Overview of the MPI-IO Parallel I/O Interface

Peter Corbett[†], Dror Feitelson[†], Sam Fineberg[*], Yarsun Hsu[†], Bill Nitzberg[*],
Jean-Pierre Prost[†], Marc Snir[†], Bernard Traversat[*], Parkson Wong[*]

†IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

http://lovelace.nas.nasa.gov/MPI-IO/

Published in the proceedings of the
Third Workshop on I/O in Parallel and Distributed Systems,
IPPS '95, Santa Barbara, CA, April 25, 1995

April 17, 1995

Abstract

Thanks to MPI, writing portable message passing parallel programs is almost a reality. One of the remaining problems is file I/O. Although parallel file systems support similar interfaces, the lack of a standard makes developing a truly portable program impossible. It is not feasible to develop large scientific applications from scratch for each generation of parallel machine, and, in the scientific world, a program is not considered truly portable unless it not only compiles, but also runs efficiently.

The MPI-IO interface is being proposed as an extension to the MPI standard to fill this need. MPI-IO supports a high-level interface to describe the partitioning of file data among processes, a collective interface describing complete transfers of global data structures between process memories and files, asynchronous I/O operations, allowing computation to be overlapped with I/O, and optimization of physical file layout on storage devices (disks).

*Computer Sciences Corporation, NASA Contract NAS 2-12961

1 Parallel I/O

It is not feasible to develop large scientific applications from scratch for each generation of parallel machine, and, in the scientific world, a program is not considered truly portable unless it not only compiles, but also runs efficiently. Thanks to MPI [20], writing portable message passing parallel programs is almost a reality. One of the remaining problems is file I/O.

The closest thing to a standard, the UNIX file interface, is ill-suited to parallel computing. The main deficiency of UNIX I/O is that UNIX is designed first and foremost for an environment where files are not shared by multiple processes at once (with the exception of pipes and their restricted access possibilities). In a parallel environment, simultaneous access by multiple processes is the rule rather than the exception. Moreover, parallel processes often access the file in an interleaved manner, where each process accesses a fragmented subset of the file, while other processes access the parts that the first process does not access [18]. UNIX file operations provide no support for such access, and in particular, do not allow access to multiple non-contiguous parts of the file in a single operation.

In addition, the significant optimizations required for efficiency (e.g. grouping [24], two-phase I/O [9], and disk-directed I/O [17]) can only be implemented as part of a parallel I/O environment if it supports a high-level interface to describe the partitioning of file data among processes and a collective interface describing complete transfers of global data structures between process memories and the file. Further efficiencies can be gained via support for asynchronous I/O, allowing computation to be overlapped with I/O, and control over physical file layout on storage devices (disks).

Parallel file systems and programming environments have typically solved the problems of data partitioning and collective access by introducing file modes. The different modes specify the semantics of simultaneous operations by multiple processes. Once a mode is defined, conventional read and write operations are used to access the data, and their semantics are determined by the mode. The most common modes are [2, 15, 16, 25]:

mode	**description**	**examples**
broadcast reduce	all processes collectively access the same data	Express singl PFS global mode CMMD sync-broadcast
scatter gather	all processes collectively access a sequence of data blocks, in rank order	Express multi CFS modes 2 and 3 PFS sync & record CMMD sync-sequential
shared offset	processes operate independently but share a common file pointer	CFS mode 1 PFS log mode
independent	allows programmer complete freedom	Express async CFS mode 0 PFS UNIX mode CMMD local & independent

The common denominator of those modes that actually attempt to capture useful I/O patterns and help the programmer is that they define how data is partitioned among the processes. Some systems do this explicitly without using modes, and allow the programmer to define the partitioning directly. Examples include Vesta [7] and the nCUBE system software [8]. Recent studies show that various simple partitioning schemes do indeed account for most of observed parallel I/O patterns [23].

In addition to the commercial offerings (IBM SP2 PIOFS [6], Intel iPSC CFS [24, 26] and Paragon PFS [11, 27], Ncube [8], and Thinking Machines CM-5 sfs [2, 19], there has been a recent flurry of activity in the research community.

PIOUS [21, 22] and PETSc/Chameleon I/O [13] are both widely available non-proprietary portable parallel I/O interfaces. PIOUS is a PVM-based parallel file interface. Files can be declustered across disks in a round robin fashion. Access modes support globally shared and independent file pointers, and file per node accesses. PETSc/Chameleon I/O runs on top of vendor file systems as well as p4, PICL, and PVM. Two file layouts are supported: sequential (sequential files in the underlying file system), and parallel (where every node accesses its own disk). Global array operations are supported via specific collective calls to perform broadcast/reduce or scatter/gather operations.

PPFS [14] is a general Parallel I/O system built on top of NXLIB intended as a workbench for studying issues and algorithms in Parallel I/O. Modules controlling prefetching, caching, data consistency, access patterns, and file layout can be plugged-in, or defined procedurally, in order to evaluate (and optimize) parallel I/O performance.

Vesta [4, 5, 7, 12] is a library-based parallel file system which runs on the IBM SP2. It provides user-defined parallel views of files for data partitioning, collective operations for data access, and asynchronous operations. Vesta is designed to scale to hundreds of compute nodes, with no sequential bottlenecks in the data-access path.

Jovian [1], PASSION [3, 31], and VIP-FS [10] target out-of-core algorithms. Panda [28, 29, 30] supports a collective global array interface, and optimizes file access by making the file layout correspond to the global array distribution—a 3-d array is stored in 3-d "chunks" in the file.

2 Overview of MPI-IO

The goal of the MPI-IO interface is to provide a widely used standard for describing parallel I/O operations within an MPI message-passing application. The interface establishes a flexible, portable, and efficient standard for describing independent and collective file I/O operations by processes in a parallel application. In a nutshell, MPI-IO is based on the idea that I/O can be modeled as message passing: writing to a file is like sending a message, and reading from a file is like receiving a message. MPI-IO intends to leverage the relatively wide acceptance of the MPI interface in order to create a similar I/O interface. The MPI-IO interface is intended to be submitted as a proposal for an extension of the MPI standard in support of parallel file I/O. The need for such an extension arises from three main reasons. First, the MPI standard does not cover file I/O. Second, not all parallel machines support the same parallel or concurrent file system interface. Finally, the traditional UNIX file system interface is ill-suited to parallel computing.

The MPI-IO interface was designed with the following goals:

1. It was targeted primarily for scientific applications, though it may be useful for other applications as well.

2. MPI-IO favors common usage patterns over obscure ones. It tries to support 90% of parallel programs easily at the expense of making things more difficult in the other 10%.

3. MPI-IO features are intended to correspond to real world requirements, not just arbitrary usage patterns. New features were only added when they were useful for some real world need.

4. MPI-IO allows the programmer to specify high level information about I/O to the system rather than low-level system dependent information.

5. The design favors performance over functionality.

The following, however, were not goals of MPI-IO:

1. Support for message passing environments other than MPI.

2. Compatibility with the UNIX file interface.

3. Support for transaction processing.

4. Support for FORTRAN record oriented I/O.

Emphasis has been put in keeping MPI-IO as MPI-friendly as possible. When opening a file, a communicator is specified to determine which group of processes can get access to the file in subsequent I/O operations. Accesses to a file can be independent (no coordination between processes takes place) or collective (each process of the group associated with the communicator must participate to the collective access). MPI derived datatypes are used for expressing the data layout in the file as well as the partitioning of the file data among the communicator processes. In addition, each read/write access operates on a number of MPI objects which can be of any MPI basic or derived datatype.

3 Data Partitioning in MPI-IO

Instead of defining file access modes in MPI-IO to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach which consists of expressing the data partitioning via MPI derived datatypes. Compared to a limited set of pre-defined access patterns, this approach has the advantage of added flexibility and expressiveness.

MPI derived datatypes are used in MPI to describe how data is laid out in the user's buffer. We extend this use to describe how the data is laid out in the file as well. Thus we distinguish between two (potentially different) derived datatypes that are used: the **filetype**, which describes the layout in the file, and the **buftype**, which describes the layout in the user's buffer. In addition, both **filetype** and **buftype** are derived from a third MPI datatype, referred to as the *elementary* datatype **etype**. The purpose of the elementary datatype is to ensure consistency between the type signatures of **filetype** and **buftype** and to enhance portability by basing them on datatypes rather than bytes. Offsets for accessing data within the file are expressed as an integral number of **etype** items.

The filetype defines a data pattern that is replicated throughout the file (or part of the file—see the concept of displacement below) to tile the file data. It should be noted that MPI derived datatypes consist of fields of data that are located at specified offsets. This can leave "holes" between the fields, that do not contain any data. In the context of tiling the file with the filetype, the process can only access the file data that matches items in the filetype. It cannot access file data that falls under holes (see Figure 1).

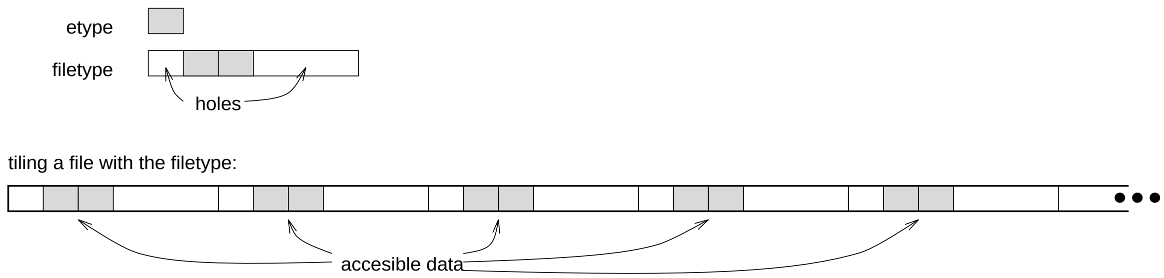

Figure 1: Tiling a file using a filetype

Data which resides in holes can be accessed by other processes which use complementary filetypes (see Figure 2). Thus, file data can be distributed among parallel processes in disjoint chunks.

MPI-IO provides filetype constructors to help the user create complementary filetypes for common distribution patterns, such as broadcast/reduce, scatter/gather, and HPF distributions. In general, we expect most MPI-IO programs will use filetype constructors exclusively, never needing to generate a complicated MPI derived datatype by hand.

Figure 2: Partitioning a file among parallel processes

In MPI-IO, the **filetype** and **etype** are specified at file open time. This is the middle ground between specifying the data layout during file creation (or filesystem creation) and during data access (read/write). The former is too restrictive, as it prohibits accessing a file using multiple patterns simultaneously. In addition, static data layout information must be stored as file metadata, inhibiting file portability between different systems. Specifying the **filetype** at data access time is cumbersome, and it is expected that filetypes will not be changed too often.

In order to better illustrate these concepts, consider a 2-D matrix, stored in row major order in a file, that is to be transposed and distributed among a group of three processes in a row cyclic manner (see Figure 3). The **filetype**s implement the row cyclic data distribution, and can easily be defined via a filetype constructor. The transpose is performed by defining a **buftype** which corresponds to a column of the target matrix in processor memory. Since all processes are performing the same transpose operation, identical **buftype**s can be used for all processes. Note that the elementary datatype allows one to have a generic implementation that applies to any type of 2-D matrix (see appendix for example code).

Note that using MPI derived datatypes leads to the possibility of very flexible patterns. For example, the filetypes need not distribute the data in rank order. In addition, there can be overlaps between the data items that are accessed by different processes. The extreme case of full overlap is the broadcast/reduce pattern.

Using the filetype allows a certain access pattern to be established. But it is conceivable

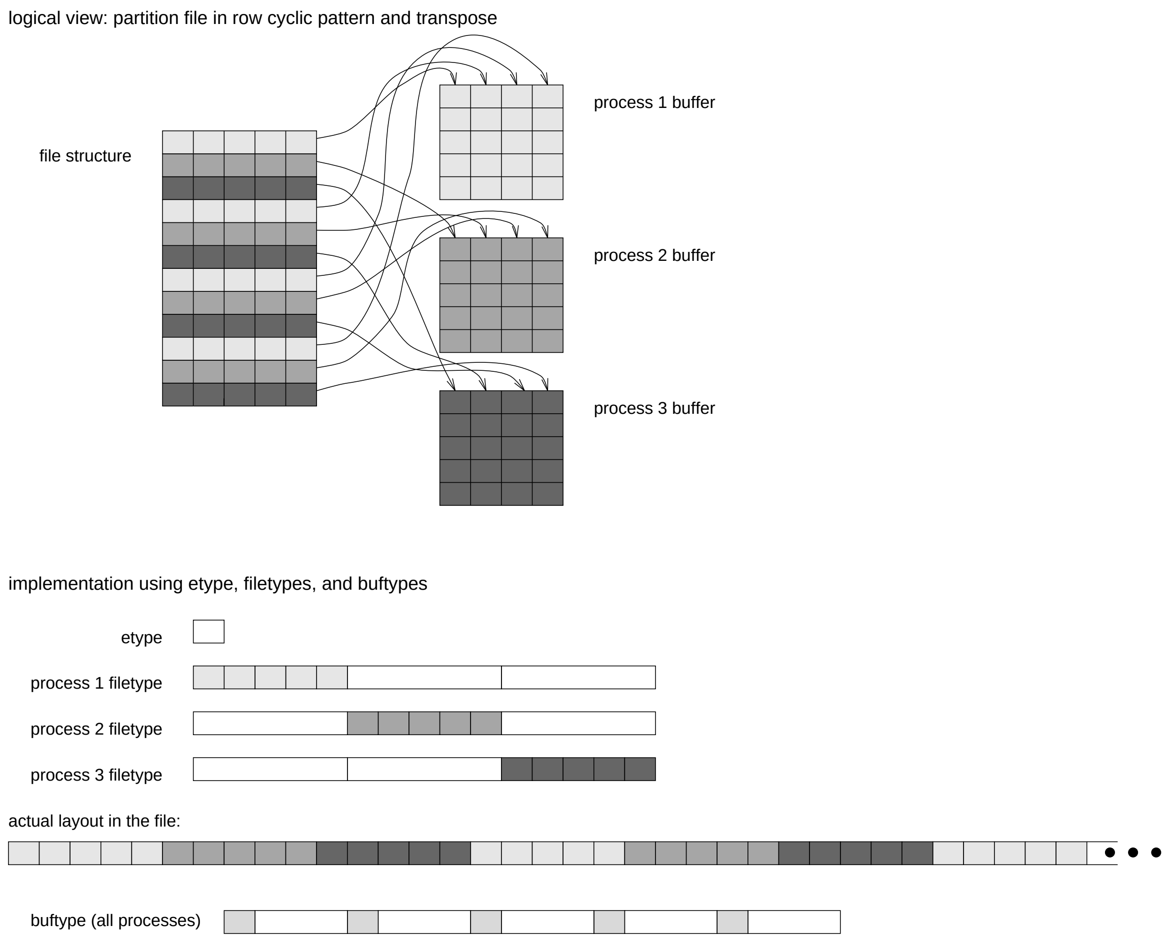

Figure 3: Transposing and partitioning a 2-D matrix

that a single pattern would not be suitable for the whole file. The MPI-IO solution is to define a displacement from the beginning of the file, and have the access pattern start from that displacement. Thus if a file has two or more segments that need to be accessed in different patterns, the displacement for each pattern will skip over the preceding segment(s). This mechanism is also particularly useful for handling files with some header information at the beginning (see Figure 4). Use of file headers could allow the support of heterogeneous environments by storing a "standard" codification of the file data.

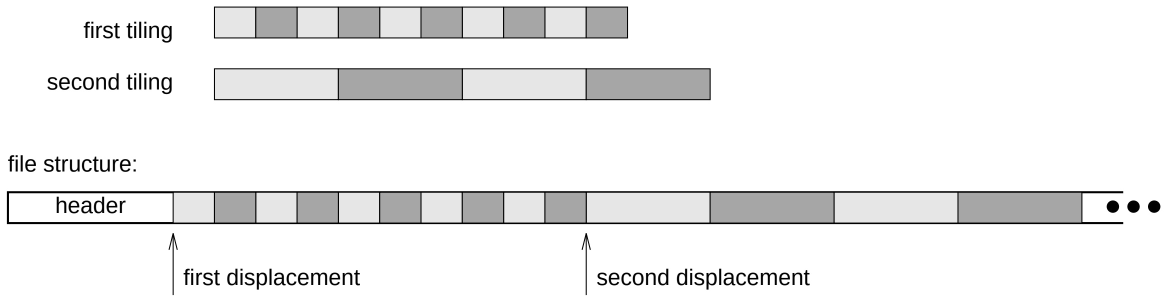

Figure 4: Displacements

4 MPI-IO Data Access Functions

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset vs. implicit file pointer), synchronism (blocking vs. nonblocking), and coordination (independent vs. collective). MPI-IO provides all combinations of these data access functions, including two types of file pointers:

positioning	**synchronism**	**coordination**	
		independent	*collective*
explicit offset	*blocking* (synchronous)	MPIO_Read MPIO_Write	MPIO_Read_all MPIO_Write_all
	nonblocking (asynchronous)	MPIO_Iread MPIO_Iwrite	MPIO_Iread_all MPIO_Iwrite_all
individual file pointers	*blocking* (synchronous)	MPIO_Read_next MPIO_Write_next	MPIO_Read_next_all MPIO_Write_next_all
	nonblocking (asynchronous)	MPIO_Iread_next MPIO_Iwrite_next	MPIO_Iread_next_all MPIO_Iwrite_next_all
shared file pointer	*blocking* (synchronous)	MPIO_Read_shared MPIO_Write_shared	MPIO_Read_shared_all MPIO_Write_shared_all
	nonblocking (asynchronous)	MPIO_Iread_shared MPIO_Iwrite_shared	MPIO_Iread_shared_all MPIO_Iwrite_shared_all

UNIX read() and write() are blocking, independent operations, which use individual file pointers; the MPI-IO equivalents are MPIO_Read_next() and MPIO_Write_next().

4.1 Positioning

UNIX file systems traditionally maintain a system file pointer specifying what offset will be used for the read or write operation. The problem with this interface is that it was primarily designed for files being accessed by a single process. In a parallel environment, we must decide whether a file pointer is shared by multiple processes or if an individual file pointer will be maintained by each process. In addition, parallel programs do not generally exhibit locality of reference within a file [18]. Instead, they tend to move between distinct non-contiguous regions of a file. This means that the process must seek on almost every read or write operation. In addition, in multithreaded environments or when performing I/O asynchronously, it is difficult to ensure that the file pointer will be in the correct position when the read or write occurs.

MPI-IO provides separate functions for positioning with explicit offsets, individual file pointers, and a shared file pointer. The explicit offset operations require the user to specify an offset, and act as atomic seek and read or seek and write operations. The individual and shared file pointer operations use the implicit system maintained offsets for positioning. The different positioning methods are orthogonal; they may be mixed within the same program, and they do not affect each other. In other words, an individual file pointer's value will be unchanged by executing explicit offset operations or shared file pointer operations. The MPI-IO data access functions which accept explicit offsets have no extensions (e.g. MPIO_xxx), the individual file pointer functions have _next appended (e.g. MPIO_xxx_next), and the shared file pointer functions have _shared appended (e.g. MPIO_xxx_shared). In order to allow the implicit offset to be set, two seek functions are also provided (MPIO_Seek and MPIO_Seek_shared).

In general, file pointer operations have the same semantics as explicit offset operations, with the offset argument set to the current value of the system-maintained file pointer.

4.1.1 Offsets

MPI-IO uses two "keys" to describe locations in a file: an MPI datatype and an offset. MPI datatypes are used as templates, tiling the file, and an offset determines an initial position for transfers. Offsets are expressed as an integral number of elementary datatype (etype) items. The etype argument is associated with a file and used to express the filetype, buftype and offset arguments. Therefore, the filetype and buftype datatypes must be directly constructed (i.e. derived datatype) from etype, or their type signatures must be a multiple of the etype signature.

One can view an offset into a file from the global perspective, as an *absolute* count of etypes, or from an individual process's perspective, as a count of etypes *relative* to the filetype. MPI-IO provides both views (see Figure 5). An absolute offset is one that ignores the file partitioning pattern, and is based on the canonical view of a file as a stream of etypes. Absolute offsets can point to anywhere in the file, so they can also point to an item that is inaccessible by this process. In this case, the offset will be advanced automatically to the next accessible item. Therefore specifying any offset in a hole is functionally equivalent to specifying the offset of the first item after the hole. A relative offset is one that only includes the parts of a file accessible by this process, excluding the holes of the filetype associated with the process.

Absolute offsets may be easier to understand if accesses to arbitrary random locations are combined with partitioning the file among processes using filetypes. If such random accesses are not used, relative offsets may be preferable. If the file is not partitioned (all filetypes are identical), absolute and relative offsets are the same.

4.1.2 File Pointers

When a file is opened in MPI-IO, the system creates a set of file pointers to keep track of the current file position. One is a global file pointer, *shared* by all the processes in the communicator group. The others are individual file pointers local to each process in the communicator group, and can be updated *independently.* A shared file pointer only makes sense if all the processes can access the same dataset. This means that all the processes should use the same filetype when opening the file.

The main semantic issue with system-maintained file pointers is how they are updated

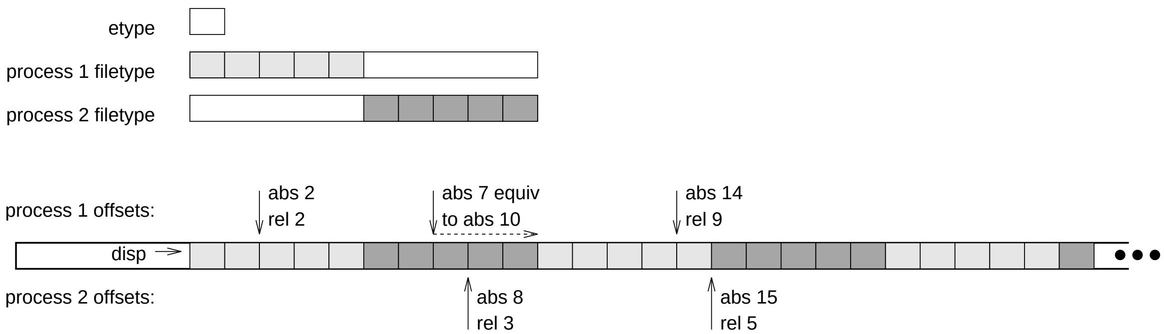

Figure 5: Absolute and relative offsets

by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that was accessed. This principle applies to both types of offsets (absolute and relative), to both types of file pointers (individual and shared), and to all types of I/O operations.

When absolute offsets are used, the file pointer is left pointing to the next **etype** after the last one that was accessed. This **etype** may be accessible to the process, or it may not be accessible. If it is not, then the next I/O operation will automatically advance the file pointer to the next accessible **etype**. With relative offsets, only accessible **etype**s are counted. Therefore it is possible to formalize the update procedure as follows:

$$new\_file\_position = old\_position + \frac{size(buftype) \times bufcount}{size(etype)}$$

Another complication with UNIX I/O operations, is that the system-maintained file pointer is normally only updated when the operation completes. At that stage, it is known exactly how much data was actually accessed (which can be different from the amount requested), and the file pointer is updated by that amount. When MPI-IO nonblocking accesses are made using an individual or the shared file pointer, the update cannot be delayed until the operation completes, because additional accesses can be initiated before that time by the same process (for both types of file pointers) or by other processes (for the shared file pointer). Therefore, the file pointer must be updated at the outset, by the amount of data requested.

Similarly, when blocking accesses are made using the shared file pointer, updating the file pointer at the completion of each access would have the same effect as serializing all blocking accesses to the file. In order to prevent this, the shared file pointer for blocking accesses is updated at the beginning of each access by the amount of data requested. For blocking accesses using an individual file pointer, updating the file pointer at the completion of each access would be perfectly valid. However, in order to maintain the same semantics for all types of accesses using file pointers, the update of the file pointer in this case is also made at the beginning of the access by the amount of data requested.

Although consistent, and semantically cleaner, updating the file pointer at the initiation of all I/O operations differs from accepted UNIX practice, and may lead to unexpected results. Consider the following scenario:

```
MPIO_Read_Next(fh, buff, buftype, bufcount, &status);
MPIO_Write_Next(fh, buff, buftype, bufcount, &status);
```

If the first read reaches the end of the file before completing, the file pointer will be incremented by the amount of data *requested* rather than the amount of data read. Therefore, the file pointer will point beyond the current end of the file, and the write will leave a hole in the file. However, such a problem only occurs if reads and writes are mixed without checking for the end of the file. Although common in a single process workstation environment, we believe this is uncommon in a parallel scientific environment.

4.2 Synchronism

MPI-IO supports the explicit overlap of computation with I/O, hopefully improving performance, through the use of nonblocking data access functions. MPI-IO provides both blocking and nonblocking versions of these functions. As in MPI, the nonblocking versions of the calls are named MPIO_Ixxx, where the I stands for immediate.

A *blocking* I/O call will block until the I/O request is completed. A *nonblocking* I/O call only initiates an I/O operation, but does not wait for it to complete. A separate *request complete* call (MPIO_Wait or MPIO_Test) is needed to complete the I/O request, i.e., to certify that data has been read/written, and it is safe for the user to reuse the buffer. With suitable hardware, the transfer of data out/in the user's buffer may proceed concurrently with computation.

Note that just because a nonblocking (or blocking) data access function completes does not mean that the data is actually written to "permanent" storage. All of the data access functions may buffer data to improve performance. The only way to guarantee data is actually written to storage is by using the MPIO_File_sync call. However, one need not be concerned with the converse problem—once a read operation completes, the data is always available in the user's buffer.

4.3 Coordination

Global data accesses have significant potential for automatic optimization, provided the I/O system can recognize an operation as a global access. Collective operations are used for this purpose. MPI-IO provides both independent and collective versions of all data access operations. Every independent data access function MPIO_xxx, has a collective counterpart MPIO_xxx_all, where _all means that "all" processes in the communicator group which opened the file must participate.

Independent calls do not imply any coordination among processes. Collective calls imply that all processes belonging to the communicator associated with the opened file must participate. However, as in MPI, no synchronization pattern between those processes is enforced by the MPI-IO definition. Any required synchronization may depend upon a specific implementation. Collective calls can be used to achieve certain semantics, as in a scatter-gather operation, but they are also useful to advise the system of a set of independent accesses that may be optimized if combined.

An *independent* I/O request is a request which is executed individually by any of the processes within a communicator group. A *collective* I/O request is a request which is executed by *all* processes within a communicator group. The completion of an independent call only depends on the activity of the calling process. On the other hand, a process can (but is not required to) return from a collective call as soon as its participation in the collective operation is completed. The completion of the call, however, does not indicate that other processes have completed or even started the I/O operation. Thus, a collective call may, or may not, have the effect of synchronizing all calling processes.

From a semantic view, the only difference between collective operations and their independent counterparts is potential synchronization. From a performance view, however, collective operations have the potential to be much faster than their independent counterparts.

5 Miscellaneous Features

5.1 File Layout in MPI-IO

MPI-IO is intended as an interface that maps between data stored in memory and a file. Therefore, the basic access functions only specify how the data should be laid out in a virtual file structure (the file type), not how that file structure is to be stored on one or more disk. This was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information will be necessary in order to optimize disk layout. MPI-IO allows a user to provide this information as *hints* specified when a file is created. These hints do not change the semantics of any of the MPI-IO interfaces, instead they are provided to allow a specific implementation to increase I/O throughput. However, the MPI-IO standard does not enforce that any of the hints will be used by any particular implementation.

5.2 Read/Write Atomic Semantics

When concurrent data accesses involve overlapping data blocks, it is desirable to guarantee consistent interleaving of the accesses. For example, the UNIX read/write interface provides *atomic* access to files. Suppose process A writes a 64K block starting at offset 0, and process B writes a 32K block starting at offset 32K (see Figure 6). The resulting file will have the 32K overlapping block (starting from offset 32K), either come from process A, or from process B. The overlapping block will not be intermixed with data from both processes A and B.

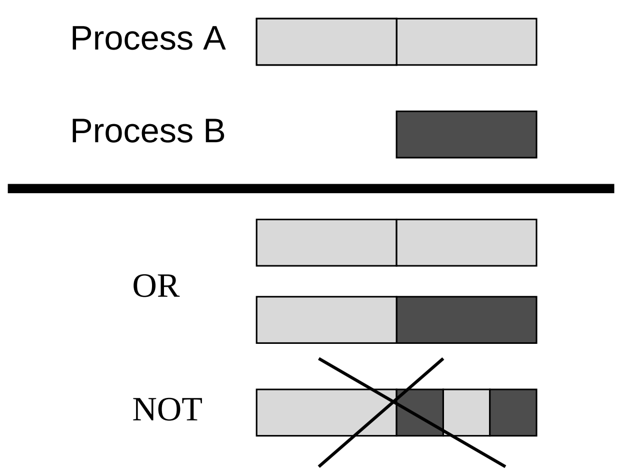

Figure 6: UNIX Atomic Semantics

Similarly, if process A writes a 64K block starting at offset 0, and process B reads a

64K block starting at offset 32K, process B will read the overlapping block, as either old data, or as new data written by process A, but not mixed data.

For performance, most parallel file systems decluster files across multiple storage servers. In this environment, providing atomic data access can be expensive, requiring synchronization and adding overhead to merely check for read/write overlaps. However, as parallel applications rarely issue concurrent, overlapping read and write accesses, MPI-IO does not provide atomic data access by default. To guarantee atomicity, MPI-IO provides a *cautious* mode, enabled via MPIO_File_control.

6 Current Status and Future Developments

Currently, two implementations of MPI-IO are in progress. NASA Ames is working on a portable implementation, primarily targeted at workstation clusters. IBM Research is working on an implementation for the SP2, built on top of IBM's PIOFS file system. Prototypes for the IBM and NASA implementations are targeted for completion by May 15, 1995, with full versions by the end of the year.

General information, copies of the latest draft, and an archive of the MPI-IO mailing list, can be obtained via the WWW at `http://lovelace.nas.nasa.gov/MPI-IO/`. To join the MPI-IO mailing list, send your request to `mpi-io-request@nas.nasa.gov` (see the Web page for details).

7 Acknowledgments

The authors would like to thank William Gropp, David Kotz, and the other contributors to the MPI-IO mailing list who helped with this work.

References

[1] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[2] 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.

[3] 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.

[4] 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), pages 7–14, December 1993.

[5] 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.

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

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

[8] 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.

[9] 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), pages 31–38, December 1993.

[10] Juan Miguel del Rosario, Michael Harry, and Alok Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

[11] Rüdiger Esser and Renate Knecht. Intel Paragon XP/S — architecture and software environment. Technical Report KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Jülich, Germany, r.esser@kfa-juelich.de, April 26 1993.

[12] Dror G. Feitelson, Peter F. Corbett, and Jean-Pierre Prost. Performance of the Vesta parallel file system. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995. To appear.

[13] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[14] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[15] Intel Supercomputer Systems Division. *iPSC/2 and iPSC/860 User's Guide*, April 1991. Order number: 311532-007.

[16] Intel Supercomputer Systems Division. *Intel Paragon XP/S User's Guide*, April 1993. Order number: 312489-01.

[17] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994. Revised November 8, 1994.

[18] 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.

[19] 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.

[20] Message Passing Interface Forum. MPI: A message-passing interface standard, May 1994.

[21] Steven A. Moyer and V. S. Sunderam. A parallel I/O system for high-performance distributed computing. In *Proceedings of the IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, 1994.

[22] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[23] Nils Nieuwejaar and David Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, September 1994.

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

[25] Parasoft Corp. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[26] 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.

[27] Brad Rullman and David Payne. An efficient file i/o interface for parallel applications. Draft distributed at Frontiers '95, 1995.

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

[29] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 218–227, September 1994.

[30] 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*, February 1995. To appear.

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

Appendix: Transposing a 2-D Matrix

The following C code implements the example depicted in Figure 3. A 2-D matrix is to be transposed in a row-cyclic distribution onto m processes. For the purpose of this example, we assume that matrix A is a square matrix of size n by n.

```
read_matrix(char *filename,     /* File containing matrix "A[n][n]" */
            int n,              /* Number of rows (columns) of matrix */
            MPI_Datatype etype, /* Matrix element type (e.g. MPI_DOUBLE) */
            void *localA)       /* Target for transposed matrix */
{
        MPIO_File fh;
        MPI_Datatype filetype, buftype;
        MPI_Status status;
        MPI_Datatype column_t;
        int m, rank, nrows, sizeofetype;

        /*
         * Create row-cyclic filetype for data distribution
         */
        MPIO_Type_hpf_cyclic(MPI_COMM_WORLD, n * n, n, etype, &filetype);
        MPI_Type_commit(&filetype);

        /*
         * Create buftype to transpose matrix into process memory
         */
        MPI_Comm_size(MPI_COMM_WORLD, &m);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        nrows = (rank < n % m) ? (n / m + 1) : (n / m);
        MPI_Type_extent(etype, &sizeofetype);
        MPI_Type_vector(n, 1, nrows, etype, &column_t);
        MPI_Type_hvector(nrows, 1, sizeofetype, column_t, &buftype);
        MPI_Type_commit(&buftype);
        MPI_Type_free(&column_t);

        /*
         * Read, distribute, and transpose the matrix (and cleanup)
         */
        MPIO_Open(MPI_COMM_WORLD, filename, MPIO_RDONLY, MPIO_OFFSET_ZERO,
                  etype, filetype, MPIO_OFFSET_RELATIVE, NULL, &fh);
        MPIO_Read_all(fh, MPIO_OFFSET_ZERO, localA, buftype, 1, &status);
        MPIO_Close(fh);
        MPI_Type_free(&filetype);
        MPI_Type_free(&buftype);
}
```