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. We discuss Galley's file structure and application interface, as well as an application that has been implemented using that interface.

1 Introduction

While massively parallel computers have been steadily increasing in computational power for years, the power of their I/O subsystems 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 parallel file systems is also to blame. One of the primary reasons that parallel file systems have not improved at the same rate as other aspects of multiprocessors is that, until recently, there has been limited information available about how applications were using existing parallel 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 parallel file systems were incorrect [KN94, NK95, PEK+95]. It was commonly believed that parallel scientific applications would behave like sequential and vector scientific applications: accessing large files in large, consecutive chunks [Pie89, PFDJ89, LIN+93, MK91]. Studies of two parallel file-system workloads, running a variety of applications in a variety of scientific domains, at two sites 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 [NKP+95]. These studies suggest that most parallel file systems have been optimized for a workload that is very different than that which actually exists.

Using the results from these workload characterizations and from performance evaluations of existing parallel file systems, we have developed a new parallel file system that is intended to deliver high performance to a variety of applications running under 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 specific needs of a specific community of users.

In this paper we describe the features and design of the system. The performance and scalability of the system are examined in greater detail in [NK96].

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 parallel 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 discusses one such application in detail. In Section 7 we discuss several other parallel file systems, and finally in Section 8 we summarize and describe our future plans.

2 Design Goals

Most current parallel file systems were designed based primarily on hypotheses about how scientific applications would perform I/O. Galley's design is the result of examining how parallel scientific applications actually do I/O. Accordingly, Galley is designed to satisfy several goals:

- Efficiently handle a variety of access sizes and patterns.
- Allow applications to explicitly control parallelism in file access.

This research was supported in part by the NASA Ames Research Center under Agreements numbered NCC 2-849 and NSG 2-936.

1	2	3	4	5	6	1	2	3	4	5	6
4	5	6	1	2	3	4	5	6	1	2	3
3	1	2	6	4	5	3	1	2	6	4	5
6	4	5	3	1	2	6	4	5	3	1	2
2	3	1	5	6	4	2	3	1	5	6	4
5	6	4	2	3	1	5	6	4	2	3	1

Figure 1: An example of a 2-dimensional, cyclically-shifted block layout. 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.

- Be flexible enough to support a wide variety of interfaces and policies, implemented in libraries.
- Allow easy and efficient implementations of libraries.
- Scale to many compute and I/O processors.
- Minimize memory and performance overhead.

3 File Structure

Most existing multiprocessor file systems are based on the conventional Unix-like file-system interface in which a file is seen as an addressable, linear sequence of bytes [BGST93, Pie89, LIN+93]. The 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 should lead to greater flexibility and performance.

3.1 Subfiles

The linear model can give 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. However, the declustering unit size is frequently measured in kilobytes (e.g., 4KB in Intel's CFS [Pie89]), while our workload characterization studies show that the typical request size in a parallel application is much smaller: frequently under 200 bytes [NKP+95]. This disparity 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 problem with 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 [TG96]. 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.

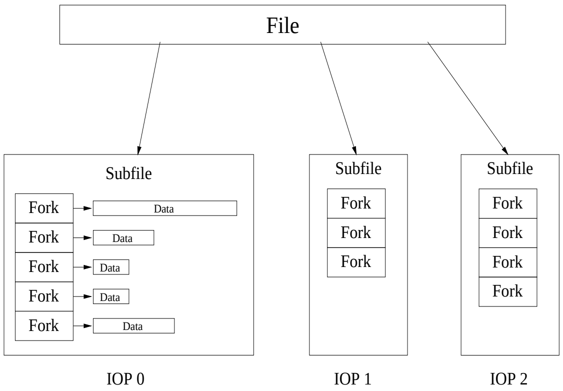

Figure 2: Three dimensional structure of files in the Galley File System. The portion of the file residing on disk 0 is shown in greater detail than the portions on the other two disks.

To address these problems, Galley does not automatically decluster 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* [CK93]. 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*, each of which resides entirely on a single disk, and which may be directly addressed by the application. This structure 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.

3.2 Forks

Each subfile in Galley is structured as a collection of one or more independent *forks*. Each fork is a named, linear, addressable sequence of bytes, similar to a traditional Unix file. While the number of subfiles in a file is fixed at file-creation time, 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. 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. 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 is available independently.

Forks are most likely to be useful when implementing libraries. In addition to data in the traditional sense, many libraries also need to store persistent, library-specific 'meta-data' independently of the data proper. One example of such a library would be a compression library similar to that described in [SW95], which compresses a data file in multiple independent chunks. The library could store the compressed data chunks in one fork and index information in another.

Another example of the use of this type of file structure may be found in the problem of genome-sequence comparison, which requires searching a large database to find approximate matches between strings [Are91]. The raw database used in [Are91] 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* [MHQ96]. Briefly, Stream* divides a file into three distinct segments, each of which corresponds to a particular set of access semantics. Although one could use a different fork for each segment, Stream* was actually designed to store them all in a single file. In addition to the raw data, Stream* maintains several kinds of metadata, which are currently stored in three different files: `.meta`, `.first`, `.dir`. In a Galley-based implementation of Stream*, it would be natural to store this metadata in separate forks rather than separate files.

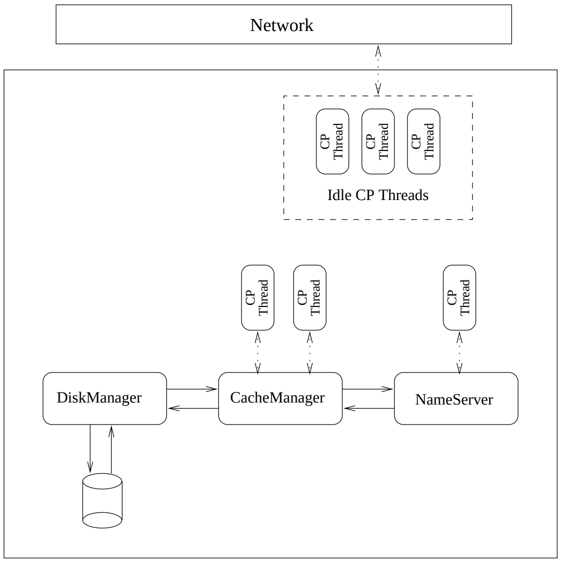

Figure 3: Internal structure of a Galley I/O Processor, showing two active data requests waiting for the CacheManager, one active metadata request waiting for the NameServer, and three idle CP threads.

4 System Structure

The Galley parallel file system follows the client-server model, and is based on a 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 a 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 compute and I/O processors.

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, which in turn means that applications may use whichever communication software (e.g., MPI, PVM, P4) is most suitable to the given problem.

We expect that most applications will use a higher-level library or language layered above the Galley run-time library. One such library will be one that supports a linear, Unix-like file model, which will reduce the effort required to port applications to Galley. Other libraries currently being implemented provide Panda [SCJ+95] and Vesta [CFP+95] interfaces. We also plan to implement ViC*, a variant of C* designed for out-of-core computations, on top of Galley [CC94].

4.2 I/O Processors

I/O servers in Galley are composed of several independent units (as shown in Figure 3). Each unit is implemented as a single thread. Furthermore, each IOP also has one thread designated to handle incoming I/O requests for each compute processor. When an IOP receives a request from a CP, the appropriate CP Thread interprets the request, passes it on to the appropriate worker thread, and then handles the transfer of data between the IOP and the CP. This multithreading makes it easy for an IOP to service requests from many clients simultaneously.

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 our performance tests [NK96]. 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.

Galley's metadata (not to be confused with application-specific 'metadata' discussed above) is distributed across all IOPs, so there is no single point of contention that could limit scalability. Thus, each IOP acts both as a data server and as a metadata server. When a request arrives for a metadata operation (e.g., file open, close, delete), the CP's thread hands the request on to the NameServer, waits for the NameServer to complete the operation, and then passes the result back to the requesting CP. For most operations, the NameServer will need to submit a request to the CacheManager for data stored on disk.

4.2.1 CP Threads

When a request for a data transfer arrives, the CP thread responsible for handling the request creates a list of all the disk blocks that will be required to satisfy the request. It then passes the whole block list on to the CacheManager. The CP thread then waits on a queue of buffers returned by the CacheManager. Although this model does not necessarily restrict each CP to a single outstanding request to each IOP, for performance reasons our current implementation does impose such a restriction.

As buffers become ready, the CP thread handles the transfer of data between the requesting CP and that buffer. When all the requested data has been transferred into or out of that buffer, the thread decreases that buffer's reference count, and handles the next buffer in the queue. 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. 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 comes in.

4.2.2 CacheManager

The CacheManager maintains a separate list of requested disk blocks for each thread. When multiple threads submit requests to the CacheManager, 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. Identifying more sophisticated and effective techniques for providing fair service is a subject of ongoing research.

The CacheManager maintains a global LRU 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 another area of ongoing work.

For efficiency, the CacheManager also maintains a hash table, which contains 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 first buffer in the LRU list with a reference count of 0 is scheduled to be replaced with the new block. The buffer is marked 'not ready', and is added to the requesting thread's ready queue. Then 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 maintains a list of blocks that are scheduled to be read or written. Galley uses 32 KB as its disk block size. 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 [SCO90]. When a block has been read from disk, the DiskManager updates the cache status of that block from 'not ready' to 'ready', and notifies any threads that may have been waiting for that block.

For portability, Galley does not use a 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.

5 Application Interface

Given the new file model provided by Galley, and the observed frequency of strided access patterns in parallel file system workloads, it was not sufficient to simply provide applications with a traditional Unix interface. Galley's interface is primarily intended to allow the easy implementation of libraries. These libraries will provide the higher-level functionality needed by most users.

5.1 File Operations

Files in Galley are created using the `gfs_create_file()` call. In addition to specifying a file name, the application may specify on how many IOPs, and even on which IOPs, the file is to be created. File creation is a three-step process. The first step is to verify that the name chosen is available, and if so, to reserve it. This is done with a single message to the IOP that will be responsible for maintaining the metadata for the new file. The responsible IOP is determined by applying a simple hash function to the file name. The next step is to create subfiles on each of the appropriate IOPs. Subfile creation involves allocating a *subfile-header block* to the file. A subfile-header block is analogous to an inode in a Unix file system, in that it will contain all the metadata information for that subfile. Unlike the Unix practice of statically creating inodes, any block in the file system may become a subfile-header block. If this step fails (e.g., if one or more disks have no more room), then the reserved file name is released, and the appropriate error code is returned to the application. If the operation succeeds, each IOP will return the ID of the subfile-header block to the calling CP. The final step of the file-creation process is to store the file name, along with all the subfile-header block IDs, on disk at the responsible IOP and to return a success code to the application. After the file is created, the subfiles are empty; that is, no forks are created as part of the file-creation process.

When an application opens a file in Galley, using the `gfs_open_file()` call, that processor sends a request to the appropriate metadata server (again, determined by hashing the file name). If the file exists, the IOP returns a success code and the list of all the subfile-header block IDs to the requesting CP. The run-time library assigns the open file a *file ID*, and caches the list of header block IDs in an *open-file table* to avoid repeated requests to the metadata server. Since these IDs do not change during the course of the file's lifetime, consistency is not an issue.

5.2 Fork Operations

Forks in Galley are created using the `gfs_create_fork()` call. Each call takes the ID of an open file, the number of the subfile in which the fork is to be created, and the name of the fork. The run-time library looks up the header-block ID for the appropriate subfile, and sends the header ID and the fork name to the appropriate IOP. By sending the header ID to the IOP, there is no need for an extra indexing operation to take place at the IOP; the IOP is able to retrieve the appropriate subfile-header block immediately. The IOP inserts the name of the fork into the subfile-header block, and returns a success or error code to the CP. For the convenience of application programmers, Galley also provides a `gfs_all_create()` call that will create a fork of the given name in every subfile of a file.

Forks in Galley are opened using the `gfs_open_fork()` call, which takes the same parameters as the fork-creation call. If a fork is successfully opened, Galley returns a *fork ID*, which is used in subsequent calls, much like a file descriptor is used in Unix. Forks are closed with `gfs_close_fork()`, and deleted with `gfs_delete_fork()`. If a CP attempts to delete a fork that has been opened by it, or by any other CP, that fork is marked for deletion, but is not actually deleted until it is closed by every CP that has it opened. For convenience, there are `gfs_all_open`, `gfs_all_close`, and `gfs_all_delete` calls as well.

5.3 Data Operations

Most parallel file systems present applications with an application interface similar to that of Unix [Pie89, RP95, BGST93]. While this interface is simple and familiar to programmers, it was not designed to allow parallel applications to access parallel disks. In particular, it does not allow programmers to issue the highly structured requests that we have observed to be common among parallel, scientific applications [NKP+95]. Indeed, if an interface were available that allowed an application to issue such highly regular requests, the number of I/O requests issued in one production file-system workload could have been reduced by over 90% [NK95]. Such structured operations can also lead to significant performance improvements [Kot94].

In addition to simple `read()/write()` operations, Galley supports simple-strided, nested-strided, and nested-batched operations. Descriptions of these operations and the interfaces required to support them may be found in [NK95]. The tremendous performance improvements achieved using these interfaces in Galley are described in [NK96].

In addition to these structured operations, Galley provides a more general file interface, which we call a *list* interface. This interface accepts an array of (file offset, memory offset, size) triples from the application. 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. The more structured interfaces are actually implemented on top of the list interface.

While all of these interfaces specify the order of data in the buffer, the order in which the individual pieces are transferred between the IOP to the CP is not specified. This freedom allows Galley to transfer the data from the disk to the IOP's memory and from the IOP to the CP in the most efficient order rather than strictly sequentially. This ability to reorder data transfers can lead to remarkable performance gains [NK96], and is a distinct advantage of these interfaces over any interface where the user must request one small piece of data at a time, forcing the file system to service requests in a particular order.

To avoid complicating these interfaces further, Galley does not provide an explicit interface to request data from multiple forks or subfiles. Users may achieve similar results by submitting multiple requests asynchronously, one to each desired fork.

6 Example: FITS

We present an example of how the features described above may be used in practice. The Flexible Image Transport System (FITS) data format is a standard format for astronomical data [NAS94]. A FITS file begins with an ASCII header that describes the contents of the file and structure of the records in the file. The remainder of the file is a series of records, stored in binary form. Each record is composed of a key, with one or more fields, and one or more data elements. Each record within a single FITS file has an identical size and structure. Records may appear in any order within the file.

For this paper, we created a system that was able to handle a specific type of FITS file in use at the National Radio Astronomy Observatory (NRAO), and generic queries on those files. A library that was capable of handling many kinds of queries and FITS files is a perfect example of the type of domain-specific library we expect to be implemented on Galley.

6.1 FITS at NRAO

One specific example of how FITS files are used in practice is described in [KGF93, KFG94]. This type of FITS file contains records with 6 keys, describing the frequency domain (U, V, W), the baseline, and the time the data was collected. The baseline is a single number that indicates which antenna or combination of antennas generated that record. The data portion of each record contains a pair of data elements, one for each of two polarizations. Each data element contains floating-point triples for each of 31 frequencies. The triples represent a single complex number and a weighting factor. Thus, a single data element contains 372 bytes of data and each record contains 24 bytes of key information and 744 bytes of data.

These files are used in many different ways by different users at NRAO. The most common types of use involve scanning subvolumes of the full, multi-dimensional sparse data set, where the subvolumes may be defined along one or more of the axes. For example, a user may want to examine all the records within a given time range, and sorted along the U axis.

Previous work on these files has focused on increasing locality along several dimensions simultaneously. In [KFG94], the authors examine studied the effectiveness of Piecewise Linear Order-Preserving Hashing (PLOP) files at reducing the amount of time required to perform common queries, by increasing certain kinds of locality within the files. While locality can also improve performance in parallel file systems, too much locality can reduce the number of disks being accessed at any time, actually leading to lower performance.

6.2 FITS on Galley

Since most of the queries common at NRAO include subranges of time as at least one of the constraints, we sorted the records by time before distributing them across the IOPs. The data was distributed in CYCLIC fashion, in blocks of 1024 records. That is, in a system with 4 IOPs, IOP 0 would hold records 0 to 1023, 4096 to 5119, and so on, while IOP 1 would hold records 1024 to 2047, 5120 to 6143, and so on.

For many queries, we were unable to determine *a priori* which data records would satisfy the query. As a result, we frequently examined many keys to identify the small number of data records that were relevant to the query. To improve performance, we chose to store the keys in one fork and the data in another. This setup allowed us to achieve higher performance when reading the keys, since we were not paying for the cost of retrieving uninteresting data from disk. Although we stored all the data in a single fork on each subfile, another reasonable choice would have been to store the data for each polarization in its own fork. Since many of the queries involved data from only a single polarization, this setup would also have reduced the amount of uninteresting data that was read from disk.

To evaluate the efficacy of their PLOP-file implementation, the authors performed several queries, which were intended to be representative of those that were most commonly used in practice at NRAO [KGF93]. Their tests were performed on a single-processor, single-disk system. We performed the same set of queries, using the same data set, on our implementation. Our tests were performed on a cluster of IBM RS/6000s connected by an FDDI network. Since the original queries were performed on a single-node processor, we used a single CP. We used four IOPs, each with a single disk. Each IOP used a raw disk partition to store its data, thus avoiding skewing the results by retrieving data stored in AIX's buffer cache.

The specific queries performed in both cases are briefly described below. More detail about each query, and why it is commonly used at NRAO, may be found in [KGF93].

1. Read the full data set.
2. Read the full data set, sorting records by time.
3. Read the full data set, sorting records by baseline.
4. Read a subvolume of the data including 10% of the time range.
5. Read a subvolume of the data including 10% of the time range, sorting the records by U.
6. Read the subvolume for a single time and polarization.
7. Read a subvolume including 10% of the time range and one polarization.
8. Read a subvolume including 50% of the time range, a single baseline, and one polarization.
9. Read a subvolume including 50% of the time range, antenna #1, and one polarization.
10. Read a subvolume including 50% of the time range, antenna #14, and one polarization.
11. Read a subvolume including 50% of the time range, antenna #27, and one polarization.
12. Read a subvolume containing a single baseline and a single polarization, sorting records by time.

Although many of these queries could have been most efficiently expressed using some form of strided request, our system was designed to handle generic queries. As a result these queries were all performed using Galley's list interface. The buffer cache on each IOP was flushed prior to each query.

Table 1 shows the length of time required to complete each query for both the PLOP-file and Galley implementations. Since the PLOP-file results were obtained on a different system with only a single disk, we cannot directly compare the time required to complete the queries. Instead, we compare the amount of time required to complete a query relative to the time required to read all the data. This crude normalization allows us to make some effort at comparison.

	Data	PLOP-file		Galley	
Query	Elements	Secs.	Normal.	Secs.	Normal.
1	126092	55.49	1.00	11.20	1.00
2	126092	70.50	1.27	11.72	1.05
3	126092	181.80	3.28	13.96	1.25
4	12636	4.10	0.07	1.00	0.09
5	12636	6.85	0.12	1.53	0.14
6	351	0.27	0.00	0.17	0.01
7	6318	2.33	0.04	0.62	0.05
8	186	1.45	0.03	0.55	0.05
9	4836	4.03	0.07	1.45	0.13
10	4836	14.50	0.26	2.10	0.19
11	4836	20.30	0.37	2.27	0.20
12	180	1.49	0.03	1.57	0.14

Table 1: Timing results for PLOP files on a uniprocessor system, and for Galley files on a 4-IOP, 1-CP system. Results are shown in 'raw' form, as well as normalized to the time required to read the full data set with no filtering or sorting. The full data set contained 63,046 records, with 126,092 data elements.

While our implementation on top of Galley was far simpler than the PLOP-file implementation (about 1/10 the number of lines of code), it performed significantly better in 4 out of 11 cases (disregarding the first case, which is used as a baseline), and had competitive performance in 5 of the remaining cases. Galley performed particularly well on queries 2 and 3. While the PLOP-file implementation had to sort the whole dataset in memory, Galley's interface allowed us to read just the keys from their fork, sort them, and then read the actual data into memory in sorted order. Galley also performed relatively well on queries 10 and 11. While the PLOP-file implementation had to read in 3 to 5 times as many records as they were interested in, we were able to filter out the interesting records by looking only at the data in the key-fork. Galley's relative performance was worst on queries 9 and 12. In these two cases, Galley had to examine a large number of keys to identify a small number of interesting records, while the PLOP files were carefully structured to reduce the number of records they had to examine for these queries. This same structure also caused the PLOP-file implementation to be noticeably worse than Galley's on queries 10 and 11.

7 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.

Intel's Concurrent File System (CFS) [Pie89, Nit92], and its successor, PFS, are examples of parallel file systems that provide a linear file model with a Unix-like interface. Support for parallel applications is limited to *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 parallel file systems with this style of interface 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 [HER+95]. In PPFS, however, the basic transfer unit is an application-defined *record* rather than a byte. PPFS maps the logical, linear stream of records onto an underlying two-dimensional model, indexed with a `(disk, record)` pair. PPFS provides several mapping functions, which correspond to common data distributions, and allows an application to provide its own mapping function as well.

One of the most interesting parallel file systems is the Vesta file system (and its commercial version, PIOFS) [CF94, CFP+95]. 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. Unlike Galley, 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 sub-blocks of a single size. Like Galley, Vesta uses a hashing scheme to distribute metadata. In addition to the functionality of Vesta, PIOFS provides applications with a Unix-like interface. Work is underway on a library that will provide a Vesta interface for Galley.

8 Summary and Future Work

Based on several studies of parallel file systems being used in production environments, we have designed a new parallel file system that is intended to provide high performance to a variety of libraries and applications. Galley is based on a new three-dimensional structuring of files. This structuring provides tremendous flexibility to applications and libraries, as well as opportunities to explicitly control the degree of parallelism in an application's file accesses. Galley provides several new forms of I/O request that reduce the aggregate latency of multiple small requests and allows the file system to optimize the disk accesses required to satisfy the request.

The case studies contained in this paper, as well as performance evaluations described elsewhere [NK96], suggest that Galley rectifies many of the shortcomings of existing parallel file systems. In particular, we demonstrated the usefulness of Galley's "fork" structure and higher-level interfaces.

Galley has been completely implemented. While Galley currently runs only on a cluster of IBM RS/6000s and IBM's SP2 multiprocessor, porting to other architectures should be fairly straightforward and will be explored in the near future. Work continues on improving the stability of the system in general. Future work will focus on efficiently supporting multiple applications, which may place conflicting demands on the system.

Acknowledgments

Thanks to NASA Ames for the use of their SP2. Thanks also to John Karpovich of the University of Virginia for his help in understanding the use and interpretation of FITS files. Finally, thanks to the reviewers for their helpful comments.

References

[Are91] 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.

[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.

[CC94] 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.

[CF94] 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.

[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.

[HER+95] 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.

[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.

[KGF93] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Breaking the I/O bottleneck at the National Radio Astronomy Observatory (NRAO). Technical Report CS-94-37, University of Virginia, August 1993.

[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.

[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.

[MHQ96] 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.

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

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

[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.

[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.

[RP95] Brad Rullman and David Payne. An efficient file I/O interface for parallel applications. DRAFT presented at the Workshop on Scalable I/O, Frontiers '95, February 1995.

[SCJ+95] 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.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Conference*, pages 313–324, 1990.

[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.

[TG96] 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.

[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.