NASA-CR-203564

NCC 2-349
IN 62-CR
013268

Performance of the Galley Parallel File System

Nils Nieuwejaar David Kotz

Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510
{nils,dfk}@cs.dartmouth.edu

Abstract

As the I/O needs of parallel scientific applications increase, file systems for multiprocessors are being designed to provide applications with parallel access to multiple disks. Many parallel file systems present applications with a conventional Unix-like interface that allows the application to access multiple disks transparently. This interface conceals the parallelism within the file system, which increases the ease of programmability, but makes it difficult or impossible for sophisticated programmers and libraries to use knowledge about their I/O needs to exploit that parallelism. Furthermore, most current parallel file systems are optimized for a different workload than they are being asked to support. We introduce Galley, a new parallel file system that is intended to efficiently support realistic parallel workloads. Initial experiments, reported in this paper, indicate that Galley is capable of providing high-performance I/O to applications that access data in patterns that have been observed to be common.

1 Introduction

Multiprocessors have been steadily increasing in computational performance, but the power of the I/O subsystem has not kept pace. This disparity is partly due to the physical limitations of storage hardware, but a more significant reason for this performance gap is the limitations of current parallel file systems. Most modern parallel file systems were designed around several key assumptions about how scientific applications would use such systems. Several recent analyses of file-system workloads on production multiprocessors running primarily scientific applications show that many of these assumptions are incorrect [KN94, PEK+95, NKP+95]. Specifically, it was commonly believed that parallel, scientific applications would have behavior similar to that of existing sequential and vector scientific applications [Pie89, PFDJ89, LIN+93]. These applications tend to access large files in large, consecutive chunks [MK91, PP93]. Studies of two parallel file-system workloads, supporting many users and running a variety of applications in a variety of scientific domains, under both data-parallel and control-parallel programming models, show that many parallel, scientific applications make many small, non-consecutive requests to the file system [KN94, PEK+95, NKP+95, NK95]. These studies suggest that most parallel file systems have been optimized for a workload that is different than that which actually exists.

Using the results from these workload studies and from performance evaluations of existing parallel file systems, we have developed a new parallel file system that is able to deliver high performance to a variety of applications, such as those observed in actual workloads.

The remainder of this paper is organized as follows. In Section 2 we describe existing parallel file systems, how they are used in practice, and how they fail to meet the needs of the applications that rely on them. In Section 3 we describe a new way of structuring files and the interface that is used to access those files. Section 4 examines the performance of Galley. In Section 5 we discuss some related work, and finally, in Section 6, we conclude and describe our future plans.

Copyright ©1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

2 Background

2.1 Parallel File Systems

Most existing multiprocessor file systems are based on the conventional Unix-like file-system interface in which files are seen as an addressable, linear stream of bytes [BGST93, Pie89, LIN+93, WMR+94]. To provide higher throughput, the file system typically *declusters* files (i.e., scatters the blocks of each file across multiple disks), thus allowing parallel access to the file, reducing the effect of the bottleneck imposed by the relatively slow disk speed. Although the file is actually scattered across many disks, the underlying parallel structure of the file is hidden from the application by the higher-level abstraction. The interface is limited to such operations as `open()`, `close()`, `read()`, `write()`, and `seek()`, all of which manipulate an implicit *file pointer*.

One enhancement to the conventional interface, which is offered by several multiprocessor file systems, is a file pointer that is shared among the processes in an application and provides a mechanism for regulating access to a shared file by those processes [Pie89, BGST93]. The simplest shared file pointer is one which supports an atomic-append mode (as in [LMKQ89], page 174). Most parallel file systems provide this mode in addition to several more structured access modes (e.g., round-robin access to the file pointer).

Copyright 1996 by ACM. Appeared in IOPADS '96, May 1996, pp. 83-94.
IOPADS is the Workshop on I/O in Parallel and Distributed Systems.

JAN 24 1997

We compare Galley to other, more sophisticated, parallel file systems in Section 5.

2.2 Workload Characterization

Experience has shown that the simple, Unix-like model of a file is well suited to uniprocessor applications that tend to access files in a simple, sequential fashion [OCH+85]. It has similarly proven to be appropriate for scientific, vector applications that also tend to access files sequentially [MK91]. Until recently, however, there had been no investigation into whether this file model and interface were well suited to massively parallel scientific applications.

To determine whether this model was appropriate, we examined the file-system workloads on two different massively parallel processors, running two different application workloads [KN94, PEK+95]. These studies show that sequential access to consecutive portions of a file is much less common in a multiprocessor environment than in uniprocessor or supercomputer environments. In [NK95, NKP+95], we looked more closely at the specific patterns in which applications accessed the files in a parallel file system. We found that these applications frequently accessed files in regular, repeating patterns. For example, the most common pattern was a series of requests, all of the same size, separated by a common *stride* within the file. This pattern is likely to arise if, for example, a two-dimensional matrix is stored on disk in row-major order, and an application distributes the columns of the matrix across its processes in a CYCLIC fashion (using High Performance Fortran terminology [HPF93]).

In addition to assuming that parallel scientific applications would access files consecutively, most parallel file system implementations assume that these files would be accessed in large chunks — hundreds of kilobytes or megabytes at a time. Our workload characterization studies show that while some parallel scientific applications do issue a relatively small number of large requests, there are many applications that issue thousands or millions of small ($<$ 200 bytes) requests, putting a great deal of stress on current file systems.

While the standard Unix-like interface has worked well in the past, it seems clear that it is not well suited to parallel applications, which have more complicated access patterns than uniprocessor and supercomputer applications. Furthermore, the tracing study described in [KN94] found that shared file pointers were rarely used in practice and suggests that poor performance and a failure to match the needs of applications are the likely causes. This finding indicates that the simple extensions offered by most of today's parallel file systems are not a sufficient adaptation of this interface.

3 File Structure

While most existing multiprocessor file systems are based on the linear file model, the underlying parallel structure of the file is hidden from the application. Galley uses a more complex file model that should lead to greater flexibility and performance. In addition to providing high performance, Galley was designed to be 'library friendly', giving programmers the capability to easily layer abstractions above the file system. We summarize the model here; full details of this structure may be found in [NK96].

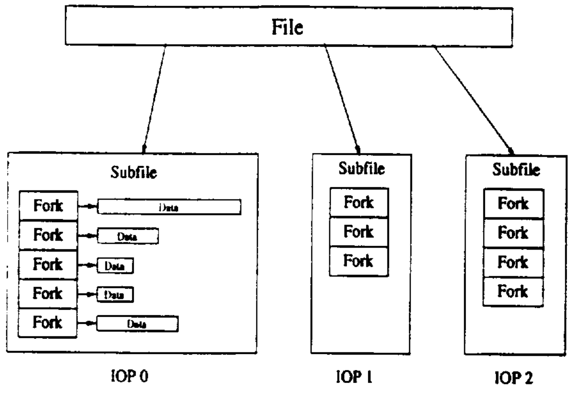

Figure 1: Three dimensional structure of files in the Galley File System. The portion of the file residing on IOP 0 is shown in greater detail than the portions on the other two IOPs.

3.1 Subfiles

The linear model can allow good performance when the request size generated by the application is larger than the declustering-unit size, as multiple disks are being used in parallel for each request. The declustering-unit size is frequently measured in kilobytes (e.g., 4KB in Intel's CFS [Pie89]), however, while our workload characterization studies show that the typical request size in a parallel application is much smaller: frequently under 200 bytes. This disparity means that most of the individual requests generated by parallel applications are not being executed in parallel. Another problem with the linear file model is that a dataset may have a natural, parallel mapping onto multiple disks that is not easily captured by the standard cyclic block-declustering schemes. One such example may be seen in the Flexible Image Transport System (FITS) data format, which is used for astronomical data [NAS94]. A FITS file is organized as a series of records, each of which contains a key with multiple fields and one or more data elements. It is not clear that blindly striping these records across multiple disks is the optimal approach in a parallel file system. Rather, one could distribute the data across the disks using the keys and knowledge about how the dataset will be used to determine a partitioning scheme that results in highly parallel access. Finally, the parallel-I/O algorithms community has frequently argued for this kind of increased control over declustering [CK93, WGRW93].

To address these problems, Galley allows applications to fully control the way in which data is declustered across the IOPs, as well as which IOP they wish to access in each request. To allow this behavior, files are composed of one or more *subfiles*, each of which resides entirely on a single IOP, and which may be directly addressed by the application.

3.2 Forks

Each subfile in Galley is structured as a collection of one or more independent *forks*, each of which is a linear, addressable collection of bytes, similar to a traditional Unix file. This three-dimensional file structure is illustrated in Figure 1. Note that there is no requirement that all subfiles have the same number of forks, or that all forks have the same size.

The use of forks allows further application-defined structuring. This structuring may include storing distinct types of data in separate forks (e.g., a list of pressures in one fork and a list of temperatures in another), or it may involve storing metadata in one fork and 'real' data in another (e.g., a compression library similar to that described in [SW95] could store compressed data chunks in one fork and directory information in another).

An example of using forks for both data and metadata may be found in data files like those described in [KFG94]. The style of FITS file described in this study contained records with 6 keys, describing the frequency domain, the antenna, and the time the data was collected. The data portion of each record contained a pair of data elements, one for each of two polarizations, and each data element contains floating-point triples for each of 31 frequencies. Although most queries to the data only involved one of the two polarizations, the two sets of data were stored together in a single file to reduce the total amount of diskspace (by not replicating the key information). Unfortunately, this meant that an application generally read all the data for each polarization when it was only interested in one set. Using the Galley file system, a programmer could choose to store the keys in one fork, the data for one polarization in a second fork, and the data for the other polarization in a third fork. In addition to reducing the amount of data we need to read when we are only interested in one of the two sets of data, by isolating the keys in their own fork we reduce the amount of data we need to read when scanning for a given key pair. Unlike segmenting a traditional file into three regions, the fork-based structure allows each fork to grow as more records are added.

A further discussion of the applications and benefits of this structure can be found in [NK96].

3.3 Data Access Interface

The standard Unix interface provides only simple primitives for accessing the data in files. These primitives are limited to `read()`ing and `write()`ing consecutive regions of a file. As discussed above, recent studies show that these primitives do not match the needs of many parallel applications [NK95, NKP+95]. Specifically, parallel scientific applications frequently make many small requests to a file, with *strided* access patterns.

We define two types of strided patterns. A *simple-strided* access pattern is one in which all the requests are the same size, and there is a constant distance between the beginning of one request and the beginning of the next. A group of requests that form a strided access pattern is called a *strided segment*. A *nested-strided* access pattern is similar to a simple-strided pattern, but rather than repeating a single request at regular intervals, the application repeats a strided segment at regular intervals. Studies show that both simple- and nested-strided patterns are common in parallel, scientific applications [NK95, NKP+95]. Indeed, in one study, over 90% of the requests in the entire workload were part of one of these two patterns.

Galley provides three interfaces that allow applications to explicitly make regular, structured requests such as those described above, as well as one interface for unstructured requests. These interfaces allow us to combine many small requests into a single, larger request, which can lead to improved performance in two ways. First, reducing the number of requests can lower the aggregate latency costs, particularly for those applications that issue thousands or millions of tiny requests. Second, recent work has shown that providing a file system with more information about an application's access patterns can lead to tremendous performance improvements by introducing opportunities for intelligent scheduling of I/O and communication [Kot94].

The higher-level interfaces offered by Galley are summarized below. These interfaces are described in greater detail, and examples are provided, in [NK96, NK95].

3.3.1 Simple-strided Requests

```
gfs_read_strided(int fid, void *buf, ulong offset,
                 ulong rec_size, int f_stride,
                 int m_stride, int quant)
```

Beginning at `offset`, the file system will read `quant` records of `rec_size` bytes, where the offset of each record is `f_stride` bytes greater than that of the previous record. The records are stored in memory beginning at `buf`. The offset into the buffer is changed by `m_stride` bytes after each record is transferred. Note that either the file stride (`f_stride`) or the memory stride (`m_stride`) may be negative. The call returns the number of bytes transferred.

When `m_stride` is equal to `rec_size`, data will be *gathered* from disk, and stored contiguously in memory. When `f_stride` is equal to `rec_size`, data will be read from a contiguous region of a file, and *scattered* in memory. It is also possible for both `m_stride` and `f_stride` to be different than `rec_size`, and possibly each other.

Naturally, there is a corresponding `gfs_write_strided()` call.

3.3.2 Nested-strided Requests

```
gfs_read_nested(int fid, void *buf, ulong offset,
                ulong rec_size, struct stride *vec,
                int levels)
```

The `vec` is a pointer to an array of (`f_stride, m_stride, quantity`) triples listed from the innermost level of nesting to the outermost. The number of levels of nesting is indicated by `levels`.

3.3.3 Nested-batched Requests

While we found that most of the small requests in the observed workloads were part of strided patterns, there may well be applications that could benefit from some form of high-level, regular request, but would find the nested-strided interface too restrictive. For those applications, we provide a *nested-batched* interface. The nested-batched interface allows applications to make an arbitrary series of requests, which may then be repeated at regular intervals. This interface is *nested* in that any of the requests in the arbitrary series may themselves be a batched request. To conserve space, we do not present the details here (see [NK96, NK95]).

3.3.4 List Requests

Finally, in addition to these structured operations, Galley provides a more general file interface, called the *list* interface, which is similar to the POSIX `lio_listio()` interface [IBM94]. This interface simply takes an array of (file offset, memory offset, size) triples from the application. This interface is useful for applications with access patterns that do not have any inherently regular structure. While this interface essentially functions as a series of simple reads and writes, it provides the file system with enough information to make intelligent disk-scheduling decisions, as well as the ability to coalesce many small pieces of data into larger messages for transferring between CPs and IOPs.

4 Performance

Performance studies of parallel file systems tend to focus on the performance of large, sequential requests. Indeed, most do not even examine the performance of requests of fewer than many kilobytes [Nit92, BBH95, KR94]. As discussed above, recent workload characterizations show that parallel file systems are frequently called upon to service many small requests. This disparity means that most performance studies actually fail to examine how a file system can be expected to perform when running real applications in a production environment.

4.1 Experimental Platform

The Galley File System was designed to be portable across workstation clusters and massively parallel processors. The results in this paper were obtained on the IBM SP-2 at NASA Ames' Numerical Aerodynamic Simulation facility. This system has 160 nodes, each running AIX 3.2.5, but no more than 140 are available for general use. Each node has a 66.7 MhZ POWER2 processor, at least 128 megabytes of memory, and is connected to IBM's high-performance switch. While the switch allows throughput of up to 34 MB/s using one of IBM's message-passing libraries (PVMe, MPL, or MPI), those libraries cannot operate in a multithreaded environment. Furthermore, neither MPL nor MPI allow applications to be implemented as persistent servers and transient clients. As a result of these limitations, Galley is implemented on top of TCP/IP, with a maximum throughput of approximately 17 MB/s on the SP-2.

Each IOP in Galley controls a single disk, which it logically partitions into 32K blocks. Each IOP also maintains a cache of the most recently used blocks from the disk it controls. For this study, the size of each cache was 34 megabytes, large enough to hold 1100 blocks. Galley does not attempt to prefetch data for two reasons. First, indiscriminate prefetching can cause the cache to thrash [Nit92]. Second, prefetching is based on the assumption that the system can intelligently guess what an application is going to request next. Using the higher-level requests described above, there is frequently no need for Galley to make guesses about an application's behavior; the application is able to explicitly provide that information to each IOP.

Although each node on the SP-2 has a local disk, access to that disk must be performed through AIX's Journaling File System. While Galley was originally implemented to use these disks, we felt that our performance results were being inflated by the prefetching and caching provided by JFS. Specifically, we frequently measured apparent throughputs of over 10 MB/s from a single disk. Accordingly, the performance results presented here were obtained through the use of a simulation of an HP 97560 SCSI hard disk, which has an average seek time of 13.5 ms and a maximum sustained throughput of 2.2 MB/s [HP91]. Each IOP provides access to one simulated disk.

Our implementation of the disk model was based on earlier implementations [RW94, KTR94]. Among the factors simulated by our model are head-switch time, track-switch time, SCSI-bus overhead, controller overhead, rotational latency, and the disk cache. To validate our model, we used a trace-driven simulation, using data provided by Hewlett-Packard and used by Ruemmler and Wilkes in their study.[1] Comparing the results of this trace-driven simulation with the measured results from the actual disk, we obtained a demerit figure (see [RW94] for a discussion of this measure) of 5.0%, indicating that our model was extremely accurate.

The simulated disk is integrated into Galley by creating a new thread on each IOP to execute the simulation. When the thread receives a disk request, it calculates the time required to complete the request, and then suspends itself for that length of time. While, in most cases, the disk thread does not actually load or store the requested data, metadata blocks must be preserved. To avoid losing that data, the disk thread maintains a small cache, which is used to store 'important' data. When the simulation thread copies data to or from its cache, the amount of time required to complete the copy is deducted from the amount of time the thread is suspended. It should be noted that the remainder of the Galley code is unaware that it is accessing a simulated disk.

4.2 Access Patterns

We examine the performance of Galley under several different access patterns, each of which is composed of a series of requests for fixed-size pieces of data, or *records*. The patterns we examine are shown in Figure 2. While these patterns do not directly correspond to a particular 'real world' application, they are representative of the general patterns we observed to be most common in production multiprocessor systems. Our analysis is done with a single file that contains a single subfile (each with a single fork) on each IOP, and the patterns shown in Figure 2 reflect the patterns that we access *from each IOP*. The correspondence between the IOP-level access patterns we use in this study, and the file-level patterns observed in actual applications, is discussed for each pattern below.

The simplest access pattern we call *broadcast*. With this access pattern every compute node reads the whole file (i.e., the IOPs *broadcast* the whole file to all the CPs). This access pattern models the series of requests we would expect to see when all the nodes in an application read a shared file, such as a configuration file or the initial state for a simulation. Since an application that wants to access all the data in a file must access all the data in every subfile, a broadcast pattern at the file level clearly corresponds to a broadcast pattern at each subfile. Although it seems counterintuitive for an application to access large, contiguous regions of a file in small chunks, such behavior does occur in practice. One likely reason that data would be accessed in this fashion is that records stored contiguously on disk are to be stored non-contiguously in memory. In the simplest case, this pattern would be similar to the interleaved pattern described below, with the interleaving occurring in memory rather than on disk. Since it seems unlikely that an application would want every node to write to the entire file, we did not measure the performance of the broadcast-write case.

[1]Kindly provided to us by John Wilkes and HP. Contact John Wilkes at wilkes@hplabs.hp.com for information about obtaining the traces.

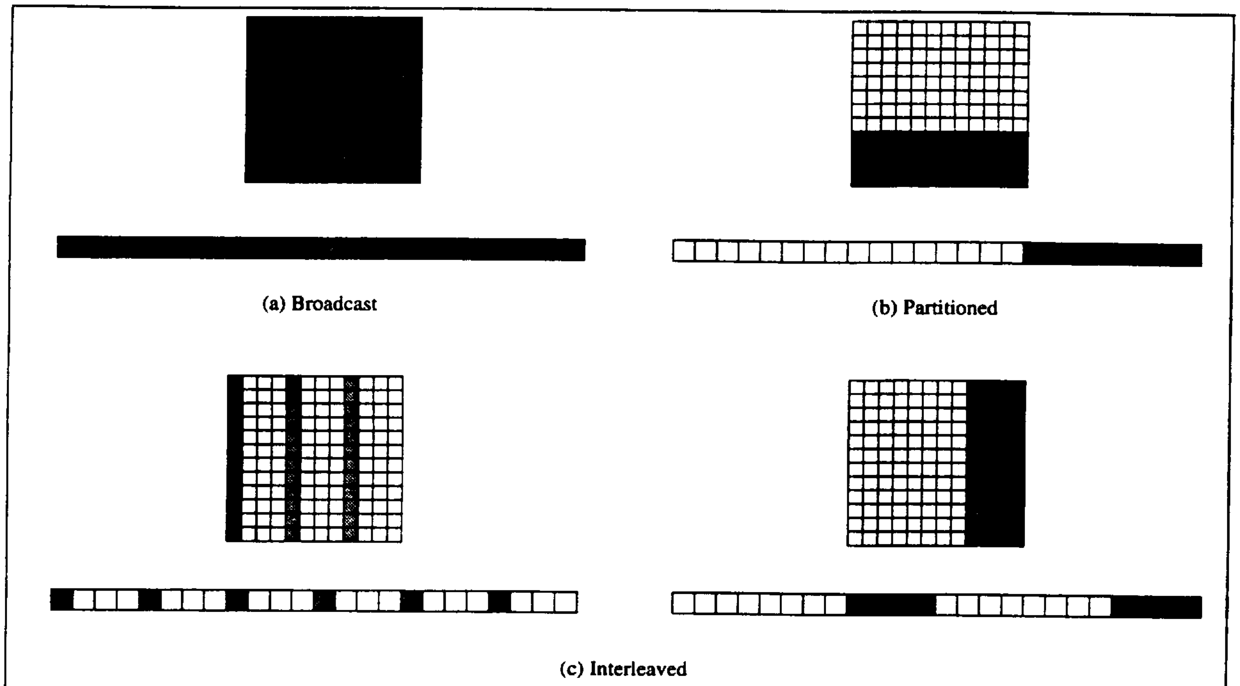

Figure 2: The three access patterns examined in this study. Each pattern is displayed with two types of view: the pattern as applied to a linear file, and matrix distributions that could give rise to the pattern. We assume that the matrices are stored in row-major order. Each block corresponds to a single *record* in the file, and the highlighted blocks represent the records accessed by a single compute node in a group of four.

The next access pattern we refer to as *partitioned*. With this pattern, each compute node accesses a distinct, contiguous region of each file. This pattern could represent either a one-dimensional partitioning of data or the series of accesses we would expect to see if a two-dimensional matrix were stored on disk in row-major order, and the application distributed the rows of the matrix across the compute nodes in a BLOCK fashion (using HPF terminology [HPF93]). A partitioned access pattern at the file level can map onto two different access patterns at the IOP level. The first pattern arises if the file is distributed across the disks in a BLOCK fashion; that is the first $1/n$ of the file bytes in the file are mapped onto the first of the n IOPs, and so forth. For each IOP, this mapping results in an access pattern similar to a broadcast pattern with 1 compute processor. The other, more interesting, mapping distributes blocks of data across the disks in a CYCLIC fashion, as in most implementations of a linear file model. This distribution results in accesses by each CP to each IOP. In a system with 4 CPs, the first CP would access the first 1/4 of the data in each subfile, and so forth. Thus a partitioned pattern at the file level leads to a partitioned pattern at each IOP. As with the broadcast pattern, applications may access data in this pattern using a small record size if the the data is to be stored non-contiguously in memory.

The final access pattern is an *interleaved* pattern. In this pattern, each compute node requests a series of noncontiguous, but regularly spaced, records from a file. For our testing, the interleaving was based on the record size. That is, if 16 compute nodes were reading a file with a record size of 512 bytes, each node would read 512 bytes and then skip ahead 8192 (16*512) bytes before reading the next chunk of data. This pattern models the accesses generated by an application that distributes the columns of a two-dimensional matrix across the processors in an application, in a CYCLIC fashion, if the matrix is stored in a linear file in row-major order. Assume the linear file is distributed traditionally, with blocks distributed across the subfiles in a CYCLIC fashion. In the simplest case, the block size might be evenly divisible by the product of the record size and the number of CPs. In this case, every block in the file is accessed with the same interleaved pattern, and any rearrangement of the blocks (between or within disks) will result in the same subfile-access pattern. Thus, the blocks can be declustered across the subfiles, but the access pattern within each subfile will still be interleaved. There are, of course, more complex mappings of an interleaved file-level pattern to an IOP-level pattern, but we focus on the simplest case.

For each test, we held the number of compute processors constant at 16, and varied the number of IOPs (each with one disk) from 4 to 64. Thus, the CP:IOP ratio varied from 1:4 to 4:1. Each test began with an empty buffer cache on each IOP, and each write test included the time required for all the data to actually be written to disk. Each fork was laid out contiguously on disk, allowing us to better understand how each access pattern affects the system's performance. While the size of each fork was fixed, the amount of data accessed for each test was not. Since the system's performance on the fastest tests was several orders of magnitude faster than on the slowest tests, there was no fixed amount of data that would provide useful results across all tests. Thus, the amount of data accessed for each test varied from 4 megabytes (writing 64-byte records to 4 IOPs) to 2 gigabytes (eading 64-kilobyte records from 64 IOPs).

During preliminary testing, we found that communications on the SP-2 would occasionally appear to freeze for a period of time. This problem would lead to anomalous results; one run in a series would take many times longer to complete than the others. Fortunately, this problem was severe enough that we could easily detect when it had occured. To work around this problem, we performed each test three times. We discarded any outliers (defined as throughput less than 10% of the average of the three runs), and recorded the average of the remaining runs. Disregarding outliers, results from repeated runs were generally within one or two kilobytes per second of one another.

4.3 Traditional Interface

To provide a baseline for future comparison, we first examined the performance of Galley using a traditional read/write interface. This interface forced each CP to make a separate request for each record from each fork. The tests in this section were performed by issuing asynchronous requests to each fork for a single record. When a request from one fork completed, a request for the next record from that fork was issued. By issuing asynchronous requests to each IOP, we were generally able to keep all the IOPs in the system busy. Since each CP accessed its portion of each subfile sequentially, and since the forks were laid out contiguously on disk, the IOPs were frequently able to schedule disk accesses effectively, even with the small amount of information offered by the traditional interface. Furthermore, the CPs were generally able to issue requests in phase. That is, when an IOP completed a request for CP 1, it would handle requests from CPs 2 through n. By the time the IOP had completed the request from CP n, it had received the next request from CP 1. Thus, even without explicit synchronization among the CPs, the IOPs were able to service requests from each node fairly, and were able to make good use of the disk.

Figure 3 shows the total throughput achieved when reading a file with various record sizes for each access pattern. Figure 4 presents similar results for write performance. The performance curves generally look similar to typical throughput curves in other systems; that is, as the record size increased, so did the performance. As in most systems, eventually a plateau was reached, and further increases in the record size did not result in further performance increases. The precise location of this plateau varied between patterns and CP:IOP ratios. As in most systems, when accessing data in small pieces, the total throughput was limited by software overhead and by the high latency of transferring data across a network, regardless of the access pattern.

When reading data in large chunks, the access pattern had a greater effect on the performance. Under the partitioned and interleaved access patterns, for small numbers of IOPs, the bottleneck appeared to be the speed of the disk. The partitioned pattern was limited to 50% of the peak throughput, due to excessive seeking between file regions, while the interleaved pattern was limited only by the disks' sustainable throughput.

When the system had 32 or 64 IOPs, however, the performance with large requests was greatly affected by the network. The CPs were clearly capable of handling large amounts of data, but when that data was being received from many sources at once, congestion drastically degraded overall performance. This degradation became evident at larger request sizes, as the chances and cost of congestion increased. It is not clear whether this congestion was an effect of the heavyweight TCP/IP protocol, or of the network hardware. More seriously, this contention caused the compute processors to become unsynchronized; their requests then arrived at each IOP widely separated in time, which seriously degraded the effectiveness of the disk scheduler. Indeed, this effect is inherent in the structure of the traditional interface and could also occur when the CPs are unsynchronized for other reasons (e.g., if a CP's I/O is interspersed with computation). The disk scheduler's ineffectiveness caused extra seeks in the partitioned pattern, and missed disk rotations and thrashing of the disk's buffer in the interleaved pattern. In an attempt to reduce this con-

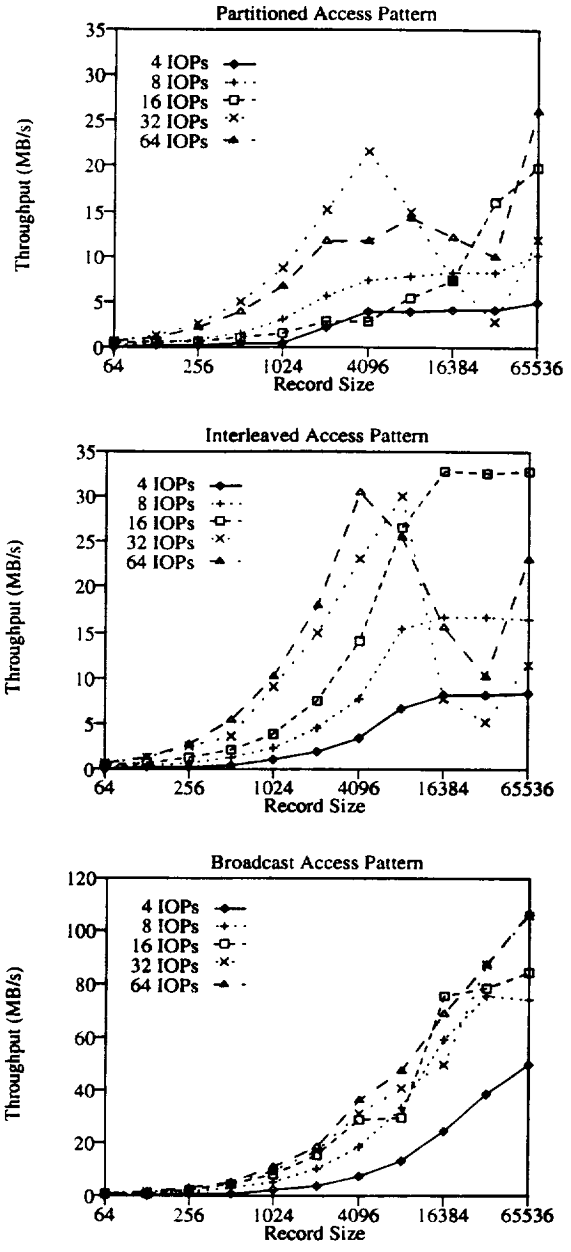

Figure 3: Throughput for read requests using the traditional Unix-like interface. There were 16 CPs in every case. Note the different scales on the y-axis.

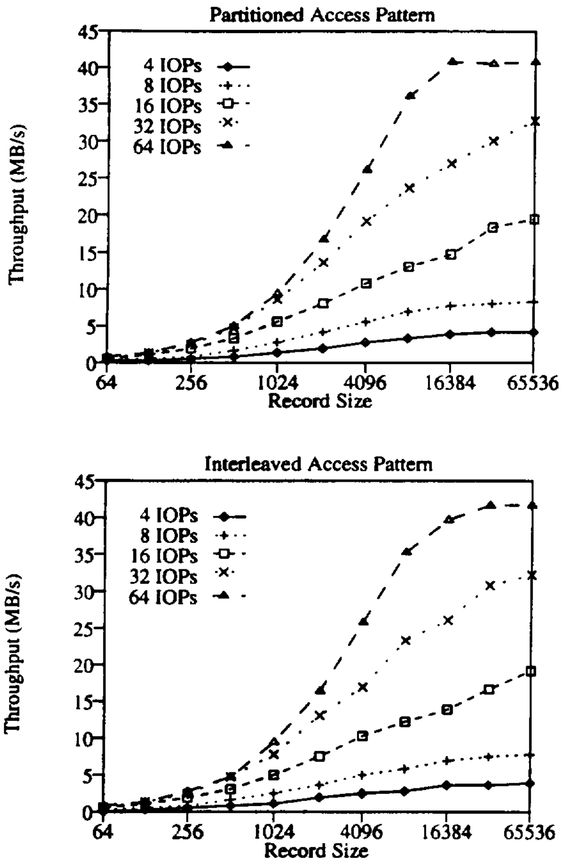

Figure 4: Throughput for write requests using the traditional Unix-like interface. There were 16 CPs in every case.

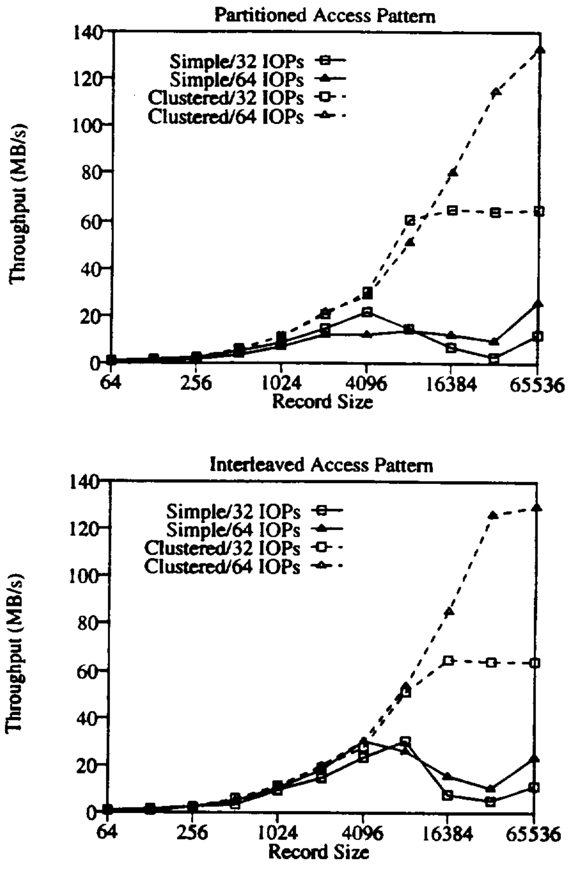

Figure 5: Throughput for read requests using both a simple strategy and a clustering strategy. There were 16 CPs in every case.

tention, we experimented with a *clustering* strategy. Rather than having all 16 CPs attempt to access all 32 or 64 IOPs simultaneously, we had the first 4 CPs access the first 16 IOPs, and so on. When a CP finished reading all the data from its first *cluster*, it began reading data from the next cluster. Nitzberg experimented with a similar strategy on CFS to reduce contention for cache space on the IOP [Nit92]. Figure 5 shows the results of our clustering experiment. Clearly, reducing the number of active sockets reduced the congestion at each CP, and improved overall performance.

Under the broadcast access pattern, data was read from the disk once, when the first compute processor requested it, and stored in the IOP's cache. When subsequent CPs requested the same data, it was retrieved from the cache rather than the disk. Since each piece of data was used many times, the cost of accessing the disk was amortized over a number of requests, and the limiting factors were software and network overhead. Again, network contention affected performance for large numbers of IOPs, but since every CP accessed exactly the same disk blocks, there was no further degradation caused by poor disk performance. As a result, when broadcasting we merely see a slower rate of increase for large numbers of IOPs rather than an actual reduction in performance.

When writing data, the access pattern appeared to have less of an impact on performance. While the write performance for a partitioned pattern was comparable to the read performance on the same pattern, the performance on interleaved patterns was significantly lower than when reading. This difference in performance was primarily caused by Galley's write protocol. Reading data is a simple process: when a compute processor wants to read data, it issues a request to an I/O processor, and waits for the data to be transferred. Writing is more complicated: when a compute processor wants to write data to a block, it sends a request to the I/O processor, waits for an 'ack', and only then begins sending the data. This ack is used to ensure that the I/O processor has space in its file cache to receive the incoming data. Writing is particularly expensive for requests smaller than the file system's block size. When an IOP is asked to write a single record to a block, the whole block must be read from disk before we can write the new, small piece of data. Reading this block increases the amount of disk I/O, leaving less bandwidth for the application.

Network contention was not a significant issue when writing data. When reading data, the bottleneck discussed above was caused by contention at the receiving side of a network connection. In this case, the request-response write protocol functions as a form of flow control; an IOP will not request more data than it is able to handle.

4.4 Strided Interface

When reading data with a traditional interface, in many cases we were able to achieve about 95% of the disks' peak sustainable performance. This best-case performance seems respectable, but our performance with small record sizes was certainly less than satisfactory. The goal of our new interfaces is to provide high performance for the whole range of record sizes, with particular emphasis on providing high throughput for small records. As described in [NK96], our higher-level interfaces are essentially different faces on the same underlying mechanism, and the performance of one is indicative of the performance of the others. The tests in this section were again performed by issuing asynchronous requests to each fork. Rather than issuing a series of single-record requests to each IOP, we used the strided interface to issue only a single request to each IOP. That single request identified all the records that should be transferred to or from that IOP. All other experimental conditions were identical to those in the previous section.

Figure 6 shows the total throughput achieved when reading a file with various record sizes for each access pattern using the new interface, and Figure 7 shows corresponding results for writing. The most striking difference between these graphs and those for the traditional interface is that, in most cases we were able to achieve peak performance with records as small as 64 bytes—two or three orders of magnitude smaller than the request sizes required to achieve peak throughput using the traditional interface. Other than increased opportunities for intelligent disk scheduling, the primary performance benefit of our interface was a reduction in the number of messages, accomplished by *packing* small chunks of data into larger packets before transmitting them to the receiving node.

When using the strided interface to read or write an interleaved access pattern, or to write a partitioned access pattern, the maximum throughput increased slightly over the traditional interface for small numbers of IOPs. When reading a partitioned access pattern using the strided interface with a small number of IOPs, however, the peak throughput nearly doubled. This increase in peak performance for partitioned reads was a result of the IOP having complete information about every CP's access pattern. This information allowed the IOP to intelligently schedule tens or hundreds of disk accesses when the requests initially arrived. Using the traditional interface, each IOP was limited to arranging a schedule based on only a single request per CP.

Once again, network contention was a problem for large numbers of IOPs. Unlike in the traditional interface, the contention did not interfere with disk scheduling, because the strided interface provides complete information up front, enabling a perfect disk schedule. Unfortunately, the best disk schedule is often the worst network schedule, as in the partitioned pattern, where all IOPs first served CP 1, then CP 2, and so forth. A similar clustering strategy might improve performance here as well.

While it is clear that the strided interface allowed the file system to deliver much better performance, the throughput plots shown in Figures 6 and 7 present only part of the picture. Figure 8 shows the speedup of the strided-read interface over a traditional read interface, and Figure 9 shows similar results for the write interfaces.[2] When using either a partitioned or interleaved pattern, the strided interface

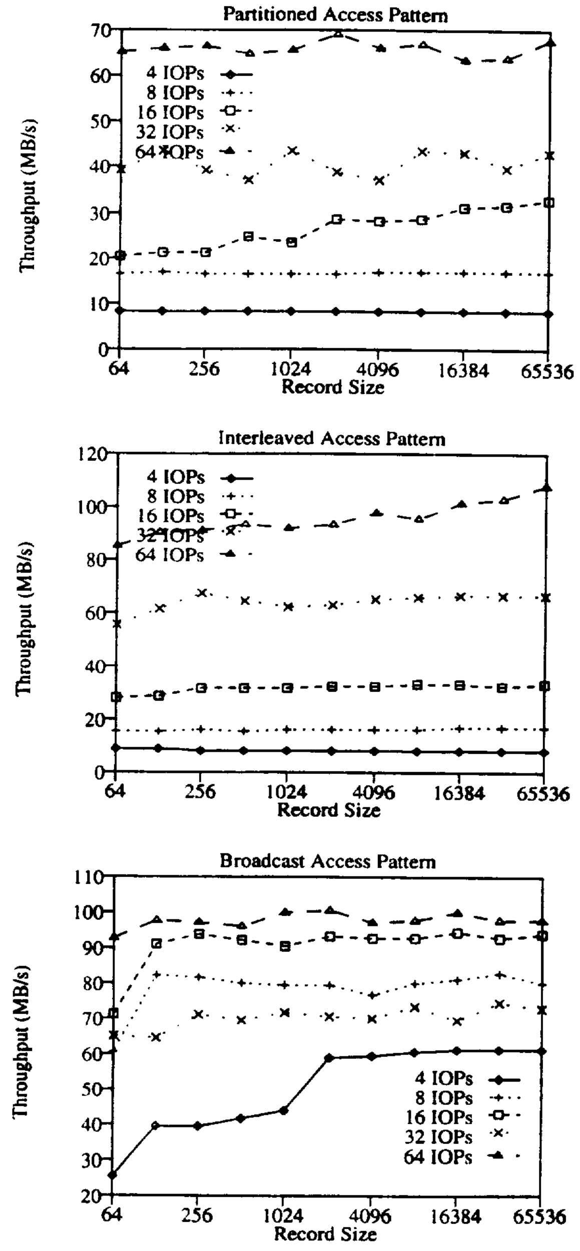

Figure 6: Throughput for read requests using the strided interface. There were 16 CPs in every case. Note the different scales on the y-axis.

read small records 55 to 95 times faster than the traditional interface, and wrote small records 22 to 55 times faster. Generally, the configurations with fewer IOPs experienced a greater increase in performance, due to the network contention described above. The broadcast-read pattern had the largest speedups for small records, ranging from 90 to 183. Although there was less room for improvement with large records, better disk scheduling in the partitioned-read pattern significantly improved some cases. Note that since each fork was contiguously laid out on disk, the speedup due to disk scheduling is lower than we would expect to see in a production system, where forks might be scattered across a

[2]These speedup results are based on the performance using the simple strategy—not the clustered IOP strategy.

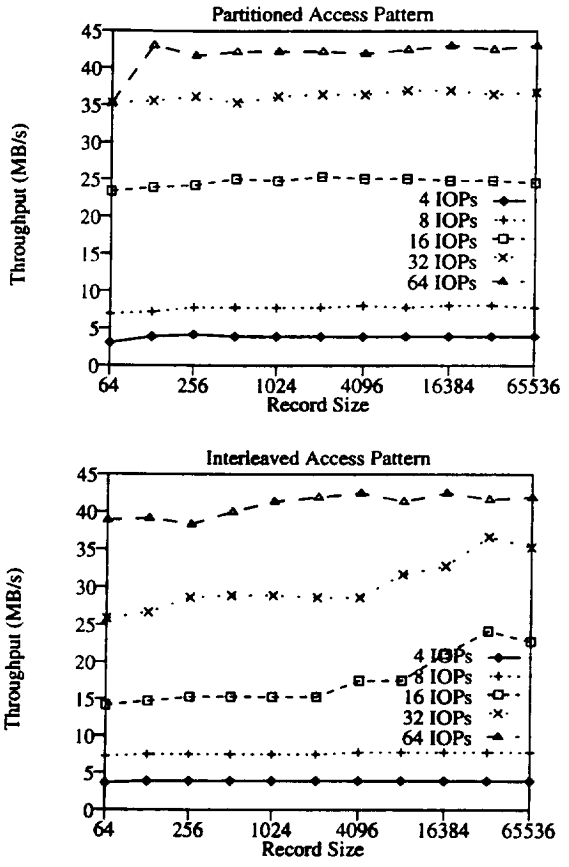

Figure 7: Throughput for write requests using the strided interface. There were 16 CPs in every case.

disk.

5 Related Work

Many different parallel file systems have been developed over the past decade. While many of these were similar to the traditional Unix-style file system, there have been also several more ambitious attempts.

Bridge, one of the earliest parallel file systems, has disks on every node — their model does not distinguish between CPs and IOPs. ridge provides both a traditional Unix-like interface, and a more complex interface that allows applications to explicitly access the local file systems on each node [Dib90].

Intel's Concurrent File System (CFS) [Pie89, Nit92], frequently cited as the canonical first-generation parallel file system, and its successor, PFS, are examples of file systems that provide a linear file model to the applications, and offer a Unix-like interface to the data. Other examples of this type of parallel file system are SUNMOS (and its successor, PUMA) [WMR+94], *sfs* [LIN+93], and CMMD [BGST93].

PPFS provides the end user with a linear file that is accessed with primitives that are similar to the traditional `read()/write()` interface. In PPFS, however, the basic transfer unit is an application-defined *record* rather than a byte. PPFS includes a number of predefined data distributions, which map the logical, linear stream of records to an

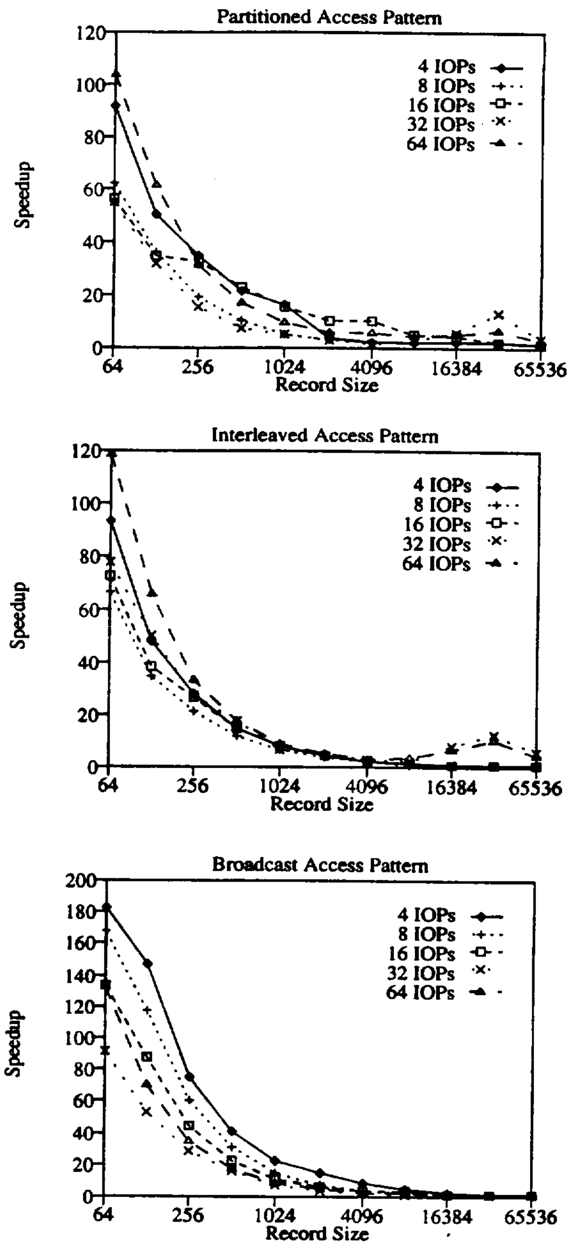

Figure 8: Increase in throughput for read requests using the strided interface. Note the different scales on the y-axis.

underlying `(disk, record)` pair, and also allows an application to provide its own mapping function.

The ELFS system [GP91] and the Hurricane File System [Kri94] provide object-oriented interfaces. These interfaces allow library designers to implement complex functionality (e.g., transparent replication of data, application-specific caching algorithms) in their files, but to hide that complexity from end users.

The Vesta file system, and its commercial version, PIOFS, address some of the same issues as Galley [CFP+95]. Most importantly, both recognize that data structures stored in a single file on disk are likely to be partitioned across multiple processes in a parallel application, and that new

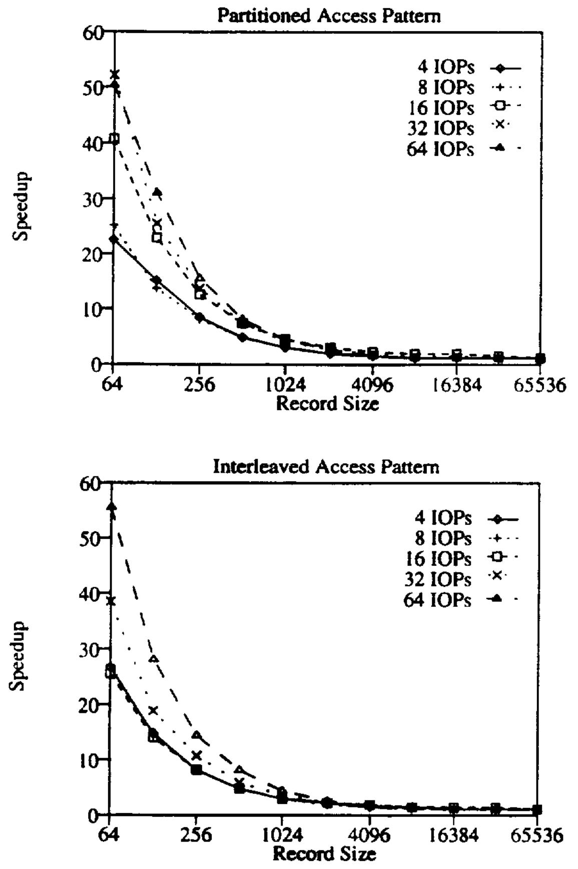

Figure 9: Increase in throughput for write requests using the strided interface.

interfaces are required to express this partitioning. Galley was designed as a bottom-up approach to the problem, by examining which access patterns are actually being used by applications, and supporting those patterns efficiently without regard to the higher-level semantics of those patterns. Vesta adopts a top-down approach. Vesta begins with the assumption that all shared data structures can be represented as a rectangular array, and allows the application to describe how the array should be partitioned across the processors. This high-level description gives Vesta much the same information as Galley's interfaces. Indeed, a Vesta-style interface could be easily implemented on top of Galley's low-level primitives.

While Vesta's approach offers many of the benefits of Galley's interfaces, it also has several limitations. The first is that there is no easy way to work with irregular data structures under Vesta. Unless your data can be mapped onto a rectangular array, you cannot make use of Vesta's partitioning schemes. Second, Vesta's partitioning schemes do not allow for irregular partitioning. Even if your data can be fit into a rectangular model, Vesta only allows the data to be partitioned into regularly-distributed, rectangular sub-blocks of a single size. Examples in [CFP+95] illustrate both the flexibilty and limitations of Vesta's approach to partitioning. Finally, Vesta does not provide an easy way for two processes to access overlapping regions of a file; each process's partition is strictly disjoint from every other process's partition. Since many models of physical events require logically adjacent nodes to share boundary information, this could be an important restriction. Indeed, we have observed that such overlapping file access is likely to occur in practice. Results in [NKP+95], show that most read-only files had at least some bytes that were accessed by multiple processors. We should note that the same results show that in many cases, the strictly disjoint partitioning offered by Vesta may match the applications' needs for write-only files.

In addition to full file systems, there are numerous interfaces that are designed to allow programmers to describe their I/O needs at a higher semantic level. These interfaces are sometimes tightly integrated into a particular language such as HPF [BGMZ92, HPF93] or CMF [Thi94]. There are also many language-independent libraries to support parallel I/O, usually to support distributed matrices [TBC+94, SW94]. The Jovian project explores the issues relating to the storage of irregular structures [BBS+94]. Finally, there are also plans to extend the MPI standard to include parallel I/O operations [MPI94, CFF+95].

These systems and their interfaces could all be considered candidates for implementation on top of Galley. Indeed, Galley is specifically designed as a low-level file system capable of supporting multiple high-level interfaces.

6 Summary

Based on the results of several workload characterization studies, we have designed Galley, a new parallel file system that attempts to rectify some of the shortcomings of existing file systems. Galley is based on a new three-dimensional structuring of files, which provides tremendous flexibility and control to applications and libraries. We show how Galley's strided I/O request reduced the aggregate latency of multiple small requests and allowed the file system to optimize the disk accesses required to satisfy the request.

The results of our experiments indicate that our new style of interface increased performance by several orders of magnitude. More importantly, this new interface allows high performance on access patterns that are known to be common in scientific applications, and which are known perform poorly on most current parallel file systems.

Future Work. This performance study reveals several areas for further work. First, Galley currently only achieves about 50% of the potential throughput for write operations. While it is not surprising that write performance should lag read performance, a factor of 2 seems excessively slow. Furthermore, Galley currently supports only a single disk per IOP. Since our maximum throughput is frequently limited by the disk's maximum throughput, adding support for multiple disks at the IOP is a high priority. Finally, we have only examined the performance of the system running microbenchmarks. To really understand Galley's performance, we plan to study how real applications perform on Galley.

Availability. Although Galley is still alpha-quality software, there are several projects underway to implement libraries, applications, and a compiler on top of it. We hope that these projects will help identify weak points in our implementation, and lead to a more robust system. We anticipate making Galley publicly available in the near future.

References

[BBH95] Sandra Johnson Baylor, Caroline B. Benveniste, and Yarson Hsu. Performance evaluation of a parallel I/O architecture. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 404–413, Barcelona, July 1995.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994. ftp://hpsl.cs.umd.edu/pub/papers/splc94.ps.Z.

[BGMZ92] Peter Brezany, Michael Gernt, Piyush Mehotra, and Hans Zima. Concurrent file operations in a High Performance FORTRAN. In *Proceedings of Supercomputing '92*, pages 230–237, 1992.

[BGST93] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[CFF+95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on I/O in Parallel and Distributed Systems*, pages 1–15, Santa Barbara, CA, April 1995.

[CFP+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, pages 222–248, 1995.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

[GP91] Andrew S. Grimshaw and Jeff Prem. High performance parallel file objects. In *Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[HP91] Hewlett Packard. *HP97556/58/60 5.25-inch SCSI Disk Drives Technical Reference Manual*, second edition, June 1991. HP Part number 5960-0115.

[HPF93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993. http://www.erc.msstate.edu/hpff/report.html.

[IBM94] IBM. *AIX Version 3.2 General Programming Concepts*, twelfth edition, October 1994.

[KFG94] John F. Karpovich, James C. French, and Andrew S. Grimshaw. High performance access to radio astronomy data: A case study. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 240–249, September 1994. Also available as UVA TR CS-94-25.

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[KR94] Thomas T. Kwan and Daniel A. Reed. Performance of the CM-5 scalable file system. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 156–165, July 1994.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, University of Toronto, October 1994.

[KTR94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[MK91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.0 edition, May 5 1994. http://www.mcs.anl.gov/Projects/mpi/standard.html.

[NAS94] NASA/Science Office of Standards and Technology, NASA Goddard Space Flight Center, Greensbelt, MD 020771. *A User's Guide for the Flexible Image Transport System (FITS)*, 3.1 edition, May 1994.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[NK95] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *IPPS '95 Workshop on I/O in Parallel and Distributed Systems*, pages 47–62, April 1995.

[NK96] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996. To appear.

[NKP+95] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. Technical Report PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. Submitted to IEEE TPDS.

[OCH+85] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[PEK+95] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[PFDJ89] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[PP93] Barbara K. Pasquale and George C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, 1993.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[SW95] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

[TBC+94] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[Thi94] Thinking Machines Corporation, Cambridge, Mass. *CM Fortran User's Guide*, 2.1 edition, January 1994.

[WGRW93] David Womble, David Greenberg, Rolf Riesen, and Stephen Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.

[WMR+94] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65, 1994.