A Parallel I/O System for High-Performance Distributed Computing*

Steven A. Moyer
V. S. Sunderam

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

Abstract

PIOUS is a parallel file system architecture for providing process groups access to permanent storage within a heterogeneous network computing environment. PIOUS supports parallel application development by providing coordinated access to *parafile* file objects with guaranteed consistency semantics and a dynamically-selectable fault tolerance level. For performance, PIOUS declusters files to exploit the combined file I/O and buffer cache capacities of multiple interconnected computer systems.

1 Introduction

Programming environments for developing distributed applications with groups of cooperating processes are becoming increasingly important for cost-effective, high-performance computing. Existing parallel-distributed computing environments, such as PVM [Sun90b] and Linda [ABC+90], provide interprocess communication, synchronization and concurrency control, fault tolerance, and process management. However, many parallel applications require or could benefit from a unified parallel I/O system that such environments generally lack.

PIOUS, the Parallel Input/OUtput System, is a parallel file system that provides process groups access to permanent storage within a heterogeneous network computing environment. This paper focuses on the PIOUS architecture and programming model.

*Research supported by the National Science Foundation, under Award No. CCR-9118787, U. S. Department of Energy, under Grant No. DE-FG05-91ER25105, and the Office of Naval Research under grant N00014-93-1-0278.

2 Basic Architecture Principles

The PIOUS architecture embodies a number of principles that we feel are essential to developing a flexible, reliable, and high-performance network parallel file system:

- transport and native file system independence to enhance portability; PIOUS assumes only that mechanisms exist to reliably transport data between co-operating machines, and to access data on permanent storage from a subset of those machines;
- an asynchronous model of operation to increase system parallelism; components of the PIOUS architecture operate independently without the need for explicit synchronization;
- data declustering for cost-effective scalable performance; PIOUS declusters files over multiple interconnected systems to aggregate transfer rate and buffer cache capacities and load-balance access requests;
- integrated concurrency control and fault tolerance mechanisms for guaranteed consistency semantics and reliability; PIOUS provides sequential consistency of access and tolerance of system failures by requiring components of the architecture to interoperate via transaction-based protocols;
- access mechanism and policy independence to facilitate experimentation in file structure and user interface design; components of the PIOUS architecture that provide consistent and reliable data access are separate from components that implement the file system model.

3 PIOUS Software Architecture

The PIOUS software architecture is depicted in Figure 1. PIOUS consists of a set of data servers, a service coordinator, and library routines linked with client processes. An underlying transport mechanism is assumed to carry messages between client processes and components of the PIOUS architecture. PIOUS data servers are assumed to access permanent storage via a native file system.

A PIOUS Data Server (PDS) resides on each machine over which files are declustered. Each PDS provides transaction-based access to the local files that represent a portion of a declustered file. PDS do not interpret the global structure of a file, however, thus separating file access mechanisms from policies. PDS are independent and hence do not communicate, enhancing system parallelism and scalability. Though ideally each PDS accesses a file system local to the machine on which it resides, two PDS may share a file system via a network file service.

A single PIOUS Service Coordinator (PSC) initiates activities within the 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

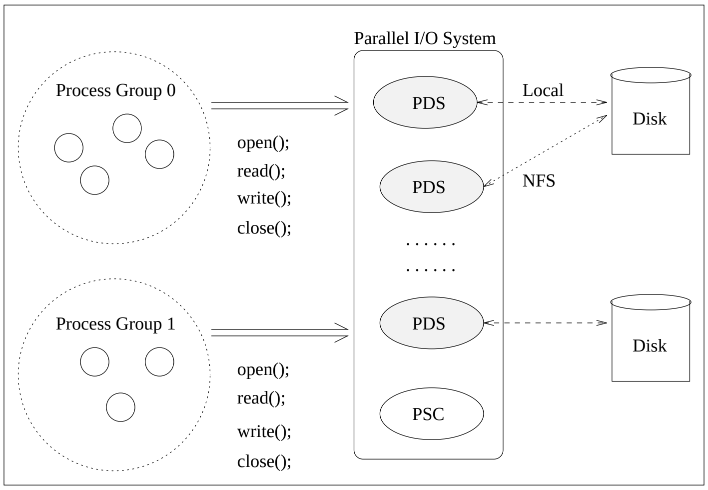

Figure 1: PIOUS software architecture

with those of other processes in the group. Note that the PSC only participates in major system events and not general file access, thus it does not represent a system bottleneck.

Finally, each PIOUS client process is linked with library routines that translate file operations into PDS/PSC service requests. Global file structure and access policies are implemented at this level, facilitating experimentation in user interface design without requiring re-implementation of the PDS or PSC.

The PIOUS architecture emphasizes asynchronous operation and scalable performance via the independence of parallel data servers, the interpretation of global file structure in client library routines, and the lack of a centralized manager for all but major events. Furthermore, the architecture is relatively free of assumptions concerning the underlying transport and storage subsystems.

4 Transactions in PIOUS

Parallel I/O operations in PIOUS are performed within the context of transactions [BHG87], transparently to the user, to provide sequential consistency [Lam79] of access and tolerance of system failures. Referring to Figure 1, client library routines act as transaction managers for the data servers participating in a distributed transaction satisfying a user request.

The PIOUS architecture defines two distinct transaction types: *stable* and *volatile*. Both transaction types guarantee serializability of access. A stable transaction is a "traditional" transaction that employs logging and synchronous disk write operations to also guarantee that coherence is maintained in the event of a system failure. A volatile transaction is a "light-weight" transaction that does not guarantee fault tolerance. Providing two distinct transaction types allows user-selectable levels of fault tolerance to be implemented.

Specifying that PIOUS components interoperate via transaction-based protocols might seem to be an implementation detail rather than an architectural issue. However, employing transaction-based protocols allows elements of the PIOUS architecture to operate independently and obviates the need for special concurrency control and fault tolerance components.

5 PIOUS File Structure and User Interface

A PIOUS file structure and interface design are presented below. To better support parallel applications, our file system model is based on a parallel access file object, called a *parafile*, and a user interface that directly supports the coordinated interaction of cooperating processes.

For the remainder of this text the term PIOUS refers to our particular implementation of the PIOUS architecture. Thus we discuss features of PIOUS not directly specified by the architectural description; e.g. file structure details.

5.1 Parafile Objects

Parafiles are implemented in PIOUS to facilitate parallel data mapping. Parafiles are logically single files 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 parafile is set at the time of creation and does not change for the life of the file.

Each parafile segment resides on a single PIOUS data server. If the number of parafile segments exceeds the maximum number of PDS for declustering the file, then segments are mapped to data servers in a round-robin fashion.

Figure 2 illustrates a parafile object with four segments declustered over two data servers. At each data server a directory contains file entries that represent parafile segments and associated meta-data; the directory path name is the logical file path name at the given server. Implementing parafiles on top of a conventional file system necessitates representing each parafile segment as a separate file.

Because parafiles are two-dimensional, array and tabular data map naturally. Thus developing parallel scientific, database, and sorting applications, among others, should be greatly simplified. Parafiles are based on Kotz's *multifiles* [Kot93] and McElrath's *virtual files* [McE93], two segmented file schemes proposed independently. The terms *file* and *parafile* are used interchangeably for the remainder of this text, as a parafile is the only file object implemented in PIOUS.

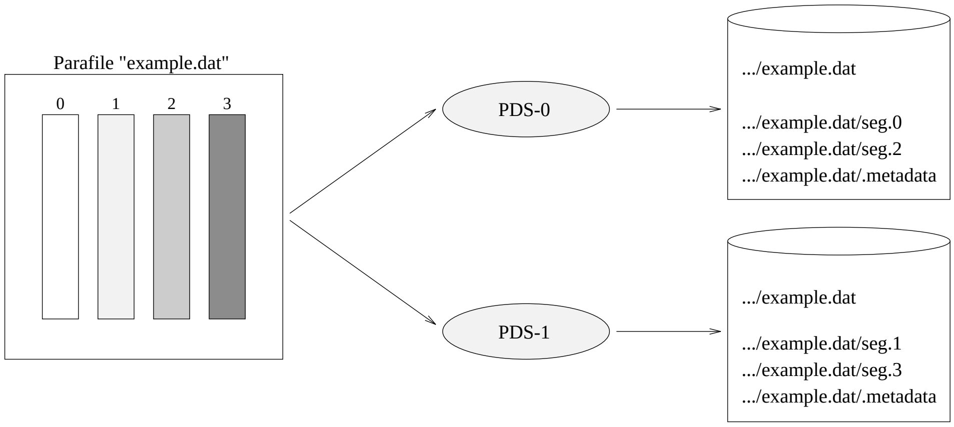

Figure 2: Parafile object

5.2 Parafile Access Semantics

The PIOUS interface provides a process group with three *views* of a parafile object: global, independent, and segmented. Each parafile 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; datum access and file pointer update are atomic.

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

segmented The segmented structure of a parafile is exposed; each process accesses a segment via a local pointer; datum access is atomic.

Under the views defined, parafiles are either accessed by segment or logically mapped to a linear sequence of bytes. A linear byte sequence is defined by ordering the bytes in a parafile by fixed size blocks taken round-robin from each segment. Thus for a block size of b, the first b bytes in the linear sequence are the first b bytes in the lowest numbered segment, the next b bytes in the linear sequence are the first b bytes in the next segment, and so on.

Note that a view only defines the way a parafile is accessed and does not alter the physical representation. Thus a given parafile can always be opened with any view. All processes in a group open a file with the same view.

Parallel file systems are relatively new and an area of active research. Currently no particular file structure or user interface has been embraced as being clearly superior; in fact, most commercial parallel file systems implement a traditional Unix[1] model. The set of file views currently supported in PIOUS is limited, though sufficient to implement a super-set of the common parallel file access patterns identified by Crockett [Cro89]. We feel that until further experience is gained with the system, development of a more sophisticated interface is premature.

5.3 Consistency and Fault Tolerance Semantics

PIOUS provides all processes sequential consistency of access to files opened under any view. Between processes of different groups, some form of flexible file locking is being considered as well.

PIOUS provides parafile 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 [BHG87] or as computed parity information [PGK88]. Such techniques are independent of the file system architecture, provided that multiple independent storage devices can be accessed.

5.4 PIOUS Interface Definition

A PIOUS user interface can now be defined that supports the file model and usage semantics discussed above. For portability of sequential programs, a subset of the PIOUS interface is POSIX [Ins88] equivalent. Interface definitions use C-language bindings, though Fortran language bindings will be implemented as well.

5.4.1 File Primitives

Prior to performing input-output operations, a file must be opened with one of the following functions:

int pious_pfopen(*group*, *path*, *view*, *map*, *faultmode*, *oflag* [, *mode* [, *seg*]])
int pious_pfopens(*servers*, *group*, *path*, *view*, *map*, *faultmode*, *oflag* [, *mode* [, *seg*]])
int pious_open(*path*, *oflag* [, *mode*])

char ***servers*, **group*, **path*;
int *view*, *map*, *faultmode*, *oflag*, *seg*;
pious_modet *mode*;

Pious_pfopen() opens for a process in group *group* the parafile *path* with view *view* and returns a file descriptor for subsequent I/O operations. If the file view

[1] Unix is a registered trademark of X/Open Company Ltd.

specifies a linear mapping, i.e. is global or independent, then the number of bytes taken round-robin from each segment is specified by *map*; otherwise, if the view is segmented, *map* determines which segment is to be accessed. Fault tolerance mode is specified by *faultmode* as either stable or volatile. File access mode is specified via *oflag* as one of read-only, write-only, or read-write; a file creation flag determines if the file is to be created if it does not exist. Upon file creation, *mode* specifies the file permission bits, in accordance with the standard Unix access control mechanism, and the number of parafile segments is determined by *seg*, with a default value equal to the number of data servers. Data is declustered over a predefined set of default data servers.

Pious_pfopens() is identical to pious_pfopen() except that the set of data servers over which the file is declustered is specified via the *servers* argument. This feature allows the user to choose the set of data servers that best meet the needs of the application; e.g. a set of RAID [PGK88] servers for media fault tolerance.

Pious_open() is a POSIX equivalent open function that results in Unix file semantics in most cases. Pious_open(*path*, *oflag*, *mode*) is equivalent to pious_pfopen(NULL, *path*, INDEPENDENT, MAPDEFAULT, VOLATILE, *oflag*, *mode*).

Functions are also provided to close a file and to query file status and configuration; e.g. to determine the number of data servers on which a file is declustered.

5.4.2 Input-Output Primitives

Once a file has been opened, data may be read or written using the following functions:

pious_ssizet pious_read(*fildes*, *buf*, *nbyte*)
pious_ssizet pious_reado(*fildes*, *buf*, *nbyte*, *offset*)
pious_ssizet pious_write(*fildes*, *buf*, *nbyte*)
pious_ssizet pious_writeo(*fildes*, *buf*, *nbyte*, *offset*)
pious_offt pious_lseek(*fildes*, *offset*, *whence*)

int *fildes*, *whence*;
char **buf*;
pious_sizet *nbyte*;
pious_offt *offset*;

Pious_read() attempts to read *nbyte* bytes from file *fildes* into the buffer *buf*. The read operation starts at the file position indicated by the appropriate file pointer, as determined by the view; the file pointer is incremented accordingly.

Pious_reado() performs the same function as pious_read(), except that reading starts at the file position *offset*, relative to the file view, and does not alter any file pointer.

The pious_write() and pious_writeo() functions are the obvious extensions of the corresponding read operations, with data moved from the user buffer to the indicated file.

Finally, the pious_lseek() function repositions the appropriate file pointer for the given view relative to either the current file pointer position or the beginning of the file, as indicated by *whence*.

5.4.3 Maintenance Primitives

In addition to file access primitives, the following basic parafile maintenance primitives are defined to provide a more complete environment:

int pious_chmod(*path*, *mode*)
int pious_chmods(*servers*, *path*, *mode*)
int pious_unlink(*path*)
int pious_unlinks(*servers*, *path*)
int pious_mkdir(*path*, *mode*)
int pious_mkdirs(*servers*, *path*, *mode*)
int pious_rmdir(*path*)
int pious_rmdirs(*servers*, *path*)

char ***servers*, **path*;
pious_modet *mode*;

The maintenance functions extend standard Unix functions to parafiles in the PIOUS environment. Pious_chmod() sets the file access control bits of file *path* to *mode*. Pious_unlink() removes parafile *path*. Pious_mkdir() and pious_rmdir() create and remove directory *path*, respectively, at each data server.

All PIOUS maintenance functions have a "standard" version, that refers to the default data servers, and a "server" version, that refers to specific data servers.

5.4.4 Transaction Primitives

The PIOUS I/O operations of section 5.4.2 are performed within the context of a transaction, transparently to the user, as discussed in section 4. For most applications, the programming model is simplified by hiding the mechanism that is providing concurrency control and fault tolerance. However certain applications may desire to perform multiple access operations all within the context of a single transaction. This functionality is supported in PIOUS via the following functions:

int pious_tbegin(*faultmode*)
int pious_tabort()
int pious_tend()

int *faultmode*;

Pious_tbegin() and pious_tend() mark the beginning and end of a transaction, respectively. All PIOUS access operations that occur between a transaction begin and end point are part of that transaction. The transaction fault tolerance mode is specified by *faultmode* as either stable or volatile, as defined in section 4. An application can abort the current transaction at any time via pious_tabort().

5.4.5 Record Support

As defined, PIOUS read-write operations operate on uninterpreted blocks of bytes. While this is sufficient in an environment with a homogeneous data representation, in a heterogeneous data environment the burden is placed on the application to convert data to some universal format such as XDR [Sun90a].

In a later version of the PIOUS interface we intend to explore record support to address this issue. A solution will probably involve a form of record descriptor, such as employed in the MPI [Mes93] environment.

6 Related Work

The PIOUS architecture has roots in a number of network and parallel computer file systems. A brief overview of this related work is presented below.

Parallel machines commonly provide a parallel file system interface; familiar examples include the Intel CFS [Pie89] and nCUBE Parallel I/O System [DdR92]. Such systems provide parallel applications with varying degrees of file access coordination and control and often employ data declustering to increase aggregate performance. The IBM Vesta [CFPB93] multicomputer file system implements two-dimensional files and a unique stencil-based logical partitioning operator to coordinate parallel file access.

Swift [CL91] and Zebra [HO93] are research network file systems that implement disk striping across multiple servers with RAID-4/5 [PGK88] fault tolerance of media failures. Zebra is unique in that a log-structured file system is employed to allow striping on a per client basis. However, Swift and Zebra are not parallel file systems as they lack support for parallel application development.

Express is a CUBIX-model [Sal87] parallel computing environment that implements a parallel file system interface. However, in network environments, Express does not employ data declustering to aggregate the performance of distributed resources.

PIOUS extends the work of existing parallel file systems by incorporating transactions as a generalized concurrency control and fault tolerance mechanism. Consequently, PIOUS provides parallel applications with guaranteed consistency semantics, asynchronous file operations, and dynamically-selectable fault tolerance modes. Furthermore, to our knowledge PIOUS is the first network parallel file system to employ data declustering for scalable performance.

7 Conclusions

PIOUS is a parallel file system architecture for providing process groups access to permanent storage within a heterogeneous network computing environment. The PIOUS file structure and interface implement a programming model that supports the coordinated interaction of cooperating processes, thereby simplifying parallel applications development. The PIOUS software architecture employs data declustering to exploit distributed resources transparently to the user.

A PIOUS prototype has been implemented and benchmarked [MS94], however space limitations prevent us from presenting performance results in this paper. To summarize, the overhead introduced by the PIOUS software is minimal for modest size data transfers. Aggregate system performance for volatile mode access is limited by the latency and bandwidth of our transport mechanism for two or more data servers; thus we are unable to measure scalability in this mode. Aggregate system performance for stable mode access is disk bandwidth limited; performance scales proportional to the number of disks employed.

8 Acknowledgements

The authors wish to thank the referees for their advice on improving this paper to better meet the conference objectives.

References

[ABC+90] M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore. Adventures with network Linda. *Supercomputer Review*, 10(3):42–46, October 1990.

[BHG87] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[CFPB93] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra J. Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[CL91] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4), Fall 1991.

[Cro89] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[DdR92] 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.

[HO93] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[Ins88] The Institute of Electrical and Electronics Engineers, Inc. *POSIX: IEEE Standard Portable Operating System Interface for Computer Environments*, 1988. IEEE Std 1003.1-1988.

[Kot93] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[McE93] Rodney McElrath. MRJ Inc., Personal Communication, June 1993.

[Mes93] The Message Passing Interface Forum. *Document for a Standard Message-Passing Interface (Draft)*, November 1993.

[MS94] Steven A. Moyer and V. S. Sunderam. PIOUS: A scalable parallel I/O system for distributed computing environments. In *1994 Scalable High Performance Computing Conference*. May, 1994. To appear.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[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.

[Sal87] John Salmon. CUBIX: Programming hypercubes without programming hosts. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 3–9, 1987.

[Sun90a] Sun Microsystems, Inc. *Network Programming Guide*, 1990. External Data Representation Standard: Protocol Specification.

[Sun90b] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.