Parallel I/O for Distributed Systems: Issues and Implementation*

V. S. Sunderam
Steven A. Moyer

Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, U.S.A.
{vss, moyer}@mathcs.emory.edu

Abstract

Parallel and distributed computing have matured sufficiently for their adoption in production environments, consequently necessitating effective, robust, and efficient frameworks for input and output. A number of concurrent I/O initiatives have evolved in response to these needs, some system specific, others proposing an abstract framework and portable interface. These parallel and distributed I/O efforts focus either on characterizing file access patterns, language extensions, runtime libraries, or low level primitives. Important systems issues in each approach, technical details of various strategies, and their effect on performance are analyzed. The PIOUS system, a transport independent, scalable I/O framework for parallel and distributed systems is presented, and experiences with its use are reported.

1 Introduction

It is well documented that the performance of many scientific and commercial applications, including the Grand Challenge problems [4], is often limited by the performance of the I/O subsystem [11]. Furthermore, computational capability continues to grow at a much higher rate than I/O capability; while microprocessor performance increases at an average rate of 40% per year, raw disk seek time, a major factor in overall disk performance, improves at a rate of

*Research supported by the National Science Foundation, under awards CCR-9118787 and ASC-9214149, the U. S. Department of Energy, under grant DE-FG05-91ER25105, the Office of Naval Research, under grant N00014-93-1-0278, the Army Research Office, under grant ARO-93G0045, and the National Aeronautics and Space Administration, under grant NAG2-828.

only 7% per year [22]. To make matters worse, data set sizes are increasing and additional demands are being placed on the I/O system by new visualization and multimedia tools.

Parallel applications are particularly sensitive to the growing disparity in computational and I/O capability, and thus have provided the impetus for the development of parallel file systems. Parallel file systems address the I/O bottleneck by logically aggregating multiple storage devices into a single high-performance storage subsystem. Capacity increases naturally as a function of the number of devices. Deliverable bandwidth is increased by employing two orthogonal methods of achieving access concurrency: independent disk addressing and data declustering. Addressing storage devices independently allows the file system to concurrently access data from different files. Declustering the data from a single file across multiple devices allows the system to access the file in parallel.

Today most parallel machines implement some form of parallel file system. Familiar examples include the Intel CFS [23] and nCUBE Parallel I/O System [9]. Parallel file systems have also been successfully implemented for distributed computing environments, such as PIOUS [19, 20], for PVM [27, 13]. Parallel and distributed I/O systems have many issues in common, but some that are distinct. In this paper, our focus is on architectural, interface, and performance factors which may be treated within a unified framework.

Figure 1 depicts the generic parallel file system architecture employed in most parallel processors and distributed computing environments. A parallel file system is implemented over a set of independent I/O nodes that can be accessed by a set of compute nodes via a high-speed interconnect. An I/O daemon executing on each I/O node implements general file access methods, disk scheduling, data caching and prefetching, and other management activities. Library routines linked with applications executing on the compute nodes implement the file system interface and manage all communication with the I/O nodes.

When an application executes a read or write library function call, the minimal event sequence required for data access proceeds as follows. First it is determined which I/O nodes contain the file data to be accessed. Messages are then sent to the appropriate I/O daemons requesting that data be read or written. Independently, each I/O daemon satisfies each request and replies with the result. After all results have been collected, the library function returns control to the caller.

Because current parallel file system architectures are generally very similar, whether intended for use on a parallel machine or within a distributed environment, most of the first-order issues are the same: language and/or library support for expressing high-level semantic information about the data access requirements of an application [3], file system primitives for enabling efficient implementation of language and library level mechanisms [21], prefetching mechanisms that exploit high-level semantic information about file access pat-

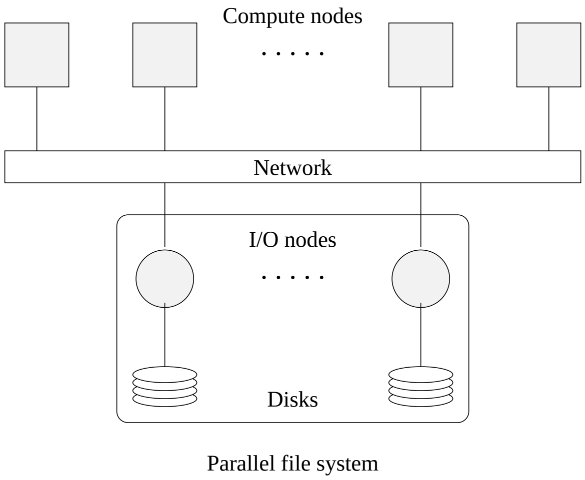

Figure 1: Generic parallel file system architecture

terns [26], and concurrency control [18] and cache coherence [16] protocols that maintain guaranteed consistency semantics. Additional first-order issues specific to distributed environments include support for the transparent conversion of heterogeneous data [6], authentication of clients and encryption of data transfers, and system configuration and management [20].

In this paper we present an overview of some current research efforts that address a number of the issues listed above. We then present a user-level view of our PIOUS parallel file system for the PVM distributed computing environment, including an architectural overview, an outline of the API and the implementation, and general performance results.

2 Overview of Current Initiatives

Parallel file systems offer a *potential* aggregate bandwidth proportional to the number of storage devices employed. Unfortunately, this potential is rarely realized. Two factors contribute significantly to the insufficient performance of current parallel file systems: the lack of effective language and/or library support for expressing high-level semantic information about the data access requirements of an application, and the lack of effective file system primitives for enabling efficient implementation of language and library level mechanisms. Thus these areas are beginning to receive considerable attention by the research community.

The following sections provide an overview of major efforts aimed at characterizing parallel file-access patterns, developing language and library mechanisms for expressing high-level data access requirements, and defining low-level file-access primitives to efficiently support common application requirements.

2.1 Characterizing Parallel File-Access Patterns

The CHARISMA project and system [17, 24] is based on the premise that a thorough understanding of the expected workload is valuable in optimizing the performance of parallel and distributed filesystems. To this end, Kotz et al have embarked on a project aimed at characterizing the file and data access patterns of parallel filesystems in production environments. Their methodology has been to trace a multiprocessor filesystem that has been in use for production scientific computing; based on availability, they selected the Intel iPSC/860 with the Concurrent File System (CFS) at NASA Ames which is used for numerical aerodynamic simulation and computational fluid dynamics codes. The data that they report in [17] is based on a filesystem with a total capacity of 7.6GB accessed via a network with only 10MB/s bandwidth. Data was collected for about 156 hours over a period of three weeks, with individual trace files covering time slots ranging from 30 minutes to 22 hours. Traces included only I/O done via the CFS and contained, among other data, timestamps, file size information, numbers of reads and writes, their frequency, size of each I/O operation, file offset of each I/O operation, and number of clients.

After detailed analyses of over 700MB of trace data, some interesting conclusions were reached. First, while the workload had some common features with previous studies of scientific applications and filesystems, parallelism had a significant effect on workload characteristics. Among them was that request sizes were smaller, and intra-job concurrent file sharing was high; moreover, nonconsecutive sequential access and interprocess spatial locality were pronounced. The study suggested that parallelism leads to new, interleaved access patterns with no temporal locality, and high interprocess spatial locality at the I/O node. The ultimate conclusion was that the filesystem interface needed to be changed for efficient parallel I/O. The current interface forces the programmer to break down large I/O activities into small, non-contiguous requests; interface primitives for strided and collective I/O requests, coupled with caching at the I/O and compute nodes would significantly improve parallel filesystem access performance.

2.2 Language Extensions and Runtime Libraries

A second orientation in parallel and distributed filesystems projects and research focuses on languages, compilers, and runtime libraries for improving the effectiveness of parallel input and output. One effort, by Choudhary et. al., [2] is based on the argument that efficient compilation of out-of-core programs requires information regarding the distribution of arrays *on disks* as well as

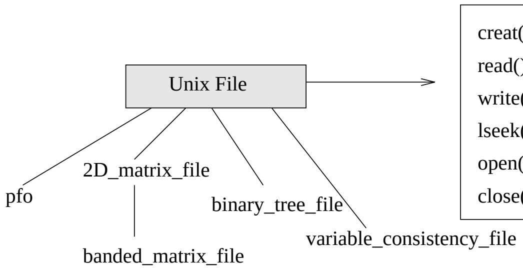

Figure 2: The ELFS class hierarchy

on processors. Their project therefore proposes several primitives intended to assist the compiler and runtime systems in optimizing accesses. For example, the primitive "**DISKS** D(8,8)" indicates that disks are logically arranged as a 2-dimensional 8x8 grid. Another primitive is **FILEPROC** which indicates the number and logical topology of processors that actually participate in the I/O. As another example, "**FILEDISTR** F(D,[FP]))" declares a file template F which is distributed over D disks, and further associates this template with the processors declared in FP. And finally, "**FILEALIGN** F:: F1,F2,F3" aligns a list of associated files to a template, while **ASSOCIATE** associates a file-template with the corresponding array template. All of these primitives have processor/memory duals, and the same principles in using such information for efficient in-core operations may be applied to I/O operations.

The ELFS project [14] is another effort aimed at providing language and runtime extensions. However, the strategy in ELFS is to use an object oriented interface and a file-class hierarchy. An example is shown in Figure 2. In this figure, the class *2D_matrix_file* inherits all of the *Unix_file* operations and provides additional member functions based on the semantics and structure of the underlying abstraction. Since the implementation of the class is aware of the underlying structure and further, can be specifically tailored for any kind of specialized operation via its member functions, prefetching and data caching can be done more intelligently and therefore more effectively than in a filesystem where this knowledge is lacking. Moreover, these benefits may be directly inherited by derived classes (such as *banded_matrix_file* in the example), or, when appropriate, operator overloading and specialized member functions in the derived class may be used to override the mechanisms built into the base class. Thus flexibility and reusability are retained while delivering the potential for optimized operation.

The PASSION project [2, 10], in addition to proposing HPF-like extensions for specifying parallel file layout, also incorporates a "two-phase runtime access strategy". In this approach, data is first read in a manner conforming to the

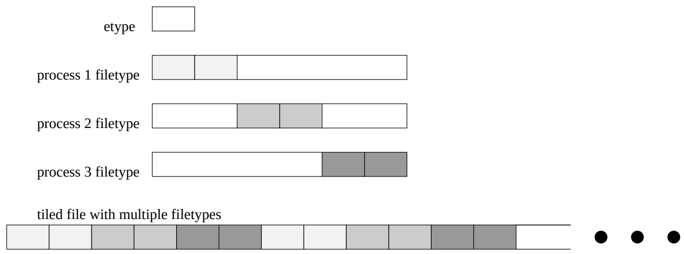

Figure 3: MPI-IO filetypes and file layout

distribution on disks, and then redistributed among the processors to obtain the target distribution. The data access cost is the metric used to measure efficiency — in this approach, it is the sum of the file access cost and the data distribution cost. As external data movement, i.e. the file access, is generally predominant and overshadows the data distribution costs, this strategy allows file layouts to be (almost) independent of the target data distribution, and yet achieve substantial benefits. To test the efficacy of this method, several experiments comparing the two-phase access strategy with direct I/O (i.e. disk layout corresponding to target distribution) were made, and significant and sometimes overwhelming improvements were observed. This effort has established therefore, that at least on the multiprocessor and filesystems tested, optimal disk layouts that are different from target processor layouts are often much more effective than a mirror image mapping, despite the redistribution that is required.

A more recent effort following the language extensions/libraries approach is the MPI-IO initiative by Corbett et al [5, 6]. This proposed standard for filesystem access in message passing systems supports a high level interface to describe the partitioning of data among processes, a collective interface describing complete transfers of global data structures between process memories and files, asynchronous I/O operations, allowing I/O to be overlapped with computation, and optimization of physical layout on storage devices. The essence of the MPI-IO strategy is to use the notion of "derived datatypes" that is an integral MPI facility to describe the layout of a message. MPI-IO adopts this concept to describe the layout of data in the file, using a derived datatype `filetype` and in the user's buffer, using the type `buftype`. Both are derived from a third MPI datatype referred to as the *elementary* datatype `etype`. The idea behind doing this is to ensure consistency between the type signatures of `filetype` and `buftype`, and to ensure portability by basing these on datatypes than on bytes.

In MPI-IO the filetype defines a pattern that is continuously repeated throughout a file, as shown in Figure 3. Constructors are provided to help the user create complementary filetypes corresponding to common distribution patterns, such

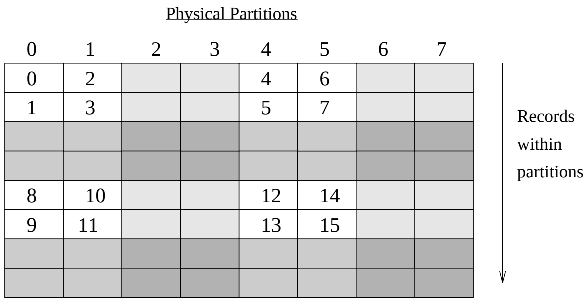

Figure 4: Logical partitions in a 2D Vesta file

as broadcast/reduce, scatter/gather, and HPF distributions. When opening a file, both the `filetype` and `etype` are specified. Data is moved between files and processes via `read` and `write` calls, each of which has semantics specified in terms of three orthogonal attributes, namely positioning (explicit offset vs. implicit file pointer), synchrony (blocking vs. nonblocking), and coordination (independent vs. collective). In addition to these basic primitives in the proposed MPI-IO standard, the interface is evolving to support hints for optimizing disk layout, atomic read/write file access semantics, and declustering in cautious-mode to guarantee consistent access.

2.3 Access Primitives

In addition to characterizing parallel filesystems and developing language extensions and runtime libraries to provide convenient interfaces and optimize performance, low level access primitives for parallel filesystems are also essential. One project that has focused on this issue, in addition to providing access interfaces, is the Vesta system developed by Corbett et al [7]. Vesta implements 2D files and provides low level primitives to access regular partitions of a matrix. Files under this system are comprised of multiple physical partitions, with multiple "records" within each physical partition; these may be thought of as defining the horizontal and vertical dimensions. Logical partitions, i.e. file units visible to the user, are superimposed on the physical layout in a manner that can be specified at the time of open'ing the file. There are four parameters that define the logical partitioning — the vertical and horizontal *group size*, and the vertical and horizontal *interleave*. An example in which each of these parameters has the value 2 is shown in Figure 4.

The "vertical interleave" in a Vesta file specifies the number of different logical partitions interleaved within a physical partition. Each of these logical partitions gets the "vertical group size" number of consecutive records of the physical partition. Similarly, the "horizontal group size" specifies the number of physical partitions spanned by a logical partition. Finally, the number of distinct logical partitions that appear in the horizontal dimension is specified by the "horizontal interleave". This organization allows for different access patterns to be implemented efficiently, by translating an offset into a logical partition into the physical location of the data depending upon the values of the four logical partitioning parameters. Experiences with such an organization in the Vesta system have been encouraging, and the unique feature of providing an *explicit* parallel interface to the filesystem has enabled a wide variety of parallel access patterns to be supported effectively.

In the same vein as Vesta, Kotz et al [21] have recently proposed a comprehensive set of low level parallel file access primitives. These are sophisticated Unix-like functions based on, and derived from, the patterns seen in the CHARISMA study. If implemented fully, they would subsume all the other proposed interfaces, including those found in Vesta and MPI-IO. The primitives proposed in this project may be classified as:

- *Strided*: beginning at a given offset, the filesystem reads a specified number of records, offsetting each by a specified stride size — but stores these contiguously in memory (assuming a read operation).
- *Nested strided*: is, as its name implies, an access method where multiple levels of striding, with different stride sizes in each level, can be specified.
- *Nested batched*: is the most general of access methods allowed, but owing to this generality, requires that each access be specified, and then issued to the filesystem as a group (as a request "vector" or completely general request "structure").

In evaluating the efficacy of providing such low level file access primitives, this research uncovered some interesting results. First was the observation that over 80% of all the files examined were accessed entirely with a strided pattern. Moreover, if nested-strided patterns were included, almost all the remaining accesses could be classified as those with a two-level or three-level nesting. Interestingly, it was found that three-level nested-strided accesses were far more common than two, and suggested the use of multidimensional matrices. Based on traces carried out in the CHARISMA experiment, and by consolidating these observations, it was concluded that the number of I/O requests could have been reduced from over 25 million to just over 80,000 if strided and nested-strided primitives had been available. Low level primitives for parallel and distributed filesystems are therefore established as being essential to performance and to lowering overhead, as well as to providing natural programming interfaces.

3 PIOUS for PVM

PIOUS is a parallel file system for the PVM distributed computing environment. PIOUS supports parallel applications by providing coordinated access to file objects with guaranteed consistency semantics. For performance, PIOUS declusters file data to exploit the combined file I/O and buffer cache capacities of networked computer systems.

The PIOUS project started in the Summer of 1993 at Emory University as part of a larger research initiative aimed at understanding and developing flexible, reliable, and scalable network parallel file systems. At that time, no freely available software capable of supporting basic parallel file system research existed. Thus it was decided that the first phase of the initiative would be the development of a software platform to serve as a vehicle for further research. The PIOUS software is the result of that effort. PIOUS has been made available to the user community at large, and is intended to support high-performance parallel applications[1].

This section presents a user-level view of the PIOUS file system, providing information on the system architecture, configuration management, the programming interface and usage. Performance results are then presented for a simple benchmark application.

3.1 Software Architecture

The PIOUS software architecture is depicted in Figure 5. PIOUS implements a parallel file system within the PVM environment that consists of a service coordinator (PSC), a set of data servers (PDS), and library routines linked with client processes (PVM tasks). PVM provides both the process management facilities required to initiate the components of the PIOUS architecture, and the transport facilities required to carry messages between client processes and the PIOUS file system.

A single PIOUS Service Coordinator initiates major activities within the file system. For example, when a process opens a file, the PSC is contacted to obtain file meta-data and to ensure that the requested file access semantics are consistent with those of other processes in a logical process group. Note that the PSC only participates in major system events, not general file access, and thus does not represent a system bottleneck.

A PIOUS Data Server resides on each machine over which a file is declustered. Data-servers access disk storage in response to client requests via the native file system interface. Though ideally each PDS accesses a file system local to the machine on which it resides, two or more PDS may share a file system via a network file service.

Finally, each client process is linked with library functions that define the

[1] The PIOUS project home page is located at http://www.mathcs.emory.edu/pious.html

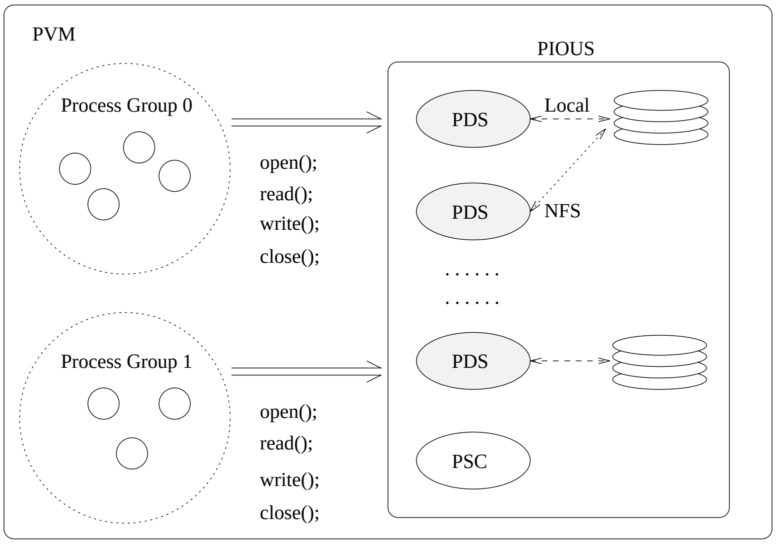

Figure 5: PIOUS software architecture

PIOUS file model and interface (API). Essentially, library functions translate file operations into PSC/PDS service requests.

The PIOUS architecture emphasizes asynchronous operation and scalable performance by employing parallel independent data-servers for declustering file data. Coordinated file access is provided via a transaction-based data-server protocol [18]. Further design and implementation details can be found in [19].

3.2 Configuration Management

PIOUS library functions allow the user to dynamically specify a set of hosts for declustering file data. Each PIOUS file can be declustered on any subset of the machines in the PVM environment. PIOUS will automatically spawn a data server on each host that contains file data on a demand basis.

Optionally, a set of default data-server hosts can be specified when PIOUS is started. An application can then operate on PIOUS files without specifying location information in the form of hosts for declustering file data. The PIOUS system assumes that if no location information is provided, file data is declustered on the default data servers.

The simplest method for using the PIOUS file system is to predetermine a set of hosts to be used for declustering all PIOUS files. A data-server host file, termed a *dsfile*, is created specifying the chosen hosts. The dsfile is used each time PIOUS is started, thereby relieving applications of ever specifying any location information.

The PIOUS dsfile is similar in syntax to the PVM hostfile. Each dsfile entry specifies information concerning a particular data-server host in the form:

hostname [sp= *path*] [lp= *path*]

A data-server host entry consists of the name of the host and the optional arguments:

sp= *path* specifies the data-server search root path. The *path* argument is an absolute pathname that refers to a directory file to be used as the root directory for all PIOUS pathname resolution at the data-server host. That is, the *path* argument is used by PIOUS as a path prefix for all file access at the named host.

Specifying a search root path for each data-server host accommodates hosts with different directory structures. For example, it allows declustering a PIOUS file on two data-server hosts where a home directory on one is `/home/fred`, and the corresponding home directory on the other is `/users/people/fred`.

lp= *path* specifies the data-server log directory path. Each PIOUS data server maintains several log files, including an error log. The *path* argument is an absolute pathname that refers to a directory file to be used by the data server on the named host for maintaining log files.

Two or more PIOUS data servers may share the same log directory file simultaneously, as all log files are distinguished by unique host and user identifiers.

Default values for both data-server search root and log directory paths are specified via a host entry with a *hostname* argument value of *. To be valid, each dsfile host entry must specify both a search root and log directory path, either directly or by default.

Figure 6 depicts a data-server host file for configuring the PIOUS file system. Note that dsfile comments begin with the occurrence of the # character and proceed to the end-of-line.

3.3 File Model and Interface

The PIOUS interface is very similar in many respects to a traditional Unix-style interface, implementing open, close, read, write, and seek file operations. Whenever possible, PIOUS function arguments and behavior are the same as, or a logical extension of, the corresponding Unix functions. Thus anyone familiar with a Unix-style file system interface can easily develop PIOUS applications.

PIOUS diverges from the traditional Unix model by implementing two-dimensional file objects and sophisticated access coordination mechanisms designed to simplify parallel applications development. These features of the

```
# Data-server host file example

# Set default search root and log directory paths

* sp= /users/fred/piousfiles    lp= /users/fred/piouslogs

# Host         : snoopy
# Search Root  : /users/fred/piousfiles
# Log Directory: /users/fred/piouslogs

snoopy

# Host         : linus
# Search Root  : /users/fred/BIG-piousfiles
# Log Directory: /users/fred/piouslogs

linus sp= /users/fred/BIG-piousfiles
```

Figure 6: PIOUS data-server host file for configuration management

PIOUS file system are discussed in detail below. A complete description of the current PIOUS interface is presented in [20].

3.3.1 File Objects

A PIOUS file is a single object composed of one or more physically disjoint segments. Each segment is composed of a linear sequence of zero or more bytes. The number of segments in a file is set at the time of creation and does not change for the life of the file. Figure 7 depicts a PIOUS file with four segments.

PIOUS declusters file data on a set of specified data servers, as previously discussed. Though the method employed for declustering data is not important from a functional point of view, understanding declustering is important for developing high-performance applications.

A PIOUS file is declustered by placing each segment on a single data server. If the number of segments exceeds the maximum number of data servers for declustering the file, then segments are mapped to data servers in a round-robin fashion.

For example, the file in Figure 7 is shown declustered on two data servers. To reduce name space pollution at each data-server host, the logical file pathname actually refers to a directory file that contains entries representing file segments. Note that the logical PIOUS file structure directly reflects the physical structure that naturally results from declustering file data.

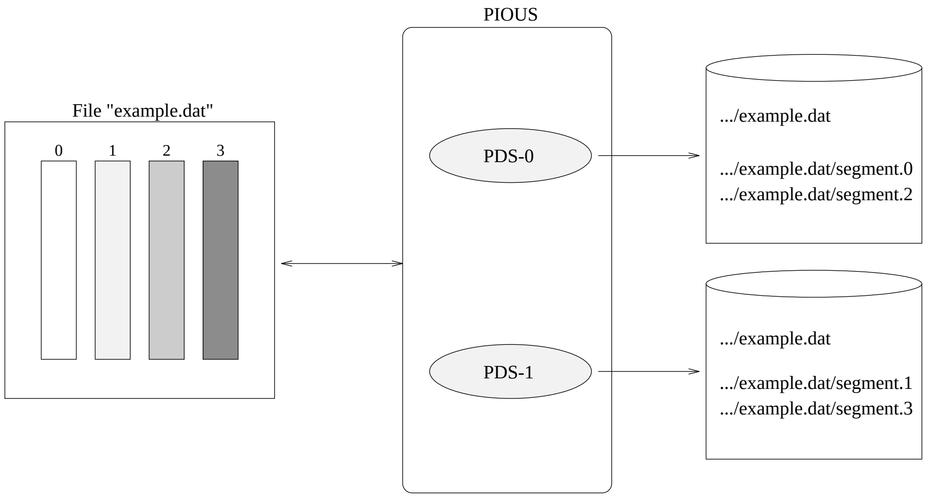

Figure 7: PIOUS file object

Because PIOUS files are two-dimensional, array and tabular data map naturally. Thus developing parallel scientific, database, and sorting applications, among others, should be simplified.

3.3.2 File Views and Access Semantics

The PIOUS interface provides a process group[2] with three logical *views* of a file object: global, independent, and segmented. Each file view provides a process group with a distinct set of access semantics, as defined below:

global A file appears as a linear sequence of data bytes; all processes in a group share a single file pointer.

independent A file appears as a linear sequence of data bytes; each process in a group maintains a local file pointer.

segmented A file appears in the natural segmented form; each process accesses a specified segment via a local file pointer.

For the global and independent file views, a linear byte sequence is defined by ordering the bytes in a file by fixed-size blocks taken round-robin from each segment; the block size is specified at the time the file is opened. Note that if the number of file segments is a multiple of the number of data servers on which

[2] Group membership is a convenient abstraction expressed at the time a file is opened. There is no relationship between logical groups in PIOUS and the PVM process group mechanism.

the file is declustered, then a linear file view results in an access pattern that is equivalent to traditional disk striping [25]. A view only defines the way a file is accessed and does not alter the physical representation. Thus a given file can always be opened with any view. All processes in a group must open a file with the same view.

PIOUS two-dimensional files simplify data mapping for many parallel applications. However, segmenting files does impact familiar access semantics versus a traditional Unix linear file. The most obvious difference is the notion of EOF (end-of-file). When accessing a PIOUS file in the segmented view, each process accesses a particular segment with a well defined EOF. However, when accessing a multi-segment file as a linear byte sequence, in either the global or independent view, the notion of EOF is not clear. For example, a read access at a particular file offset can hit the EOF at one segment, yet a read access at a greater file offset can return data from a different segment.

A related issue concerns "holes" in a file. Unix-style file systems specify that a read from an unwritten portion of a file prior to the EOF returns bytes with value zero. When accessing a PIOUS file in the segmented view, each segment accessed by a process will exhibit this behavior. However, when accessing a multi-segment file as a linear byte sequence, this behavior will only be exhibited in the case where each write access that creates a hole results in data being written to all segments.

Fortunately, working with PIOUS files in the global and independent views is relatively straightforward if the underlying file structure is kept in mind. Applications that consistently access a file as a linear byte sequence and explicitly write all data that is to be read will not encounter any of the issues discussed above.

3.3.3 Consistency and Fault Tolerance Semantics

PIOUS provides all processes with linearizable [15] (and hence sequentially consistent) access to files opened under any view. Both data access and file pointer update are guaranteed to be atomic with respect to concurrency.

PIOUS also supports file access in two user-selectable fault tolerance modes: *volatile* and *stable*. Stable mode guarantees file consistency in the event of a system failure; volatile mode does not. Volatile mode is intended for non-critical applications that require high-performance. Applications requiring fault tolerance can access files in the lower-performance, but guaranteed consistent, stable mode.

The current PIOUS implementation does not provide applications with fault tolerance of media failures. Media fault tolerance is achieved by data replication on independent devices, either as multiple file copies [1] or as computed parity information [22]. Such techniques are independent of the file system architecture, provided that multiple independent storage devices can be accessed.

3.3.4 Transaction Facilities

PIOUS files are normally accessed via traditional read and write operations that are guaranteed to be linearizable. To further enhance coordinated file access, PIOUS also supports the grouping of read and write operations within the context of a single user-transaction[3] [1].

User-transactions are an atomic unit with respect to concurrency, guaranteeing linearizability with respect to all other user-transactions and individual access operations. As with single access operations, user-transactions can be performed in either volatile or stable mode.

Utilizing user-transactions in a PIOUS application is straightforward. The functions pious_tbegin() and pious_tend() mark the beginning and end of a user-transaction, respectively. All PIOUS access operations that occur between a transaction begin and end point are part of that user-transaction. At any time after calling pious_tbegin(), but prior to calling pious_tend(), a user-transaction can be aborted via pious_tabort().

Any number of files can be accessed within the context of a user-transaction, and any PIOUS function can be called, though certain actions may be denied.

3.3.5 Future Interface Development

Parallel file systems are relatively new and an area of active research. To date, no particular file model or interface has been embraced as being clearly superior. The current PIOUS interface represents only a limited extension of the Unix model, and was designed to support the set of parallel file-access patterns identified by Crockett [8]. However, much work is still needed to define a comprehensive set of primitives that will enable the efficient implementation of a wide range of parallel applications. The first phase of our research focused on the development of the initial PIOUS system; the second phase focuses on interface definition.

3.4 Usage

PVM applications utilize the PIOUS file system by simply calling PIOUS library functions. PIOUS functions will not interfere with PVM functions in any way.

Figure 8 depicts a PVM application that accesses two PIOUS files. The first file accessed is presumed to reside on a default set of data-server hosts, while the second file accessed resides on a set of hosts directly specified in the program text. Though this particular example uses C-language bindings, Fortran-language bindings are supported as well.

[3] To avoid confusion with the transaction-based data-server protocol that PIOUS employs transparently for all read and write operations, a transaction performed at the application level is termed a user-transaction.

```
/* PIOUS example -
 *  Access two files, one on the default set of data-server hosts
 *  and the other on a set of hosts specified in the program text.
 */
#include <pious1.h>

main()
{ int fd;
  char buf[4096];
  struct pious_dsvec dsv[2];

  /* access file on default set of data-server hosts */

  fd = pious_open("file1.dat",
                  PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
                  PIOUS_IRUSR | PIOUS_IWUSR);
  pious_write(fd, buf, (pious_sizet)4096);
  pious_lseek(fd, (pious_offt)0, PIOUS_SEEK_SET);
  pious_read(fd, buf, (pious_sizet)4096);
  pious_close(fd);

  /* access file on hosts marcie and patty */

  dsv[0].hname = "marcie";
  dsv[0].spath = "/users/fred/piousfiles";
  dsv[0].lpath = "/users/fred/piouslogs";

  dsv[1].hname = "patty";
  dsv[1].spath = "/users/fred/piousfiles";
  dsv[1].lpath = "/users/fred/piouslogs";

  fd = pious_sopen(dsv, 2,
                   "file2.dat",
                   PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
                   PIOUS_IRUSR | PIOUS_IWUSR);
  pious_write(fd, buf, (pious_sizet)4096);
  pious_lseek(fd, (pious_offt)0, PIOUS_SEEK_SET);
  pious_read(fd, buf, (pious_sizet)4096);
  pious_close(fd);
}
```

Figure 8: PIOUS usage example

P0	P1	P2	P3

Figure 9: Partitioned sequential access pattern

3.5 Performance

PIOUS performance results are presented for a simple PVM benchmark application. Our intent is to demonstrate the potential benefits of the PIOUS file system. A comprehensive performance study is beyond the scope of this paper.

We define the partitioned access throughput (PAT) benchmark to be a parallel program that reads or writes a file using the partitioned sequential access pattern [8] employed by many parallel computations. Partitioned access divides a linear file into contiguous blocks, one per process, with each block accessed sequentially by a different process. Figure 9 illustrates this file access pattern for a parallel computation consisting of four processes.

The PAT benchmark specifies that file data must be declustered according to the distribution pattern known as disk striping [25]. In disk striping, a linear sequence of fixed size data blocks, called *striping-units*, are distributed in round-robin fashion across some number of storage devices. A *stripe* consists of the set of striping-units distributed in a single round. File size must be sufficient such that each process' partition is a "reasonable" multiple of the stripe size, and each file operation must access exactly one stripe. Benchmark results must be given in terms of the minimum sustained aggregate rate [12], the data rate obtained by all processes as perceived by an outside observer.

To achieve consistent results in our implementation of the PAT benchmark for PIOUS, file access parameters have been chosen so as to minimize variance in the performance of the system software on which PIOUS is layered. File size is $3q$ megabytes, where q is the number of data-servers across which the file is declustered. Scaling file size minimizes variance in the performance of the data-server hosts' native file systems by accessing the same amount of data on each. Striping-unit size is fixed at 8 kilobytes. Employing a single striping-unit size minimizes variance in the performance of the communication subsystem, which is PVM message passing, by keeping data-transport message size constant. Though PIOUS data-servers implement caching, this feature is disabled during benchmarking. Furthermore, file data is flushed from the system buffers of all data-server hosts prior to each execution of the benchmark application.

Performance measurements were collected on the Heterogeneous Environment and Testbed (HEAT) network located at Sandia National Laboratories. Ten SGI R4000 Indigo machines running IRIX 4.0.5 were employed, all directly connected to a DEC Gigaswitch; each machine contained 80 MB of RAM and 2 GB of hard disk storage. PAT benchmark performance was measured for

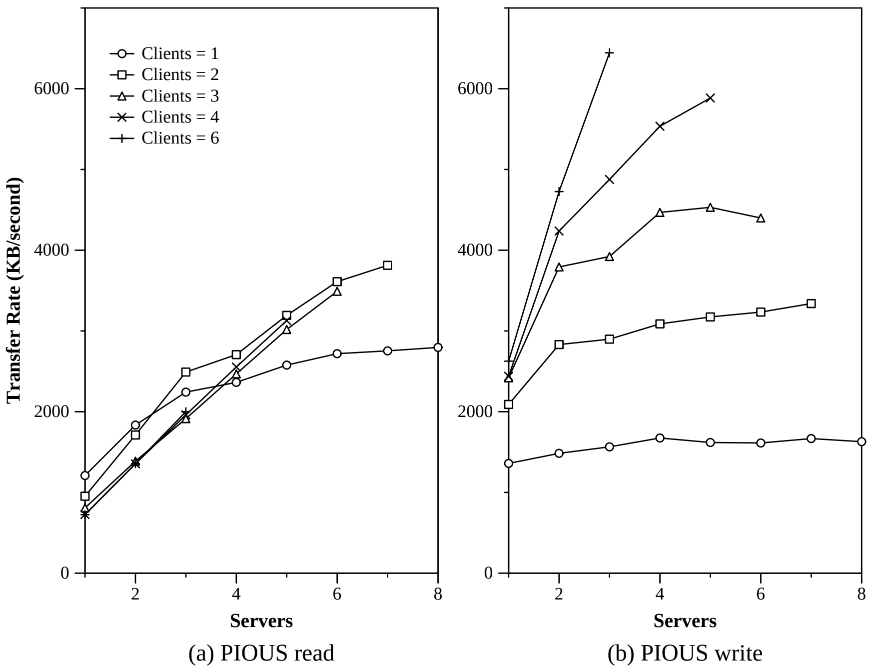

Figure 10: PIOUS read and write performance

configurations of one to six client processes accessing files declustered on one to eight data-servers. To avoid having PAT processes share hosts with PIOUS system processes, the sum of the PAT and PIOUS processes was limited to nine, leaving one machine for performance monitoring.

Figure 10(a) depicts PIOUS read[4] performance, in terms of the observed aggregate bandwidth, as a function of the number of data servers on which the accessed file is declustered. PIOUS read exhibits nearly linear scaling in performance as the number of servers is increased, given a sufficient number of clients. For a small number of servers, PIOUS read exhibits a decrease in absolute performance as the number of clients is increased; this effect is due to the increased complexity in disk access patterns at each server.

Figure 10(b) depicts PIOUS write performance. Though PIOUS write exhibits good scalability for six clients, it is likely that an insufficient number of clients was available to achieve maximum performance for more than one server. For a single client, PIOUS write does not exhibit a significant increase in performance as the number of servers is increased; this is due to a lengthy

[4] PAT read (write) of PIOUS files is referred to simply as PIOUS read (write).

request phase, resulting in little server parallelism, as the client process sends one striping-unit of data to each server.

PAT benchmark results indicate that the PIOUS parallel file system can provide scalable bandwidth when the file access pattern maps well to the data distribution.

4 Conclusions

The field of parallel and distributed I/O is maturing, and research projects as well as commercial systems have evolved to address many of the important issues that impact this critical facet of concurrent computing. These concurrent I/O systems generally emphasize one or more of the following factors: characterizing file access patterns to deliver optimized performance, language extensions to enable appropriate and efficient mapping between disk storage and memory, run time libraries to implement caching and prefetching, concurrency control to ensure correct consistency semantics, low level interface primitives to specify file layout and access, declustering facilities, and failure resilience. All of these are important to a greater or lesser extent, depending upon the underlying hardware platform or distributed system, as well as the application involved. As described, characterizing file access patterns and designing primitives appropriately can reduce the overhead of parallel I/O requests by up to an order of magnitude. Similarly, performance improvements of the order of 10 to 90% can be obtained by using an optimized disk mapping, followed by internal message passing to achieve the target memory mapping. From a different viewpoint, scalability, with concurrency control, at sustained performance levels approaching that of the underlying communications fabric, can be attained via a combination of pipelined requests and transaction based access. Logical partitioning of files based upon user-specified layouts enables a more natural application to file mapping. Finally, systems and communications failures can be tolerated when required, such that no data is lost, with reasonable decreases in performance. These and similar results currently define the status of parallel and distributed I/O systems which will continue to evolve into mature, robust technologies.

References

[1] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[2] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[3] Alok Choudhary, Ian Foster, Geoffrey Fox, Ken Kennedy, Carl Kesselman, Charles Koelbel, Joel Saltz, and Marc Snir. Languages, compilers, and runtime systems support for parallel input-output, 1994. Scalable I/O Initiative Working Paper Number 3. On WWW at http://www.ccsf.caltech.edu/SIO/SIO.html.

[4] Committee on Physical, Mathematical, and Engineering Sciences Federal Coordinating Council for Science, Engineering and Technology, Washington, D.C. *High-Performance Computing and Communications, Grand Challenges 1993 Report*, 1993.

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

[6] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.

[7] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[8] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[9] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, April 1992.

[10] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93*

Workshop on Input/Output in Parallel Computer Systems, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[11] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[12] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[13] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[14] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Univ. of Virginia Computer Science Department, July 1991.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] Fumihide Itoh, Takashi Chikayama, Takeshi Mori, Masaki Sato, Tatsuo Kato, and Tadashi Sato. The design of the PIMOS file system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 1, pages 278–285. ICOT, 1992.

[17] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. Technical Report PCS-TR94-211, Dept. of Math and Computer Science, Dartmouth College, April 1994. Revised May 11, 1994. To appear in *IEEE Parallel and Distributed Technology*.

[18] Steven A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. Technical Report CSTR-950601, Emory University, June 1995.

[19] Steven A. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. *International Journal of Supercomputer Applications*, 9(2), 1995.

[20] Steven A. Moyer and V. S. Sunderam. *PIOUS for PVM: User's Guide and Reference Manual - Version 1*, 1995. Software and documentation available via the PIOUS home page at http://www.mathcs.emory.edu/pious.html.

[21] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[22] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[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] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. Technical Report CS-1994-33, Dept. of Computer Science, Duke University, October 1994.

[25] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[26] Tarvinder Pal Singh and Alok Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[27] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.