The Galley Parallel File System

Nils Nieuwejaar, David Kotz

{nils,dfk}@cs.dartmouth.edu

Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510

Most current multiprocessor file systems are designed to use multiple disks in parallel, using the high aggregate bandwidth to meet the growing I/O requirements of parallel scientific applications. Many multiprocessor file systems provide applications with a conventional Unix-like interface, allowing the application to access multiple disks transparently. This interface conceals the parallelism within the file system, increasing the ease of programmability, but making it difficult or impossible for sophisticated programmers and libraries to use knowledge about their I/O needs to exploit that parallelism. In addition to providing an insufficient interface, most current multiprocessor 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 scientific multiprocessor workloads. We discuss Galley's file structure and application interface, as well as the performance advantages offered by that interface.

Key words: Parallel I/O. Multiprocessor file system. Performance evaluation. IBM SP-2. Scientific Computing.

1 Introduction

While the speed of most components of massively parallel computers have been steadily increasing for years, the I/O subsystem has not been keeping pace. Hardware limitations are one reason for the difference in the rates of performance increase, but the slow development of new multiprocessor file systems is also to blame. One of the primary reasons that multiprocessor file-system performance has not improved at the same rate as other aspects of multiprocessors is that, until recently, there has been limited information available

⋆ This research was funded by NSF under grant number CCR-9404919 and by NASA Ames under agreement numbers NCC 2-849 and NAG 2-936.

about how applications were using existing multiprocessor file systems and how programmers would like to use future file systems.

Several recent analyses of production file-system workloads on multiprocessors running primarily scientific applications show that many of the assumptions that guided the development of most multiprocessor file systems were incorrect [12,18,25]. It was generally assumed that scientific applications designed to run on a multiprocessor would behave in the same fashion as scientific applications designed to run on sequential and vector supercomputers: accessing large files in large, consecutive chunks [23,24,15,16]. Studies of two different multiprocessor file-system workloads, running a variety of applications in a variety of scientific domains, on two architectures, under both data-parallel and control-parallel programming models, show that many applications make many small, regular, but non-consecutive requests to the file system [20]. These studies suggest that the workload that most multiprocessor file systems were optimized for is very different than the workloads they are actually being asked to serve.

Using the results from these two workload characterizations and from performance evaluations of existing multiprocessor file systems, we have developed a new multiprocessor file system called Galley. Galley is designed to deliver high performance to a variety of parallel, scientific applications running on multiprocessors with realistic workloads. Rather than attempting to design a file system that is intended to directly meet the specific needs of every user, we have designed a simpler, more general system that lends itself to supporting a wide variety of libraries, each of which should be designed to meet the needs of a specific community of users.

The remainder of this paper is organized as follows. In Section 2 we describe the specific goals Galley was designed to satisfy. In Section 3 we discuss a new, three-dimensional way to structure files in a multiprocessor file system. Section 4 describes the design and current implementation of Galley. Section 5 discusses the interface available to applications that intend to use Galley, and Section 6 shows how Galley's interface can improve an application's performance. In Section 7 we discuss several other multiprocessor file systems, and finally in Section 8 we summarize and describe our future plans.

2 Design Goals

Most current multiprocessor file-system designs are based primarily on hypotheses about how parallel scientific applications would use a file system. Galley's design is the result of examining how parallel scientific applications actually use existing file systems. Accordingly, Galley is designed to satisfy

several goals:

- Allow applications and libraries to explicitly control parallelism in file access.
- Efficiently handle a variety of access sizes and patterns.
- Be flexible enough to support a wide variety of interfaces and policies, implemented in libraries.
- Allow easy and efficient implementations of libraries.
- Be scalable enough to run well on multiprocessors with dozens or hundreds of nodes.
- Minimize memory and performance overhead.

  Galley is targeted at distributed memory, MIMD machines such as IBM's SP-2 or Intel's Paragon.

3 File Structure

Most existing multiprocessor file systems use a Unix-like file model [3,23,15]. Under this model, a file is seen as an addressable, linear sequence of bytes. Applications can issue requests to read or write data contiguous subranges of that sequence of bytes. A parallel file system typically *declusters* files (i.e., scatters the blocks of each file across multiple disks), allowing parallel access to the file. This parallel access reduces 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.

Galley uses a more complex file model that allows greater flexibility, which should lead to higher performance.

3.1 Subfiles

The linear file model offered by most multiprocessor file systems can give good performance when the request size generated by the application is larger than the declustering unit size, as a single request will involve data from multiple disks. Under these conditions, the file system can access multiple disks in parallel, delivering higher bandwidth to the application, and possibly hiding any latency caused by disk seeks. The drawback of this approach is that most multiprocessor file systems use a declustering unit size measured in kilobytes (e.g., 4 KB in Intel's CFS [23]), but our workload characterization studies show that the typical request size in a parallel application is much smaller: frequently under 200 bytes [20]. This disparity between the request size and

the declustering unit size means that most of the individual requests generated by parallel applications are not being executed in parallel. In the worst case, the compute processors in a parallel application may issue their requests in such a way that all of an application's processes may first attempt to access disk 0 simultaneously, then all attempt to access disk 1 simultaneously, and so on.

Another drawback of the linear file model is that a dataset may have an efficient, parallel mapping onto multiple disks that is not easily captured by the standard declustering scheme. One such example is the two-dimensional, cyclically-shifted block layout scheme for matrices, shown in Figure 1, which was designed for SOLAR, a portable, out-of-core linear-algebra library [31]. This data layout is intended to efficiently support a wide variety of out-of-core algorithms. In particular, it allows blocks of rows and columns to be transferred efficiently, as well as square or nearly-square submatrices.

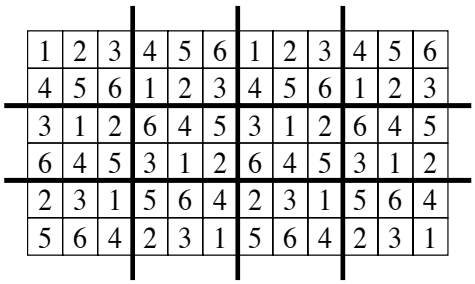

Fig. 1. An example of a 2-dimensional, cyclically-shifted block layout, as described in [31]. In this example there are 6 disks, logically arranged into a 2-by-3 grid, and a 6-by-12 block matrix. The number in each square indicates the disk on which that block is stored.

To avoid the limitations of the linear file model, Galley does not impose a declustering strategy on an application's data. Instead, Galley provides applications with the ability to fully control this declustering according to their own needs. This control is particularly important when implementing *I/O-optimal algorithms* [8]. Applications are also able to explicitly indicate which disk they wish to access in each request. To allow this behavior, files are composed of one or more *subfiles*, which may be directly addressed by the application. Each subfile resides entirely on a single disk, and no disk contains more than one subfile from any file. The application may choose how many subfiles a file contains when the file is created. The number of subfiles remains fixed throughout the life of the file.

The use of subfiles gives applications the ability both to control how the data is distributed across the disks, and to control the degree of parallelism exercised on every subsequent access. Of course, many application programmers will not want to handle the low-level details of data declustering, so we anticipate that most end-users will use a user-level library that provides an appropriate

declustering strategy.

3.2 Forks

Each subfile in Galley is structured as a collection of one or more independent *forks*. A fork is a named, addressable, linear sequence of bytes, similar to a traditional Unix file. Unlike the number of subfiles in a file, the number of forks in a subfile is not fixed; libraries and applications may add forks to, or remove forks from, a subfile at any time. The final, three-dimensional file structure is illustrated in Figure 2. There is no requirement that all subfiles have the same number of forks, or that all forks have the same size.

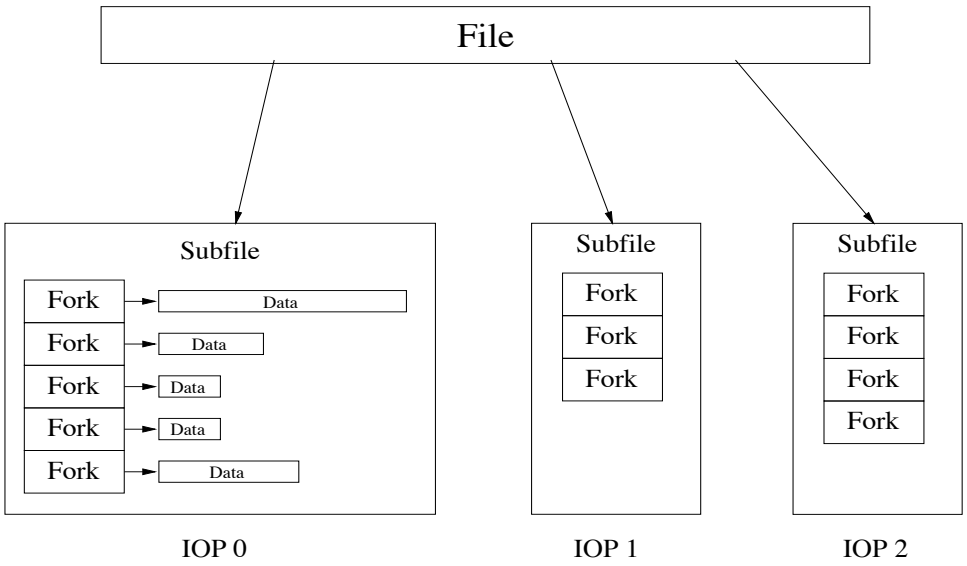

Fig. 2. Three-dimensional structure of files in the Galley File System. The portion of the file residing on disk 0 is shown in greater detail than the portions on the other two disks.

The use of forks allows further application-defined structuring. For example, if an application represents a physical space with two matrices, one containing temperatures and other pressures, the matrices could be stored in the same file (perhaps declustered across multiple subfiles) but in different forks. In this way, related information is stored logically together but may be accessed independently.

While typical application programmers may find forks helpful, they are most likely to be useful when implementing libraries. In addition to storing data in the traditional sense, many libraries also need to store persistent, library-specific 'metadata' independently of the data proper. One example of such a library would be a compression library similar to that described in [28], which compresses a data file in multiple independent chunks. Such a library

could store the compressed data chunks in one fork and index information in another.

Another instance where this type of file structure may be useful is in the problem of genome-sequence comparison. This problem requires searching a large database to find approximate matches between strings [1]. The raw database used in [1] contained thousands of genetic sequences, each of which was composed of hundreds or thousands of bases. To reduce the amount of time required to identify potential matches, the authors constructed an index of the database that was specific to their needs. Under Galley, this index could be stored in one fork, while the database itself could be stored in a second fork.

A final example of the use of forks is Stream*, a parallel file abstraction for the data-parallel language C* [17]. Briefly, Stream* divides a file into three distinct segments, each of which corresponds to a particular set of access semantics. While the current implementation of Stream* stores all the segments in a single file, one could use a different fork for each segment. In addition to the raw data, Stream* maintains several kinds of metadata, which are currently stored in three different files: `.meta`, `.first`, and `.dir`. In a Galley-based implementation of Stream*, it would be natural to store this metadata in separate forks rather than separate files.

Users of linear-file based file systems would generally use multiple files in the cases described above. Although that is certainly an option in Galley, forks provide two significant advantages. First, forks are lighter-weight entities than files. Second, forks allow libraries to hide metadata information safely. In a traditional file system, a library would either have to store its metadata directly in the file itself or in separate files. Storing the metadata in the data file has the side effect of making it difficult for other libraries and applications to get at the raw data. Storing the metadata separately from the data makes it easy for the data to become separated from the metadata, for example, if one of the files is moved or deleted. This approach can also lead to namespace collisions, as with two Stream* files each wanting to store their metadata in the `.meta`, `.first`, and `.dir` files.

4 System Structure

The Galley parallel file system is structured as a set of clients and servers. This model is based on the typical multiprocessor architecture that dedicates some processors to computation and dedicates the rest to I/O. In this system, the *Compute Processors* (CPs) function as clients and the *I/O Processors* (IOPs) act as servers.

4.1 Compute Processors

A client in Galley is simply any user application that has been linked with the Galley run-time library, and which runs on a compute processor. The run-time library receives file-system requests from the application, translates them into lower-level requests, and passes them (as messages) directly to the appropriate servers, running on I/O processors. The run-time library then handles the transfer of data between the I/O processors and the compute node's memory.

As far as Galley is concerned, every compute processor in an application is completely independent of every other compute processor. Indeed, Galley does not even assume that one compute processor is even aware of the existence of other compute processors. This independence means that Galley does not impose any communication requirements on a user's application. As a result, applications may use whichever communication software (e.g., MPI, PVM, P4) is most suitable to the given problem.

Like most multiprocessor file systems, Galley offers both blocking and non-blocking I/O. To simplify the implementation, and to avoid binding Galley too tightly to a single architecture, Galley originally used multithreading to implement non-blocking I/O. Unfortunately, most of the major communications packages cannot function in a multithreaded environment. As a result, Galley is currently forced to use signals to implement non-blocking I/O, using a TCP/IP communications substrate. If support for multithreaded environments ever becomes commonplace in message-passing packages, we will reexamine this decision.

Although applications may interact directly with Galley's interface, we expect that most applications will use a higher-level library or language layered on top of the Galley run-time library. One such library implements a Unix-like file model, which should reduce the effort required to port legacy applications to Galley [21]. Other libraries that have been implemented on top of Galley provide Panda [27,30] and Vesta [5] interfaces, as well as support for ViC*, a variant of C* designed for out-of-core computations [6,7].

4.2 I/O Processors

Galley's I/O servers are composed of several functional units, which are described in detail below. A high-level view of the internal structure of an IOP, which shows the paths of communication between the units, is shown in Figure 3. Each functional unit is implemented as a separate thread. Furthermore, each IOP also has one thread designated to handle incoming I/O requests

for each compute processor. This multithreading makes it easy for an IOP to service requests from many clients simultaneously.

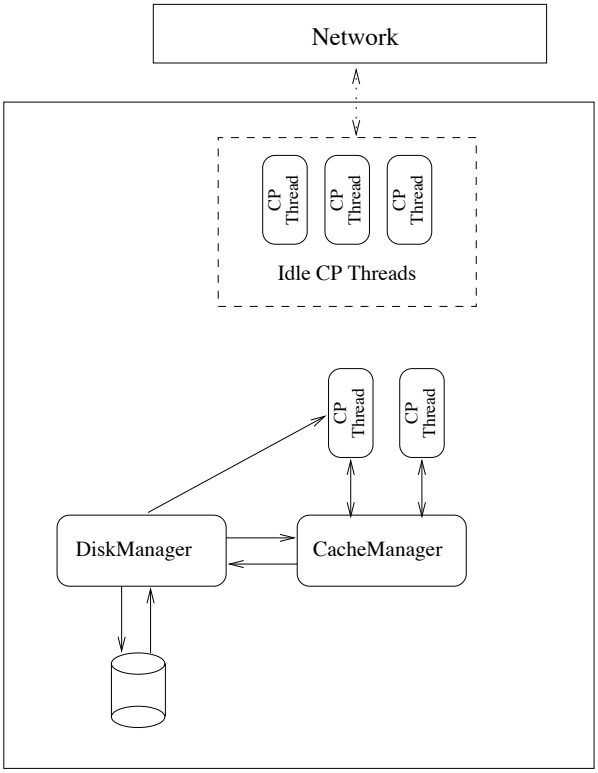

Fig. 3. High-level view of the internal structure of a Galley I/O Processor, showing the communication paths between the functional units. In this example, there two active requests waiting for data from the buffer cache or from disk, and three idle CP Threads waiting for new requests to arrive.

While one potential concern is that this thread-per-CP design may limit the scalability of the system, we have not observed such a limitation in the performance tests shown in Section 6. One may reasonably assume that a thread that is idle (i.e., not actively handling a request) is not likely to noticeably affect the performance of an IOP. By the time the number of active threads on a single IOP becomes great enough to hinder performance, the IOP will most likely be overloaded at the disk, the network interface, or the buffer cache, and the effect of the number of threads will be minor relative to these other factors. We intend to explore this issue further as we port Galley to different architectures, which may offer different levels of thread support.

4.2.1 CP Threads

CP Threads remain idle until a request arrives from the corresponding CP. After being awakened to service a new request, a CP Thread creates a list of all the disk blocks that will be required to satisfy the request. The CP Thread then passes the full list of blocks to the CacheManager, and waits on a queue

of buffers returned by the CacheManager and DiskManager. As a CP Thread receives buffers on its queue, it handles the transfer of data between its CP and those buffers. When a CP Thread completes the transfer of data to or from a buffer, it decreases that buffer's reference count, and handles the next buffer in the queue. When the whole request has been satisfied, or if it fails in the middle, the thread passes a success or failure message back to its CP, and idles until another request arrives.

The order in which a fork's blocks are placed on the CP Thread's buffer queue is determined by which blocks are present in the buffer cache and the order in which that fork's blocks are laid out on disk. As a result, it is not possible for Galley's client-side run-time library to know in advance the order in which an IOP will satisfy the individual pieces of a request. So, when reading, before the IOP can send data to the CP, it must first send a message indicating what data will be sent. Similarly, when writing, the IOP must send a message to the CP indicating which portion of the data the IOP is ready to receive. When writing, this approach is somewhat unusual in that the IOP is essentially 'pulling' the data from the CP, rather than the traditional model, where the CP 'pushes' the data to the IOP.

There is a further complication in transferring data between CPs and IOPs: packing. Rather than sending lots of small packets across the network, when possible Galley packs multiple small chunks of data into a larger packet, and sends the larger packet when it is full. This packing reduces the aggregate latency, and increases the effective data-transfer bandwidth. In the current implementation, the list of data chunks is precomputed on the CP, and the whole list is sent to the IOP. On our testbed systems, the speed of the network relative to the speed of the processors is high enough that sending the list across the network makes more sense than computing the list on both the CPs and the IOPs.

For simplicity, within a single packet the IOP will only pack chunks in the order they appear in the chunk list. If an out-of-order block is placed on a CP Thread's queue, the current packet is flushed, even if it is not full, and a new packet is started. An early implementation of Galley supported out-of-order packing within a packet, but that approach required that a fairly large packet of 'control' data be sent to the CP with each flushed packet. The current implementation is less flexible, but appears to have higher performance on our testbeds. On a system with a higher-bandwidth, lower-latency network, out-of-order packing might be more efficient, as the cost of the extra control data would be reduced.

4.2.2 CacheManager

Each IOP has a buffer cache that is maintained by the CacheManager. In addition to deciding which blocks are kept in the buffer, the CacheManager does all the work involved in locating blocks in the buffer cache for CP Threads. To perform these lookups, the CacheManager maintains a separate list of disk blocks requested by each CP Thread. When the CacheManager has outstanding request lists from multiple threads, it services requests from each list in round-robin order. This round-robin approach is an attempt to provide fair service to each requesting CP.

The CacheManager maintains a global Least-Recently-Used list of all the blocks resident in the cache. When a new block is to be brought into the cache, this list is used to determine which block is to be replaced. Providing applications with more control over cache policies is one area of ongoing work.

Rather than performing lookups by scanning through the entire LRU list, for efficiency the CacheManager also maintains a hash table, containing a list of all the blocks in the cache. For each disk block requested, the CacheManager searches its hash table of resident blocks. If the block is found, its reference count is increased, and a pointer to that buffer is added to the requesting thread's ready queue. If the block is not resident in the cache, the CacheManager finds the first block in the LRU list with a reference count of 0, and schedules it to be replaced by the requested block. The buffer is then marked 'not ready', and a request is issued to the DiskManager to write out the old block (if necessary), and to read the new block into the buffer.

4.2.3 DiskManager

The DiskManager is responsible for actually reading data from and writing data to disk. To increase portability, Galley does not use a system-specific low-level driver to directly access the disk. Instead, Galley relies on the underlying system (presumably Unix) to provide such services. Galley's DiskManager has been implemented to use raw devices, Unix files, or simulated devices as "disks". Galley's disk-handling primitives are sufficiently simple that modifying the DiskManager to access a device directly through a low-level device driver is likely to be a trivial task.

The DiskManager maintains a list of blocks that the CacheManager has requested to be read or written. As new requests arrive from the CacheManager, they are placed into the list according to the disk scheduling algorithm. The DiskManager currently uses a Cyclical Scan algorithm [29]. When using either simulated disks or raw devices, this disk scheduling helps deliver high performance. When the underlying storage medium is a Unix file, the layout of that file on disk is unrelated to the layout of data within Galley's file system, so

the DiskManager's scheduling is less likely to help performance.

When a block has been read from disk, the DiskManager updates the cache status of that block's buffer from 'not ready' to 'ready', increases its reference count, and adds it to the requesting thread's ready queue.

Galley's DiskManager does not attempt to prefetch data for two reasons. First, indiscriminate prefetching can cause thrashing in the buffer cache [22]. 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 below, 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.

5 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 are not sufficient to meet the needs of many parallel applications [18,20]. 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 either a simple-strided or nested-strided segment at regular intervals. Studies show that both simple-strided and nested-strided patterns are common in parallel, scientific applications [18,20].

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 the file system 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, providing the file system with this level of information allows it to make intelligent disk-scheduling decisions, leading to fewer disk-head seeks, and to better utilization of the disks' on-board caches.

The higher-level interfaces offered by Galley are summarized below. These interfaces are described in greater detail, and examples are provided, in [18,21]. Note that each request accesses data from a single fork; Galley has no notion of a file-level read or write request.

5.1 Simple-strided Requests

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

Beginning at *offset* in the open fork indicated by *fid*, the file system will read *quant* records, of *rec_size* bytes each. 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*, and 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 different than each other.

Naturally, there is a corresponding gfs_write_strided() call.

5.2 Nested-strided Requests

gfs_read_nested(int *fid*, void **buf*, long *offset*, long *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*.

5.3 Nested-batched requests

gfs_read_batched(int *fid*, void **buf*, struct gfs_batch **vec*, int *quant*);

While we found that most of the small requests in the observed workloads were part of either simple-strided or nested-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. One example of such an application is given in [21]. For those applications, we provide a *nested-batched* interface. A nested-batched request is composed of one or more batched requests, each of which is described using the data structure shown in Figure 4.

```
struct gfs_batch {
    int32 f_off;                    /* File offset */
    int32 m_off;                    /* Memory offset */
    char f_absolute;                /* Is the file offset absolute? */
    char m_absolute;                /* Is the memory offset absolute? */
    char sub_vector;                /* Is the sub-request a vector? */
    int32 quant;                    /* Number of repetitions */
    int32 f_stride;                 /* File stride between repetitions */
    int32 m_stride;                 /* Memory stride between repetitions */
    int32 subvec_len;               /* Number of elements in subvec */
    union {
        int32 size;                 /* Size for simple request */
        struct gfs_batch *subvec; /* Vector of batch requests */
    } sub;
};
```

Fig. 4. Data structure involved in a nested-batched I/O request.

A single instance of this data structure essentially represents a single level in a nested-strided request. That is, with one gfs_batch structure, you can represent a "standard" request, a simple-strided request, or one level of nesting in a nested-strided request. Galley's batched interface allows an application to submit a vector of batched requests, which allows an application to submit a list of strided requests, a list of standard requests, a list of nested-strided requests, or arbitrarily complex combinations of those requests.

As with a nested-strided request, a batched request allows an application to specify that a particular pattern will be repeated a number of times, with a regular stride between each instance of the pattern. However, a nested-strided request requires that the repeated pattern be either a simple- or a nested-strided requests. The batched interface allows applications to repeat batched requests with a regular stride between them. Hence the name "nested-batched". This capability allows applications to repeat arbitrary access patterns with a regular stride.

A full gfs_read_batched() or gfs_write_batched() request will typically combine multiple gfs_batch structures into vectors, trees, vectors of trees, trees of

vectors, and so on. For example, a doubly-nested-strided request would be a two-level tree. The root of the tree would describe the outer level of striding, and that node's child would describe the inner level of striding. An application with two such strided requests could combine them into a single batched request. In that case, there would be a vector of two trees, and each tree would have two levels.

The first two elements in the data structure contain the initial file and memory offsets of the request. The second two elements of the data structure indicate whether these offsets are specified absolutely (as is done with all other Galley requests), or relatively. If the offsets are relative, and if the request is the first element in a new vector, these offsets are specified relative to the offset of that vector's parent. Otherwise, a relative offset is specified relative to the offset of the previous element in the vector.

The fifth element in the structure (char *sub_vector*) indicates whether the pattern to be repeated is a simple data request or another batch vector. The sixth element (*quant*) indicates how many times the pattern should be repeated. The next two elements contain the strides that should be applied to the file and memory offsets between repetitions of the pattern. The ninth element in the structure only applies when the pattern to be repeated is a batched request. In that case, it indicates how many elements are in the sub-request.

Finally, the sub-request is described. The sub-request can be a simple data transfer (in the case of a standard or a simple-strided request), or it can be a vector of gfs_batch structures (in the case of a nested-strided, or more complex request).

An example of when this interface is useful is shown in [21].

5.4 List Requests

Finally, in addition to these structured operations, Galley provides a simple, more general file interface, called the *list* interface, which has functionality similar to the POSIX *lio_listio()* interface [11]. This interface allows an application to simply specify an array of (file offset, memory offset, size) triples that it would like transferred between memory and disk. 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 transfer between CPs and IOPs.

6 Performance

Most studies of multiprocessor file systems have focused primarily on the systems' performance on large, sequential requests. Indeed, most do not even examine the performance of requests of fewer than many kilobytes [22,2,14]. As discussed earlier, multiprocessor file-system workloads frequently include many small requests. This disparity between the observed and benchmarked workloads 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.

6.1 Experimental Platform

The Galley Parallel File System was designed to be easily ported to a variety of workstation clusters and massively parallel processors. The results presented here were obtained on the IBM SP-2 at NASA Ames' Numerical Aerodynamic Simulation facility. This system had 160 nodes, each running AIX 4.1.3, but only 140 were available for general use. Each node had a 66.7 MhZ POWER2 processor and at least 128 megabytes of memory. Each node was connected to both an Ethernet and IBM's high-performance switch. While the switch allowed 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 multi-threaded environment. Furthermore, neither MPL nor MPI allow applications to be implemented as persistent servers and transient clients. As a result of these limitations, and to improve portability, Galley was implemented on top of TCP/IP.

6.1.1 TCP/IP Performance

To determine what effect, if any, our use of TCP/IP would have on the overall performance of our system, we benchmarked the SP-2's TCP/IP performance. According to IBM, and verified by our own testing, the maximum TCP/IP throughput between two nodes on the SP-2 is approximately 17 MB/s. Unfortunately, as the number of communicating nodes increases, they are unable to maintain this throughput at each node, as shown in Figure 5.

For each test shown in that figure, we used 16 *sinks*, and varied the number of *sources* from 4 to 64. For a given test, each source sent the same amount of data to each sink, in a series of messages, using a fixed record size. For each sink/source configuration, we measured the throughput for a variety of message sizes. As the throughput ranged over several orders of magnitude, we varied the total amount of data transferred as well, from 1.5 MB with 4 sources

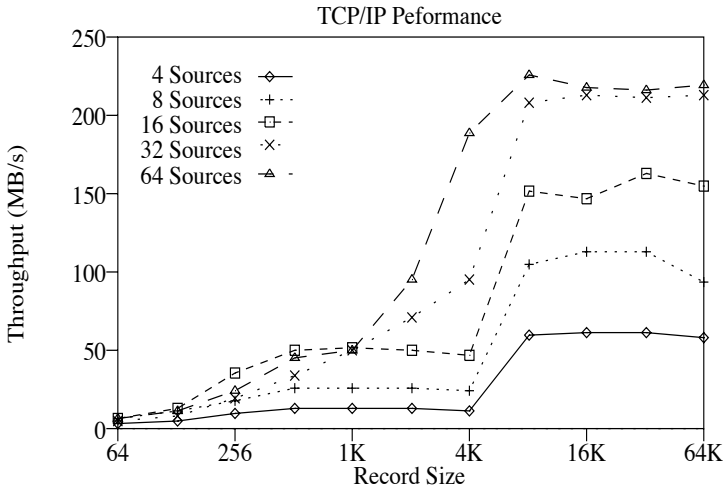

Fig. 5. Measured TCP/IP throughout on the SP-2. For each test, there were 16 *sinks* (similar to CPs reading a file), and a variable number of *sources* (similar to IOPs servicing read requests).

and a 64-byte record size, to over 800 MB with 64 sources and a 64-kilobyte record size.

In each of these tests, we used *select()* to identify sockets with pending I/O, but we did not attempt to use any flow-control beyond that provided by TCP/IP. As the figure shows, the achieved maximum throughput increases with the number of sources, until the number of sources exceeds 32. Even with many sources, we are only able to achieve about 220 MB/s, or less than 14 MB/s at each sink.

6.1.2 Simulated Disk

Each IOP in Galley controls a single disk, logically partitioned into 32 KB blocks. For this study, each IOP had a buffer cache of 24 megabytes, large enough to hold 750 blocks. Although each node on the SP-2 has a local disk, that disk must be accessed through AIX's Journaling File System. While Galley was originally implemented to use these disks, our performance results appeared to be inflated by the prefetching and caching provided by JFS. Specifically, we frequently measured apparent throughputs of over 10 MB/s from a single disk. To avoid these inflated results, we examined Galley's performance using 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 [9].

Our implementation of the disk model was based on earlier implementations [26,13][1]. Among the factors simulated by our model are head-switch

[1] The source code for this disk simulator is available online at `http://www.cs.dartmouth.edu/~nils/disk.html`, and is distributed with

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.[2] Comparing the results of this trace-driven simulation with the measured results from the actual disk, we obtained a demerit figure (see [26] 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 pool of buffers, which is used to store 'important' data. When the disk simulation thread copies data to or from a buffer, the amount of time required to complete the copy (which we calculate at system startup) 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.

6.2 Access Patterns

We examined the performance of Galley under several different access patterns, shown in Figure 6, each of which is composed of a series of requests for fixed-size pieces of data, or records. Although 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, as described above. Our experiments used a file that contained a subfile on each IOP, and a single fork within each subfile. To allow us to better understand the system's performance, by removing one variable, each fork was laid out contiguously on disk. The patterns shown in Figure 6 reflect the patterns that we access *from each fork*, and hence, from each IOP. The correspondence between the file-level patterns observed in actual applications, and the IOP-level access patterns used in this study, is discussed below.

The simplest access pattern is called *broadcast*. With this access pattern every compute node reads the whole file. In other words, 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 the initial state for a simulation. Since, to read all the data in a file, an

the Galley source code.

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

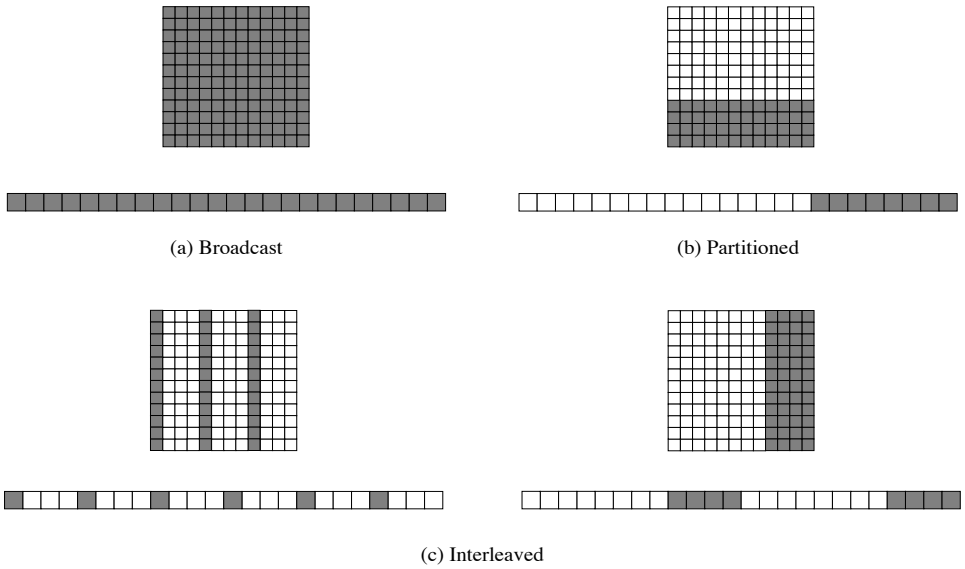

Fig. 6. The three access patterns examined in this study. Two views of each pattern are displayed: the pattern as applied to a linear file, and matrix distributions that could give rise to the pattern. For these examples, we assume that the matrices are stored on disk in row-major order. Each square corresponds to a single *record* in the file, and the highlighted squares represent the records accessed by a single compute node in a group of four.

application must read all the data in every subfile, a broadcast pattern at the file level clearly corresponds to a broadcast pattern at each subfile. Although it may seem counterintuitive for an application to access large, contiguous regions of a file in small chunks, we observed such behavior in practice [20]. 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. Another possible cause for such behavior is that the I/O was added to an existing loop as an afterthought. Since it seems unlikely that an application would want every node to rewrite the entire file, we did not measure the performance of the broadcast-write case.

Under a *partitioned* 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. There are two different ways a partitioned access pattern at the file level can map onto access patterns at the IOP level. The simpler mapping, which is not shown in the figure, occurs 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 only one compute processor. The other mapping, shown in the figure above, distributes

blocks of data across the disks in a CYCLIC fashion. This second mapping is more interesting and corresponds to the mapping used by 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, using the second mapping, 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.

In an *interleaved* pattern, each compute node requests a series of noncontiguous, but regularly spaced, records from a file. For the results presented here, the interleaving was based on the record size. That is, if 16 compute nodes were reading a fork 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. To see how this file-level pattern maps onto an IOP-level pattern, 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 this performance analysis, 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. 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 (reading 64-KB records from 64 IOPs). We performed each test five times. We disregarded the lowest and highest results, and present the average of the remaining three.

6.3 Traditional Interface

We first examined the performance of Galley using the standard read/write interface. This interface required each CP to issue separate requests for each record from each fork. Each CP issued asynchronous requests to all the forks, for a single record from each fork. When a request from one fork completed, a request for the next record from that fork was issued. By issuing asynchronous requests to all IOPs simultaneously, the CPs were generally able to keep all the IOPs in the system busy. Since each CP accessed its portion of each subfile sequentially, 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 for 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 frequently able to service requests from each node fairly, and were able to make good use of the disk.

Figure 7 shows the total throughput achieved when reading a file with various record sizes for each access pattern. Figure 8 presents similar results for write performance when overwriting an existing file, and Figure 9 shows Galley's performance when writing to a new file. The performance curves have the same general shape as throughput curves in most 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. Not surprisingly, when accessing data in small pieces, the total throughput was limited by a combination of software overhead and by the high latency of transferring data across a network, regardless of the access pattern.

The choice of access pattern had the greatest effect on performance when reading data with large blocks. When reading an interleaved pattern, the system's peak performance was limited by the sustainable throughput of the disks on each IOP (about 2.2 MB/s). Interestingly, there was a small dip in performance as the record size increased from 2 KB to 4 KB. With records of 2 KB or smaller, every CP reads data from every block. So, regardless of the order in which CPs' requests arrive at an IOP, that IOP reads all of the blocks in its fork, in order. With a record size of 4 KB, each CP reads data only from alternate blocks. As a result, it is possible for a request for block $n + 1$ to arrive before a request for block n, possibly causing a miss in the disk cache and an extra head seek, slightly degrading disk performance. Even more time was spent seeking when accessing data in a partitioned pattern. Indeed, with that pattern, the time spent seeking from one region of the file to another was

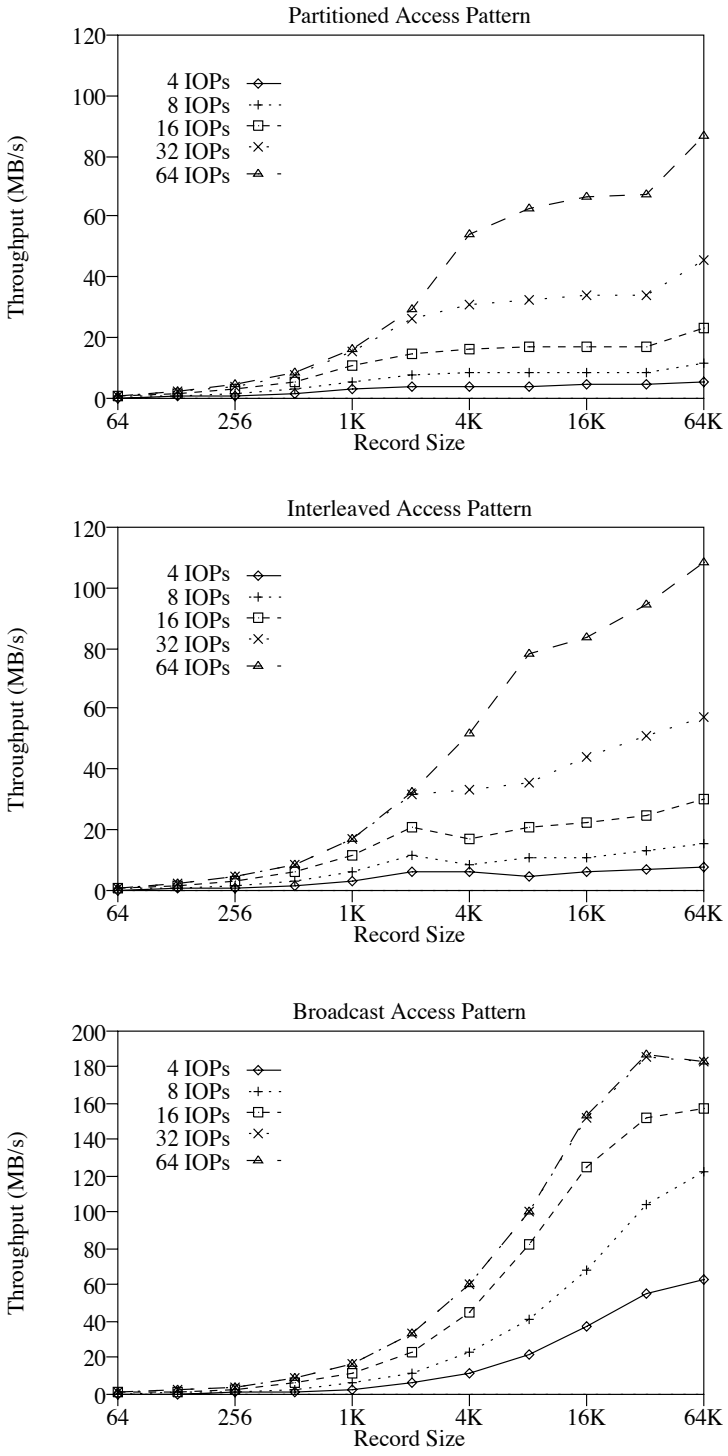

Fig. 7. Throughput for read requests using the traditional Unix-like interface. There were 16 CPs in every case. Note the different scales on the y-axis.

the limiting factor in the system's performance.

When testing an earlier version of Galley we found that with large numbers of IOPs, the network congestion at the CPs was so great that the CPs were un-

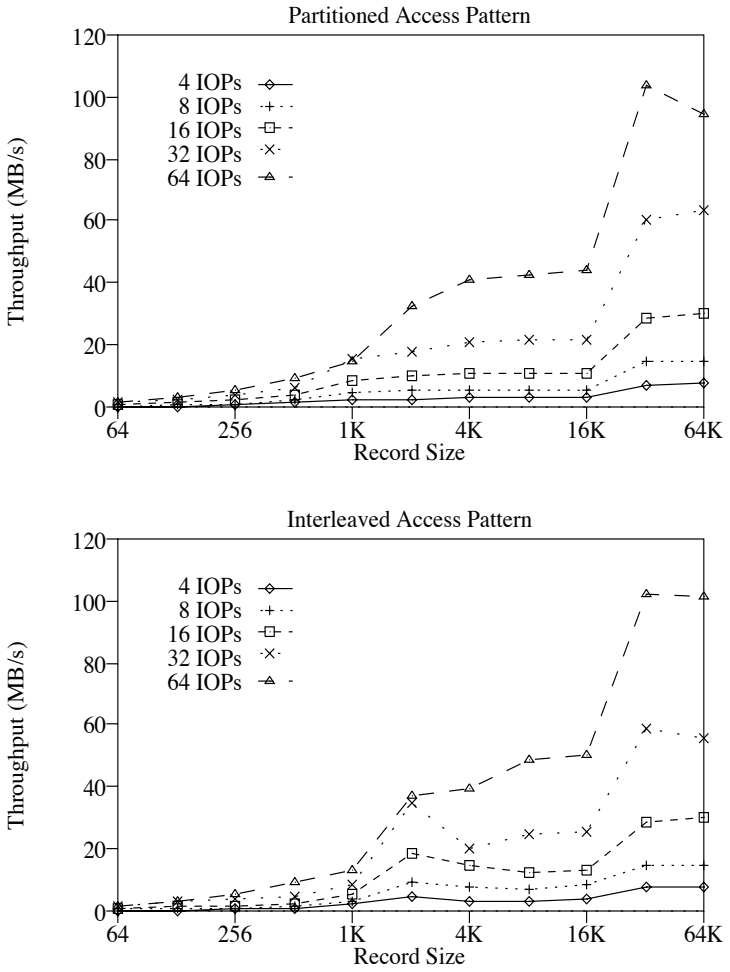

Fig. 8. Throughput for write requests using the traditional Unix-like interface when overwriting an existing file. There were 16 CPs in every case.

able to receive data and issue new requests to the IOPs in a timely fashion [19]. As a result, the fortuitous synchronization discussed above broke down, so the DiskManagers on the IOPs were unable to make intelligent disk scheduling decisions, causing excess disk-head seeks and thrashing of the on-disk cache. The combination of the network congestion and the poor disk scheduling led to dramatically reduced performance with large record sizes in the interleaved and partitioned patterns. To avoid this problem, we added a simple flow-control protocol to Galley's data-transfer mechanism. This flow control essentially requires an IOP to obtain permission from a CP before sending each chunk of data. By limiting the number of outstanding permissions, the CP can reduce or avoid this network congestion. Simple experiments on the SP-2 showed that choosing a limit between 2 and 8 led to the highest, and most consistent, performance. While this limit is currently a compile-time option, it may be worth exploring the possibility of allowing the CP to set it dynamically as well. All the experiments shown here used a limit of 2 outstanding permissions.

Under the broadcast access pattern, data was read from the disk once, when

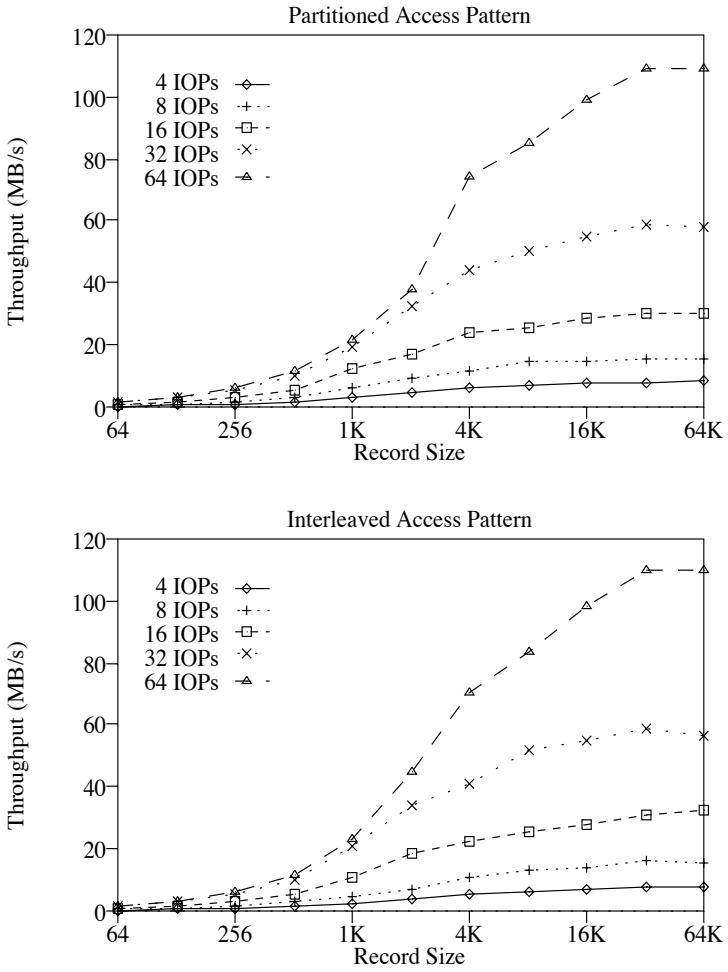

Fig. 9. Throughput for write requests using the traditional Unix-like interface when writing a new file. There were 16 CPs in every case.

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 from 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. In this case, the total throughput of the system was limited by the SP-2's TCP/IP performance, as discussed above.

We now consider Figure 8. When overwriting an existing file, and using records of less than 32 KB, the file system had to read each block off the disk before the new data could be copied into it. Without this requirement, any data that was stored in that block would be lost — even data that was not being modified by the write request. As a result, the system's performance was significantly slower when writing small records than when reading them. As when reading data, the interleaved pattern had the higher throughput because the partitioned pattern forced the disk to spend time seeking between one region of the file and another. The performance difference between the two

was smaller when writing since many of the disk accesses in the write case occurred at the end of the test, when the benchmark forced each IOP to write all dirty blocks to disk (with a `gfs_sync()` call). Since most of the disk accesses occurred at once, the DiskManager was able to schedule those accesses efficiently.

When the record size reached 32 KB, the write performance of both patterns increased dramatically. With the record size at least as large as the file system's block size, Galley did not have to read each data block off the disk before copying the new data in. Since the file system could simply write the new data to disk (rather than read-modify-write), the number of disk accesses in each pattern was cut in half.

We finally consider Figure 9. In these tests we measured the time to write data to a new file, rather than to overwrite an existing file. Note we did not use Galley's `gfs_extend()` call (which preallocates disk space for a fork) for these tests; new blocks were assigned to the fork on the fly, as it grew. Not only was writing to a new file generally faster than overwriting an existing file, in many cases it was faster than reading a file. For small requests, writing a new file was faster than overwriting an existing file because there was no need to read the original data off of disk. There is some additional overhead involved when writing a new file, as new blocks must be assigned to the file, but this cost was significantly less than the cost of the read-modify-write cycle. In those cases where writing a new file was faster than reading a file, the write tests benefited from the nearly perfect disk schedule during the `gfs_sync()` call, as discussed above.

6.4 Strided Interface

When reading data with a traditional interface, in many cases we were able to achieve nearly 100% of the disks' peak sustainable performance. This best-case performance seems respectable, but as with most systems, Galley's performance with small record sizes was certainly less than satisfactory. The goal of Galley's new interfaces is to provide high performance for the whole range of record sizes, with particular emphasis on providing high throughput for small records.

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 for the entire test. All other experimental conditions were identical to those in the previous section.

Figure 10 shows the total throughput achieved when reading a file with various record sizes for each access pattern using the new interface. Figure 11 shows corresponding results for overwriting an existing file and Figure 12 shows the results when writing to a new file.

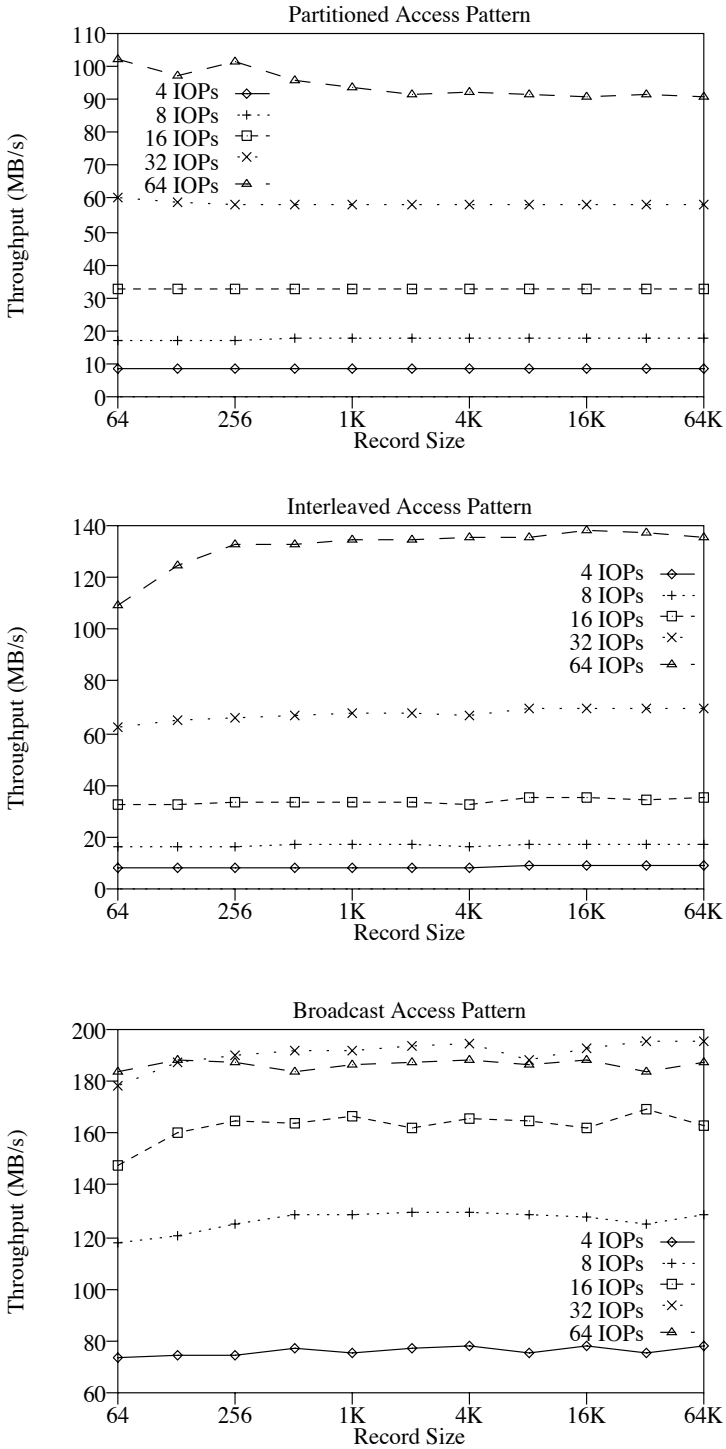

Fig. 10. Throughput for read requests using the strided interface. There were 16 CPs in every case. Note the different scales on the y-axis.

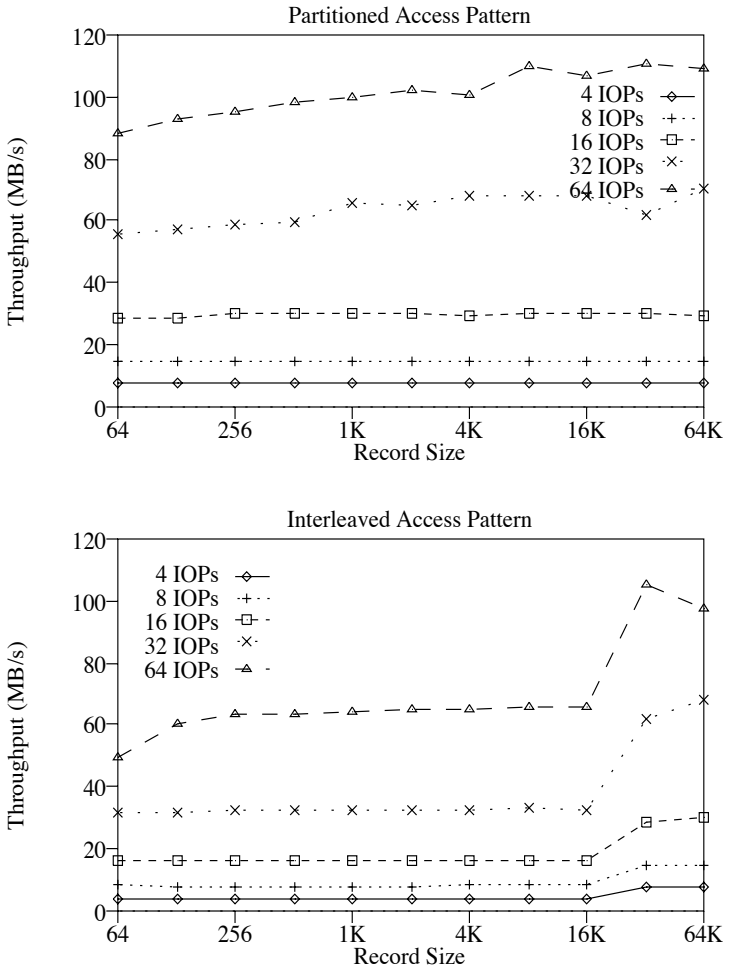

Fig. 11. Throughput for write requests using the strided interface when overwriting an existing file. There were 16 CPs in every case.

Given the traditional interface, the disk scheduler had to handle each request in the order they arrived from the CPs. This requirement led to excess disk-head movement, primarily in the partitioned pattern, but also in the interleaved pattern when the record size was larger than 2 KB (32 KB/16 CPs). Since all the CPs accessed the same disk blocks in the broadcast case, and in an interleaved pattern with small records, the disk schedule was optimal even with the traditional interface. Since many of the disk accesses in the traditional write cases occurred after a call to `gfs_sync()`, the disk scheduler was able to make intelligent decisions then as well. Therefore, the tests on which the new interface led to the greatest improvements in the disk schedule were the interleaved and partitioned read tests, and these were the two tests where the peak throughput to the CPs improved most dramatically.

Once again, network contention was a problem for large numbers of IOPs. The peak throughput on the broadcast pattern was limited to 13-14 MB/s to each CP. The best disk schedule can also be the worst network schedule, as in the partitioned pattern, where all IOPs first served CP 1, then CP 2, and so

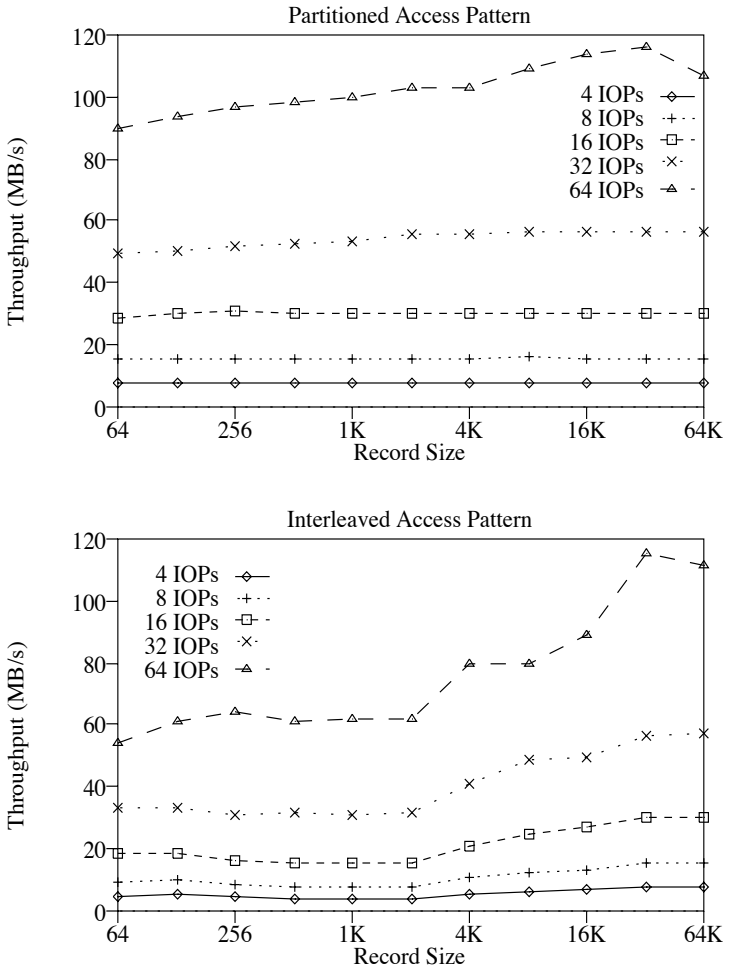

Fig. 12. Throughput for write requests using the strided interface when creating a new file. There were 16 CPs in every case.

forth. This disk schedule, combined with the limits of TCP/IP on the SP-2, contributed to the interleaved-read pattern having higher performance than the partitioned-read pattern using the strided interface.

While the increase in peak performance is interesting, the most striking difference between the two sets of tests is that, in most cases, Galley was 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 new 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.

The one case where Galley was not able to achieve maximum throughput with a small record size was in writing a new file in an interleaved pattern. When a CP Thread on an IOP receives the first request to write to a new fork, that

CP Thread *locks* the metadata for that fork. The CP Thread then examines the list of requests for the fork, and asks the DiskManager to assign however many blocks are necessary for the new file. Only after all the blocks have been assigned does the CP Thread *unlock* the fork's metadata, allowing the other CP Threads to start processing their requests. It appears that the delay caused by this long-term locking noticeably affects the system's throughput. This delay is less significant with the partitioned pattern because the number of requests is smaller; each CP has at most one request per block in the partitioned pattern, while they may have as many as 32 per block in the interleaved case.

While it is clear that the strided interface allowed the file system to deliver much better performance, the throughput plots shown in Figures 10 and 11 present only part of the picture. Figure 13 shows the speedup of the strided-read interface over a traditional read interface, and Figures 14 and 15 show similar results for the write interfaces, for both new files and overwriting preexisting files. When using an interleaved pattern with small records, the strided interface led to speedups of up to 98 times when reading, 30 times when overwriting an old file, and 23 times when writing a new file. There was a similar increase in performance for small records in a partitioned pattern: up to 92 times when reading, 56 times when rewriting, and 35 when writing a new file. The broadcast-read pattern had the largest speedups for small records, ranging from 150 to over 350.

Although there was less room for improvement with large records, better disk scheduling when reading interleaved and partitioned patterns occasionally led to higher performance even for large records. When reading, the minimum speedups within the range of record sizes we examined, were between 1 and 2, and occurred with the largest record sizes. When writing, the minimum speedups were mostly between .95 and 1.25. Again, the minimum speedups in the write tests were smaller than the read tests because much of the writing with the traditional interface was performed during the `gfs_sync()` call, so the IOP was able to perform more efficient disk scheduling.

7 Related Work

A variety of multiprocessor file systems have been developed over the past ten years or so. While many of these were similar to the traditional Unix-style file system, there have been also several more ambitious attempts.

Intel's Concurrent File System (CFS) [23,22], and its successor, PFS, are examples of multiprocessor file systems that use a linear file model and provide applications with a Unix-like interface. Both systems provide limited support

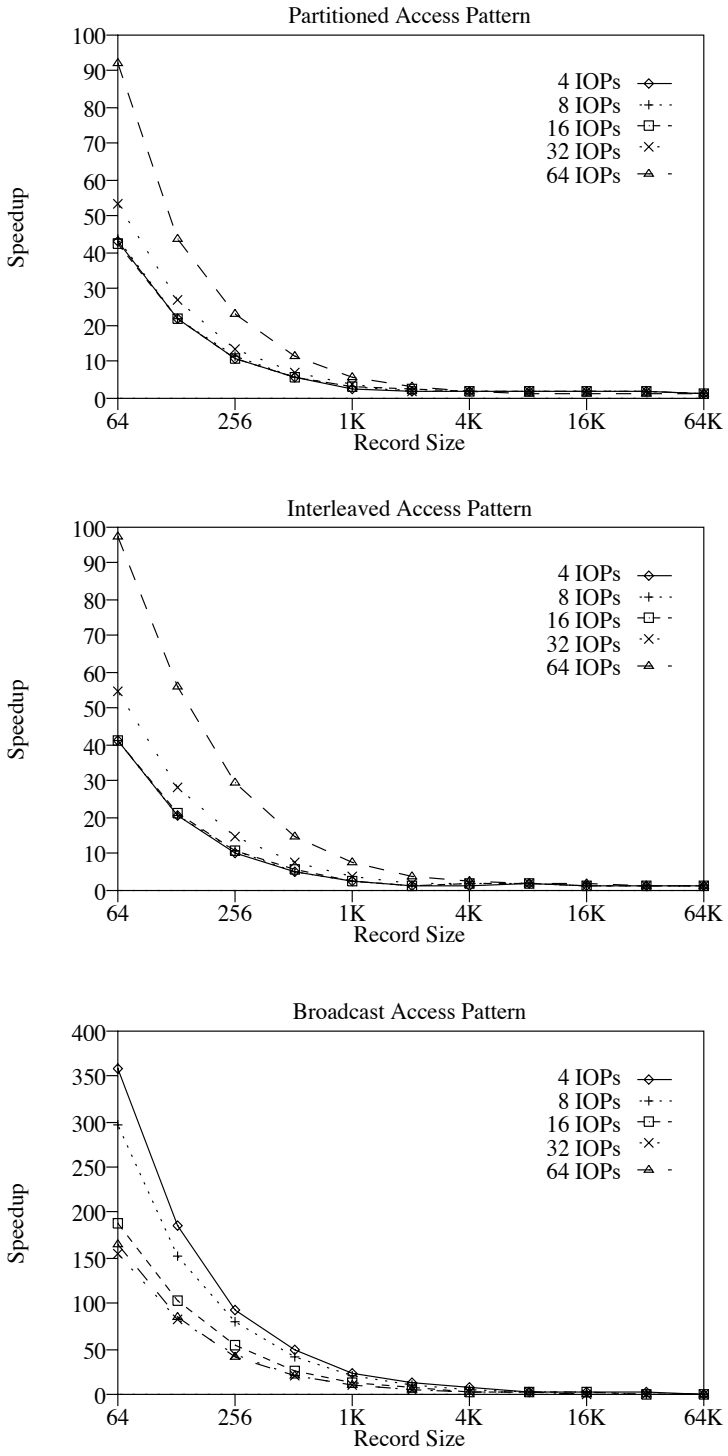

Fig. 13. Increase in throughput for read requests using the strided interface. Note the different scales on the y-axis.

to parallel applications in the form of *file pointers* that may be shared by all the processes in the application. CFS and PFS provide several *modes*, each of which provides the applications with a different set of semantics governing how the file pointers are shared. Other multiprocessor file systems with this

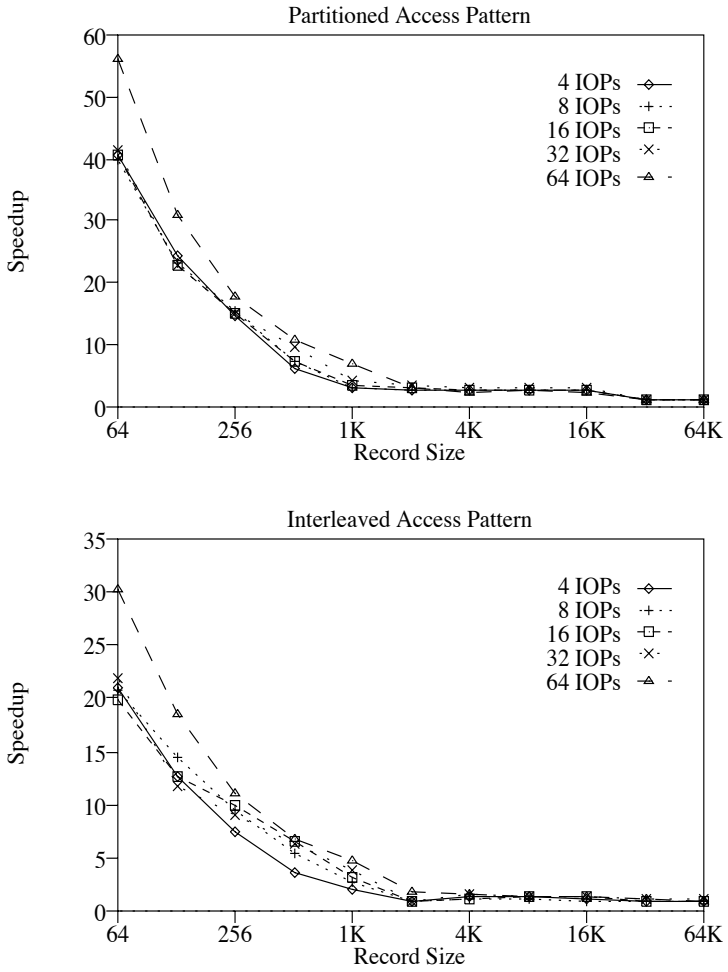

Fig. 14. Increase in throughput for write requests using the strided interface when overwriting an existing file.

style of interface are SUNMOS and its successor, PUMA [32], *sfs* [15], and CMMD [3].

Like the systems mentioned above, PPFS provides the end user with a linear file that is accessed with primitives that are similar to the traditional `read()/write()` interface [10]. In PPFS, however, the basic transfer unit is an application-defined *record* rather than a byte. PPFS maps requests against the logical, linear stream of records to an underlying two-dimensional model, indexed with a `(disk, record)` pair. Several different mapping functions, corresponding to common data distributions, are built into PPFS. An application is able to provide its own mapping function as well.

Ironically, the multiprocessor file system most removed from the traditional Unix-like model also provides the most Unix-like interface. PIOFS, the file system for IBM's SP-2, allows users and applications to interact with it exactly as they would interact with any AIX file system. Administrators and advanced users may also choose to interact with PIOFS's underlying parallel file system,

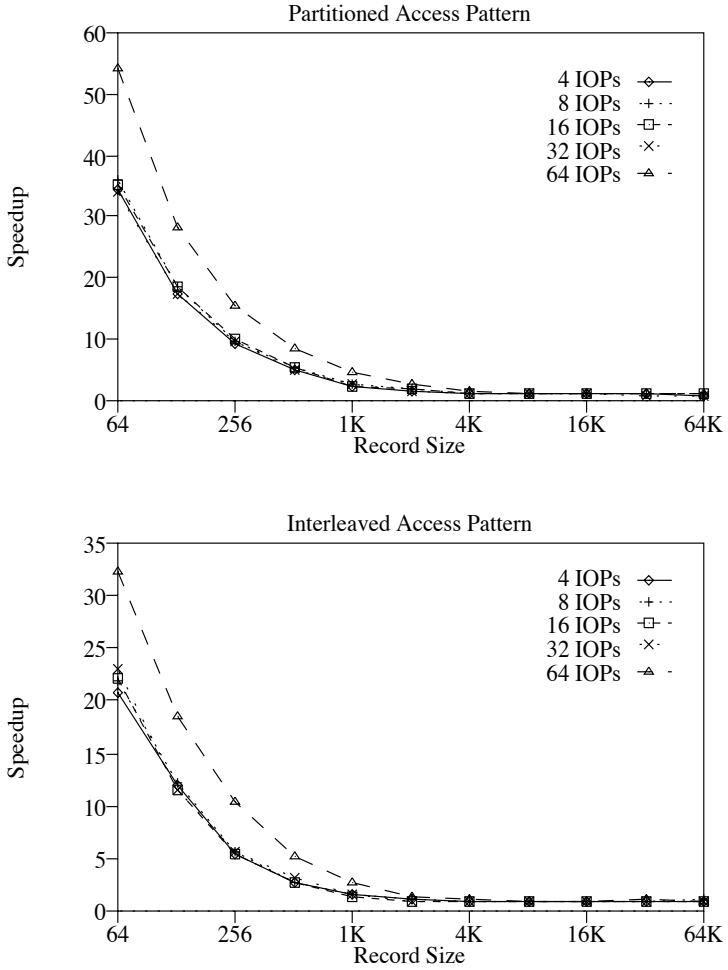

Fig. 15. Increase in throughput for write requests using the strided interface when creating a new file.

which is based on the Vesta file system [4,5]. Files in Vesta are two-dimensional, and are composed of multiple *cells*, each of which is a sequence of *basic striping units*. BSUs are essentially records, or fixed-sized sequences of bytes. Like Galley's subfiles, each cell resides on a single disk. While Galley only allows a file to have a single subfile per disk, in Vesta a single disk may contain many cells. Equivalent functionality could be achieved on Galley by mapping cells to forks rather than subfiles. Vesta's interface includes *logical views* of the data. These views are essentially rectangular partitionings of the two-dimensional file, and can provide the application with much of the functionality of Galley's strided interfaces. Vesta provides users with a different and powerful way of thinking about data storage. Its largest drawback is that it is ill-suited to datasets that cannot be partitioned into rectangular, non-overlapping sub-blocks of a single size. In addition to the functionality of Vesta, PIOFS provides applications with a Unix-like interface. We have built a library that provides a Vesta-like interface for Galley.

8 Summary and Future Work

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 have shown how Galley's higher-level I/O requests provide the file system with the information necessary to deliver high performance, particularly on those access patterns that are known to be common in scientific applications, and which are known perform poorly on most current multiprocessor file systems. This high performance was achieved by combining multiple small records into larger buffers before transferring them across the network, reducing the aggregate latency, and by allowing the file system to perform effective disk scheduling, reducing the amount of disk-head movement and making better use of the disks' on-board cache.

Future Work. We are exploring several areas for further work. First, 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. Second, we intend to examine how Galley performs when asked to service requests from multiple applications to multiple files at once. Finally, we intend to explore the issue of moving some of an application's I/O related code from the CP to the IOP. This functionality would allow applications to perform data-dependent filtering and distribution at the IOP, reducing the amount of data transferred over the network.

Availability. The source for the Galley parallel file system and the disk simulator used in this paper are all available at
`http://www.cs.dartmouth.edu/~nils/galley.html`.

References

[1] James W. Arendt. Parallel genome sequence comparison using a concurrent file system. Technical Report UIUCDCS-R-91-1674, University of Illinois at Urbana-Champaign, 1991.

[2] 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.

[3] 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.

[4] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[5] 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*, 34(2):222–248, January 1995.

[6] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[7] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. Technical Report PCS-TR96-293, Dartmouth College Department of Computer Science, August 1996. To appear in *Parallel Computing*.

[8] 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.

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

[10] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Jason A. Moore, Phil Hatcher, and Michael J. Quinn. Efficient data-parallel files via automatic mode detection. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–14, Philadelphia, May 1996.

[18] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 9, pages 205–223. Kluwer Academic Publishers, 1996.

[19] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, May 1996.

[20] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 1996. To appear.

[21] Nils A. Nieuwejaar. *Galley: A New Parallel File System For Scientic Applications*. PhD thesis, Dartmouth College, 1996.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[28] 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.

[29] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Conference*, pages 313–324, 1990.

[30] Joel T. Thomas. The Panda array I/O library on the Galley parallel file system. Technical Report PCS-TR96-288, Dept. of Computer Science, Dartmouth College, June 1996. Senior Honors Thesis.

[31] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996.

[32] 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.