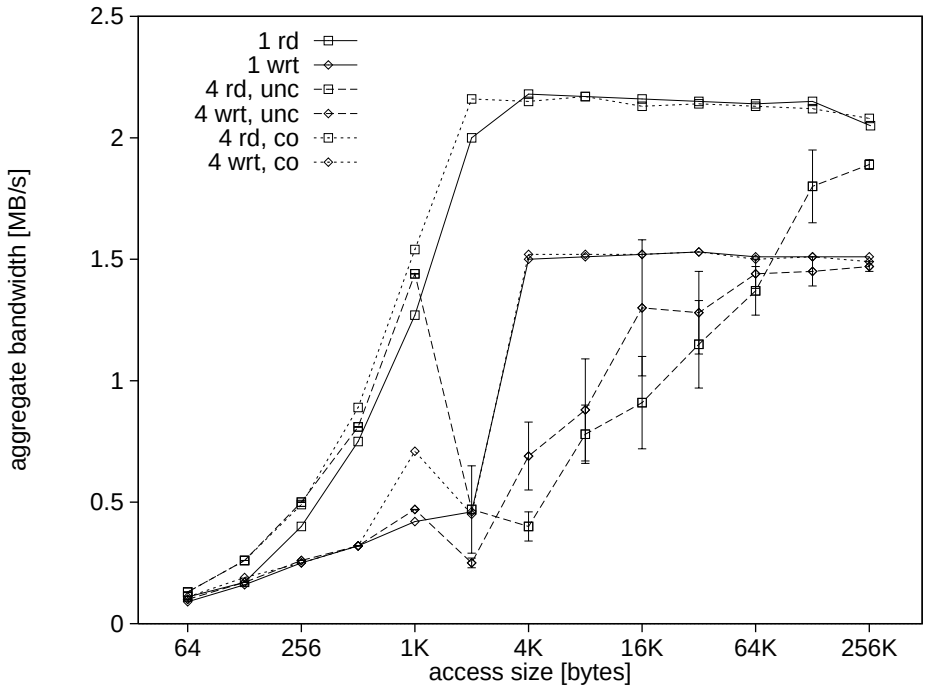


Figure 14: Performance of disk access by one and four clients, with and without coordination.

We have executed the following experiment using Vesta on an IBM SP1 to demonstrate and quantify this effect. A file of 128 MB is written to and read from a single I/O node. This is done by either a single compute node, or a set of four compute nodes. When four compute nodes are used, the data is partitioned into four disjoint subfiles that are interleaved with each other. All powers of two from 64 bytes up to 256 KB were used as the unit of interleaving. Accesses are always to a single such unit. The same access sizes are also used in the experiment with a single compute node.

The experiment with four compute nodes was run twice. In the first case, the accesses from the four nodes were not coordinated in any way. Each node simply accesses its subfile sequentially. In the second case, the nodes passed a token among themselves to ensure that the requests are issued in the order in which the data resides on disk. The token starts with the first node, which issues the first request. It then passes the token to the second node, which issues its first request, targeted at the second interleaved data unit. The third and fourth nodes issue their requests next. The first node must wait for the token to come back from the fourth node before it can issue its second request, which is targeted at the fifth data unit.

The results are shown in Fig. 14. The bandwidth achieved for small access sizes is low, because the overhead per access dominates. For reads, a single compute node achieves the disk bandwidth of 2.2 MB/s for access sizes of 2 KB and above. For

writes, the disk bandwidth of 1.5 MB/s is achieved for 4 KB and above. The sharp transition between 2 KB and 4 KB is due to the fact that the disk block size is 4 KB, so smaller accesses need to read the block off disk first before it can be modified[3].

The measurements for four compute nodes with coordinated access track those for a single compute node very closely. In some cases, the performance for four nodes is even slightly superior. When four compute nodes access the data with no coordination, their performance tracks that of a single node up to access sizes of 1 KB, and then they drop sharply. This is again a result of the 4 KB block size. When each access is smaller than 1 KB, each compute node touches all the data blocks in sequence. The first to arrive performs the disk access, then the rest hit the buffer cache. The sequence of requests seen by the disk is therefore identical to the case of a single compute node. But if the access size is larger, each compute node only touches a subset of the data blocks. The sequence of requests seen by the disk then depends on the order in which the requests arrive from the compute nodes. The performance then depends on the random interleaving order. The plotted results are averages of a number of measurements, with error bars that represent the average distance of individual measurements from this average.

The worst performance is for accesses of 2 KB, which are small enough so that the overhead for disk seeking is significant, and large enough so that each process does not touch all the disk blocks. As the access size grows larger, the relative weight of the disk seek becomes smaller. When the access size is very large, each individual request is large enough to utilize the disk efficiently. Therefore uncoordinated writes achieve full performance for 64 KB and above. The trend indicates that reads should achieve full performance for accesses larger than 512 KB.

4.2 Explicit Support for Collective I/O

When collective I/O operations are performed, the system obtains important knowledge about a whole set of I/O operations that occur at the same time. It is then possible to perform these operations in the order that would optimize disk performance.

The performance of disk access is governed by the physical properties of disks. The magnetic head must seek to the correct track for data to be accessed. The platter must rotate to the correct position before data can be transferred. Taken together, these characteristics cause sequential access to full tracks, one after the other, to be the most efficient access pattern. All other patterns achieve inferior performance, as measured by both turnaround time and achieved bandwidth.

Message passing among the compute nodes and I/O nodes is orders of magnitude faster than disk access. It has therefore been proposed that collective I/O operations representing complicated access patterns by multiple processes be performed in two phases (Fig. 15) [18]. For reading, first read all the data sequentially off the disk

[3]This experiment was performed with the initial Vesta implementation, where the server running on the I/O nodes used AIX JFS to access files. The AIX block size is 4 KB.

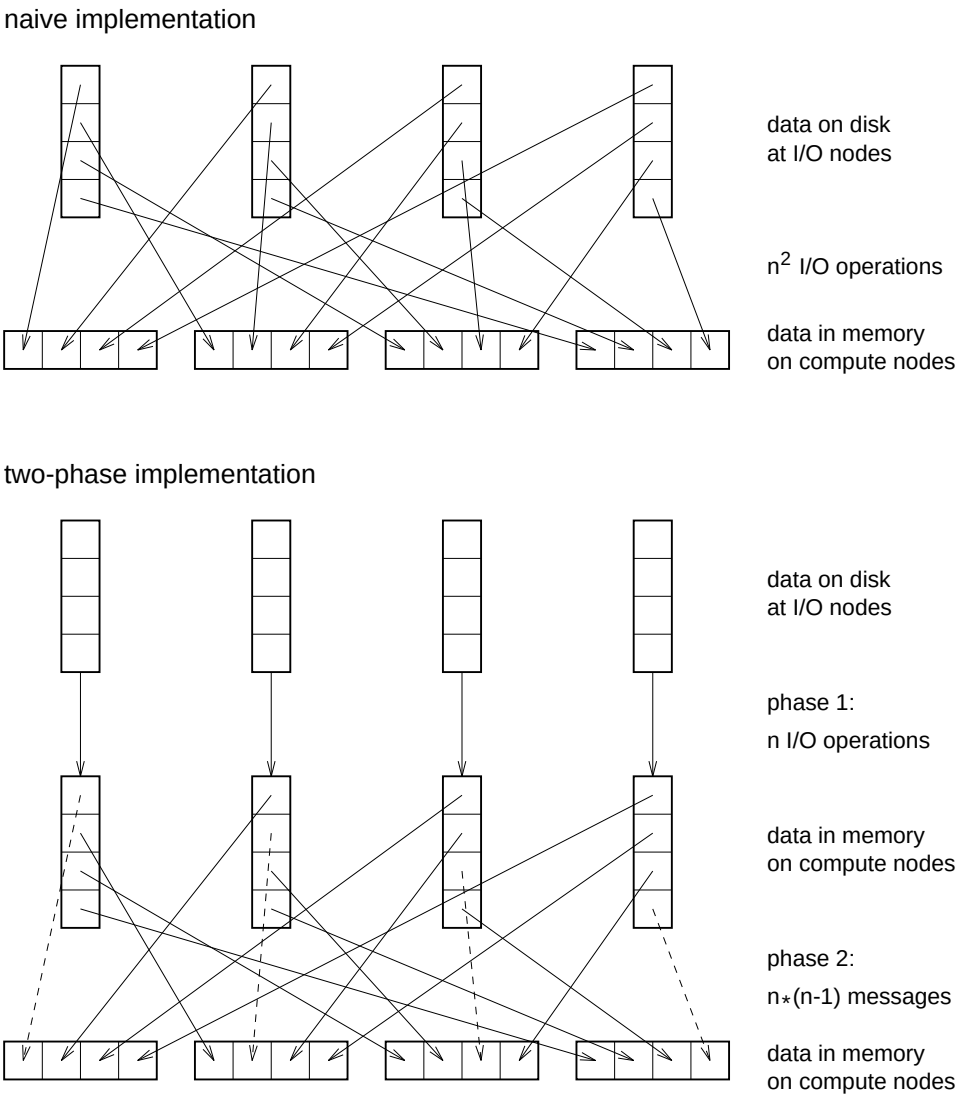


Figure 15: Partitioning by rows when a matrix is stored by columns. A two-phase implementation reduces the number of I/O operations, and performs larger I/Os, at the expense of additional message passing later.

into the memories of a select subset of compute nodes. Then reorganize the data in memory, and send each part to the compute node that requested it. For writes, the order is reversed: first compose all the data in memory, and then write it sequentially.

Experimental results obtained on the Touchstone Delta based on Intel's CFS have shown that the two-phase approach can improve performance by more than two orders of magnitude relative to the naive implementation where each component is accessed separately [18]. However, this approach also has its drawbacks. First, it requires extra buffering at the compute nodes, which might come at the expense of memory available for the user application (the exception is if all reorganization can be done

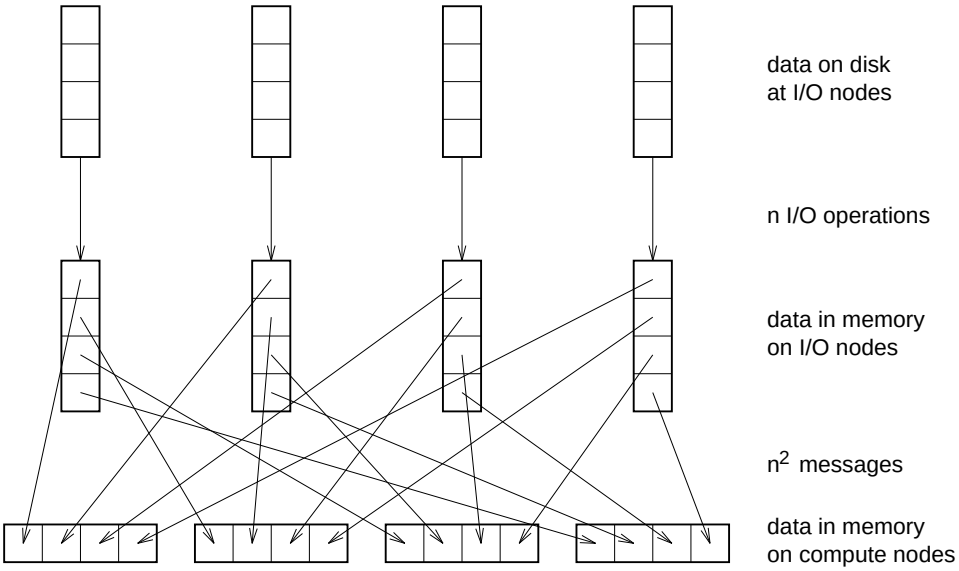


Figure 16: Optimal disk access based on reorganization in the memory of the I/O nodes.

in place). Second, it increases network traffic, because most of the data needs to be transferred twice. Finally, the re-organization phase involves concentrated message traffic that is more susceptible to congestion, whereas the transfer to and from I/O nodes is gated by the lower bandwidth of the disks.

The main advantage of two-phase I/O is that it can be implemented as a user-level library above whatever I/O system is available. Further optimizations require changes to the interface between compute nodes and I/O nodes. One approach that has been suggested is disk-directed I/O [35]. In this approach, the data partitioning involved in the collective I/O operations can be described by a small number of parameters (as in the Vesta interface). The collective operation itself first involves a barrier synchronization among the compute nodes, to ensure that all the memory buffers are ready. Then a representative compute node broadcasts the I/O request to all the I/O nodes. Each I/O node analyzes the request, and extracts the parts that resides on its local disks. If the request is a collective read, it then schedules the required disk access operations in the optimal manner. As each disk block is read off disk, the data is sent to the relevant compute nodes. Thus there is no extra message passing, no need for extra memory on compute nodes, no need for extra buffering on I/O nodes, and no need for heuristics for buffer management and prefetching.

An example is shown in Fig. 16. This is the same type of access as before, with 4 compute nodes accessing the rows of a matrix that is stored by column on 4 I/O nodes. The difference is that the data is first read into memory on the I/O nodes, rather than moved to the memories of the compute nodes. Then n^2 messages are used to redistribute the data in the desired pattern.

4.3 Caching and Prefetching

Caching file data in memory is considered an important feature in many file systems. In distributed file systems, caching is often done both on the file servers and on the clients [41]. In parallel file systems, client-side caching is more problematic, because the interleaved access patterns create significant false sharing of file blocks [38]. However, it is still possible to use client caching if the access is read only. File systems that provide partitioning can also support client caching when data is written, provided that processes are writing to disjoint partitions. The system can later reconstruct the file blocks using knowledge about the partitioning scheme.

Caching at the I/O nodes can be instrumental for supporting collective I/O operations, even without using any explicit knowledge. This is based on the observation that even when each process performs a strided access through the data, considering all the accesses by all the processes often shows, in aggregate, that all the data is being accessed (this has been called "interprocess locality" [37]). By using a large enough buffer cache, it is possible to coalesce the possibly unordered accesses from the different processes into a sequential access pattern to disk. This happens at each I/O node individually, and pertains to data stored at that I/O node. For writing data, the contributions from different processes can come in any order. As each one is received by the I/O node, it is copied into a frame in the buffer cache. If it relates to a previously unwritten block, a new frame is allocated. Using a write behind policy, the data is allowed to accumulate in the buffer cache. Assuming the accesses to the same area in the file come more or less together, a range of blocks will be filled before the system runs out of frames. These can then be written sequentially to disk (Fig. 17).

The same principle can be applied to reading, except that here the system is required to prefetch data based on a read-ahead mechanism. With a large enough buffer cache and simple prefetching algorithms, strided I/O operations from one process can pave the way for buffer cache hits for other processes. This requires the access history to be maintained on a global basis, rather than for each process individually. For example, the Vesta prefetching algorithm maintains a list of the last 32 unique blocks that have been accessed. When a new request comes in, this list is searched for a sequence of blocks preceding the newly requested one. If they are found, a read-ahead operation for the following blocks is triggered. This works even if the preceding blocks were accessed out of order and by different processes.

Buffer cache management algorithms for parallel systems have been designed with such scenarios in mind [36]. Experiments conducted with the Vesta file system have shown that prefetching based on the identification of a sequential pattern composed of interleaved strided accesses does lead to benefits, even without collective I/O calls [21][4]. But relying exclusively on buffer cache management cannot solve all problems. For one thing, not all access patterns cover the whole dataset. If the

[4]These are experiments with the new version of Vesta, which includes buffer cache management.

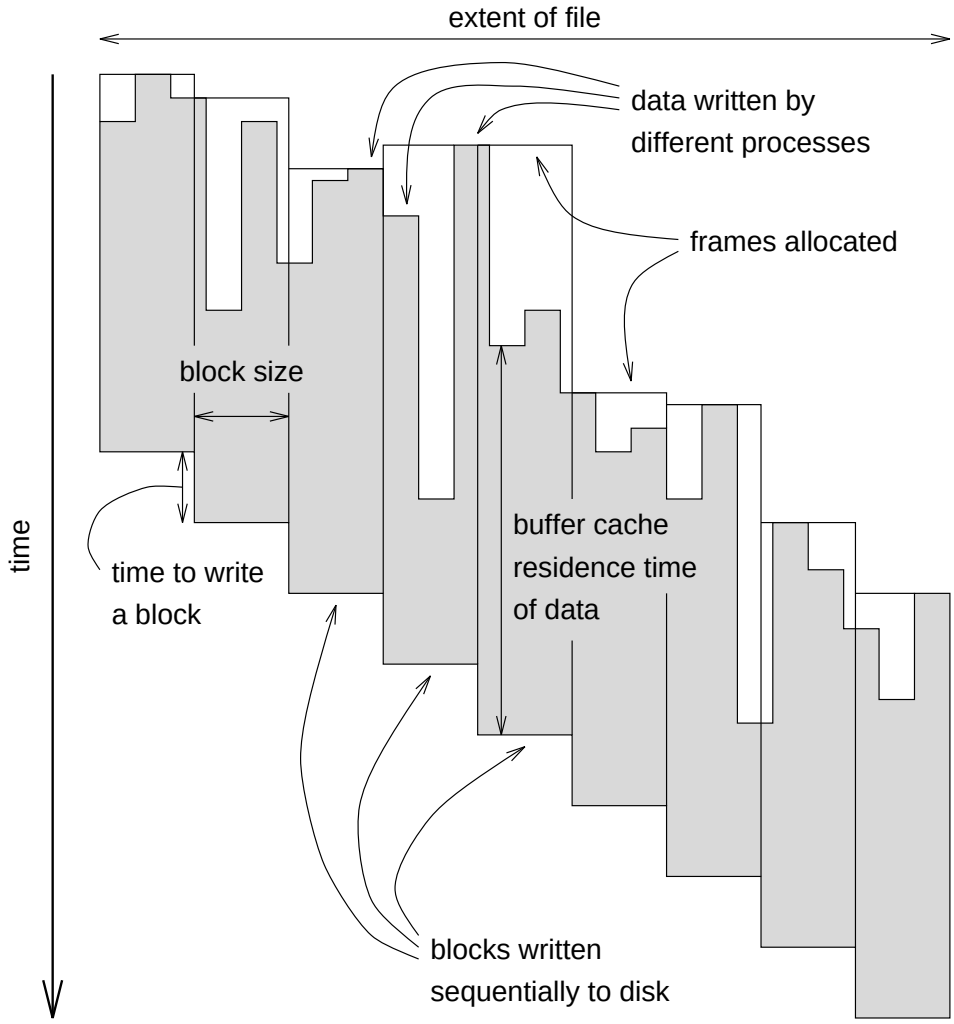


Figure 17: A buffer cache can collect writes that come in a random order, aggregate the data, and then perform sequential writes to disk.

processes are reading a 2-D slice out of a 3-D data structure, then their aggregate behavior may be a strided access, not a sequential one. This may cause excessive prefetching of unnecessary data, and tie up important disk bandwidth. A partial solution is to use more sophisticated prefetching algorithms, that can identify a strided access pattern [37]. However, the problem remains for access patterns that are less structured. Another example is when old data is overwritten in units that are smaller than the block size. Global knowledge can reveal that eventually all the data will be overwritten, but if the requests are received one at a time it is necessary to first read the old data, and then update it one part at a time. Finally, overagressive buffer cache management can cause problems by itself. This happened in CFS on Intel hypercubes, where prefetching sometimes competed with actual application I/O for

limited buffer space, especially in installations that had a high ratio of compute nodes to I/O nodes [51]. This was the reason for streaming data through the I/O nodes without any buffering in the Intel PFS design (which is based on the OSF/1 AD file system [58]).

Explicit support for collective I/O operations can also be combined with prefetching and buffer management. This is done in TIP (Transparent Informed prefetching) [55]. The idea is to provide the system with hints that inform it about future reference patterns. The system can then decide what to prefetch and when. It can even take into account the conflicting requests of multiple jobs.

5 Conclusions

In massively parallel processors, parallel I/O subsystems are required to balance I/O capabilities with computing power. The general trend in recent years is toward dedicated I/O nodes servicing I/O operations requested by compute nodes executing user applications. Compute nodes and I/O nodes communicate via the MPP's internal high-performance interconnection network, which is used both for I/O operations and for user application inter-task communication. Depending on the system architecture and I/O node placement, I/O traffic from a given job may not interfere with other jobs as long as these jobs do not perform I/O themselves.

In order to define the semantics of parallel I/O operations, different parallel interfaces are used. File modes allow one to express broadcast/reduce and uni-dimensional scatter/gather types of operations on file data. In addition, supporting a shared file pointer allows for the creation of shared log files or self-scheduled processing. However, file modes are insufficient for performing many rectilinear partitionings of file data. Multi-dimensional rectilinear partitioning, which is common in parallel applications, is supported by libraries based on array decomposition. The Vesta partitioning scheme permits all types of 2-D partitioning, and ties the partitioning ot physical data layout. The recent MPI-IO proposal expresses partitioning with MPI derived datatypes, and the extraction of file data is expressed by the same mechanism as the extraction of data for inter-process message passing in MPI. This is the most flexible mechanism to date.

Optimizations are required to implement such parallel interfaces efficiently, and there is an increasing recognition by the scientific community of the need for explicitly parallel collective I/O operations. Grouping I/O requests issued by parallel tasks allows one to minimize the number of disk accesses and to order them according to the physical location of the data on disk. Caching and prefetching are other techniques which permit the reuse of file data, and provides a measure of implicit coordination when inter-process locality exists in the application. All these techniques are complementary and ongoing research studies ways of combining them efficiently.

Undoubtedly, the field of parallel I/O is in flux, with new interfaces being designed, and new systems being implemented. There is a slow convergence, and a

move toward standards. We envision a common interface with file partitioning and collective operations, that will enable efficient and portable implementations across various platforms.

References

[1] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Benjamin/Cummings, 1989.

[2] B. Alpern, L. Carter, and J. Ferrante, "*Modeling parallel computers as memory hierarchies*". In *Programming Models for Massively Parallel Computers*, W. K. Gilio, S. Jahnichen, and B. D. Shriver (eds.), pp. 116–123, IEEE Press, 1993.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "*Measurements of a distributed file system*". In 13th *Symp. Operating Systems Principles*, pp. 198–212, Oct 1991. Correction in *Operating Systems Rev.* **27(1)**, pp. 7–10, Jan 1993.

[4] S. J. Baylor, C. Benveniste, and Y. Hsu, "*Performance evaluation of a massively parallel I/O subsystem*". In *IPPS '94 Workshop on I/O in Parallel Computer Systems*, pp. 1–15, Apr 1994. (Reprinted in *Comput. Arch. News* **22(4)**, pp. 3–10, Sep 1994).

[5] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker, "*CMMD I/O: a parallel Unix I/O*". In 7th *Intl. Parallel Processing Symp.*, pp. 489–495, Apr 1993.

[6] R. Bordawekar, J. M. del Rosario, and A. Choudhary, "*Design and evaluation of primitives for parallel I/O*". In *Supercomputing '93*, pp. 452–461, Nov 1993.

[7] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima, "*Concurrent file operations in a high performance FORTRAN*". In *Supercomputing '92*, pp. 230–237, Nov 1992.

[8] C. E. Catlett, "*Balancing resources*". *IEEE Spectrum* **29(9)**, pp. 48–55, Sep 1992.

[9] 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.

[10] P. Corbett, D. Feitelson, Y. Hsu, J-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong, *MPI-IO: A Parallel File I/O Interface for MPI, Version 0.2.* Research Report 19841 (87784), IBM T. J. Watson Research Center, Nov 1994.

[11] P. F. Corbett, S. J. Baylor, and D. G. Feitelson, "*Overview of the Vesta parallel file system*". In *Proc. IPPS '93 Workshop on I/O in Parallel Computer Systems*, pp. 1–16, Apr 1993. (Reprinted in *Comput. Arch. News* **21(5)**, pp. 7–14, Dec 1993).

[12] P. F. Corbett and D. G. Feitelson, "*Design and implementation of the Vesta parallel file system*". In *Scalable High-Performance Comput. Conf.*, pp. 63–70, May 1994.

[13] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, "*Parallel access to files in the Vesta file system*". In *Supercomputing '93*, pp. 472–481, Nov 1993.

[14] T. H. Cormen, "*Fast permuting on disk arrays*". *J. Parallel & Distributed Comput.* **17(1&2)**, pp. 41–57, Jan/Feb 1993.

[15] T. W. Crockett, "*File concepts for parallel I/O*". In *Supercomputing '89*, pp. 574–579, Nov 1989.

[16] E. DeBenedictis and J. M. del Rosario, "n*CUBE parallel I/O software*". In 11th *Intl. Phoenix Conf. Computers & Communications*, pp. 117–124, Apr 1992.

[17] E. P. DeBenedictis and S. C. Johnson, "*Extending Unix for scalable computing*". *Computer* **26(11)**, pp. 43–53, Nov 1993.

[18] 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).

[19] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[20] D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, *Satisfying the I/O Requirements of Massively Parallel Supercomputers.* Research Report RC 19008 (83016), IBM T. J. Watson Research Center, Jul 1993.

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

[22] S. Frank, H. Burkhardt, III, and J. Rothnie, "*The KSR1: bridging the gap between shared memory and MPPs*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 285–294, Feb 1993.

[23] N. Galbreath, W. Gropp, and D. Levine, "*Applications-driven parallel I/O*". In *Supercomputing '93*, pp. 462–471, Nov 1993.

[24] G. A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* MIT Press, 1992.

[25] G. A. Gibson and D. A. Patterson, "*Designing disk arrays for high data reliability*". *J. Parallel & Distributed Comput.* **17(1&2)**, pp. 4–27, Jan/Feb 1993.

[26] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "*A microprocessor-based hypercube supercomputer*". *IEEE Micro* **6(5)**, pp. 6–17, Oct 1986.

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

[28] M. Holland and G. A. Gibson, "*Parity declustering for continuous operation in redundant disk arrays*". In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 23–35, Sep 1992.

[29] IBM Corp., *Introduction to Parallel Processing and Scalable POWERparallel Systems 9076 SP1 and 9076 SP2*. Order number GG24-4344-00, May 1994.

[30] Intel Supercomputer Systems Division, *iPSC/2 and iPSC/860 User's guide*. Order number 311532-007, Apr 1991.

[31] Intel Supercomputer Systems Division, *Paragon User's Guide*. Order number 312489-003, Jun 1994.

[32] R. H. Katz, "*High-performance network and channel based storage*". *Proc. IEEE* **80(8)**, pp. 1238–1261, Aug 1992.

[33] 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.

[34] R. E. Kessler and J. L. Schwarzmeier, "*Cray T3D: a new dimension for Cray Research*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 176–182, Feb 1993.

[35] D. Kotz, "*Disk-directed I/O for MIMD multiprocessors*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 61–74, USENIX, Nov 1994.

[36] D. Kotz and C. S. Ellis, "*Caching and writeback policies in parallel file systems*". *J. Parallel & Distributed Comput.* **17(1&2)**, pp. 140–145, Jan/Feb 1993.

[37] D. Kotz and C. S. Ellis, "*Practical prefetching techniques for parallel file systems*". In 1st *Intl. Conf. Parallel & Distributed Inf. Syst.*, pp. 182–189, Dec 1991.

[38] D. Kotz and N. Nieuwejaar, "*Dynamic file-access characteristics of a production parallel scientific workload*". In *Supercomputing '94*, pp. 640–649, Nov 1994.

[39] E. K. Lee and R. H. Katz, "*The performance of parity placements in disk arrays*". *IEEE Trans. Comput.* **42(6)**, pp. 651–664, Jun 1993.

[40] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 272–285, Jun 1992.

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

[42] W. Liu, V. Lo, K. Windisch, and B. Nitzberg, "*Non-contiguous processor allocation algorithms for distributed memory multicomputers*". In *Supercomputing '94*, pp. 227–236, Nov 1994.

[43] D. B. Loveman, "*High Performance Fortran*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 25–42, Feb 1993.

[44] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, "*sfs: a parallel file system for the CM-5*". In *Proc. Summer USENIX Conf.*, pp. 291–305, Jun 1993.

[45] J. Menon and D. Mattson, "*Comparison of sparing alternatives for disk arrays*". In 19th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 318–329, May 1992.

[46] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard.* May 1994.

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

[48] K. Miura, M. Takamura, Y. Sakamoto, and S. Okada, "*Overview of the Fujitsu VPP500 supercomputer*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 128–130, Feb 1993.

[49] J. R. Nickolls, "*The MasPar scalable Unix I/O system*". In 8th *Intl. Parallel Processing Symp.*, pp. 390–395, Apr 1994.

[50] N. Nieuwejaar and D. Kotz, *A Multiprocessor Extension to the Conventional File System Interface.* Technical Report PCS-TR94-230, Dept. Computer Science, Dartmouth College, Sep 1994.

[51] B. Nitzberg, *Performance of the iPSC/860 Concurrent File System.* Technical Report RND-92-020, NASA Ames Research Center, Dec 1992.

[52] M. H. Nodine and J. S. Vitter, "*Large-scale sorting in parallel memories*". In 3rd *Symp. Parallel Algorithms & Architectures*, pp. 29–39, Jul 1991.

[53] Parasoft Corp., *Express Version 1.0: A Communication Environment for Parallel Computers.* 1988.

[54] Y. N. Patt, "*The I/O subsystem: a candidate for improvement*". *Computer* **27(3)**, pp. 15–16, Mar 1994. (guest editor's introduction to special issue).

[55] R. H. Patterson and G. A. Gibson, "*Exposing I/O concurrency with informed prefetching*". In 3rd *Intl. Conf. Parallel & Distributed Information Syst.*, pp. 7–16, Sep 1994.

[56] P. Pierce, "*A concurrent file system for a highly parallel mass storage subsystem*". In 4th *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 155–160, Mar 1989.

[57] A. L. N. Reddy, P. Banerjee, and S. G. Abraham, "*I/O embedding in hypercubes*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 331–338, Aug 1988.

[58] P. J. Roy, "*Unix file access and caching in a multicomputer environment*". In *USENIX Mach III Symp.*, pp. 21–37, Apr 1993.

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

[60] E. J. Schwabe and I. M. Sutherland, "*Improved parity-declustered layouts for disk arrays*". In 6th *Symp. Parallel Algorithms & Architectures*, pp. 76–84, Jun 1994.

[61] K. G. Shin and G. Dykema, "*A distributed I/O architecture for HARTS*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 332–342, May 1990.

[62] D. Stodolsky, G. Gibson, and M. Holland, "*Parity logging: overcoming the small write problem in redundant disk arrays*". In 20th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 64–75, May 1993.

[63] C. B. Stunkel et al., *The SP2 Communication Subsystem*. Research Report RC 19914, IBM T. J. Watson Research Center, Jan 1995.

[64] R. Thakur, R. Bordawekar, and A. Choudhary, "*Compilation of out-of-core data parallel programs for distributed memory machines*". In *IPPS '94 Workshop on I/O in Parallel Computer Systems*, pp. 54–72, Apr 1994. (Reprinted in *Comput. Arch. News* **22(4)**, pp. 23–28, Sep 1994).

[65] Thinking Machines Corp., *Connection Machine CM-5 Technical Summary*. Nov 1992.

[66] 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.