NASA-CR-205139

MPI-IO: A Parallel File I/O Interface for MPI
Version 0.3

[NAS Technical Report NAS-95-002 January 1995]

mpi-io@nas.nasa.gov

Peter Corbett, Dror Feitelson, 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

Typeset on January 30, 1995

[1]Computer Sciences Corporation, NASA Ames Research Center, under Contract NAS2-12961

Contents

1 Introduction

Thanks to MPI [9], 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. Further, the closest thing to a standard, the UNIX file interface, is ill-suited to parallel computing.

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.

The above approach can be materialized in different ways. The current proposal represents the result of extensive discussions (and arguments), but is by no means finished. Many changes can be expected as additional participants join the effort to define an interface for portable I/O.

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. It specifies what the interface currently supports and states what would need to be added to the current proposal to make the interface more complete and robust. The next seven sections contain the interface definition itself. Section 3 presents definitions and conventions. Section 4 contains functions for file control, most notably open. Section 5 includes functions for independent I/O, both blocking and nonblocking. Section 6 includes functions for collective I/O, both blocking and nonblocking. Section 7 presents functions to support system-maintained file pointers, and shared file pointers. Section 8 presents constructors that can be used to define useful filetypes (the role of filetypes is explained in Section 2 below). Section 9 presents how the error handling mechanism of MPI is supported by the MPI-IO interface. All this is followed by a set of appendices, which contain information about issues that have not been totally resolved yet, and about design considerations. The reader can find there the motivation behind some of our design choices. More information on this would definitely be welcome and will be included in a further release of this document. The first appendix contains a description of MPI-IO's "hints" structure which is used when opening a file. Appendix B is a discussion of various issues in the support for file pointers. Appendix C explains what we mean in talking about atomic access. Appendix D provides detailed examples of filetype constructors, and Appendix E contains a collection of arguments for and against various design decisions.

1.1 Background

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 [8]. 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 [10, 7, 6, 1]:

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

The common denominator of those modes that actually attempt to capture useful I/O patterns and help the programmer is that they define how data is partitioned among the processes. Some systems do this explicitly without using modes, and allow the programmer to define the partitioning directly. Examples include Vesta [3] and the nCUBE system software [4]. Recent studies show that various simple partitioning schemes do indeed account for most of observed parallel I/O patterns [8]. MPI-IO also has the goal of supporting such common patterns.

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, 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 I/O interface was designed with the following goals:

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

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

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

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

5. The design favors performance over functionality.

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

1. Support for message passing environments other than MPI.

2. Compatibility with the UNIX file interface.

3. Support for transaction processing.

4. Support for FORTRAN record oriented I/O.

1.3 History

This work is an outgrowth of the original proposal from IBM [11], but it is significantly different. The main difference is the use of file types to express partitioning in an MPI-like style, rather than using special Vesta functions. In addition, file types are now used to express various access patterns such as scatter/gather, rather than having explicit functions for the different patterns.

Version 0.2 is the one 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 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 2.5). 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 Section 7).

2 Overview of MPI-IO

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 tasks can get access to the file in subsequent I/O operations. Accesses to a file can be independent (no coordination between tasks takes place) or collective (each task 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 tasks. 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 task can only access the file data that matches items in the filetype. It cannot access file data that falls under holes (see Figure 1).

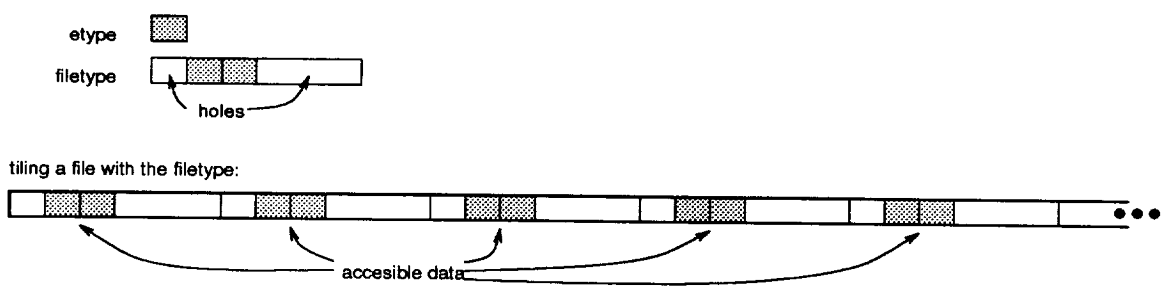

Figure 1: Tiling a file using a filetype

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

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 tasks (see Figure 3). The matrix is to be distributed among the parallel tasks in a row cyclic manner. Each task 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 D.

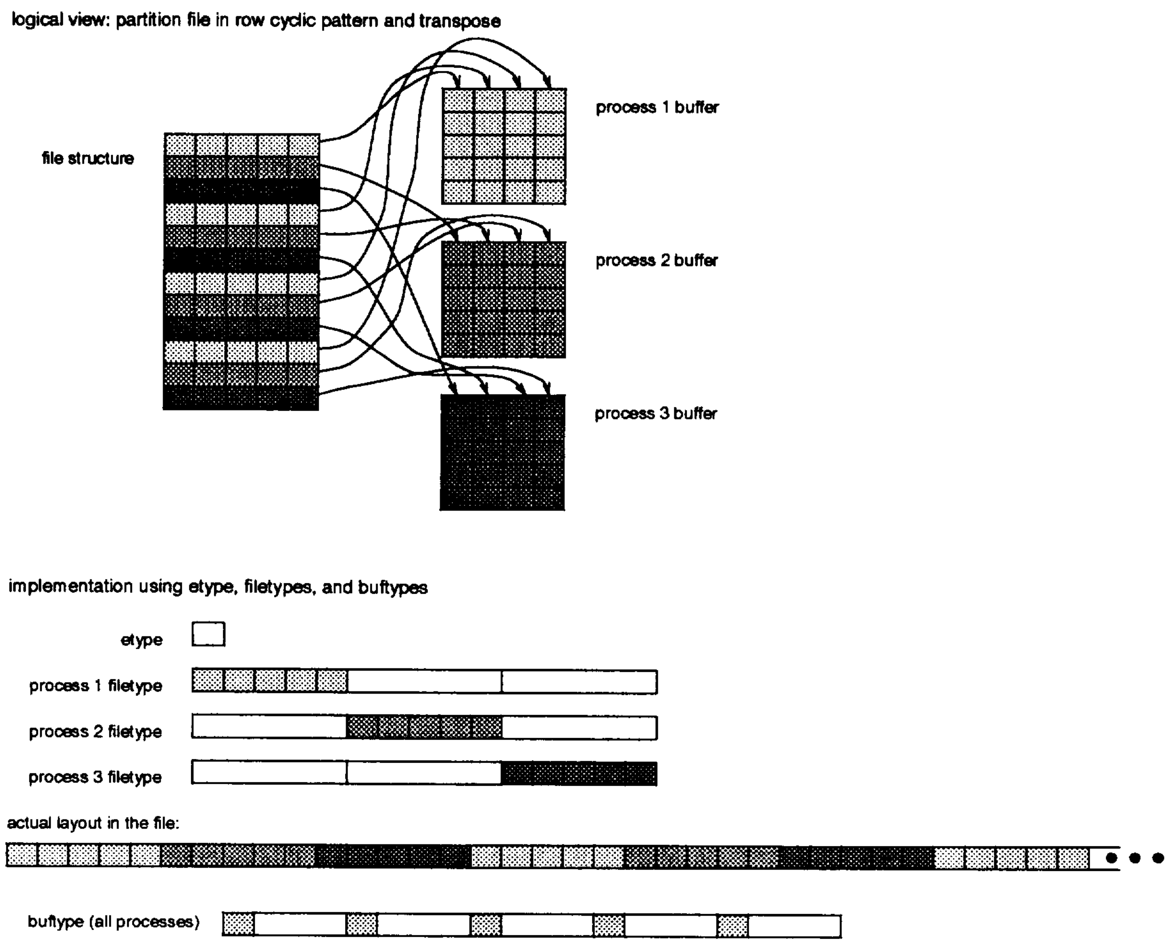

Figure 3: Transposing and partitioning a 2-D matrix

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.

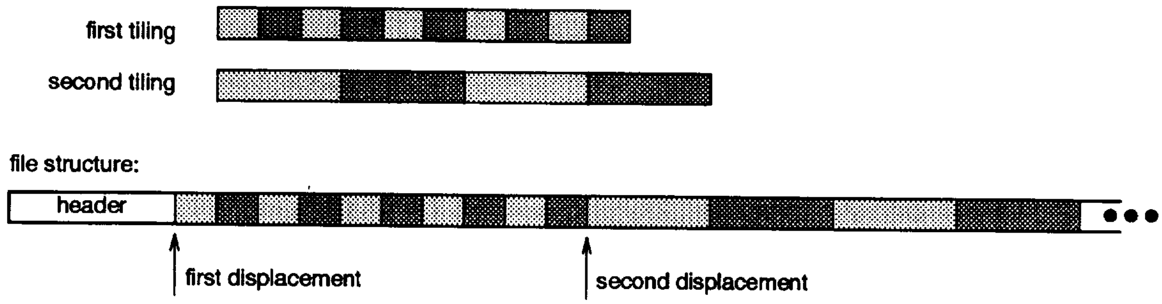

Figure 4: Displacements

2.2 MPI-IO Data Access Functions

As noted above, we have elected not to define specific calls for the different access patterns. However, there are different calls for the different synchronization behaviors which are desired, and for different ways to specify the offset in the file. The following table summarizes these calls:

offset	*synchronization*	*independent*	*collective*
explicit offset	*blocking* (synchronous)	MPIO_Read MPIO_Write	MPIO_Read_all MPIO_Write_all
	nonblocking (asynchronous)	MPIO_Iread MPIO_Iwrite	MPIO_Iread_all MPIO_Iwrite_all
independent file pointer	*blocking* (synchronous)	MPIO_Read_next MPIO_Write_next	MPIO_Read_next_all MPIO_Write_next_all
	nonblocking (asynchronous)	MPIO_Iread_next MPIO_Iwrite_next	MPIO_Iread_next_all MPIO_Iwrite_next_all
shared file pointer	*blocking* (synchronous)	MPIO_Read_shared MPIO_Write_shared	– –
	nonblocking (asynchronous)	MPIO_Iread_shared MPIO_Iwrite_shared	– –

The independent calls with explicit offsets are described in Section 5, and the collective ones in Section 6. Independent calls do not imply any coordination among the calling processes. On the other hand, collective calls imply that all tasks belonging to the communicator associated with the opened file must participate. However, as in MPI, no synchronization pattern between those tasks is enforced by the MPI-IO definition. Any required synchronization may depend upon a specific implementation. Collective calls can be used to achieve certain semantics, as in a scatter-gather operation, but they are also useful to advise the system of a set of independent accesses that may be optimized if combined.

When several independent data accesses involve multiple overlapping data blocks, it may be desirable to guarantee the atomicity of each access, as provided by Unix (see Appendix C). In this case, it is possible to enable the MPIO_CAUTIOUS access mode for the file. Note that the 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 tasks of the same MPI application.

2.3 Offsets and File Pointers

Part of the problem with the Unix interface when used by multiple processes is that there is no atomicity of seek and read/write operations. MPI-IO rectifies this problem by including an explicit offset argument in the first set of read and write calls. This offset can be *absolute*, which means that it ignores the file partitioning pattern, or *relative*, which means that only the data accessible by this process is counted, excluding the holes of the filetype associated with the task (see Figure 5). In both cases, offsets are expressed as an integral number of elementary datatype items. As absolute offsets can point to anywhere in the file, they can also point to an item that is unaccessible 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. Absolute offsets may be easier to understand if accesses to arbitrary random locations are combined with partitioning the file among processes using filetypes. If such random accesses are not used, relative offsets are better. If the file is not partitioned, absolute and relative offsets are the same.

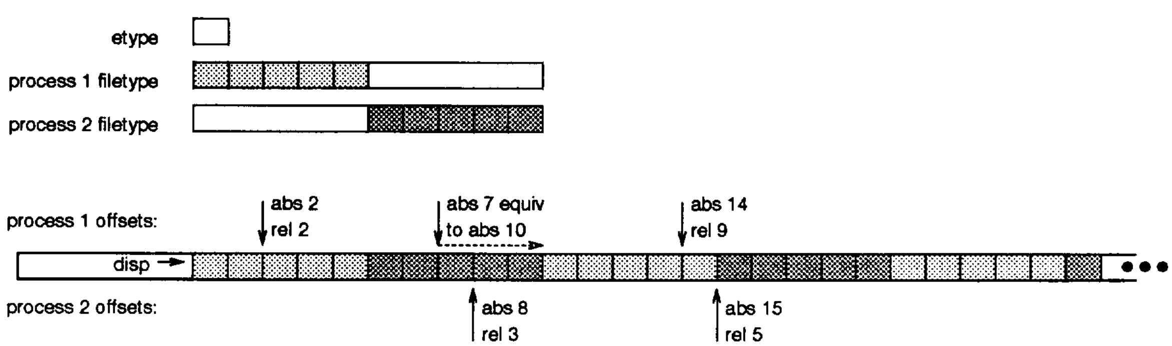

Figure 5: Absolute and relative offsets

It should be noted that the offset is a required argument in the explicit offset functions. Processes must maintain their offsets into all files by themselves. A separate set of functions, described in Section 7, provide the service of doing the next access where the previous one left off. This is especially convenient for sequential access patterns (or partitioned-sequential patterns), which are very common in scientific computing [8]. Likewise, shared file pointers are also supported. This allows for the creation of a log file with no prior coordination among the processes, and also supports self-scheduled reading of data. However, there are no collective functions using shared offsets. This issue is discussed in Appendix B.

2.4 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.

2.5 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.

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.

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.

3 Interface Definitions and Conventions

3.1 Independent vs. Collective

An *independent* I/O request is a request which is executed individually by any of the processes within a comunicator 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, collective calls can (but are not required to) return as soon as their 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. Collective calls may require that all processes, involved in the collective operation, pass the same value for an argument. We will indicate it with the "[SAME]" annotation in the function definition, like in the following example:

MPIO_CLOSE(fh)

IN	fh	[SAME] Valid file handle (handle)

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 Blocking vs. Nonblocking

One can improve performance by overlapping computation and I/O. 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 nonblocking call may return before the data has been read/written out of the user's buffer. 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 out of the user's buffer. With suitable hardware, the transfer of data out/in the user's buffer may proceed concurrently with computation.

Advice to users. The fact that a blocking or nonblocking I/O request completed does not indicate that data has been stored on permanent storage. It only indicates that it is safe to access the user's buffer. (*End of advice to users.*)

3.3 Etype, Filetype, Buftype, and Offset Relation

The **etype** argument is the elementary datatype associated with a file. etype is used to express the **filetype**, **buftype** and **offset** arguments. The **filetype** and **buftype** datatypes must be directly constructed (i.e. derived datatype) from **etype**, or their type signatures must be a multiple of the **etype** signature. 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**.

3.4 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.

3.5 Return Code and Status

All the MPI-IO Fortran interfaces return a success or a failure code in the IERROR return argument. All MPI-IO C functions also return a success or a 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.

3.6 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).

4 File Control

4.1 Opening a File (Collective)

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	Relative/Absolute offset flag (integer)
IN	hints	Hints to the file system (array of integer)
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(MPIO_HINTS_SIZE), FH, IERROR
    INTEGER*8 DISP
```

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 - deleting on close

These can be combined using the bitwise OR operator. 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.

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file), where the file is to be opened. This is used to skip headers, and when the file includes a sequence of data segments that are to be accessed in different patterns.

The **etype** argument specifies the elementary datatype used to construct the **filetype**, and also the **buftype** type used in the read/write. 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 this 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 must be interpreted. **moffset** can have 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 gives user's file access patterns, and file system specifics (see Appendix A).

Files are opened by default in the MPIO_RECKLESS read/write atomic semantics mode. Each process may pass different values for the **disp**, **filetype**, **moffset** and **hints** arguments. However, the **filename**, **comm**, **amode** and **etype** argument values must be the same.

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

Access permissions are not specified when opening a file. If the file is being created, operating system defaults apply (eg Unix command umask).

> *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
```

MPIO_Close closes the file associated with **fh**. If the file was opened with MPIO_DELETE, the file is deleted. If there are other processes currently accessing the file, the status of the file and the behavior of future accesses are implementation dependent. After closing, the content of the file handle **fh** is destroyed. All future use of fh will cause an error.

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. (*End of advice to implementors.*)

4.3 File Control (Independent/Collective)

MPIO_FILE_CONTROL(fh, size, cmd, arg)

IN	fh	[SAME] Valid file handle (handle)
IN	size	[SAME] Numbers of command passed (integer)
IN	cmd	[SAME] Command arguments (array of integer)
IN/OUT	arg	Arguments or return values to the command requests

```
int MPIO_File_control(MPIO_File fh, int size, int *cmd, void *arg)
```

```
MPIO_FILE_CONTROL(FH, SIZE, CMD, ARG, IERROR)
    INTEGER FH, SIZE, CMD(*), IERROR, ARG(*)
```

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, with the restriction that it is not allowed to mix collective and independent commands. The commands available are:

- (independent)
  - MPIO_GETCOMM: Get the communicator associated with the file.
  - MPIO_GETNAME: Get the filename.
  - MPIO_GETAMODE: Get the file access mode associated with the file.
  - MPIO_GETDISP: Get the displacement.
  - MPIO_GETETYPE: Get the elementary datatype.
  - MPIO_GETFILETYPE: Get the filetype.
  - MPIO_GETHINTS: Get the hints associated with the file.
  - MPIO_GETATOM: Get the current read/write atomic semantics enforced mode.
  - MPIO_GETINDIVIDUALPOINTER: 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_GETSHAREDPOINTER: 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).
- (Collective)
  - MPIO_SETAMODE: Set the file access mode using the arg argument. arg must be a valid amode.
  - MPIO_SETDISP: Set new displacement.
  - MPIO_SETETYPE: Set the elementary datatype associated with the file.

- MPIO_SETFILETYPE: Set the filetype associated with the file.
- MPIO_SETATOM: Set the read/write atomic semantics enforced mode. arg can be either MPIO_RECKLESS or MPIO_CAUTIOUS.
- MPIO_GETSIZE: Get the current file size.

For collective commands, all processes in the communicator group that opened the file must issue the same command. In the cases of MPIO_SETAMODE and MPIO_SETATOM, the arguments must also be identical.

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 Independent I/O

5.1 MPIO_Read

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**.

5.2 MPIO_Write

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**.

5.3 MPIO_Iread

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.

5.4 MPIO_Iwrite

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.

6 Collective I/O

6.1 MPIO_Read_all

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. MPIO_Read_all stores the number of buftype elements actually read in status.

6.2 MPIO_Write_all

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. MPIO_Write_all stores the number of **buftype** elements actually written in **status**.

6.3 MPIO_Iread_all

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 processes to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes.

6.4 MPIO_Iwrite_all

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 processes to complete). The user should not access any part of the supplied buffer after a nonblocking write is called, until the write completes.

7 File pointers

7.1 Introduction

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 which is shared by all the processes in the communicator group. The others are *individual* file pointers local to each process in the communicator group, and can be updated independently.

All the I/O functions described above in Sections 5 and 6 require an explicit offset to be passed as an argument. Those functions do not use the system-maintained file pointers, nor do those functions update the system maintained file pointers. In this section we describe an alternative set of functions that use the system maintained file pointers. Actually there are two sets: one using the individual pointers, and the other using the shared pointer. The main difference from the previous function is that an offset argument is not required. In order to allow the offset to be set, seek functions are provided.

The main semantics issue with system-maintained file pointers is how they are updated by I/O operations. In general, each I/O operation leaves the pointer pointing to the next data item after the last one that was accessed. This principle applies to both types of offsets (MPIO_OFFSET_ABSOLUTE and MPIO_OFFSET_RELATIVE), to both types of pointers (individual and shared), and to all types of I/O operations (read and write, blocking and nonblocking). The details, however, may be slightly different.

When absolute offsets are used, the 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 (see the discussion in Section 2). If it is not, then the next I/O operation will automatically advance the 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:

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

In all cases (blocking or nonblocking operation, individual or shared file pointer, absolute or relative offset), the file pointer is updated when the operation is initiated (see Appendix B.2 for the reasons behind this design choice), in other words before the access is performed.

> *Advice to users.* This update reflects the amount of data that is requested by the access, not the amount that will be actually accessed. Typically, these two values will be the same, but they can differ in certain cases (e.g. a read request that reaches EOF). This differs from the usual Unix semantics, and the user is encouraged to check for EOF occurrence in order to account for the fact that the file pointer may point beyond the end of file. In rare cases (e.g. a nonblocking read reaching EOF followed by a write), this can cause problems (e.g. creation of holes in the file). (*End of advice to users.*)

7.2 Shared File Pointer I/O Functions

These functions use and update the global current file position maintained by the system. The individual file pointers are not used nor updated. Note that only independent functions are currently defined. It is debatable whether or not collective functions are required as well. This issue is addressed in Appendix B.3.

> *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 should use the same **filetype** when opening the file. (*End of advice to users.*)

7.2.1 MPIO_Read_shared (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.

7.2.2 MPIO_Write_shared (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.

7.2.3 MPIO_Iread_shared (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.

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.

7.2.4 MPIO_Iwrite_shared (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.

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.

7.3 Individual File Pointer Blocking I/O Functions

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 blocking functions described in Sections 5 and 6, 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 requested. For collective I/O, each individual file pointer is updated independently.

7.3.1 MPIO_Read_next (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 read in status. The file pointer is updated by the amount of data requested.

7.3.2 MPIO_Write_next(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 written in status. The file pointer is updated by the amount of data requested.

7.3.3 MPIO_Read_next_all (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**. MPIO_Read_next_all returns the number of **buftype** elements read in **status**. The file pointer of each process is updated by the amount of data requested by that process.

7.3.4 MPIO_Write_next_all (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. **MPIO_Write_next_all** returns the number of **buftype** elements written in **status**. The file pointer of each process is updated by the amount of data requested by that process.

7.4 Individual File Pointer Nonblocking I/O Functions

Like the functions described in Section 7.3, 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 nonblocking functions described in Sections 5 and 6, with the **offset** argument set to the current value of the system-maintained individual file pointer. This file pointer is updated when the I/O is initiated and reflects the amount of data requested. For collective I/O, each individual file pointer is updated independently.

7.4.1 MPIO_Iread_next (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 pointer is updated by the amount of data requested.

7.4.2 MPIO_Iwrite_next (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 pointer is updated by the amount of data requested.

7.4.3 MPIO_Iread_next_all (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 processes to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes. The pointer is updated by the amount of data requested.

7.4.4 MPIO_Iwrite_next_all (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 processes 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 pointer is updated by the amount of data requested.

7.5 File Pointer Manipulation Functions

7.5.1 MPIO_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**.

7.5.2 MPIO_Seek_shared (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., broadcast read, 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 datatypes 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 datatype created **newtype** consists of zero or more copies of the base type **oldtype**, possibly separated by holes. The extent of the new datatype is a nonnegative integer multiple of the extent of the base type. All datatype constructors return a success or failure code.

8.2 Broadcast-Read and Write-Reduce Constructors

8.2.1 MPIO_Type_read_bcast

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 broadcast read operation by using the types generated by this call.

8.2.2 MPIO_Type_write_reduce

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 type 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 types generated by this call.

> *Advice to implementors.* The choice of which process actually performs the write operation can either be always the same process (eg process with rank 0 in the process group) or arbitrary (eg 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

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 type 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 is of type oldtype.

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

8.3.2 MPIO_Type_scatterv_gatherv

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 types returned should be passed to a collective read or write operation, giving identical offsets.

8.4 HPF Filetype Constructors

The HPF [5] 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

HPF distribution of an N-dimensional array:

MPIO_TYPE_HPF(comm, ndim, dsize, distrib, darg, 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 dimension of distributee (array of nonnegative offset)
IN	distrib	[SAME] HPF distribution of 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	oldtype	[SAME] old datatype (handle)
OUT	newtype	new datatype (handle)

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

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

MPIO_Type_hpf generates a filetype corresponding to the HPF distribution of an ndim-dimensional array of oldtype specified by the arguments.

For example, in order to generate the types corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
      MPI_COMM_SIZE(comm, size, ierror)
!HPF$ PROCESSORS PROCESSES(size)

!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

The corresponding MPI-IO type would be created by the following code:

```
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;
MPIO_Type_hpf(comm, ndim, dsize, distrib, darg, oldtype, &newtype);
```

8.4.2 MPIO_Type_hpf_block

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 offset)
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, MPIO_Offset 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 a filetype corresponding to the HPF BLOCK distribution of a one-dimensional dsize element array of oldtype.

This call is a shorthand for:

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

8.4.3 MPIO_Type_hpf_cyclic

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 offset)
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, MPIO_Offset 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 a filetype corresponding to the HPF CYCLIC distribution of a one-dimensional **dsize** element array of **oldtype**.

This call is a shorthand for:

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

8.4.4 MPIO_Type_hpf_2d

HPF distribution of a *two*-dimensional array:

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

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

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

```
MPIO_TYPE_HPF_2D(COMM, DSIZE1, DISTRIB1, DARG1, DSIZE2, DISTRIB2, DARG2,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, DSIZE1, DISTRIB1, DARG1, DISIZE2, DISTRIB2, DARG2,
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_hpf_2d generates a filetype corresponding to the HPF (**distrib1(darg1)**, **distrib2(darg2)**) distribution of a two-dimensional (**dsize1,dsize2**) element array of **oldtype**.

This call is a shorthand for:

```
dsize[0]=dsize1;
distrib[0]=distrib1;
darg[0]=darg1;
dsize[1]=dsize2
distrib[1]=distrib2;
darg[1]=darg2;
MPIO_Type_hpf(comm, 2, dsize,  distrib, darg, 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_set 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.

Bibliography

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

[2] 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.

[3] 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.

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

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

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

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

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

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

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

[11] 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.

```
MPIO_TYPE_HPF_2D(COMM, DSIZE1, DISTRIB1, DARG1, DSIZE2, DISTRIB2, DARG2,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COMM, DSIZE1, DISTRIB1, DARG1, DISIZE2, DISTRIB2, DARG2,
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

MPIO_Type_hpf_2d generates a filetype corresponding to the HPF (distrib1(darg1), distrib2(darg2)) distribution of a two-dimensional (dsize1,dsize2) element array of oldtype.

This call is a shorthand for:

```
dsize[0]=dsize1;
distrib[0]=distrib1;
darg[0]=darg1;
dsize[1]=dsize2
distrib[1]=distrib2;
darg[1]=darg2;
MPIO_Type_hpf(comm, 2, dsize,  distrib, darg, 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.

interpret the hints in slightly different ways. For example, the following table outlines possible interpretations for an MPI-IO implementation based on the Vesta parallel file system:

hint	*interpretation*
striping-unit	BSU size
striping factor	number of cells
IO-node-list	base node
partitioning-pattern	Vesta partitioning parameters

B System Support for File Pointers

B.1 Interface Style

The basic MPI-IO design calls for offsets to be passed explicitly in each read/write operation. This avoids issues of uncertain semantics when multiple processes are performing I/O operations in parallel, especially mixed seek and read/write operations. It also reflects current practices, where programmers often keep track of offsets themselves, rather than using system-maintained offsets.

There are a number of ways to add support for system-maintained file pointers to the interface:

1. Add a **whence** argument to each read/write call, to specify whether the given offset is to be used directly or whether it is relative to the current system-maintained offset. To just use the system-maintained offset, the **offset** argument should be set to 0.
2. Define certain special values for the **offset** argument. For example, −1 could mean that the system maintained individual offset should be used, and −2 that the system-maintained shared offset be used.
3. Define a separate set of functions with no **offset** argument.

We have chosen the third approach for the following reasons. First, it saves overhead because the system need not update offsets unless they are actually used. Second, it makes the interface look more like a conventional Unix interface for users who use system-maintained offsets. This is preferable over an interface with extra arguments that are not used.

B.2 File Pointer Update

In normal Unix I/O operations, the system-maintained file pointer is 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 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.

This way of updating file pointers may lead to some problems in rare circumstances, like in the following scenario:

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

If the first read reaches EOF, since the file pointer is incremented by the amount of data requested, the write will occur beyond EOF, leaving a hole in the file. However, such a problem only occurs if reads and writes are mixed with no checking, which is an uncommon pattern.

B.3 Collective Operations with Shared File Pointers

The current definition of the MPI-IO interface only includes independent read and write operations using shared file pointers. Collective calls are not included, because they seem to be unnecessary. The main use of a shared pointer is to partition data among processes on the fly, with no prior coordination. Collective operations imply coordinated access by all the processes. These two approaches seem at odds with each other.

C 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 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 [2], 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

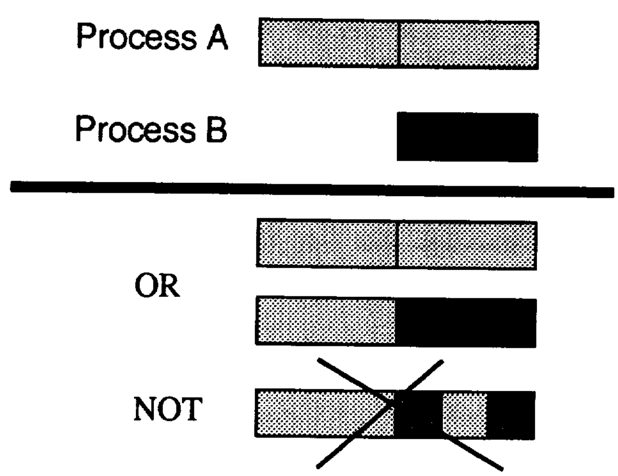

Figure 6: Unix Atomic Semantics

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.

D Filetype Constructors: Sample Implementations and Examples

D.1 Support Routines

```
/*
 * MPIO_Type_set_bounds - surround a type with holes (increasing the extent)
 */
int MPIO_Type_set_bounds(
        int displacement,          /* Displacement from the lower bound */
        int ub,                    /* Set the upper bound               */
        MPI_Datatype oldtype,      /* Old datatype                      */
        MPI_Datatype *newtype)     /* New datatype                      */
{
        int blocklength[3];
        MPI_Datatype type[3];
        MPI_Aint disp[3];

        blocklength[0] = 1;
        disp[0] =  0;
        type[0] = MPI_LB;

        blocklength[1] = 1;
        disp[1] = displacement;
        type[1] = oldtype;

        blocklength[2] = 1;
        disp[2] = ub;
        type[2] = MPI_UB;
```

```
        /*
         * newtype =
         *     { (LB, 0), (oldtype, displacement), (UB, ub) }
         */
        return MPI_Type_struct(3, blocklength, disp, type, newtype);
}
```

D.2 Sample Filetype Constructor Implementations

D.2.1 MPIO_Type_scatter_gather Sample Implementation

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

        MPI_Type_extent(oldtype,&extent);

        MPI_Comm_size(comm, &size)
        MPI_Comm_rank(comm, &rank)

        return MPIO_Type_set_bounds(rank*extent, size*extent, oldtype, newtype);
}
```

D.2.2 HPF BLOCK Sample Implementation

```
/*
 * MPIO_Type_hpf_block - generate datatypes for a HPF BLOCK(darg) distribution
 */
int 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 extent;
        int beforeblocksize;
        int myblocksize;
        int nblocks;
```

```
        int is_partial_block;
        MPI_Datatype block1;
        int rc;

        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);

        MPI_Type_extent(oldtype, &extent);

        /*
         * Compute and check distribution argument
         */
        if (darg == MPIO_HPF_DFLT_ARG)    /* [HPF, p. 27, L37] */
                darg = (dsize + size - 1) / size;
        if (darg * size < dsize)          /* [HPF, p. 27, L33] */
                return MPIO_ERROR_ARG;

        /*
         * Compute the sum of the sizes of the blocks of all processes
         * ranked before me, and the size of my block
         */
        nblocks = dsize / darg;
        is_partial_block = (dsize % darg != 0);
        if (nblocks < rank) {
                beforeblocksize = dsize;
                myblocksize = 0;
        } else if (nblocks == rank) {
                beforeblocksize = nblocks * darg;
                myblocksize = dsize % darg;
        } else {
                beforeblocksize = rank * darg;
                myblocksize = darg;
        }

        /*
         * Create filetype --- block with holes on either side
         */
        if ((rc = MPI_Type_contiguous(myblocksize, oldtype, &block1))
            == MPI_SUCCESS) {
           rc = MPIO_Type_set_bounds(beforeblocksize*extent, dsize*extent,
                                     block1, newtype));
           MPI_Type_free(&block1);
        }
        return rc;
}
```

D.2.3 HPF CYCLIC Sample Implementation

```
/*
 * MPIO_Type_hpf_cyclic - generate types for HPF CYCLIC(darg) distribution;
 *                        we assume here that dsize >= darg * size; in other
 *                        words, we do not support degenerated cases where
 *                        some processes may not have any data assigned to them
 */
int 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 extent;
        MPI_Datatype block1, block2, block3;
        int rc;

        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);

        MPI_Type_extent(oldtype, &extent);

        /*
         * Compute and check distribution argument
         */
        if (darg == MPIO_HPF_DFLT_ARG)   /* [HPF, p. 27, L42] */
                darg = 1;

        /*
         * Take care of full blocks (contains darg*size oldtype items)
         */
        nelem = dsize / (darg * size);
        if ((rc = MPI_Type_contiguous(darg, oldtype, &block1) != MPI_SUCCESS)
           return rc;
        if ((rc = MPIO_Type_set_bounds(darg*rank*extent, darg*size*extent,
                                       block1, &block2)) != MPI_SUCCESS) {
           MPI_Type_free(&block1);
           return rc;
        }
        rc = MPI_Type_contiguous(nelem, block2, &block3);
        MPI_Type_free(&block1);
        MPI_Type_free(&block2);
        if (rc != MPI_SUCCESS)
           return rc;
```

```
    /*
     * Take care of residual block
     */
    residue = dsize - nelem * (darg * size);
    if (residue > rank * darg) {
       int last_block;
       int b[2];
       MPI_Aint d[2];
       MPI_Datatype t[2];
       MPI_Datatype block4, block5;

       last_block = residue - rank * darg;
       if (last_block > darg)
          last_block = darg;
       if ((rc = MPI_Type_contiguous(last_block, oldtype, &block4))
           != MPI_SUCCESS) {
          MPI_Type_free(&block3);
          return rc;
       }
       if ((rc = MPIO_Type_set_bounds(darg*rank*extent, residue*extent,
                                      block4, &block5)) != MPI_SUCCESS) {
          MPI_Type_free(&block3);
          MPI_Type_free(&block4);
          return rc;
       }
       b[0] = 1;
       b[1] = 1;
       d[0] = 0;
       d[1] = nelem * darg * size * extent;
       t[0] = block3;
       t[1] = block5;
       rc = MPI_Type_struct(2, b, d, t, newtype);
       MPI_Type_free(&block4);
       MPI_Type_free(&block5);
    } else {
       rc = MPIO_Type_set_bounds(0, dsize*extent, block3, newtype);
    }
    MPI_Type_free(&block3);
    return rc;
}
```

D.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]
```

D.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));
```

D.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);
```

D.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
```

D.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);
```

D.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);
```

D.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 m processes. For the purpose of this example, we assume that matrix A is a square matrix of size n and that each element of the matrix is a double precision real number (etype is a MPI_DOUBLE).

```
int             m;              /* number of tasks in MPI_COMM_WORLD */
int             rank;           /* rank of the task within MPI_COMM_WORLD */

void           *Aloc;           /* local matrix assigned to the task */
int             n;              /* size (in etype) of global matrix A */
int             nrow;           /* number of rows assigned to the task */
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             sizeofetype;
MPI_Datatype    column_t;

MPI_Comm_size (MPI_COMM_WORLD, m);
MPI_Comm_rank (MPI_COMM_WORLD, rank);

/* Determine number of rows assigned to the task */
nrow = n / m;
if (rank < n % m) nrow++;

amode = MPIO_RDONLY;

/* Aloc is a matrix of MPI_DOUBLE items */
etype = MPI_DOUBLE;
MPI_Type_extent (etype, &sizeofetype);

MPIO_Type_hpf_cyclic (MPI_COMM_WORLD, n * n, n, etype, &filetype);
MPI_Type_commit (&filetype);

moffset = MPIO_OFFSET_RELATIVE;  /* relative offsets will be used */

hints = NULL;   /* hints are not fully implemented yet */

/* 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, nrow, etype, &column_t);
MPI_Type_hvector (nrow, 1, sizeofetype, 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);
```

E 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.

Argument: Absolute offsets are confusing, no good, and nobody uses them.
Answer: OK, we'll have relative offsets too.

Argument: Relative offsets are confusing, no good, and nobody uses them.
Answer: OK, we'll have absolute offsets too.

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.