MPI-IO: A Parallel File I/O Interface for MPI
Version 0.4

mpi-io@nas.nasa.gov

Peter Corbett, Yarsun Hsu, Jean-Pierre Prost, Marc Snir
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

&

Sam Fineberg[1], Bill Nitzberg[1], Bernard Traversat[1], Parkson Wong[1]
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

&

Dror Feitelson
Institute of Computer Science
The Hebrew University
91904 Jerusalem, Israel

Typeset on December 1, 1995

[1]MRJ Inc., NASA Ames Research Center, under Contract NAS2-14303

Contents

1 Introduction 1
1.1 Background 2
1.2 Design Goals 3
1.3 History 4
2 Overview of MPI-IO 4
2.1 Data Partitioning in MPI-IO 5
2.2 MPI-IO Data Access Functions 7
2.3 Positioning 7
2.4 Synchronism 10
2.5 Coordination 10
2.6 End of File 11
2.7 File Layout 11
3 Interface Definitions and Conventions 12
3.1 Collective Calls Argument Values 12
3.2 Etype, Filetype, Buftype, and Offset Relation 12
3.3 Displacement and Offset Types 12
3.4 Return Code and Status 13
3.5 Interrupts 13
3.6 Threads 13
4 File Manipulation 14
4.1 Opening a File (Collective) 14
4.2 Closing a file (Collective) 16
4.3 File Control (Independent/Collective) 17
4.4 Deleting a file (Independent) 18
4.5 Resizing a file (Collective) 19
4.6 File Sync (Collective) 19
5 Data Access With Explicit Offsets 20
5.1 Independent Operations 20
5.2 Collective Operations 23
6 Data Access With Individual File Pointers 26
6.1 Independent Operations 27
6.2 Collective Operations 29
7 Data Access With A Shared File Pointer 32
7.1 Independent Operations 32
7.2 Collective Operations 35
8 Filetype Constructors 39
8.1 Introduction 39
8.2 Read-Broadcast and Write-Reduce Constructors 40

8.3 Scatter / Gather Type Constructors 41
8.4 HPF Filetype Constructors 42
9 Error Handling 46
9.1 MPIO_Errhandler_create (independent) 47
9.2 MPIO_Errhandler_set (independent) 48
9.3 MPIO_Errhandler_get (independent) 48
10 Profiling 48
11 Current Proposal and Future Extensions 49
12 Acknowledgements 51
Bibliography 52
A File hints 55
B Unix Read/Write Atomic Semantics 57
C Filetype Constructors: Sample Implementations and Examples 58
C.1 Support Routines 58
C.2 Sample Filetype Constructor Implementations 60
C.3 Example: Row block distribution of A[100, 100] 64
C.4 Example: Column block distribution of A[100, 100] 66
C.5 Example: Transposing a 2-D Matrix in a Row-Cyclic Distribution . 67
D Justifying Design Decisions 68

1 Introduction

Thanks to MPI [21], 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. Paragraph modified

The significant optimizations required for efficiency (e.g. grouping [25], two-phase I/O [9], and disk-directed I/O [18]) 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. In addition, 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). Paragraph added

Working together, IBM Research and NASA Ames have drafted MPI-IO, a proposal to address the portable parallel I/O problem. In a nutshell, this proposal 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. 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, and optimization of physical file layout on storage devices. Paragraph modified

Let us stress here that currently, the MPI committee has not stated yet if it would adhere to this proposal for defining I/O within MPI. Therefore, no guarantee is made that the proposed interface will be part of MPI in the future. Paragraph added

This document is organized as follows. The remainder of this section includes a discussion of some issues that have shaped the style of the interface. Section 2 presents an overview of MPI-IO as it is currently defined. The next seven sections contain the interface definition itself. Section 3 presents definitions and conventions. Section 4 contains functions for file manipulation, most notably open. Section 5 includes all functions for data access using explicit offsets. Blocking and nonblocking, as well as independent and collective, operations are detailed. Sections 6 and 7 include all functions for data access using system-maintained individual file pointers and shared file pointers respectively. Section 8 presents constructors that can be used to define useful filetypes (the role of filetypes is explained in Section 2). Section 9 presents how the error handling mechanism of MPI is supported by the MPI-IO interface. Section 10 presents the way profiled MPI-IO functions can be defined and the steps involved in creating a profiled MPI-IO library, that can allow post-mortem performance analysis of MPI-IO programs. Finally, Section 11 addresses the limitations of the current proposal and presents possible extensions, that would make the interface more complete and robust. These extensions need more thinking and a better understanding before they can be included in the current proposal. All this is followed by a set of appendices, which contain information about issues that have not been totally resolved yet. The reader can also find there the motivation behind some of our design choices. The first appendix contains a description of MPI-IO "hints" which are used when opening a file. Appendix B explains what we mean when talking about atomic access. Appendix C provides detailed examples of filetype constructors, and Appendix D contains a collection of arguments for Paragraph modified

and against various design decisions.

1.1 Background

Paragraph modified

The closest thing to a standard, the UNIX file interface, is ill-suited to parallel computing. The main deficiency of Unix I/O in the context of parallel computing 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 [19]. 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.

Parallel file systems and programming environments have typically solved this problem 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, 16, 17, 26]:

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

Paragraph modified

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 [5] and the nCUBE system software [8]. Recent studies show that few CFS users use other than mode 0 [19] and that various simple partitioning schemes do indeed account for 99% of observed parallel I/O patterns [24].

Paragraph added

In addition to the commercial offerings (IBM SP2 PIOFS [7], Intel iPSC CFS [25, 27], Intel Paragon PFS [11, 29], Ncube [8], and Thinking Machines CM-5 sfs [2, 20]), there has been a recent flurry of activity in the research community.

Paragraph added

PIOUS [22, 23] 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 [15] 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. Paragraph added

Vesta [4, 5, 6, 12] is a parallel file system which runs on the IBM SP1. 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. Vesta provided the base technology to the IBM PIOFS product running on the SP2. Paragraph added

Jovian [1], PASSION [3, 33], and VIP-FS [10] target out-of-core algorithms. Panda [30, 31, 32] 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. Paragraph added

1.2 Design Goals

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 should establish a flexible, portable, and efficient standard for describing independent and collective file I/O operations by processes in a parallel application. 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, as seen in Section 1.1, 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 design 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.

1.3 History

This work is an outgrowth of the original proposal from IBM [28], but it is significantly different. The main difference is the use of filetypes to express partitioning in an MPI-like style, rather than using special Vesta functions. In addition, filetypes are now used to express various access patterns such as scatter/gather, rather than having explicit functions for the different patterns.

Version 0.2 (dated November 18, 1994) is the draft presented at the Supercomputing '94 birds-of-a-feather session, with new functions and constants prefixed by "MPIO_" rather than "MPI_" to emphasize the fact that they are not part of the MPI standard.

Version 0.3 (dated January 30, 1995) accounts for comments received as of December 31, 1994. It states more precisely what the current MPI-IO proposal covers and what it does not address (yet) (see Section 11). Error handling is now supported (see Section 9). Permission modes are not specified any longer when opening a file (see Section 4.1). Users can now inquire the current size of a file (see Section 4.3). The semantics for updating file pointers has been changed and is identical for both individual and shared file pointers, and for both blocking and nonblocking operations (see Subsection 2.3.2).

Paragraph added

Version 0.4 (dated December 1, 1995) accounts for comments received as of November 30, 1995, and presents a profiling interface (see Section 10), similar to the MPI profiling interface. The file control interface is better defined (see Section 4.3). File hints have been revised (see Section 4.1 and Appendix A) and their definition is more precise (see Section 4.1). Collective operations with shared file pointers are now supported (see Section 7.2). The definition of filetype constructors (see Section 8) and their sample implementations (see Appendix C) have been modified. Finally, the document has been reorganized into a more logical layout, and the bibliography is now more complete.

Paragraph added

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).

2 Overview of MPI-IO

Section reorganized

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 datatypes.

2.1 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**. 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).

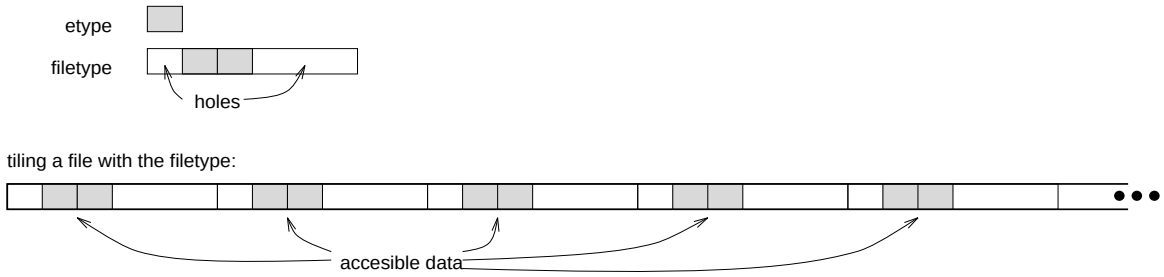

Figure 1: Tiling a file using a filetype

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 (see Section 8).

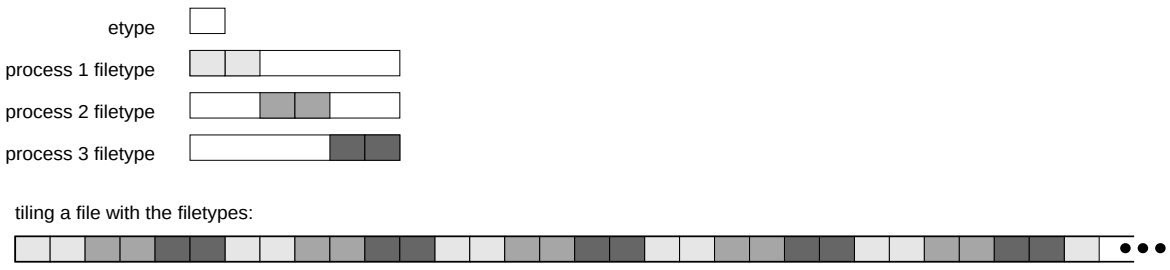

Figure 2: Partitioning a file among parallel processes

In order to better illustrate these concepts, let us consider a 2-D matrix, stored in

row major order in a file, that is to be transposed and partitioned among a group of three processes (see Figure 3). The matrix is to be distributed among the parallel processes in a row cyclic manner. Each process wants to store in its own memory the transposed portion of the matrix which is assigned to it. Using appropriate filetypes and buftypes allows the user to perform that task very easily. In addition, the elementary datatype allows one to have a very generic code that applies to any type of 2-D matrix. The corresponding MPI-IO code example is given in Appendix C.

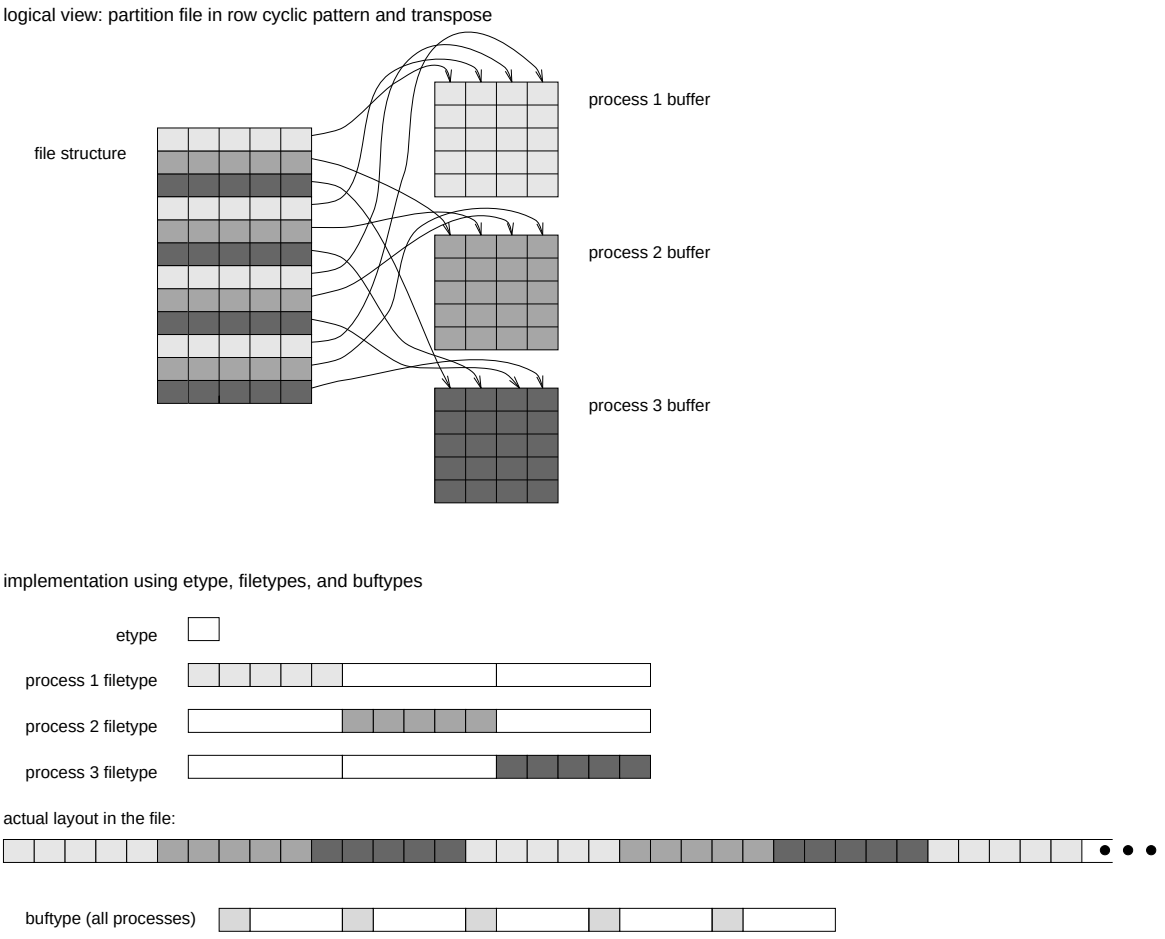

Figure 3: Transposing and partitioning a 2-D matrix

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 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 segments that need to be accessed in different patterns, the displacement for the second pattern will skip over the whole first segment. 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 data representations and data types of the file data.

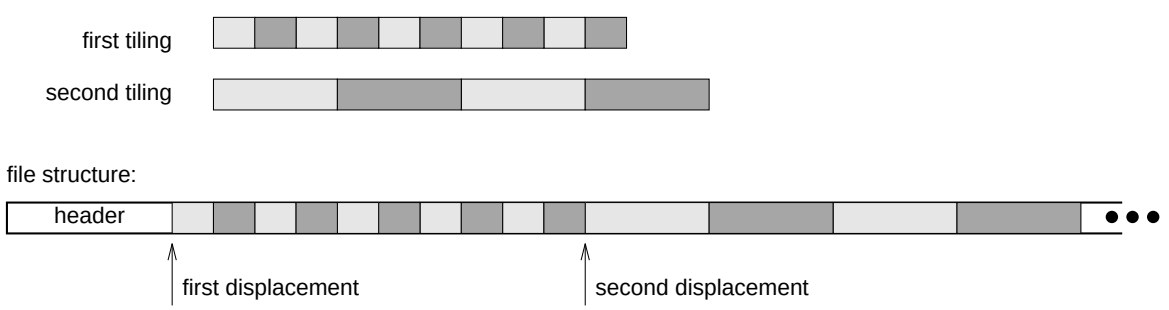

Figure 4: Displacements

2.2 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, individual and shared. Subsection entirely revised

positioning	**synchronism**	**coordination**	
		independent	*collective*
explicit offsets	*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_al l
shared file pointer	*blocking* (synchronous)	MPIO_Read_shared MPIO_Write_shared	MPIO_Read_shared_al l MPIO_Write_shared_a ll
	nonblocking (asynchronous)	MPIO_Iread_shared MPIO_Iwrite_shared	MPIO_Iread_shared_a ll MPIO_Iwrite_shared _all

UNIX read() and write() are blocking, independent operations, and use individual file pointers. The MPI-IO equivalents are MPIO_Read_next() and MPIO_Write_next(). Paragraph added

Operations with explicit offsets are described in Section 5. Operations with individual file pointers are described in Section 6. And operations with shared file pointers are described in Section 7. Paragraph modified

2.3 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, Subsection entirely revised

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 [19]. 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. Moreover, 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 shared file pointers. 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.

2.3.1 Explicit 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 derived 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.

If the file is not partitioned (all filetypes are identical), absolute and relative offsets are the same.

2.3.2 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

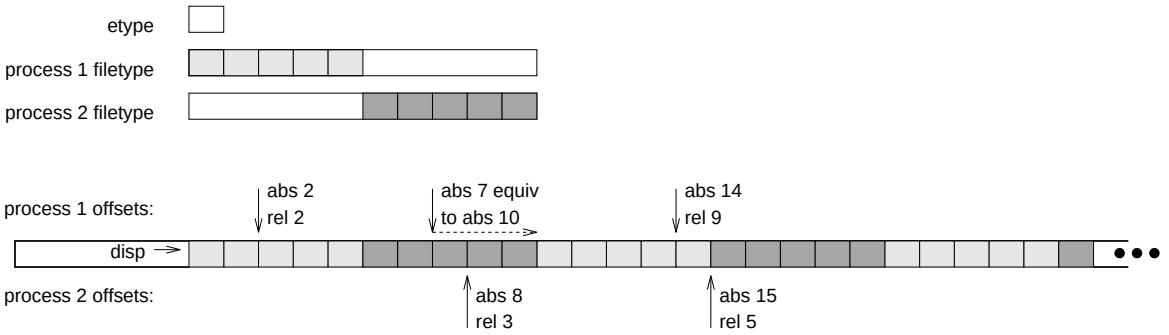

Figure 5: Absolute and relative offsets

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 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 etypes are counted. Therefore it is possible to formalize the update procedure as follows: Paragraph modified

$$new_file_position = old_position + \frac{size(buftype) \times bufcount}{size(etype)}$$

where $bufcount$ is the number of elements of type $buftype$ to be accessed and where $size(datatype)$ gives the number of bytes of actual data (excluding holes) that composes the MPI datatype $datatype$.

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.

2.4 Synchronism

Subsection added

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 functions 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 (MPI_Wait or MPI_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.

2.5 Coordination

Subsection added

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 (to the exception of collective data accesses using a shared file pointer - see Section 7.2). 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 file server(s) 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.

When several independent data accesses involve overlapping data blocks, it may be desirable to guarantee the atomicity of each access, as provided by Unix (see Appendix B). In this case, it is possible to enable the MPIO_CAUTIOUS access mode for the file. Note that the MPIO_CAUTIOUS mode does not guarantee atomicity of accesses between two different MPI applications accessing the same file data, even if they both specify the MPIO_CAUTIOUS mode. Its effect is limited to the confines of the MPI_COMM_WORLD communicator group of the processes that opened the file, typically all the processes in the job. The default access mode, referred to as MPIO_RECKLESS mode in MPI-IO, does not guarantee atomicity between concurrent accesses of the same file data by two parallel processes of the same MPI application.

2.6 End of File

Unlike Unix files, the end of file is not absolute and identical for all processes accessing the file. It depends on the filetype used to access the file and is defined for a given process as the location of the byte following the last elementary datatype item accessible by that process (excluding the holes). It may happen that data is located beyond the end of file for a given process. This data is accessible only by other processes.

When a file grows, because of more data being written to the file or the file being resized, the end of file of all processes accessing the file may change, and data now accessible, but not yet written by any of these processes, will be read as zeroes. Paragraph added

2.7 File Layout

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 filetype), not how that file structure is to be stored on one or more disks. 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 functions, 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.

3 Interface Definitions and Conventions

Section reorganized

3.1 Collective Calls Argument Values

Collective calls may require that all processes, involved in the collective operation, pass the same value for an argument. We will indicate this with the "**[SAME]**" annotation in the function definition, like in the following example:

MPIO_CLOSE(fh)

IN	fh	[SAME] Valid file handle (handle)

Paragraph added

When the "**[SAME]**" annotation is specified, the program is erroneous if each process involved does not provide the same value for the corresponding function argument. Like MPI, MPI-IO does not define the effect of executing erroneous code. In this particular instance, possible resulting effects are:

- An error message may be reported to all processes.
- An error message may be reported to at least one process.
- Different processes may receive inconsistent data.
- The program may deadlock.

It is up to each implementation to define the resulting effect of executing this type of erroneous code.

> *Advice to users.* It is dangerous to rely on synchronization side-effects of the collective I/O operations for program correctness. However, a correct program must be aware of the fact that a synchronization *may* occur. (*End of advice to users.*)
>
> *Advice to implementors.* While vendors may write optimized collective I/O operations, all collective I/O operations can be written entirely using independent I/O operations. (*End of advice to implementors.*)

3.2 Etype, Filetype, Buftype, and Offset Relation

The etype argument is the elementary datatype associated with a file. etype is the unit used to express the filetype, buftype and offset arguments. The filetype and buftype datatypes must be directly derived from etype, or their type signatures must be a multiple of the etype signature. Complete flexibility can be achieved by setting etype to MPI_BYTE. The offset argument used in the read/write interfaces will be expressed in units of the elementary datatype etype.

3.3 Displacement and Offset Types

In FORTRAN, displacements and offsets are expressed as 64 bit integers. In case 64 bit integers are not supported by a specific machine, this does not preclude the use of MPI-IO, but restricts displacements to 2 billion bytes and offsets to 2 billion elementary datatype items (substituting INTEGER*8 variables with INTEGER*4 variables). In C, a new type, MPIO_Offset, is introduced and can be seen as a long long int, if supported, or as a long int otherwise.

3.4 Return Code and Status

All the MPI-IO Fortran interfaces return a success code or a failure code in the IERROR return argument. All MPI-IO C functions also return a success or failure code. The success return code is MPI_SUCCESS. Failure return codes are implementation dependent.

If the end of file is reached during a read operation, the error MPIO_ERR_EOF is returned (either by the blocking read operation or by the function MPI_Test or MPI_Wait applied to the request returned by the nonblocking read operation). The user may write his/her own error handler and associate it with the file handle (see Section 9) in order to process this error.

The number of items actually read/written is stored in the status argument. The MPI_Get_count or MPI_Get_element MPI functions can be used to extract from status (opaque object), the actual number of elements read/written either in etype, filetype or buftype units.

3.5 Interrupts

Like MPI, MPI-IO should be interrupt safe. In other words, MPI-IO calls suspended by the occurrence of a signal should resume and complete after the signal is handled. In case the handling of the signal has an impact on the MPI-IO operation taking place, the MPI-IO implementation should behave appropriately for that situation and very likely an error message should be returned to the user and the relevant error handling take place (see Section 9).

3.6 Threads

Subsection added

Like MPI, MPI-IO should be thread safe. In other words, MPI-IO should implement all accesses to global information (e.g. shared offsets) with adequate locking mechanisms.

Paragraph added

The following sections present all the routines of the MPI-IO interface, grouped by functionalities:

- file manipulation: open, close, file_control, delete, resize, sync,
- data access with explicit offsets: independent blocking (read, write) and nonblocking (iread, iwrite) operations, collective blocking (read_all, write_all) and nonblocking (iread_all, iwrite_all) operations,
- data access with individual file pointers: independent operations (read_next, write_next, iread_next, iwrite_next, seek), and collective operations (read_next_all, write_next_all, iread_next_all, iwrite_next_all),
- data access with a shared file pointer: independent operations (read_shared, write_shared, iread_shared, iwrite_shared, seek_shared), and collective operations (read_shared_all, write_shared_all, iread_shared_all, iwrite_shared_all),
- filetype constructors: read broadcast and write reduce, scatter/gather, High-Performance FORTRAN like distributions,
- error handling,

- and profiling.

Paragraph added

For each routine, the coordination (independent or collective) is specified within parenthesis, the generic syntax along with both C and FORTRAN bindings are given, as well as a brief description of the routine and its arguments. Advices to users and implementors are also provided when appropriate.

4 File Manipulation

4.1 Opening a File (Collective)

Function bindings modified

MPIO_OPEN(comm, filename, amode, disp, etype, filetype, moffset, hints, fh)

IN	comm	[SAME] Communicator that opens the file (handle)
IN	filename	[SAME] Name of file to be opened (string)
IN	amode	[SAME] File access mode (integer)
IN	disp	Absolute displacement (nonnegative offset)
IN	etype	[SAME] Elementary datatype (handle)
IN	filetype	Filetype (handle)
IN	moffset	Offset mode (integer)
IN	hints	Hints to the file system (handle)
OUT	fh	Returned file handle (handle)

```
int  MPIO_Open(MPI_Comm comm, char *filename, MPIO_Mode amode,
             MPIO_Offset disp, MPI_Datatype etype, MPI_Datatype filetype,
             MPIO_Offset_mode moffset, MPIO_Hints hints, MPIO_File *fh)

MPIO_OPEN(COMM, FILENAME, AMODE, DISP, ETYPE, FILETYPE, MOFFSET, HINTS, FH,
             IERROR)
    CHARACTER FILENAME(*)
    INTEGER COMM, AMODE, ETYPE, FILETYPE, MOFFSET,
    INTEGER HINTS, FH, IERROR
    INTEGER*8 DISP
```

Paragraph modified

MPIO_Open opens the file identified by the file name filename, with the access mode amode. The following access modes are supported:

- MPIO_RDONLY - reading only,
- MPIO_RDWR - reading and writing,
- MPIO_WRONLY - writing only,
- MPIO_CREATE - creating file,
- MPIO_DELETE_ON_CLOSE - deleting on close.

These can be combined using the bitwise OR operator.

Advice to users. Note that the Unix append mode is not supported. This mode can be emulated by requesting the current file size (see Section 4.3) and seeking to the end of file before each write operation. (*End of advice to users.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file), where the file is to be opened.

Advice to users. Disp can be used to skip headers, or when the file includes a sequence of data segments that are to be accessed in different patterns. (*End of advice to users.*)

The etype argument specifies the elementary datatype used to construct the filetype, and also the buftype used in read/write operations. Offsets into the file are measured in units of etype. The filetype argument describes what part of the data in the file is being accessed. Conceptually, the file starting from disp is tiled by repeated copies of filetype, until the end. If filetype has holes in it, then the data in the holes is inaccessible by the calling process. However, the disp, etype and filetype arguments can be changed later to access a different part of the file.

The argument moffset specifies how offset values are interpreted. moffset can have one of the two values:

- MPIO_OFFSET_ABSOLUTE - as absolute offsets (count holes in filetype),
- MPIO_OFFSET_RELATIVE - as relative offsets (ignore holes in filetype).

Absolute offsets are interpreted relative to the full extent of the filetype. However, offsets that point to a hole in the filetype will actually access the data immediately following the hole. *Relative* offsets are interpreted relative to the accessible data only (ignoring the holes in the filetype).

The hints argument allows a user to provide various information such as file access patterns and file system specifics (see Appendix A) to direct MPI-IO's optimization. In order to allow as much generality as possible, the MPIO_Hints type is assumed to be an array of (hint_name, hint_value) pairs. Hint_name is an integer that identifies what the hint represents. All hint_names smaller than MPIO_HINT_RESERVED are reserved by MPI-IO, and carry the same meaning across different MPI-IO implementation. Implementation specific hint_names should choose values bigger than MPIO_HINT_RESERVED. Hint_value is either a long long or a pointer to an object of any type. MPIO_Hints is defined as follows: Paragraph modified

```
struct mpio_hints {
    int hint_name;
    union {
        long long value;
        void *ptr;
    } hint_value;
};

typedef struct {
    int num_hints;  /* number of hints in this structure */
    struct mpio_hints hints[ARRAY];
} mpio_hints_t;
```

```
typedef mpio_hints_t *MPIO_Hints;

#define MPIO_HINT_RESERVED      10000
```

Advice to implementors. The number of hints specified by the user (i.e. the size of the structure pointed to by MPIO_Hints) may differ from one opened file to another. If a hint is not specified by the user, a default value should be provided if that hint is used by the implementation. (*End of advice to implementors.*)

Advice to users. Hints are not guaranteed to be used by a specific implementation, and any specified hint may be ignored. (*End of advice to users.*)

Paragraph added

Filename can be prefixed with "mpio-impl:/" to identify the MPI-IO implementation that the open call would like to use if multiple MPI-IO implementation co-exists. Examples of implementations are:

- ufs: - Unix file system,
- mpirun: - mpi-io server communicating to client using mpi,
- socket: - mpi-io server communicating to client using socket,
- piofs: - IBM PIOFS.

Paragraph modified

Files are opened by default in the MPIO_RECKLESS read/write atomic semantics mode.

Processes within the group associated with the communicator comm may pass different values for the disp, filetype, moffset, and hints arguments. However, the comm, filename, amode, and etype argument values must be the same for all these processes.

The file handle returned, fh, can be subsequently used to access the file.

Advice to users. Access permissions are not specified when opening a file. If the file is being created, operating system defaults apply (e.g. Unix command umask). (*End of advice to users.*)

Advice to users. Each process can open a file independently of other processes by using the MPI_COMM_SELF communicator.

If two different MPI applications open the same file, the behavior and atomicity of the file accesses are implementation dependent. The MPIO_CAUTIOUS mode enforces read/write atomicity in the MPI_COMM_WORLD communicator group only. (*End of advice to users.*)

4.2 Closing a file (Collective)

MPIO_CLOSE(fh)

IN	fh	[SAME] Valid file handle (handle)

```
int  MPIO_Close(MPIO_File fh)

MPIO_CLOSE(FH, IERROR)
```

```
INTEGER FH, IERROR
```

Paragraph modified

MPIO_Close closes the file associated with fh. After closing, the content of the file handle fh is destroyed. Future use of fh is erroneous. If the file was opened with the access mode MPIO_DELETE_ON_CLOSE, the file is deleted. However, if there are other processes currently accessing the file, the status of the file and the behavior of future accesses by these processes are implementation dependent.

> *Advice to implementors.* If the file is to be deleted and is opened by other processes, file data may still be accessible by these processes until they close the file or until they exit. The delete operation should be effective only on the last close of that file within this application. (*End of advice to implementors.*)

Paragraph modified

4.3 File Control (Independent/Collective)

MPIO_FILE_CONTROL(fh, cmd, fcntl)

Function bindings modified

IN	fh	[SAME] Valid file handle (handle)
IN	cmd	[SAME] Command
IN/OUT	fcntl	Arguments or return values to the command

```
int  MPIO_File_control(MPIO_File fh, unsigned cmd, MPIO_Fcntl *fcntl)

MPIO_FILE_CONTROL(FH, CMD, FCNTL, IERROR)
    INTEGER FH, CMD, FCNTL, IERROR
```

MPIO_File_Control gets or sets file information about the file associated with the file handle fh. Multiple commands can be issued in one call using the bitwise OR operator. If MPIO_FCNTL_SET is set, the fields indicated by the bits set in cmd will be used to update the file information. If MPIO_FCNTL_SET is not set, the file information requested will be returned. MPIO_Fcntl is defined as follows:

```
    typedef struct MPIO_Fcntl {
        MPI_Comm           comm;
        char              *filename;
        MPIO_Mode          amode;
        MPIO_Offset_mode   moffset;
        MPIO_Offset        disp;
        MPI_Datatype       etype;
        MPI_Datatype       filetype;
        MPIO_Hints        *hints;
        MPIO_Hints        *hints_taken;
        MPIO_Atomicity     atomicity;
        MPIO_Offset        ind_fp;
        MPIO_Offset        shared_fp;
        MPIO_Offset        file_size;
        MPIO_Offset        abs_offset;
    } MPIO_Fcntl;
```

The commands available are:

- MPIO_FCNTL_SET: Set file information.
- MPIO_FCNTL_COMM: Get the communicator associated with the file.
- MPIO_FCNTL_FILENAME: Get the filename.
- MPIO_FCNTL_AMODE: Get or set the file access mode associated with the file.
- MPIO_FCNTL_MOFFSET: Get or set the offset mode associated with the file.
- MPIO_FCNTL_DISP: Get or set the displacement.
- MPIO_FCNTL_ETYPE: Get or set the elementary datatype.
- MPIO_FCNTL_FILETYPE: Get or set the filetype.
- MPIO_FCNTL_HINTS: Get or set the desired values for the hints associated with the file.
- MPIO_FCNTL_HINTS_TAKEN: Get the values for the hints actually used.
- MPIO_FCNTL_ATOMICITY: Get or set the current read/write atomic semantics enforced mode.
- MPIO_FCNTL_IND_FP: Get the current offset of the individual file pointer associated with the file (number of elementary datatype items within the file after the displacement position).
- MPIO_FCNTL_SHARED_FP: Get the current offset of the shared file pointer associated with the file (number of elementary datatype items within the file after the displacement position).
- MPIO_FCNTL_FILE_SIZE: Get the size of the file associated with fh.
- MPIO_FCNTL_ABS_OFFSET: Get the absolute byte offset of the individual file pointer.

Not all fields in fcntl are changeable. When etype or filetype is changed, all file pointers associated with fh are reset to zero.

All calls that get file information are independent calls. All calls that set file information are collective calls. When setting etype, amode, and atomicity, it is erroneous for processes to pass in different values.

4.4 Deleting a file (Independent)

MPIO_DELETE(filename)

IN	filename	Name of the file to be deleted (string)

```
int  MPIO_Delete(char *filename)
```

```
MPIO_DELETE(FILENAME, IERROR)
```

```
    CHARACTER FILENAME(*)
    INTEGER IERROR
```

MPIO_Delete deletes a file. If the file exists it is removed. If there are other processes currently accessing the file, the status of the file and the behavior of future accesses are implementation dependent. If the file does not exist, MPIO_Delete returns a warning error code.

> *Advice to implementors.* If the file to be deleted is opened by other processes, file data may still be accessible by these processes until they close the file or until they exit. (*End of advice to implementors.*)

4.5 Resizing a file (Collective)

MPIO_RESIZE(MPIO_File fh, MPIO_Offset disp)

IN	fh	[SAME] Valid file handle (handle)
IN	disp	[SAME] Displacement which the file is to be truncated at or expanded to (nonnegative offset)

```
int  MPIO_Resize(MPIO_File fh, MPIO_Offset disp)

MPIO_RESIZE(FH, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 DISP
```

MPIO_Resize resizes the file associated with the file handle fh. If disp is smaller than the current file size, the file is truncated at the position defined by disp (from the beginning of the file and measured in bytes). File blocks located beyond that position are deallocated. If disp is larger than the current file size, additional file blocks are allocated and the file size becomes disp. All processes in the communicator group must call MPIO_Resize with the same displacement.

4.6 File Sync (Collective)

MPIO_FILE_SYNC(fh)

IN	fh	[SAME]Valid file handle (handle)

```
int  MPIO_File_sync(MPIO_File fh)

MPIO_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
```

MPIO_File_sync causes the contents of the file referenced by fh to be flushed to permanent storage. All processes in the communicator group associated with the file handle fh

must call MPIO_File_sync. The MPIO_File_sync call returns *after* all processes in the communicator group have flushed to permanent storage the data they have been accessing since they opened the file.

> *Advice to users.* MPIO_File_sync guarantees that all *completed* I/O requests have been flushed to permanent storage. Pending nonblocking I/O requests that have not completed are not guaranteed to be flushed. (*End of advice to users.*)

5 Data Access With Explicit Offsets

Section reorganized

5.1 Independent Operations

5.1.1 Blocking Read (independent)

MPIO_READ(fh, offset, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
IN	offset	File offset (nonnegative offset)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (integer)
OUT	status	Status information (Status)

```
int  MPIO_Read(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Status *status)

MPIO_READ(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPIO_Read attempts to read from the file associated with fh (at the offset position) a total number of bufcount data items having buftype datatype into the user's buffer buff. The data is taken out of those parts of the file specified by filetype. MPIO_Read stores the number of buftype elements actually read in status.

5.1.2 Blocking Write (independent)

MPIO_WRITE(fh, offset, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
IN	offset	File offset (nonnegative offset)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (integer)
OUT	status	Status information (Status)

```
int  MPIO_Write(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Status *status)

MPIO_WRITE(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPIO_Write attempts to write into the file associated with fh (at the offset position) a total number of bufcount data items having buftype datatype from the user's buffer buff. The data is written into those parts of the file specified by filetype. MPIO_Write stores the number of buftype elements actually written in status.

5.1.3 Nonblocking Read (independent)

MPIO_IREAD(fh, offset, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
IN	offset	File Offset (nonnegative offset)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Request *request)

MPIO_IREAD(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPIO_Iread is a nonblocking version of the MPIO_Read interface. MPIO_Iread associates a request handle request with the I/O request. The request handle can be used later to query the status of the read request, using the MPI function MPI_Test, or wait for its completion, using the function MPI_Wait.

The nonblocking read call indicates that the system can start to read data into the supplied buffer. The user should not access any part of the receiving buffer after a nonblocking read is posted, until the read completes (as indicated by MPI_Test or MPI_Wait). MPIO_Iread attempts to read from the file associated with fh (at the offset position) a total number of bufcount data items having buftype type into the user's buffer buff. The number of buftype elements actually read can be extracted from the MPI_Test or MPI_Wait return status.

5.1.4 Nonblocking Write (independent)

MPIO_IWRITE(fh, offset, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
IN	offset	File Offset (nonnegative offset)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int  MPIO_Iwrite(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Request *request)

MPIO_IWRITE(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPIO_Iwrite is a nonblocking version of the MPIO_Write interface. MPIO_Iwrite associates a request handle request with the I/O request. The request handle can be used later to query the status of the write request, using the MPI function MPI_Test, or wait for its completion, using MPI_Wait.

The nonblocking write call indicates that the system can start to write data from the supplied buffer. The user should not access any part of the buffer after the nonblocking write is called, until the write completes (as indicated by MPI_Test or MPI_Wait). MPIO_Iwrite attempts to write into the file associated with fh (at the offset position), a total number of bufcount data items having buftype type from the user's buffer buff. The number of buftype elements actually written can be extracted from the MPI_Test or MPI_Wait return status.

5.2 Collective Operations

5.2.1 Blocking Read (collective)

MPIO_READ_ALL(fh, offset, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
IN	offset	File offset (nonnegative offset)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Read_all(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Status *status)
```

```
MPIO_READ_ALL(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPIO_Read_all is a collective version of the blocking MPIO_Read interface. All processes in the communicator group associated with the file handle fh must call MPIO_Read_all. Each process may pass different argument values for the offset, buftype, and bufcount arguments. For each process, MPIO_Read_all attempts to read, from the file associated with fh (at the offset position), a total number of bufcount data items having buftype type into the user's buffer buff. For each process, MPIO_Read_all stores the number of buftype elements actually read by that process in status.

5.2.2 Blocking Write (collective)

MPIO_WRITE_ALL(fh, offset, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
IN	offset	File offset (nonnegative offset)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Write_all(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Status *status)

MPIO_WRITE_ALL(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPIO_Write_all is a collective version of the blocking MPIO_Write interface. All processes in the communicator group associated with the file handle fh must call MPIO_Write_all. Each process may pass different argument values for the offset, buftype and bufcount arguments. For each process, MPIO_Write_all attempts to write, into the file associated with fh (at the offset position), a total number of bufcount data items having buftype type. For each process, MPIO_Write_all stores the number of buftype elements actually written by that process in status.

5.2.3 Nonblocking Read (collective)

MPIO_IREAD_ALL(fh, offset, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
IN	offset	File Offset (nonnegative offset)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread_all(MPIO_File fh, MPIO_Offset offset, void *buff,
              MPI_Datatype buftype, int bufcount, MPI_Request *request)

MPIO_IREAD_ALL(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPIO_Iread_all is a collective version of the nonblocking MPIO_Iread interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iread_all. Each process may pass different argument values for the offset, buftype and bufcount arguments. For each process in the group, MPIO_Iread_all attempts to read, from the file associated with fh (at the offset position), a total number of bufcount data items having buftype type into the user's buffer buff. MPIO_Iread_all associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual read request or wait for its completion. On each process, MPIO_Iread_all completes when the individual request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes. For each process, the number of buftype elements actually read by that process can be extracted from the MPI_Test or MPI_Wait return status.

5.2.4 Nonblocking Write (collective)

MPIO_IWRITE_ALL(fh, offset, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
IN	offset	File Offset (nonnegative offset)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int MPIO_Iwrite_all(MPIO_File fh, MPIO_Offset offset, void *buff,
             MPI_Datatype buftype, int bufcount, MPI_Request *request)

MPIO_IWRITE_ALL(FH, OFFSET, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPIO_Iwrite_all is a collective version of the nonblocking MPIO_Iwrite interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iwrite_all. Each process may pass different argument values for the offset, buftype and bufcount arguments. For each process in the group, MPIO_Iwrite_all attempts to write, into the file associated with fh (at the offset position), a total number of bufcount data items having buftype type. MPIO_Iwrite_all also associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual write request or wait for its completion. On each process, MPIO_Iwrite_all completes when the individual write request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the supplied buffer after a nonblocking write is called, until the write completes. For each process, the number of buftype elements actually written by that process can be extracted from the MPI_Test or MPI_Wait return status.

6 Data Access With Individual File Pointers

Section reorganized

These functions only use and update the individual current file position maintained by the system. They do not use nor update the shared global file pointer.

In general, these functions have the same semantics as the functions described in Section 5, with the offset argument set to the current value of the system-maintained individual file pointer. This file pointer is updated at the time the I/O is initiated and points to the next data item after the last one that will be actually accessed, possibly accounting for an end of file occurrence. For collective I/O, each individual file pointer is updated independently.

6.1 Independent Operations

6.1.1 Blocking Read (independent)

MPIO_READ_NEXT(fh, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Read_next(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Status *status)

MPIO_READ_NEXT(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Read_next attempts to read from the file associated with fh (at the system maintained current file position) a total number of bufcount data items having buftype datatype into the user's buffer buff. The data is taken out of those parts of the file specified by filetype. MPIO_Read_next returns the number of buftype elements actually read in status. The file pointer is updated by the amount of data requested.

6.1.2 Blocking Write (independent)

MPIO_WRITE_NEXT(fh, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Write_next(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Status *status)

MPIO_WRITE_NEXT(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Write_next attempts to write into the file associated with fh (at the system maintained current file position) a total number of bufcount data items having buftype datatype

from the user's buffer buff. The data is written into those parts of the file specified by filetype. MPIO_Write_next returns the number of buftype elements actually written in status. The file pointer is updated by the amount of data requested.

6.1.3 Nonblocking Read (independent)

MPIO_IREAD_NEXT(fh, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread_next(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Request *request)
```

```
MPIO_IREAD_NEXT(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iread_next is a nonblocking version of the MPIO_Read_next interface. MPIO_Iread_next associates a request handle request with the I/O request. The request handle can be used later to query the status of the read request, using the MPI function MPI_Test, or wait for its completion, using the function MPI_Wait. The file pointer is updated by the amount of data requested. The number of buftype elements actually read can be extracted from the MPI_Test or MPI_Wait return status.

6.1.4 Nonblocking Write (independent)

MPIO_IWRITE_NEXT(fh, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int  MPIO_Iwrite_next(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Request *request)
```

```
MPIO_IWRITE_NEXT(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iwrite_next is a nonblocking version of the MPIO_Write_next interface. MPIO_Iwrite_next associates a request handle request with the I/O request. The request handle can be used later to query the status of the write request, using the MPI function MPI_Test, or wait for its completion, using MPI_Wait. The file pointer is updated by the amount of data requested. The number of buftype elements actually written can be extracted from the MPI_Test or MPI_Wait return status.

6.1.5 Seek (independent)

MPIO_SEEK(fh, offset, whence)

IN	fh	Valid file handle (handle)
IN	offset	File offset (offset)
IN	whence	Update mode (integer)

```
int MPIO_Seek(MPIO_File fh, MPIO_Offset offset, MPIO_Whence whence)

MPIO_SEEK(FH, OFFSET, WHENCE)
    INTEGER FH, WHENCE
    INTEGER*8 OFFSET
```

MPIO_Seek updates the individual file pointer according to whence, which could have the following possible values:

- MPIO_SEEK_SET: the pointer is set to offset
- MPIO_SEEK_CUR: the pointer is set to the current file position plus offset
- MPIO_SEEK_END: the pointer is set to the end of the file plus offset

The interpretation of offset depends on the value of moffset given when the file was opened. If it was MPIO_OFFSET_ABSOLUTE, then offset is relative to the displacement, regardless of what the filetype is. If it is MPIO_OFFSET_RELATIVE, then offset is relative to the filetype (not counting holes). In either case, it is in units of etype.

6.2 Collective Operations

6.2.1 Blocking Read (collective)

MPIO_READ_NEXT_ALL(fh, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Read_next_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
             int bufcount, MPI_Status *status)

MPIO_READ_NEXT_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Read_next_all is a collective version of the MPIO_Read_next interface. All processes in the communicator group associated with the file handle fh must call MPIO_Read_next_all. Each process may pass different argument values for the buftype, and bufcount arguments. For each process, MPIO_Read_next_all attempts to read, from the file associated with fh (at the system maintained current file position), a total number of bufcount data items having buftype type into the user's buffer buff. For each process, MPIO_Read_next_all returns the number of buftype elements actually read by that process in status. The file pointer of each process is updated by the amount of data requested by that process.

6.2.2 Blocking Write (collective)

MPIO_WRITE_NEXT_ALL(fh, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Write_next_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
             int bufcount, MPI_Status *status)

MPIO_WRITE_NEXT_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Write_next_all is a collective version of the blocking MPIO_Write_next interface. All processes in the communicator group associated with the file handle fh must call MPIO_Write_next_all. Each process may pass different argument values for the buftype and bufcount arguments. For each process, MPIO_Write_next_all attempts to write, into the file associated with fh (at the system maintained current file position), a total number of bufcount data items having buftype type. For each process, MPIO_Write_next_all returns the number of buftype elements actually written by that process in status. The file pointer of each process is updated by the amount of data requested by that process.

6.2.3 Nonblocking Read (collective)

MPIO_IREAD_NEXT_ALL(fh, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread_next_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Request *request)
```

```
MPIO_IREAD_NEXT_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iread_next_all is a collective version of the nonblocking MPIO_Iread_next interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iread_next_all. Each process may pass different argument values for the buftype and bufcount arguments. For each process in the group, MPIO_Iread_next_all attempts to read, from the file associated with fh (at the system maintained current file position), a total number of bufcount data items having buftype type into the user's buffer buff. MPIO_Iread_next_all associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual read request or wait for its completion. On each process, MPIO_Iread_next_all completes when the individual request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes. The file pointer of each process is updated by the amount of data requested by that process. For each process, the number of buftype elements actually read by that process can be extracted from the MPI_Test or MPI_Wait return status.

6.2.4 Nonblocking Write (collective)

MPIO_IWRITE_NEXT_ALL(fh, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int  MPIO_Iwrite_next_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Request *request)
```

```
MPIO_IWRITE_NEXT_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iwrite_next_all is a collective version of the nonblocking MPIO_Iwrite_next interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iwrite_next_all. Each process may pass different argument values for the buftype and bufcount arguments. For each process in the group, MPIO_Iwrite_next_all attempts to write, into the file associated with fh (at the system maintained file position), a total number of bufcount data items having buftype type. MPIO_Iwrite_next_all also associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual write request or wait for its completion. On each process, MPIO_Iwrite_next_all completes when the individual write request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the supplied buffer after a nonblocking write is called, until the write is completed. The file pointer of each process is updated by the amount of data requested by that process. For each process, the number of buftype elements actually written by that process can be extracted from the MPI_Test or MPI_Wait return status.

7 Data Access With A Shared File Pointer

Section reorganized

These functions only use and update the global current file position maintained by the system. The individual file pointers are not used nor updated.

Paragraph modified

> *Advice to users.* A shared file pointer only makes sense if all the processes can access the same dataset. This means that all the processes must use the same filetype when opening the file. If this condition is not respected, future file accesses may lead to undefined results. (*End of advice to users.*)

7.1 Independent Operations

7.1.1 Blocking Read (independent)

MPIO_READ_SHARED(fh, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int MPIO_Read_shared(MPIO_File fh, void *buff, MPI_Datatype buftype, int
             bufcount, MPI_Status *status)

MPIO_READ_SHARED(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Read_shared has the same semantics as MPIO_Read with offset set to the global current position maintained by the system.

If multiple processes within the communicator group issue MPIO_Read_shared calls, the data returned by the MPIO_Read_shared calls will be as if the calls were serialized; that is the processes will not have read the same data. The ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After the read operation is initiated, the shared file pointer is updated to point to the next data item after the last one requested.

7.1.2 Blocking Write (independent)

MPIO_WRITE_SHARED(fh, buff, buftype, bufcount, status)

IN	fh	Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int MPIO_Write_shared(MPIO_File fh, void *buff, MPI_Datatype buftype, int
              bufcount, MPI_Status *status)

MPIO_WRITE_SHARED(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Write_shared has the same semantics as MPIO_Write with offset set to the global current position maintained by the system.

If multiple processes within the communicator group issue MPIO_Write_shared calls, the data will be written as if the MPIO_Write_shared calls were serialized; that is the processes will not overwrite each other's data. The ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After the write operation is initiated, the current global file pointer is updated to point to the next data item after the last one requested.

7.1.3 Nonblocking Read (independent)

MPIO_IREAD_SHARED(fh, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread_shared(MPIO_File fh, void *buff, MPI_Datatype buftype,
             int bufcount, MPI_Request *request)

MPIO_IREAD_SHARED(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iread_shared is a nonblocking version of the MPIO_Read_shared interface. MPIO_Iread_shared associates a request handle request with the I/O request. The request handle can be used later to query the status of the read request, using the MPI function MPI_Test, or wait for its completion, using the function MPI_Wait. The number of buftype elements actually read can be extracted from the MPI_Test or MPI_Wait return status.

If multiple processes within the communicator group issue MPIO_Iread_shared calls, the data returned by the MPIO_Iread_shared calls will be as if the calls were serialized; that is the processes will not have read the same data. The ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After the read operation is successfully initiated, the shared file pointer is updated to point to the next data item after the last one requested.

7.1.4 Nonblocking Write (independent)

MPIO_IWRITE_SHARED(fh, buff, buftype, bufcount, request)

IN	fh	Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int  MPIO_Iwrite_shared(MPIO_File fh, void *buff, MPI_Datatype buftype,
             int bufcount, MPI_Request *request)

MPIO_IWRITE_SHARED(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iwrite_shared is a nonblocking version of the MPIO_Write_shared interface. MPIO_Iwrite_shared associates a request handle request with the I/O request. The request handle can be used later to query the status of the write request, using the MPI function MPI_Test, or wait for its completion, using MPI_Wait. The number of buftype elements actually written can be extracted from the MPI_Test or MPI_Wait return status.

If multiple processes within the communicator group issue MPIO_Iwrite_shared calls, the data will be written as if the MPIO_Iwrite_shared calls were serialized; that is the processes will not overwrite each other's data. The ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After the write operation is successfully initiated, the current global file pointer is updated to point to the next data item after the last one requested.

7.2 Collective Operations

Unlike other collective operations, the MPI-IO standard enforces, in the case of shared file pointer operations, a barrier synchronization among the processes associated with the communicator involved in the operation. Section added

The semantics of a collective access using a shared file pointer is that the accesses to the file will be initiated in the order determined by the ranks of the processes within the group associated with the communicator. In addition, in order to prevent subsequent shared offset accesses by the same processes to interfere with this collective access, the call can only return after all the processes within the group have initiated their accesses. In other terms, this semantics implies a barrier synchronization among all the processes of the group, that takes place at the outset of a blocking operation or at the initiation of a nonblocking operation. When the call returns, the shared file pointer points to the next byte accessible, according to the filetype used by all processes, after the last byte requested.

7.2.1 Blocking Read (collective)

MPIO_READ_SHARED_ALL(fh, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Read_shared_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Status *status)

MPIO_READ_SHARED_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Read_shared_all is a collective version of the MPIO_Read_shared interface. All processes in the communicator group associated with the file handle fh must call MPIO_Read_shared_all. Each process may pass different argument values for the buftype,

and bufcount arguments. After all processes of the group have issued their respective calls, each process attempts to read, from the file associated with fh, a total number of bufcount data items having buftype type into the user's buffer buff. For each process, the location in the file at which data is read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had read their data. MPIO_Read_shared_all returns the number of buftype elements read in status. The global shared file pointer is updated by the amounts of data requested by all processes of the group.

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

7.2.2 Blocking Write (collective)

MPIO_WRITE_SHARED_ALL(fh, buff, buftype, bufcount, status)

IN	fh	[SAME] Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	status	Status information (Status)

```
int  MPIO_Write_shared_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Status *status)
```

```
MPIO_WRITE_SHARED_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, STATUS, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, STATUS(MPI_STATUS_SIZE), IERROR
```

MPIO_Write_shared_all is a collective version of the blocking MPIO_Write_shared interface. All processes in the communicator group associated with the file handle fh must call MPIO_Write_shared_all. Each process may pass different argument values for the buftype and bufcount arguments. After all processes of the group have issued their respective calls, each process attempts to write, into the file associated with fh, a total number of bufcount data items having buftype type contained in the user's buffer buff. For each process, the location in the file at which data is written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had written their data. MPIO_Write_shared_all returns the number of buftype elements written in status. The global shared file pointer is updated by the amounts of data requested by all processes of the group.

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

7.2.3 Nonblocking Read (collective)

MPIO_IREAD_SHARED_ALL(fh, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
OUT	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Read request handle (handle)

```
int  MPIO_Iread_shared_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
              int bufcount, MPI_Request *request)

MPIO_IREAD_SHARED_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iread_shared_all is a collective version of the nonblocking MPIO_Iread_shared interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iread_shared_all. Each process may pass different argument values for the buftype and bufcount arguments. After all processes of the group have issued their respective calls, each process attempts to read, from the file associated with fh, a total number of bufcount data items having buftype type into the user's buffer buff. For each process, the location in the file at which data will be read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had read their data. MPIO_Iread_shared_all returns immediately and does not wait for the data to be present in the user's buffer. MPIO_Iread_shared_all associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual read request or wait for its completion. On each process, MPIO_Iread_shared_all completes when the individual request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes. The global shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of buftype elements actually read by that process can be extracted from the MPI_Test or MPI_Wait return status.

7.2.4 Nonblocking Write (collective)

MPIO_IWRITE_SHARED_ALL(fh, buff, buftype, bufcount, request)

IN	fh	[SAME] Valid file handle (handle)
IN	buff	Initial address of the user's buffer (integer)
IN	buftype	User's buffer datatype (handle)
IN	bufcount	Number of buftype elements (nonnegative integer)
OUT	request	Write request handle (handle)

```
int  MPIO_Iwrite_shared_all(MPIO_File fh, void *buff, MPI_Datatype buftype,
             int bufcount, MPI_Request *request)

MPIO_IWRITE_SHARED_ALL(FH, BUFF, BUFTYPE, BUFCOUNT, REQUEST, IERROR)
    <type> BUFF(*)
    INTEGER FH, BUFTYPE, BUFCOUNT, REQUEST, IERROR
```

MPIO_Iwrite_shared_all is a collective version of the nonblocking MPIO_Iwrite_shared interface. All processes in the communicator group associated with the file handle fh must call MPIO_Iwrite_shared_all. Each process may pass different argument values for the buftype and bufcount arguments. After all processes of the group have issued their respective calls, each process attempts to write, into the file associated with fh, a total number of bufcount data items having buftype type contained in the user's buffer buff. For each process, the location in the file at which data will be written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had written their data. MPIO_Iwrite_shared_all returns immediately and does not wait for the data to be present in the user's buffer. MPIO_Iwrite_shared_all associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual write request or wait for its completion. On each process, MPIO_Iwrite_shared_all completes when the individual write request has completed (i.e. a process does not have to wait for all other requests to complete). The user should not access any part of the supplied buffer after a nonblocking write is called, until the write is completed. The global shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of buftype elements actually written by that process can be extracted from the MPI_Test or MPI_Wait return status.

7.2.5 Seek (collective)

MPIO_SEEK_SHARED(fh, offset, whence)

IN	fh	[SAME] Valid file handle (handle)
IN	offset	[SAME] File offset (offset)
IN	whence	[SAME] Update mode (integer)

```
int MPIO_Seek_shared(MPIO_File fh, MPIO_Offset offset, MPIO_Whence whence)

MPIO_SEEK_SHARED(FH, OFFSET, WHENCE)
    INTEGER FH, WHENCE
    INTEGER*8 OFFSET
```

MPIO_Seek_shared updates the global shared file pointer according to whence, which could have the following possible values:

- MPIO_SEEK_SET: the pointer is set to offset
- MPIO_SEEK_CUR: the pointer is set to the current file position plus offset
- MPIO_SEEK_END: the pointer is set to the end of the file plus offset

All the processes in the communicator group associated with the file handle fh must call MPIO_Seek_shared with the same offset and whence. All processes in the communicator group are synchronized with a barrier before the global file pointer is updated.

The interpretation of offset depends on the value of moffset given when the file was opened. If it was MPIO_OFFSET_ABSOLUTE, then offset is relative to the displacement, regardless of what the filetype is. If it is MPIO_OFFSET_RELATIVE, then offset is relative to the filetype (not counting holes). In either case, it is in units of etype.

8 Filetype Constructors

8.1 Introduction

Common I/O operations (e.g., read broadcast, rank-ordered blocks, etc.) are easily expressed in MPI-IO using the previously defined read/write operations and carefully defined filetypes. In order to simplify generation of common filetypes, MPI-IO provides the following MPI datatype constructors.

Although it is possible to implement these type constructors as local operations, in order to facilitate efficient implementations of file I/O operations, all of the filetype constructors have been defined to be *collective* operations. (Recall that a collective operation does not imply a barrier synchronization.)

The set of filetypes created by a single (collective) filetype constructor should be used together in collective I/O operations, with identical offsets, and such that the same number of etype elements is read/written by each process.

> *Advice to users.* The user is not required to adhere to this expected usage; however, the outcome of such operations, although well-defined, will likely be very confusing. (*End of advice to users.*)

Each new filetype created newtype consists of zero or more copies of the base type oldtype, possibly separated by holes. The extent of the new filetype is a nonnegative integer multiple of the extent of the base type. All filetype constructors return a success or failure code.

8.2 Read-Broadcast and Write-Reduce Constructors

8.2.1 Read-Broadcast (collective)

MPIO_TYPE_READ_BCAST(comm, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_read_bcast(MPI_Comm comm, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

MPIO_TYPE_READ_BCAST(COMM, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, OLDTYPE, NEWTYPE,IERROR
```

MPIO_Type_read_bcast generates a set of new filetypes (one for each member of the group) which, when passed to a collective read operation (with identical offsets), will *broadcast* the same data to all readers. Although semantically equivalent to MPI_Type_contiguous(1, oldtype, newtype), a good implementation may be able to optimize the read broadcast operation by using the filetypes generated by this call.

8.2.2 Write-Reduce (collective)

MPIO_TYPE_WRITE_REDUCE(comm, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_write_reduce(MPI_Comm comm, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

MPIO_TYPE_WRITE_REDUCE(COMM, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_write_reduce generates a set of new filetypes (one for each member of the group) which, when passed to a collective write operation, will result in the data from exactly one of the callers being written to the file. A write reduce operation is semantically equivalent to passing the filetype generated by MPI_Type_contiguous(1, oldtype, newtype),

to a collective write operation (with identical offsets), with MPIO_CAUTIOUS mode enabled. A good implementation may be able to optimize the write reduce operation by using the filetypes generated by this call.

Advice to implementors. The choice of which process actually performs the write operation can either be always the same process (e.g. process with rank 0 in the process group) or arbitrary (e.g. the first process issuing the call), since no checking of data identity is to be performed. (*End of advice to implementors.*)

8.3 Scatter / Gather Type Constructors

8.3.1 MPIO_Type_scatter_gather (collective)

MPIO_TYPE_SCATTER_GATHER(comm, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_scatter_gather(MPI_Comm comm, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

MPIO_TYPE_SCATTER_GATHER(COMM, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, OLDTYPE, NEWTYPE, IERROR
```

This filetype allows each process in the group to access a distinct block of the file in rank order. The blocks are identical in size and datatype; each item is of datatype oldtype.

To achieve the scatter or gather operation, the filetypes returned should be passed to a collective read or write operation, giving identical offsets. Generated filetypes will not be identical, but will have the same extent.

8.3.2 MPIO_Type_scatterv_gatherv (collective)

MPIO_TYPE_SCATTERV_GATHERV(comm, count, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	count	number of elements of oldtype in this block (nonnegative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_scatterv_gatherv(MPI_Comm comm, int count,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPIO_TYPE_SCATTERV_GATHERV(COMM, COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, COUNT, OLDTYPE, NEWTYPE, IERROR
```

This type allows each process in the group to access a distinct *block* of the file in rank order. The block sizes and types may be different; each block is defined as count repeated copies of the passed datatype oldtype (i.e. MPI_Type_contiguous(count, oldtype, oldtype)).

To achieve the scatter or gather operation, the filetypes returned should be passed to a collective read or write operation, giving identical offsets.

8.4 HPF Filetype Constructors

The HPF [14] filetype constructors create, for each process in a group, a (possibly different) filetype. When used in a collective I/O operation (with identical offsets), this set of filetypes defines the particular HPF distribution.

Each dimension of an array can be distributed in one of three ways:

- MPIO_HPF_BLOCK - Block distribution
- MPIO_HPF_CYCLIC - Cyclic distribution
- MPIO_HPF_NONE - Dimension not distributed

In order to specify a default distribution argument, the constant MPIO_HPF_DFLT_ARG is used.

For example, ARRAY(CYCLIC(15)) corresponds to MPIO_HPF_CYCLIC with a distribution argument of 15, and ARRAY(BLOCK) corresponds to MPIO_HPF_BLOCK with a distribution argument of MPIO_HPF_DFLT_ARG.

8.4.1 MPIO_Type_hpf (collective)

Subsection modified

HPF distribution of an N-dimensional array onto a P-dimensional grid of logical processors:

MPIO_TYPE_HPF(comm, ndim, dsize, distrib, darg, pdim, psize, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	ndim	[SAME] number of array dimensions (nonnegative integer)
IN	dsize	[SAME] size of distributee in each dimension (array of nonnegative integer)
IN	distrib	[SAME] HPF distribution of distributee in each dimension (array of integer)
IN	darg	[SAME] distribution argument of dimension, e.g. BLOCK(darg), CYCLIC(darg), or MPIO_HPF_NONE (array of integer)
IN	pdim	[SAME] number of processor grid dimensions (nonnegative integer)
IN	psize	[SAME] size of grid in each dimension (array of nonnegative integer)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_hpf(MPI_Comm comm, int ndim, int *dsize, MPIO_Dtype *distrib,
              int *darg, int pdim, int *psize, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

MPIO_TYPE_HPF(COMM, NDIM, DSIZE, DISTRIB, DARG, PDIM, PSIZE, OLDTYPE,
              NEWTYPE, IERROR)
    INTEGER COMM, NDIM, DSIZE(*),DISTRIB(*), DARG(*), PDIM, PSIZE(*),
    OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_hpf generates the filetypes corresponding to the HPF distribution of an ndim-dimensional array of oldtype elements onto a pdim-dimensional grid of logical processors.

Advice to users. The equation $\prod_{i=0}^{pdim-1} psize[i] = size$ must be satisfied, if size denotes the size of communicator comm. (*End of advice to users.*)

The MPIO_Dtype type is defined as follows:

```
typdedef enum {MPIO_HPF_BLOCK, MPIO_HPF_CYCLIC, MPIO_HPF_NONE} MPIO_Dtype;
```

For example, let us generate the filetypes corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following code, assuming there will be six processes attached to the run:

```
ndim = 3;
dsize[0] = 100; distrib[0] = MPIO_HPF_CYCLIC; darg[0] = 10;
dsize[1] = 200; distrib[1] = MPIO_HPF_NONE; darg[1] = 0;
dsize[2] = 300; distrib[2] = MPIO_HPF_BLOCK; darg[2] = MPIO_HPF_DFLT_ARG;
pdim = 2; psize[0] = 2; psize[1] = 3;
MPIO_Type_hpf(MPI_COMM_WORLD, ndim, dsize, distrib, darg,
              pdim, psize, oldtype, &newtype);
```

8.4.2 MPIO_Type_hpf_2d (collective)

Subsection modified

HPF distribution of a *two*-dimensional array:

MPIO_TYPE_HPF_2D(comm, dsize1, distrib1, darg1, psize1, dsize2, distrib2, darg2, psize2, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	dsize1	[SAME] size of distributee in first dimension (nonnegative integer)
IN	distrib1	[SAME] HPF distribution for first dimension (integer)
IN	darg1	[SAME] distribution argument for first dimension (integer)
IN	psize1	[SAME] size of processor grid in first dimension (integer)
IN	dsize2	[SAME] size of distributee in second dimension (nonnegative integer)
IN	distrib2	[SAME] HPF distribution for second dimension (integer)
IN	darg2	[SAME] distribution argument for second dimension (integer)
IN	psize2	[SAME] size of processor grid in second dimension (integer)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_hpf_2d(MPI_Comm comm, int dsize1, MPIO_Dtype distrib1,
             int darg1, int psize1, int dsize2, MPIO_Dtype distrib2,
             int darg2, int psize2, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPIO_TYPE_HPF_2D(COMM, DSIZE1, DISTRIB1, DARG1, PSIZE1, DSIZE2, DISTRIB2,
             DARG2, PSIZE2, OLDTYPE, NEWTYPE, IERROR)
```

```
    INTEGER COMM, DSIZE1, DISTRIB1, DARG1, PSIZE1, DSIZE2, DISTRIB2, DARG2,
    PSIZE2
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_hpf_2d generates the filetypes corresponding to the HPF (distrib1(darg1), distrib2(darg2)) distribution of a two-dimensional (dsize1,dsize2) array of oldtype elements onto a two-dimensional (psize1, psize2) grid of logical processors.

This call is a shorthand for:

```
    dsize[0]=dsize1; distrib[0]=distrib1; darg[0]=darg1; psize[0] = psize1;
    dsize[1]=dsize2; distrib[1]=distrib2; darg[1]=darg2; psize[1] = psize2;
    MPIO_Type_hpf(comm, 2, dsize, distrib, darg, 2, psize, oldtype, &newtype);
```

8.4.3 MPIO_Type_hpf_block (collective)

Subsection modified

HPF BLOCK distribution of a one-dimensional array:

MPIO_TYPE_HPF_BLOCK(comm, dsize, darg, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	dsize	[SAME] size of distributee (nonnegative integer)
IN	darg	[SAME] distribution argument, e.g. BLOCK(darg) (integer)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_hpf_block(MPI_Comm comm, int dsize, int darg,
            MPI_Datatype oldtype, MPI_Datatype *newtype)

MPIO_TYPE_HPF_BLOCK(COMM, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_hpf_block generates the filetypes corresponding to the HPF BLOCK distribution of a one-dimensional dsize element array of oldtype.

This call is a shorthand for:

```
    MPI_Comm_size(comm, &size);
    distrib = HPF_TYPE_BLOCK;
    MPIO_Type_hpf(comm, 1, dsize, distrib, darg, 1, size, oldtype, &newtype);
```

Subsection modified

8.4.4 MPIO_Type_hpf_cyclic (collective)

HPF CYCLIC distribution of a one-dimensional array:

MPIO_TYPE_HPF_CYCLIC(comm, dsize, darg, oldtype, newtype)

IN	comm	[SAME] communicator to be used in MPIO_Open (handle)
IN	dsize	[SAME] size of distributee (nonnegative integer)
IN	darg	[SAME] distribution argument, e.g. CYCLIC(darg) (integer)
IN	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPIO_Type_hpf_cyclic(MPI_Comm comm, int dsize, int darg,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

MPIO_TYPE_HPF_CYCLIC(COMM, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
```

MPIO_Type_hpf_cyclic generates the filetypes corresponding to the HPF CYCLIC distribution of a one-dimensional dsize element array of oldtype.

This call is a shorthand for:

```
MPI_Comm_size(comm, &size);
distrib = HPF_TYPE_CYCLIC;
MPIO_Type_hpf(comm, 1, dsize, distrib, darg, 1, size, oldtype, &newtype);
```

9 Error Handling

The error handling mechanism of MPI-IO is based on that of MPI. Three new error classes, called MPIO_ERR_UNRECOVERABLE, MPIO_ERR_RECOVERABLE and MPIO_ERR_EOF are introduced. They respectively contain all unrecoverable I/O errors, all recoverable I/O errors, and the error associated with a read operation beyond the end of file. Each implementation will provide the user with a list of supported error codes, and their association with these error classes.

Each file handle has an error handler associated with it when it is created. Three new predefined error handlers are defined. MPIO_UNRECOVERABLE_ERRORS_ARE_FATAL considers all I/O errors of class MPIO_ERR_UNRECOVERABLE as fatal, and ignores all other I/O errors. MPIO_ERRORS_RETURN ignores all I/O errors. And MPIO_ERRORS_ARE_FATAL considers all I/O errors as fatal.

> *Advice to implementors.* MPIO_UNRECOVERABLE_ERRORS_ARE_FATAL should be the default error handler associated with each file handle at its creation. When a fatal error (I/O related or not) occurs, open files should be closed (and optionally deleted if they were opened with the MPIO_DELETE access mode), and all I/O buffers should be flushed before all executing processes are aborted by the program. However, these issues remain implementation dependent. (*End of advice to implementors.*)

New functions allow the user to create (function MPIO_Errhandler_create) new MPI-IO error handlers, to associate (function MPIO_Errhandler_set) an error handler with an opened file (through its file handle), and to inquire (function MPIO_Errhandler_get) which error handler is currently associated with an opened file.

The attachment of error handlers to file handles is purely local: different processes may attach different error handlers to the same file handle.

9.1 MPIO_Errhandler_create (independent)

MPIO_ERRHANDLER_CREATE(function, errhandler)

IN	function	User-defined error handling function
OUT	errhandler	MPI error handler (handle)

```
int MPIO_Errhandler_create(MPIO_Handler_function function, MPI_Errhandler
              *errhandler)

MPIO_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

MPIO_Errhandler_create registers the user routine function for use as an MPI error handler. Returns in errhandler a handle to the registered error handler.

The user routine should be a C function of type MPIO_Handler_function, which is defined as:

```
    typedef void (MPIO_Handler_function)(MPIO_File *, int *, MPI_Datatype *,
                                         int*, MPI_Status *, int *, ...)
```

The first argument is the file handle in use, the second argument is the error code to be returned by the MPI routine. The third argument is the buffer datatype associated with the current access to the file (the current access to the file is either the current blocking access to the file, or the current request MPI_tested or MPI_waited for, associated with a nonblocking access to the file). The fourth argument is the number of such buffer datatype items requested by the current access to the file. The fifth argument is the status returned by the current access to the file. And the sixth argument is the request number associated with the current access to the file (this number is relevant for nonblocking accesses only). The number of additional arguments and their meanings are implementation dependent. Addresses are used for all arguments so that the error handling function can be written in FORTRAN.

9.2 MPIO_Errhandler_set (independent)

MPIO_ERRHANDLER_SET(fh, errhandler)

IN	fh	Valid file handle (handle)
IN	errhandler	New MPI error handler for opened file (handle)

```
int MPIO_Errhandler_set(MPIO_File fh, MPI_Errhandler errhandler)

MPIO_ERRHANDLER_SET(FH, ERRHANDLER, IERROR)
    INTEGER FH, ERRHANDLER, IERROR
```

MPIO_Errhandler_set associates the new error handler errhandler with the file handle fh at the calling process. Note that an error handler is always associated with the file handle.

9.3 MPIO_Errhandler_get (independent)

MPIO_ERRHANDLER_GET(fh, errhandler)

IN	fh	Valid file handle (handle)
OUT	errhandler	MPI error handler currently associated with file handle (handle)

```
int MPIO_Errhandler_get(MPIO_File fh, MPI_Errhandler *errhandler)

MPIO_ERRHANDLER_GET(FH, ERRHANDLER, IERROR)
    INTEGER FH, ERRHANDLER, IERROR
```

MPIO_Errhandler_get returns in errhandler the error handler that is currently associated with the file handle fh at the calling process.

10 Profiling

Section added

MPI-IO provides a profiling interface similar to that of MPI, that allows profiling, tracing or debugging tools to gather run time information for specific MPI-IO programs. This information can then be analyzed post-mortem, in order to characterize the I/O workload and I/O performance of these programs.

In order to enable this interface, each implementation should provide a separate set of "PMPIO_" prefixed functions used by the profiled version of the "MPIO_" prefixed functions. We will refer to the set of "PMPIO_" prefixed functions as the profiling version of the MPI-IO library.

For example, in order to accumulate the total elapsed time spent in independent blocking read operations using explicit offsets, the user could modify the implementation of the MPIO_Read function as follows:

```
    static double totaltime = 0;
    int MPIO_Read(MPIO_File fh, MPIO_Offset offset, void *buff,
```

```
                    MPI_Datatype buftype, int bufcount, MPI_Status *status) {
        double t0 = MPI_Wtime();
        int result = PMPIO_Read(fh, offset, buff, buftype, bufcount, status);

        totaltime += MPI_Wtime() - t0;
        return result;
    }
```

to create a profiled version of the MPIO_Read function.

Using the C pre-processor macro:

```
#ifdef MPIO_PROFILING
#   ifdef __STDC__
#      define FUNCTION(name) P##name
#   else
#      define FUNCTION(name) P/**/name
#   endif
#else
#   define FUNCTION(name) name
#endif
```

and defining each user visible MPI-IO function in the library as follows:

```
int FUNCTION(MPIO_Any)(/* Appropriate args */) {
   /* Useful content */
}
```

allows one to produce either the normal version or the profiling version of the MPI-IO library, by altering the state of the MPIO_PROFILING macro variable.

Without changing the source code, the user can then choose to run either a non-instrumented version of her code by issuing a command similar to:

```
cc mysource.o -lmpi -o mysource.orig
```

where library libmpi.a contains the normal version of the MPI-IO functions, or a profiled version of her code by issuing a command similar to:

```
cc mysource.o -lmympi -lpmpi -lmpi -o mysource.prof
```

where library libmympi.a contains the profiled version of the MPI-IO functions which intercept some of the MPI-IO functions, and library libpmpi.a contains the "PMPIO_" prefixed MPI-IO functions. Note that by specifying both libmympi.a and libmpi.a libraries, in that order, we guarantee the precedence of the profiled version over the normal version of each MPI-IO function, as well as the use of the normal version of each MPI-IO function for which the user did not define a profiled version.

11 Current Proposal and Future Extensions

The current proposal is not final and will evolve. Additions to it are definitely required to make the interface more complete and robust. Section moved

Currently, the problem of heterogeneity of data representations across machine architectures is not addressed. As stated above, filetypes are used to partition file data. Their purpose is not to ensure type consistency between file data accessed and user's buffer data, nor are they intended to handle type conversion between file data and user's buffer data. Therefore, file data can be currently considered as untyped data and has no data representation associated with it. Research must be carried out in order to come up with a standard for storing persistent data in a machine independent format and for encoding in the file metadata type information of the file data (a file header could be used as a repository for these metadata).

The error handling mechanism (see Section 9) is currently primitive, built on top of the MPI error handling mechanism. Further investigation is required in order to verify if this approach is appropriate and robust enough.

No real support for accessing MPI-IO files from a non MPI-IO application is currently provided. Additional functions should enable the transfer of MPI-IO files to other file systems, as well as the importation of external files into the MPI-IO environment. However, the user can easily provide the import functionality for a given external file system (eg Unix) by writing a single process program as follows:

```
int         fd;
int         nread;
char        buffer[4096];
MPIO_File   fh;
MPIO_offset offset;
MPIO_Status status;

fd = open("source_file", O_RDONLY);
MPIO_Open(MPI_COMM_WORLD, "target_file", MPIO_CREATE||MPIO_WRONLY,
          MPIO_OFFSET_ZERO, MPI_BYTE, MPI_BYTE, MPIO_OFFSET_ABSOLUTE,
          NULL, &fh);
offset = MPIO_OFFSET_ZERO;
while ((nread = read(fd, buffer, 4096)) != 0) {
   MPIO_Write(fh, offset, buffer, MPI_BYTE, nread, &status);
   offset += nread;
}
close(fd);
MPIO_Close(fh);
```

A very similar program could be written to export an MPI-IO file.

Let us also stress that nothing currently prevents the user from creating an MPI-IO file with a given number of processes and accessing it later with a different number of processes. This can be achieved by reopening the file with the appropriate filetypes and communicators.

Paragraph added

Nothing prevents two distinct applications to concurrently update the same MPI-IO file. Currently, MPI-IO does not provide any locking mechanism which prevents data inconsistency and enforce access atomicity across applications. Locking read and write operations would expand the MPIO_CAUTIOUS mode to concurrent applications (in a more constrained way however, since this would result in a serialization of concurrent accesses).

The current proposal also lacks availability of status information about MPI-IO files. The user currently has no way of inquiring any information about a non opened MPI-IO

file, nor has (s)he the possibility of inquiring the identity of the file owner, the dates and times of the creation/last modification of the file, or the access permissions to the file.

These issues are only some of the main issues that need to be addressed by a real standard for parallel I/O. They will be incorporated into our proposal incrementally, as time permits. Our emphasis has been to first define the basic functions a standard for parallel I/O should provide to allow concurrent access to shared file data in a user-friendly and efficient way. This initial set composing the current interface is designed in such a way that extensions to it can be introduced easily and progressively.

12 Acknowledgements

The current MPI-IO proposal greatly benefits from comments made by MPI-IO mailing list participants. Among them, let us thank here William Gropp (Argonne National Laboratory), David Kotz (Dartmouth College), and John May (Lawrence Livermore National Laboratory) for their help in improving the current definition of MPI-IO.

Section added

Bibliography

Reference added

[1] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz, "*Jovian: a framework for optimizing parallel I/O*". In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, IEEE Computer Society Press, pp. 10–20, Oct 1994.

[2] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker, "*CMMD I/O: a parallel Unix I/O*". In 7th *International Parallel Processing Symposium*, pp. 489–495, Apr 1993.

Reference added

[3] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur, "*PASSION: parallel and scalable software for input-output*". Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, Sep 1994.

Reference added

[4] P. F. Corbett, S. J. Baylor, and D. G. Feitelson, "*Overview of the Vesta parallel file system*". In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pp. 1–16, 1993. Also published in Computer Architecture News 21(5), pp. 7–14, Dec 1993.

[5] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, "*Parallel access to files in the Vesta file system*". In *Supercomputing '93*, pp. 472–481, Nov 1993.

[6] P. F. Corbett and D. G. Feitelson, "*Design and implementation of the Vesta parallel file system*". In *Scalable High-Performance Comput. Conf.*, pp. 63–70, May 1994.

Reference added

[7] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, M. Chi, R. Colao, B. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek, "*Parallel file systems for the IBM SP computers*". *IBM Systems Journal* 34(2), pp. 222–248, Jun 1995.

[8] E. DeBenedictis and J. M. del Rosario, "*nCUBE parallel I/O software*". In 11th *Intl. Phoenix Conf. Computers & Communications*, pp. 117–124, Apr 1992.

Reference added

[9] J.-M. del Rosario, R. Bordawekar, and A. Choudhary, "*Improved parallel I/O via a two-phase run-time access strategy*". In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pp. 56–70, 1993. Also published in Computer Architecture News 21(5), pp. 31–38, Dec 1993.

Reference added

[10] J. M. del Rosario, M. Harry, and A. 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] R. Esser and R. Knecht, "*Intel Paragon XP/S — architecture and software environment*". Technical Report KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Jülich, Germany, Apr 1993. Reference added

[12] D. G. Feitelson, P. F. Corbett, and J.-P. Prost, "*Performance of the Vesta parallel file system*". In *Proceedings of the Ninth International Parallel Processing Symposium*, Apr 1995. Reference added

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

[14] High Performance Fortran Forum, "*High performance fortran language specification*". May 1993.

[15] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal, "*PPFS: A high performance portable parallel file system*". Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, Jan 1995. Reference added

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

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

[18] D. Kotz, "*Disk-directed I/O for MIMD multiprocessors*". Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, Jul 1994. Revised Nov 1994. Reference added

[19] D. Kotz and N. Nieuwejaar, "*Dynamic file-access characteristics of a production parallel scientific workload*". In *Supercomputing '94*, pp. 640–649, Nov 1994.

[20] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, "sfs*: A parallel file system for the CM-5*". In *Proceedings of the 1993 Summer USENIX Conference*, pp. 291–305, 1993. Reference added

[21] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*. May 1994.

[22] S. 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. Reference added

[23] S. 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*, pp. 71–78, 1994. Reference added

[24] N. Nieuwejaar and D. Kotz, "*A multiprocessor extension to the conventional file system interface*". Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, Sep 1994. Reference added

[25] B. Nitzberg, "*Performance of the iPSC/860 Concurrent File System*". Technical Report RND-92-020, NAS Systems Division, NASA Ames, Dec 1992. Reference added

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

Reference added [27] P. Pierce, "*A concurrent file system for a highly parallel mass storage system*". In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pp. 155–160, 1989.

[28] J-P. Prost, M. Snir, P. F. Corbett, and D. G. Feitelson, *MPI-IO, A Message-Passing Interface for Concurrent I/O*. Research Report 19712 (87394), IBM T. J. Watson Research Center, Aug 1994.

Reference added [29] B. Rullman and D. Payne, "*An efficient file i/o interface for parallel applications*". Draft distributed at Frontiers '95, 1995.

Reference added [30] 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*, pp. 218–227, Sep 1994.

Reference added [31] K. E. Seamons and M. Winslett, "*An efficient abstract interface for multidimensional array I/O*". In *Proceedings of Supercomputing '94*, pp. 650–659, Nov 1994.

Reference added [32] 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*, Feb 1995.

Reference added [33] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh, "*PASSION runtime library for parallel I/O*". In *Proceedings of the Scalable Parallel Libraries Conference*, pp. 119–128, Oct 1994.

A File hints

File hints let users, when opening a file or altering an opened file characteristics, specify file access patterns, or file system specifics to the underlying file system to optimize file access. Hints do not change the semantics of any of the MPI-IO interfaces. Hints may allow a specific implementation to increase I/O throughput. However, the MPI-IO standard does not guarantee that any of the hints given will be used by any implementation. Section modified

It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. In particular, file control operations, setting or inquiring values to unsupported hints, should be handled as if these hints were not specified. Needless to say, hints are not mandatory. However, for each hint used by a specific implementation, a default value must be provided in order to be used when the user does not specify a value for this hint.

Some potentially useful hints are outlined below. Specific implementations are free to define these hints as well as additional hints. However, in order to maintain compatibility between implementations, additional hints should not reuse the same hint names for other purposes than these listed here. In addition, if a specific implementation supports any of these hints, the associated hint names should have values which are less than MPIO_HINT_RESERVED.

These hints are mainly concerned with layout of data on parallel I/O devices, and with access patterns. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. If specified, the "**[SAME]**" annotation stipulates that the hint values provided by all participating processes must be identical, otherwise the program is erroneous.

MPIO_STRIPING_UNIT**[SAME]:** This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data taken from one I/O node (resp. storage device) before progressing to the next node (resp. storage device), when striping across a number of nodes (resp. storage devices); it is expressed in bytes. This hint is relevant only when the file is created, and if used, this data should be maintained by the file server(s) as part of the metadata associated with the file. A good size for a striping unit is the amount of consecutive data accessed independently by a process, or if this is too small, a multiple of that size.
The hint value is an integer contained in the hint_value.value field of the hint.

MPIO_STRIPING_FACTOR**[SAME]:** This hint specifies the number of I/O nodes (resp. storage devices) that the file should be striped across, and is relevant only when the file is created.
The hint value is an integer contained in the hint_value.value field of the hint.

MPIO_IO_NODE_LIST**[SAME]:** In addition to defining the striping factor, one may want to give the explicit list of I/O nodes (resp. storage devices) that should be used. This hint is relevant only when the file is created.
The hint value (field hint_value.ptr) points to the list. The type of the list items is implementation dependent.

MPIO_NB_PROC**[SAME]:** This specifies the number of parallel processes that will typically be assigned to run programs that access this file, and can be used at file creation.
The hint value is an integer contained in the hint_value.value field of the hint.

MPIO_ACCESS_STYLE: This hint specifies the manner in which the file will be accessed until the file is closed or until the program terminates.
The hint value (field hint_value.value) is an integer equal to one of the constants MPIO_READ_ONCE, MPIO_WRITE_ONCE, MPIO_READ_MOSTLY, MPIO_WRITE_MOSTLY, ORed by one of the constants MPIO_SEQUENTIAL, MPIO_REVERSE_SEQUENTIAL, MPIO_RANDOM.

MPIO_PARTITIONING_PATTERN[**SAME**]: This hint specifies how the file data will be partitioned among the computing processes in the most common access pattern. A specific implementation can then elect to use a layout that is optimized for this pattern rather than a simple striping across all I/O nodes (resp. storage devices). This hint is relevant only when the file is created.
The partitioning pattern can be expressed concisely via a displacement, an elementary datatype and a filetype constructor. Therefore, the hint value (field hint_value.ptr) points to an object of the following type:

```
typedef struct {
   MPIO_Offset      displacement;
   MPI_Datatype     etype;
   MPIO_Constructor constructor;
} mpio_partitioning_pattern;
```

The `MPIO_Constructor` type is defined as follows:

```
typedef struct {
   int  type;
   union {
      struct {
         MPI_Datatype  oldtype;
      } bcast_reduce_param;     /* for a broadcast/reduce constructor */
      struct {
         int           nproc;
         int           count;
         MPI_Datatype  oldtype;
      } scatter_gather_param;   /* for a scatter/gather constructor */
      struct {
         int           ndim;
         int          *dsize;
         MPIO_Dtype   *distrib;
         int          *darg;
         int           pdim;
         int          *psize;
         MPI_Datatype  oldtype;
      } hpf_param;              /* for a general HPF constructor */
      struct {
         int           dsize;
         int           darg;
         MPI_Datatype  oldtype;
```

```
            } hpf_block_param;        /* for a block HPF constructor */
            struct {
               int           dsize;
               int           darg;
               MPI_Datatype  oldtype;
            } hpf_cyclic_param;       /* for a cyclic HPF constructor */
         } param;
      } MPIO_Constructor;
```

where type has one of the values MPIO_BROADCAST_REDUCE, MPIO_SCATTER_GATHER, MPIO_HPF, MPIO_HPF_BLOCK, MPIO_HPF_CYCLIC.

It is expected that, as implementations of MPI-IO emerge, additional hints will be introduced. Possible candidates are the following:

- **preallocation size:** preallocating disk space for a new file can cut the overhead induced by repeated automatic extensions of the file;
- **amount of data sharing:** specifying how much data is expected to be accessed by more than one process allows one to determine before hand how much buffer cache should be allocated for accessing the file;
- **rate of data use:** this is an estimate of the throughput needs of the application for accessing the file, that can be used to better allocate disk blocks, buffer caches and better define the prefetching policy (e.g. data size and prefetching operation frequency);
- **caching mode:** cache or do not cache file data;
- **prefetching mode:** prefetch or do not prefetch file data.

In addition, it should be noted that specific implementations are free to interpret the hints in slightly different ways. For example, the following table outlines possible interpretations for an MPI-IO implementation based on the IBM Parallel I/O File System [7]:

hint name	*interpretation*
MPIO_STRIPING_UNIT	BSU size
MPIO_STRIPING_FACTOR	number of cells
MPIO_IO_NODE_LIST	base node

B Unix Read/Write Atomic Semantics

The Unix file system read/write interfaces provide *atomic* access to files. For example, 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). With no synchronization, 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.

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

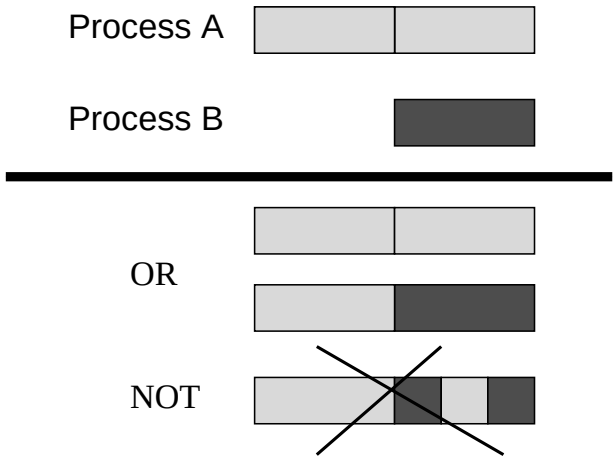

Figure 6: Unix Atomic Semantics

data, or as new data written by process A, but not mixed data. When files are declustered on multiple storage servers, similar read/write atomicities need to be guaranteed. All data of a single read that spans multiple parallel storage servers must be read entirely before or after all data of a write to the same data has proceeded. A simple and inefficient solution to enforce this semantics is to serialize all overlapped I/O. Actually, it is worse than that, all I/O would need to be synchronized, and checked for overlap before they could proceed. However, more efficient techniques are available to ensure correct ordering of parallel point sourced reads and writes without resorting to full blown synchronization and locking protocols. Some parallel file systems, like IBM Vesta [6], provide support to implement such checking. If it is known, that no overlapping I/O operations will occur, or the application is only reading the file, I/O can proceed in a *reckless* mode (i.e. no checking). Reckless mode is the default mode when opening a file in MPI-IO. This implies that users are responsible for writing correct programs (i.e. non-overlapping I/O requests or read only). MPI-IO also supports a *cautious* mode, that enforces read/write atomic semantics. Be aware that this mode may lead to lower performance.

C Filetype Constructors: Sample Implementations and Examples

Section entirely revised

C.1 Support Routines

```
/*
 *  MPIO_Type_set_bounds_absolute - surround a datatype with holes
 *                                  (increasing the extent);
 *                                  displacement and upper bound in bytes
 */
MPIO_Type_set_bounds_absolute(
        MPI_Aint displacement,  /* Displacement from the
                                   lower bound in bytes  */
        MPI_Aint ub,            /* Upper bound in bytes  */
        MPI_Datatype oldtype,   /* Old datatype          */
        MPI_Datatype *newtype)  /* New datatype          */
{
```

```
1           int block[3];
2           MPI_Datatype type[3];
3           MPI_Aint disp[3];
4           int count;
5
6           if (displacement == 0) {
7                   count = 2;
8                   block[0] = 1;
9                   type[0] = oldtype;
10                  disp[0] = 0;
11                  block[1] = 1;
12                  type[1] = MPI_UB;
13                  disp[1] = ub;
14          } else {
15                  count = 3;
16                  block[0] = 1;
17                  type[0] = MPI_LB;
18                  disp[0] = 0;
19                  block[1] = 1;
20                  type[1] = oldtype;
21                  disp[1] = displacement;
22                  block[2] = 1;
23                  type[2] = MPI_UB;
24                  disp[2] = ub;
25          }
26          /*
27           * newtype = { (LB, 0), (oldtype, displacement), (UB, ub) }
28           */
29          return (MPI_Type_struct(count, block, disp, type, newtype));
30  }
31
32  /*
33   *  MPIO_Type_set_bounds - surround a datatype with holes
34   *                         (increasing the extent);
35   *                         displacement and upper bound in counts of extent
36   */
37  MPIO_Type_set_bounds(
38          int disp,               /* Displacement from the
39                                     lower bound in extents */
40          int ub,                 /* Upper bound in extents */
41          MPI_Datatype oldtype,   /* Old datatype           */
42          MPI_Datatype *newtype)  /* New datatype           */
43          int ub,
44          MPI_Datatype oldtype,
45          MPI_Datatype *newtype)
46  {
47          int extent;
48
```

```
        MPI_Type_extent(oldtype, &extent);
        return (MPIO_Type_set_bounds_absolute(disp * extent, ub * extent,
                                              oldtype, newtype));
}

/*
 *  MPIO_Type_bounds - create a datatype composed of a hole
 */
MPIO_Type_bounds(
        MPI_Aint lb,             /* Lower bound in bytes */
        MPI_Aint ub,             /* Upper bound in bytes */
        MPI_Datatype *newtype)   /* New datatype         */
{
        int block[2];
        MPI_Datatype type[2];
        MPI_Aint disp[2];

        block[0] = 1;
        type[0] = MPI_LB;
        disp[0] = lb;
        block[1] = 1;
        type[1] = MPI_UB;
        disp[1] = ub;
        /*
         * newtype = { (LB, lb), (UB, ub) }
         */
        return (MPI_Type_struct(2, block, disp, type, newtype));
}
```

C.2 Sample Filetype Constructor Implementations

C.2.1 MPIO_Type_scatter_gather Sample Implementation

```
/*
 *  MPIO_Type_scatter_gather - generate scatter/gather filetypes to
 *                             access data blocks in rank order;
 *                             blocks are identical in size and datatype
 */
MPIO_Type_scatter_gather(
        MPI_Comm comm,                  /* Communicator group           */
        MPI_Datatype oldtype,           /* Block datatype               */
        MPI_Datatype *newtype)          /* New datatype                 */
{
        int size, rank;

        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);
        return (MPIO_Type_set_bounds(rank, size, oldtype, newtype));
```

```
}
```

C.2.2 HPF BLOCK Sample Implementation

```
/*
 *  MPIO_Type_hpf_block - generate filetypes for an
 *                        HPF BLOCK(darg) distribution
 *
 *             oldtype ARRAY(dsize)
 *       !HPF$ PROCESSORS COMM(size)
 *       !HPF$ DISTRIBUTE ARRAY(BLOCK(darg)) ONTO COMM
 */
MPIO_Type_hpf_block(
        MPI_Comm comm,                  /* Communicator group           */
        int dsize,                      /* Size of distributee          */
        int darg,                       /* Distribution argument        */
        MPI_Datatype oldtype,           /* Old datatype                 */
        MPI_Datatype *newtype)          /* New datatype                 */
{
        int size, rank;
        int nblocks, nprocessors;
        MPI_Datatype block1;
        MPI_Aint extent;
        int rc;

        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);
        /*
         * Compute and check distribution argument
         */
        if (darg == MPIO_HPF_DFLT_DARG)
                darg = (dsize + size - 1) / size;
        if (darg * size < dsize)
                return (MPIO_ERR_ARG);
        nprocessors = (dsize + (darg - 1)) / darg;
        /*
         * Compute size of my block
         */
        if ((rank + 1) < nprocessors) {
                nblocks = darg;                         /* whole block   */
        } else if ((rank + 1) == nprocessors) {
                nblocks = dsize - rank * darg;          /* partial block */
        } else {
                nblocks = 0;                            /* empty block   */
        }
        MPI_Type_extent(oldtype, &extent);
        /*
         * Create filetype -- block with holes on either side
```

```
         */
        if (nblocks != 0) {
                rc = MPI_Type_contiguous(nblocks, oldtype, &block1);
                if (rc != MPI_SUCCESS)
                        return (rc);
                MPI_Type_commit(&block1);
                rc = MPIO_Type_set_bounds_absolute(darg * rank * extent,
                                dsize * extent, block1, newtype);
                MPI_Type_free(&block1);
        } else {
                rc = MPIO_Type_bounds(0, dsize * extent, newtype);
        }
        return (rc);
}
```

C.2.3 HPF CYCLIC Sample Implementation

```
/*
 * MPIO_Type_hpf_cyclic - generate filetypes for an
 *                        HPF CYCLIC(darg) distribution
 *
 *            oldtype ARRAY(dsize)
 *      !HPF$ PROCESSORS COMM(size)
 *      !HPF$ DISTRIBUTE ARRAY(CYCLIC(darg)) ONTO COMM
 */
MPIO_Type_hpf_cyclic(
        MPI_Comm comm,                  /* Communicator group           */
        int dsize,                      /* Distributee size             */
        int darg,                       /* Distribution argument        */
        MPI_Datatype oldtype,           /* Old datatype                 */
        MPI_Datatype *newtype)          /* New datatype                 */
{
        int size, rank;
        int myblocks, nblocks, residue;
        MPI_Datatype block1, block2, block3;
        MPI_Aint extent, ub;
        int rc;

        if (darg < 0)
                return (MPIO_ERR_ARG);
        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);
        MPI_Type_extent(oldtype, &extent);
        ub = extent * dsize;
        /*
         * Compute and check distribution argument
         */
        if (darg == MPIO_HPF_DFLT_DARG)
```

```
1                darg = 1;
2        /*
3         * nblocks = dsize / darg rounded up, this is the number of blocks
4         *      needed to distributed among the processeors
5         */
6        nblocks = (dsize + (darg - 1)) / darg;
7        if (nblocks <= size) {
8                /*
9                 * go around once - some processors make have less or none
10                */
11               myblocks = dsize - darg * rank;
12               if (myblocks > darg)
13                       myblocks = darg;
14               if (myblocks <= 0) {
15                       rc = MPIO_Type_bounds(0, ub, newtype);
16               }
17               else {
18                       rc = MPI_Type_contiguous(myblocks, oldtype, &block1);
19                       if (rc != MPI_SUCCESS)
20                               return (rc);
21                       MPI_Type_commit(&block1);
22                       rc = MPIO_Type_set_bounds_absolute(
23                                                darg * rank * extent,
24                                                ub, block1, newtype);
25                       MPI_Type_free(&block1);
26               }
27               return (rc);
28       }
29       /*
30        * block1 is the basic units to be distributed
31        */
32       if ((rc = MPI_Type_contiguous(darg, oldtype, &block1)) != MPI_SUCCESS)
33               return (rc);
34       MPI_Type_commit(&block1);
35       rc = MPIO_Type_set_bounds(rank, size, block1, &block2);
36       MPI_Type_free(&block1);
37       if (rc != MPI_SUCCESS) {
38               return (rc);
39       }
40       myblocks = dsize / darg / size;
41       MPI_Type_commit(&block2);
42       rc = MPI_Type_contiguous(myblocks, block2, &block3);
43       MPI_Type_free(&block2);
44       if (rc != MPI_SUCCESS)
45               return (rc);
46       residue = dsize - myblocks * darg * size;
47       if (residue >= rank * darg)
48               residue = residue - rank * darg;
```

```
        if (residue > darg)
                residue = darg;
        if (residue == 0) {
                MPI_Type_commit(&block3);
                rc = MPIO_Type_set_bounds_absolute(0, ub, block3, newtype);
                MPI_Type_free(&block3);
                return (rc);
        } else {
                int block[3];
                MPI_Datatype type[3];
                MPI_Aint disp[3];

                block[0] = 1;
                type[0] = block3;
                disp[0] = 0;
                MPI_Type_extent(block3, &disp[1]);
                disp[1] += rank * darg * extent;
                type[1] = oldtype;
                block[1] = residue;
                block[2] = 1;
                type[2] = MPI_UB;
                disp[2] = ub;
                MPI_Type_commit(&block3);
                rc = MPI_Type_struct(3, block, disp, type, newtype);
                MPI_Type_free(&block3);
                return (rc);
        }
}
```

C.3 Example: Row block distribution of A[100, 100]

Consider an application (such as one generating visualization data) which saves a timestep of a 2-dimensional array A[100][100] in standard C-order to a file. Say we have 10 nodes. The array A is distributed among the nodes in a simple row block decomposition.

The array is distributed to nodes as (each number represents a 10x10 block):

```
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
```

in other words:

```
    Node 0:
        A[0, 0], A[0, 1], A[0, 2], ..., A[0, 99],
        A[1, 0], A[1, 1], A[1, 2], ..., A[1, 99],
        A[2, 0], A[2, 1], A[2, 2], ..., A[2, 99],
                          ...
        A[9, 0], A[9, 1], A[9, 2], ..., A[9, 99]

    Node 1:
        A[10, 0], A[10, 1], A[10, 2], ..., A[10, 99],
        A[11, 0], A[11, 1], A[11, 2], ..., A[11, 99],
        A[12, 0], A[12, 1], A[12, 2], ..., A[12, 99],
                          ...
        A[19, 0], A[19, 1], A[19, 2], ..., A[19, 99]

    ...

    Node 9:
        A[90, 0], A[90, 1], A[90, 2], ..., A[90, 99],
        A[91, 0], A[91, 1], A[91, 2], ..., A[91, 99],
        A[92, 0], A[92, 1], A[92, 2], ..., A[92, 99],
                          ...
        A[99, 0], A[99, 1], A[99, 2], ..., A[99, 99]
```

C.3.1 Intel CFS Implementation

The CFS code might look like:

```
    double myA[10][100];
    int fd;

    fd = open(filename, O_WRONLY, 0644);
    setiomode(fd, M_RECORD);
    /* Compute new value of myA */
    write(fd, &myA[0][0], sizeof(myA));
```

C.3.2 MPI-IO Implementation

The equivalent MPI-IO code would be:

```
    double myA[10][100];
    MPIO_Offset disp = MPIO_OFFSET_ZERO;
    MPIO_Offset offset;
    MPI_Datatype myA_t, myA_ftype;
    MPIO_File fh;
    MPI_Status status;
    char filename[255];

    MPI_Type_contiguous(1000, MPI_DOUBLE, &myA_t);
    MPIO_Type_scatter_gather(MPI_COMM_WORLD, myA_t, &myA_ftype);
```

```
MPI_Type_commit(&myA_t);
MPI_Type_commit(&myA_ftype);
MPIO_Open(MPI_COMM_WORLD, filename, MPIO_WRONLY,
          disp, MPI_DOUBLE, myA_ftype, MPIO_OFFSET_RELATIVE, 0, &fh);
/* Compute new value of myA */
offset = disp;
MPIO_Write_all(fh, offset, &myA[0][0], myA_t, 1, &status);
```

C.4 Example: Column block distribution of A[100, 100]

Again, consider an application which saves a timestep of a 2-dimensional array A[100][100] in standard C-order to a file, run on 10 nodes. For this example, the array A is distributed among the nodes in a simple column block decomposition.

The array is distributed to nodes as (each number represents a 10x10 block):

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

C.4.1 Intel CFS Implementation

The CFS code might look like:

```
double myA[100][10];
int fd;
int i;

fd = open(filename, O_WRONLY, 0644);
setiomode(fd, M_RECORD);
/* Compute new value of myA */
for (i = 0; i < 100; i++)
    write(fd, &myA[i][0], sizeof(myA)/100);
```

C.4.2 MPI-IO Implementation

The equivalent MPI-IO code would be:

```
double myA[100][10];
MPIO_Offset disp = MPIO_OFFSET_ZERO;
MPIO_Offset offset;
MPI_Datatype subrow_t, row_t, myA_ftype;
MPIO_File fh;
```

```
    MPI_Status status;
    char filename[255];

    MPI_Type_contiguous(10, MPI_DOUBLE, &subrow_t);
    MPIO_Type_scatter_gather(MPI_COMM_WORLD, subrow_t, &row_t);
    MPI_Type_contiguous(100, row_t, &myA_ftype);
    MPI_Type_commit(&myA_ftype);
    MPI_Type_free(&subrow_t);
    MPI_Type_free(&row_t);
    MPIO_Open(MPI_COMM_WORLD, filename, MPIO_WRONLY,
              disp, MPI_DOUBLE, myA_ftype, MPIO_OFFSET_RELATIVE, 0, &fh);
    /* Compute new value of A */
    offset = disp;
    MPIO_Write_all(fh, offset, &myA[0][0], MPI_DOUBLE, 1000, &status);
```

C.5 Example: Transposing a 2-D Matrix in a Row-Cyclic Distribution

The following code implements the example depicted in Figure 3 in Section 2. A 2-D matrix is to be transposed in a row-cyclic distribution onto nproc processes. For the purpose of this example, we assume that matrix A is a matrix of size (n, m) and that each element of the matrix is a double precision real number (etype is MPI_DOUBLE).

```
int             nproc;         /* number of processes in MPI_COMM_WORLD */
int             rank;          /* rank of the process within MPI_COMM_WORLD */
void           *Aloc;          /* local matrix assigned to the process */
int             n;             /* size (in etype) of global matrix A */
int             m;
int             nrows;         /* number of rows assigned to the process */
int             sizeofAloc;    /* size (in bytes) of local matrix Aloc */
char            mat_A[10] = "file_A"; /* name of the file containing matrix A */
                                      /* the file is assumed to exist         */
MPIO_Offset disp = MPIO_OFFSET_ZERO;  /* file_A is supposed to have no header */
MPIO_Mode       amode;    /* access mode */
MPI_Datatype    etype;    /* elementary datatype */
MPI_Datatype    filetype; /* filetype associated with an HPF row_cyclic */
                          /* distribution                                */
int             moffset;  /* relative/absolute offset flag */
MPIO_Hints     *hints;    /* hints */
MPIO_File       fh;       /* file handle */
MPIO_Offset     offset;   /* offset into file_A */
MPI_Datatype    buftype;  /* buffer type used to read in the transposed local */
                          /* matrix                                           */
int             bufcount; /* number of buftype items to read at once */
MPI_Status      status;   /* status information of read operation */
/* temporary variables */
int             extent;
MPI_Datatype    column_t;
```

```
MPI_Comm_size (MPI_COMM_WORLD, &nproc);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
/* Determine number of rows assigned to the process */
nrows = (rank < m % nproc) ? (m / nproc + 1) : (m / nproc);
amode = MPIO_RDONLY;
/* Aloc is a matrix of MPI_DOUBLE items */
etype = MPI_DOUBLE;
MPI_Type_extent (etype, &extent);
MPIO_Type_hpf_cyclic (MPI_COMM_WORLD, n * m, n, etype, &filetype);
MPI_Type_commit (&filetype);
moffset = MPIO_OFFSET_RELATIVE;  /* relative offsets will be used */
hints = NULL;   /* no hints are provided */
/* Open file containing matrix A */
MPIO_Open (MPI_COMM_WORLD, mat_A, amode, disp, etype,
           filetype, moffset, hints, &fh);
/* Define buffer type that transposes each row of the matrix read in and */
/* concatenates the resulting columns                                    */
MPI_Type_vector (n, 1, nrows, etype, &column_t);
MPI_Type_hvector (nrows, 1, extent, column_t, &buftype);
MPI_Type_commit (&buftype);
MPI_Type_free (&column_t);
/* Allocate memory for local matrix Aloc */
MPI_Type_extent (buftype, &sizeofAloc);
Aloc = (void *) malloc (sizeofAloc);
/* Read in local matrix Aloc */
offset = disp;
bufcount = 1;
MPIO_Read (fh, offset, Aloc, buftype, bufcount, &status);
```

D Justifying Design Decisions

This section contains a haphazard collection of arguments for other designs and against the one we chose, with explanations of why they were rejected.

Argument: Filetype should be defined in the read/write operation, not in the open call. This is similar to having the sendtype and recvtype in MPI scatter/gather calls.
Answer: This is more cumbersome, especially since it is expected that filetypes will not be changed often (if at all). Also, the filetype may be much larger than the buftype (or much smaller), which makes it harder to understand how they are aligned. The MPI case does not have this problem because the sizes must match.

Paragraph modified

Argument: Are absolute offsets really required, and if so, what should be their exact semantics, in particular when filetypes with non monotonic displacements are used.
Answer: We are still debating these issues. Therefore, absolute offsets are currently part of the proposal, with a pretty hazy semantics. No guarantee is made at this point that absolute offsets will remain in the final definition of MPI-IO, and if they will, their semantics may

differ from the current one.

Argument: MPI-like functions with informative names should be used, e.g. Read_Broadcast, Write_Single, Read_Scatter, Write_Gather.
Answer: This causes confusion if the filetype is used as well, because the same effect can be achieved in very different ways. The reason to prefer the filetype approach over the specific-functions approach is that it is more flexible and provides a mechanism to express additional new access patterns.