Appeared in *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999, pp. 23–32. © 1999 ACM.

# On Implementing MPI-IO Portably and with High Performance

Rajeev Thakur William Gropp Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, gropp, lusk} @mcs.anl.gov

## Abstract

We discuss the issues involved in implementing MPI-IO portably on multiple machines and file systems and also achieving high performance. One way to implement MPI-IO portably is to implement it on top of the basic Unix I/O functions (`open`, `lseek`, `read`, `write`, and `close`), which are themselves portable. We argue that this approach has limitations in both functionality and performance. We instead advocate an implementation approach that combines a large portion of portable code and a small portion of code that is optimized separately for different machines and file systems. We have used such an approach to develop a high-performance, portable MPI-IO implementation, called ROMIO.

In addition to basic I/O functionality, we consider the issues of supporting other MPI-IO features, such as 64-bit file sizes, noncontiguous accesses, collective I/O, asynchronous I/O, consistency and atomicity semantics, user-supplied hints, shared file pointers, portable data representation, and file preallocation. We describe how we implemented each of these features on various machines and file systems. The machines we consider are the HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, SGI Origin2000, and networks of workstations; and the file systems we consider are HP HFS, IBM PIOFS, Intel PFS, NEC SFS, SGI XFS, NFS, and any general Unix file system (UFS).

We also present our thoughts on how a file system can be designed to better support MPI-IO. We provide a list of features desired from a file system that would help in implementing MPI-IO correctly and with high performance.

## 1 Introduction

Portable parallel programming has long been hampered by the lack of a standard, portable application programming interface (API) for parallel I/O. Most parallel file systems have a Unix-like API with variations that are nonportable. Furthermore, the Unix API is not an appropriate API for parallel I/O: it lacks some of the features necessary to express access patterns common in parallel programs, such as noncontiguous accesses and collective I/O, resulting in poor performance [35]. To overcome these limitations, the MPI Forum defined a new API for parallel I/O (commonly referred to as MPI-IO) as part of the MPI-2 standard [19]. MPI-IO is a comprehensive API with many features intended specifically for I/O parallelism, portability, and high performance. Implementations of MPI-IO, both portable and machine-specific, are already available [7, 13, 23, 24, 34].

In this paper, we discuss the issues involved in implementing MPI-IO portably on multiple machines and file systems and also achieving high performance. We argue that if an implementation uses just the basic Unix I/O functions in order to achieve portability, it will have limitations in both functionality and performance. We describe an alternative approach, called ADIO, that achieves portability and performance by combining a large portion of portable code with a small portion of code that is optimized separately for different machines and file systems. We have used this approach in our portable MPI-IO implementation, ROMIO.[1]

In addition to implementing basic I/O functionality (open, close, read, write, seek), we consider the issues of supporting other MPI-IO features, such as 64-bit file sizes, noncontiguous accesses, collective I/O, asynchronous I/O, consistency and atomicity semantics, user-supplied hints, shared file pointers, portable data representation, and file preallocation. We describe how we implemented each of these features on various machines and file systems. The machines we consider are the HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, SGI Origin2000, and networks of workstations; and the file systems we consider are HP HFS, IBM PIOFS, Intel PFS, NEC SFS, SGI XFS, the Network File System (NFS), and any general Unix file system (UFS).

We also describe how a file system can be designed to better support MPI-IO. We provide a list of features desired from a file system that would help in implementing MPI-IO correctly and with high performance.

## 2 Achieving Portability and Performance

The basic Unix I/O functions (`open`, `lseek`, `read`, `write`, and `close`) [29] are supported without variation on all machines with a Unix-like operating system. One way to implement MPI-IO portably, therefore, is to implement MPI-IO functions on top of these basic Unix I/O functions. Since the Unix I/O functions are portable, such an MPI-IO implementation will be portable to many machines and file systems. This approach, however, has limitations in both functionality and performance, as explained below:

1. The basic Unix I/O functions are not sufficient to implement all of MPI-IO on all file systems for the following reasons:
   - The basic Unix I/O functions are blocking functions. Many file systems provide a different set of (nonportable)

[1] URL: `http://www.mcs.anl.gov/romio`

functions for nonblocking I/O.[2]

- On many file systems, the basic Unix I/O functions work only on files of size less than 2 Gbytes. Different functions must be used for larger files, and these functions are also nonportable. (We note that an MPI-IO implementation is not *required* to support large file sizes, but most high-quality implementations will.)
- Some file systems allow the user to control file-striping attributes with special, nonportable functions (e.g., IBM PIOFS and Intel PFS).
- Some file systems support additional features such as file preallocation (e.g., SGI XFS, Intel PFS, HP HFS) and a choice of atomic and nonatomic file-access modes (e.g., IBM PIOFS and Intel PFS). The corresponding functions are also nonportable.

Since all these features are available at the MPI-IO level, an MPI-IO implementation cannot support them if it uses only the basic Unix I/O functions.

2. Although the basic Unix I/O functions are supported on all file systems, they are often not the recommended functions (for performance) on all file systems. For example,

   - On the Intel Paragon, the recommended functions are `cread` and `cwrite`.
   - On SGI IRIX 6.5, the recommended functions are `pread64` and `pwrite64`; on IRIX 6.4 and earlier, they are called `pread` and `pwrite`.
   - On HP machines running the SPPUX operating system (and not HPUX), the recommended functions are `pread64` and `pwrite64`.

3. When using the Network File System (NFS), it is not sufficient to call just the Unix `read`/`write` functions. Since NFS performs noncoherent client-side caching by default, file consistency is not guaranteed if multiple processes write to a common file [28]. Client-side caching must be disabled by locking the portion of the file being accessed, by using `fcntl`. A lock and unlock are therefore needed across the `read`/`write` call.

4. Many research file systems provide their own APIs [9, 3, 11, 15, 20]. Implementing MPI-IO on top of Unix I/O functions will not be portable to these file systems.

An alternative is to implement MPI-IO on top of the POSIX I/O interface [12] instead of the basic Unix I/O functions. The POSIX interface is an international standard with greater functionality than basic Unix I/O. For example, POSIX supports asynchronous I/O and list-directed I/O. This approach, however, also has limitations. Although POSIX is a standard, it is not yet widely implemented. One, therefore, cannot assume that POSIX I/O functions will be available on all file systems. Furthermore, many vendors do not follow the POSIX standard strictly. They implement only parts of it, and even the implemented portion may not conform strictly to the standard (particularly in the case of asynchronous I/O). Some vendors provide a separate set of functions for 64-bit file sizes. POSIX also does not support some features that MPI-IO supports, for example, file preallocation and varying file-striping attributes. Nonstandard functions must be used on file systems that support these features. In all, implementing MPI-IO on top of POSIX I/O is not sufficient either.

We believe that the only way to implement MPI-IO portably with complete functionality and high performance is to have a mechanism that can utilize the special features and functions of each file system. We describe such an architecture, called ADIO, which we use in our MPI-IO implementation, ROMIO [34].

### 2.1 Abstract-Device Interface for I/O

ADIO [31], an abstract-device interface for I/O, is a mechanism specifically designed for implementing parallel-I/O APIs portably on multiple file systems. We developed ADIO before MPI-IO became a standard, as a means to implement and experiment with various parallel-I/O APIs that existed at the time.

ADIO consists of a small set of basic functions for *parallel* I/O. Any parallel-I/O API can be implemented portably on top of ADIO, and ADIO itself is implemented separately on each different file system. ADIO thus separates the machine-dependent and machine-independent aspects involved in implementing an API. The ADIO implementation on a particular file system is optimized for that file system. We used ADIO to implement Intel's PFS API and subsets of IBM's PIOFS API and the original MPI-IO proposal [36] on multiple file systems. By following such an approach, we achieved portability with very low overhead [31].

Now that MPI-IO has emerged as the standard, we use ADIO as a mechanism for implementing MPI-IO portably (see Figure 1). This MPI-IO implementation is called ROMIO [34]. ROMIO runs on the following machines: IBM SP; Intel Paragon; Cray T3E; HP Exemplar; SGI Origin2000; NEC SX-4; other symmetric multiprocessors from HP, SGI, Sun, DEC, and IBM; and networks of workstations (Sun, SGI, HP, IBM, DEC, Linux, and FreeBSD). Supported file systems are IBM PIOFS, Intel PFS, HP HFS, SGI XFS, NEC SFS, NFS, and any Unix file system (UFS). All functions defined in the MPI-2 I/O chapter except support for file interoperability, I/O error handling, and I/O error classes have been implemented in ROMIO. (The missing functions will be implemented in a future release.) ROMIO is designed to be used with *any* MPI-1 implementation—both portable and vendor-specific implementations. It works with, and is included as part of, three MPI implementations: MPICH, HP MPI, and SGI MPI.

Another application of ADIO is for implementing remote I/O. An MPI-IO implementation can enable a program running on one machine to access files from remote machines by providing an ADIO implementation that accesses data from an ADIO server running at a remote site. Such an implementation is described in [8] and also illustrated in Figure 1.

A similar abstract-device interface is used in MPICH [10] for implementing MPI portably.

## 3 Implementing MPI-IO

We describe how we implemented each feature of MPI-IO on various machines and file systems. The many variations among machines clearly demonstrate the need for an ADIO-like approach to implementing MPI-IO portably, where the variations are accounted for in the ADIO implementation.

### 3.1 Basic File Access

We first consider the basic file-access operations: open, close, read, write, and seek. We consider reads and writes in which data is contiguous in both memory and file; noncontiguous accesses are considered in Section 3.2.

[2] It is possible, however, to implement nonblocking I/O by spawning a thread that calls a blocking I/O function.

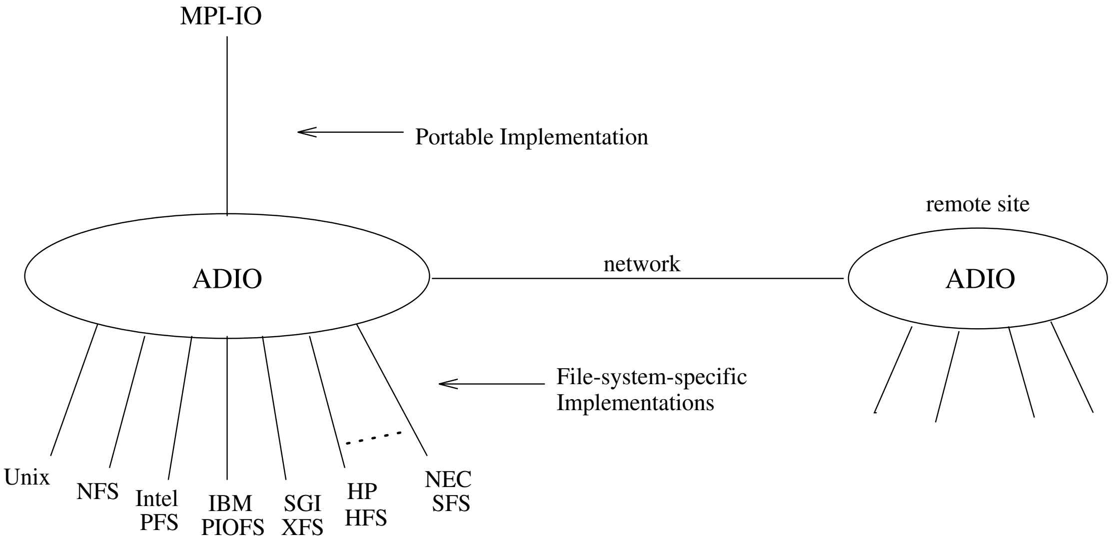


Figure 1: ROMIO architecture: MPI-IO is implemented portably on top of an abstract-device interface called ADIO, and ADIO is optimized separately for different file systems.

#### 3.1.1 Open

`MPI_File_open` is a *collective* function. One of its arguments is an MPI communicator [18] that specifies the group of processes that will call this open function and any other collective MPI-IO function that the user may choose to use thereafter on the open file. Most file systems, other than Intel PFS, support only the regular Unix `open` and do not have collective open functions. On these file systems, ROMIO just calls the `open` function on each process. Intel PFS supports two open functions: regular Unix `open` and `gopen`. The `gopen` function is a "global open," recommended to be used when *all* processes in the application open a common file. It cannot be used when a subset of processes open the file; the function will hang if all processes do not call it. MPI-IO also supports a few additional file-access modes that are not defined in Unix or POSIX.

#### 3.1.2 Close

The `close` function on most file systems is identical to `close` in Unix or POSIX. `MPI_File_close` can be implemented in a straightforward manner on top of Unix `close`. If the file was opened with the mode `MPI_MODE_DELETE_ON_CLOSE`, the implementation must delete the file. Most file systems support the Unix function `unlink` for deleting a file.

#### 3.1.3 Large Files

Most file systems distinguish between files of size less than 2 Gbytes and greater than or equal to 2 Gbytes. The reason is that file offsets and file sizes are usually represented by 4-byte integers in the regular I/O functions. The largest number that can be represented by a 4-byte signed integer is (2 Gbytes – 1). With the regular file-system functions, it is therefore not possible to access data from locations beyond 2 Gbytes. To overcome this problem, many file systems provide separate functions that use 8-byte integers to represent file offsets.

In MPI-IO, file offsets are of type `MPI_Offset`, which is a data type defined by the MPI-IO implementation. The implementation is free to define it to be of any size; the MPI standard does not mandate that the implementation support large files. In ROMIO, however, on those file systems that support large files (such as IBM PIOFS, HP HFS, NEC SFS, and SGI XFS), all files are treated as large files; that is, ROMIO defines `MPI_Offset` as an 8-byte integer and uses the corresponding file-system functions for large files (even though the file may be smaller than 2 Gbytes). On file systems that do not support large files, ROMIO also does not support large files and defines `MPI_Offset` as a 4-byte integer.

#### 3.1.4 Seek

MPI-IO has two kinds of file pointers, individual and shared, and, correspondingly, two seek functions to move these file pointers. Most file systems (other than Intel PFS), however, support only individual file pointers. In Section 3.9 we describe how an MPI-IO implementation can implement shared file pointers on top of individual file pointers.

Most file systems support the Unix `lseek` function. On some file systems we need to use a different function for large files: `lseek64` on SGI XFS, HP HFS, and NEC SFS; `llseek` on IBM PIOFS.

#### 3.1.5 Contiguous Reads and Writes

Contiguous reads and writes in MPI-IO can be mapped directly onto the reads and writes of the underlying file system. The read/write functions recommended for highest performance vary considerably among machines, however. ROMIO uses the following functions:

- `_cread/_cwrite` on Intel PFS.
- `pread64/pwrite64` on HP HFS if the operating system is SPPUX and `read/write` if it is HPUX.
- `pread64/pwrite64` on SGI XFS if the operating system is IRIX 6.5. On IRIX 6.4 and earlier, the same functions are called `pread/pwrite`.
- `read/write` elsewhere.

The functions `pread64/pwrite64` take the file offset as an argument; therefore, a separate `lseek64` is not required.

### 3.2 Noncontiguous Accesses

MPI-IO allows users to access noncontiguous data from a file into noncontiguous memory locations with a single I/O function call. The user can specify noncontiguous locations in the file by creating a *file view* with MPI's derived datatypes [19]. Noncontiguous locations in memory can be specified by using a derived datatype in the read/write call.

The ability of users to specify noncontiguous accesses in a single function call is very important, because noncontiguous accesses are very common in parallel applications [1, 4, 21, 26, 27, 32]. Most file systems, however, do not provide functions for noncontiguous I/O. The Unix functions `readv`/`writev` are widely supported, but they allow noncontiguity only in memory and not in the file. Noncontiguous memory accesses are not as commonly needed in parallel applications as noncontiguous file accesses. Furthermore, most file systems impose a limit of at most sixteen noncontiguous memory locations in a single `readv`/`writev` call.

Some file systems support the POSIX list-directed I/O function `lio_listio`, which allows users to submit multiple I/O requests at a time. This function also has limitations because of the way it is defined. The POSIX standard [12] allows a mixture of read and write requests in the list and says that each of the requests will be submitted as a separate nonblocking (asynchronous) I/O request. Therefore, POSIX implementations cannot optimize I/O for the entire list of requests. Furthermore, since the `lio_listio` interface is not collective, implementations also cannot perform collective I/O.

In the absence of proper support from the file system for noncontiguous I/O, one way to implement a noncontiguous MPI-IO request is to access each contiguous portion of the request separately by using the regular contiguous read/write functions of the file system. Such an implementation, however, results in a large number of small requests to the file system, and performance degrades drastically [33]. ROMIO instead performs an optimization, called *data sieving*, to access noncontiguous data with high performance. The basic idea in data sieving is to make large I/O requests to the file system and extract, in memory, the data that is really needed. Details of this optimization can be found in [33].

### 3.3 Collective I/O

MPI-IO provides collective-I/O functions, which must be called by all processes that together opened the file.[3] This property enables the MPI-IO implementation (or file system) to analyze and merge the requests of different processes. In many cases, the merged request may be large and contiguous, although the individual requests of each process are noncontiguous. The merged request can therefore be serviced efficiently. Such optimization is broadly referred to as *collective I/O*. Collective I/O has been shown to be a very important optimization in parallel I/O and can improve performance significantly [5, 14, 25, 30, 33].

Since none of the file systems on which ROMIO is implemented perform collective I/O, ROMIO performs two-phase collective I/O on top of the file system. In the communication phase, interprocess communication is used to rearrange data into large chunks. In the I/O phase, processes perform parallel I/O in large chunks and therefore obtain high I/O performance. ROMIO has a very general implementation of two-phase I/O: it supports any noncontiguous access pattern as described by MPI datatypes, and the user can specify by means of hints the amount of temporary buffer space ROMIO can use for collective I/O and the number of processes that should actually perform I/O in the I/O phase of the two-phase operation. Details of ROMIO's collective-I/O implementation can be found in [33].

Figure 2 shows the performance of an astrophysics application template, DIST3D, when I/O is performed in three ways: using Unix-style independent I/O, data sieving, and collective I/O. This application accesses a three-dimensional distributed array of size $512 \times 512 \times 512$ from a file. On some machines data sieving performed only slightly better than Unix-style independent I/O; on others it performed considerably better. Collective I/O always performed the best and resulted in I/O bandwidths ranging from 51 Mbytes/sec to 563 Mbytes/sec, depending on the machine. For detailed performance results, see [33].

### 3.4 Split Collective I/O

MPI-IO provides a restricted form of nonblocking collective I/O called *split collective I/O*. The user can call a "begin" function to start the collective-I/O operation and an "end" function to complete the operation. The implementation is free to implement the collective-I/O operation either entirely during the begin function or entirely during the end function or in the "background," between the begin and end functions. The MPI standard allows the user to have at most one active split collective operation on a particular file handle at any time. In other words, the user cannot issue two "begin" functions on the same file handle without calling an "end" function to complete the first begin.

The most natural way to implement split collective I/O in a nonblocking fashion is to spawn a thread that performs the entire collective-I/O operation in the background. The results in [6], however, indicate that, on most machines, this approach performs much worse than if collective I/O were done entirely in the main thread during the begin function. The performance is much better if only the I/O portion of collective I/O is done in a separate thread and the rest is done in the main thread. The split-collective-I/O functions in ROMIO, at present, perform the entire collective-I/O operation in the main thread during the begin function. We plan to implement true nonblocking collective I/O in ROMIO by incorporating the results of [6].

### 3.5 Nonblocking (Asynchronous) I/O

Many file systems support nonblocking I/O. One way to implement MPI-IO's nonblocking I/O functions is to use the nonblocking functions of the file system. Intel PFS supports nonstandard functions called `iread` and `iwrite`. Other vendors (SGI, IBM, DEC, Sun) support POSIX asynchronous I/O (aio) functions, but, in many cases, they do not follow the POSIX definition strictly. IBM supports nonblocking I/O on Unix and NFS file systems, but not on PIOFS. HP supports nonblocking I/O only on HPUX version 11.0 and higher, but not on SPPUX or earlier versions of HPUX. Nonblocking I/O functions are not yet available in Linux, FreeBSD, or the NEC SX-4.

Another way to implement nonblocking I/O is by explicitly using threads that call blocking I/O functions. This approach, however, requires *good* thread support on the machine and a thread-safe MPI implementation, neither of which is common on parallel machines as yet.

ROMIO implements nonblocking I/O by using the nonblocking I/O functions of the file system where available. On machines and file systems that do not support nonblocking I/O, ROMIO just calls the corresponding blocking I/O functions.

[3] An MPI communicator is used in the open call to specify the participating processes. The communicator could represent any subset (or all) of the processes of the application.

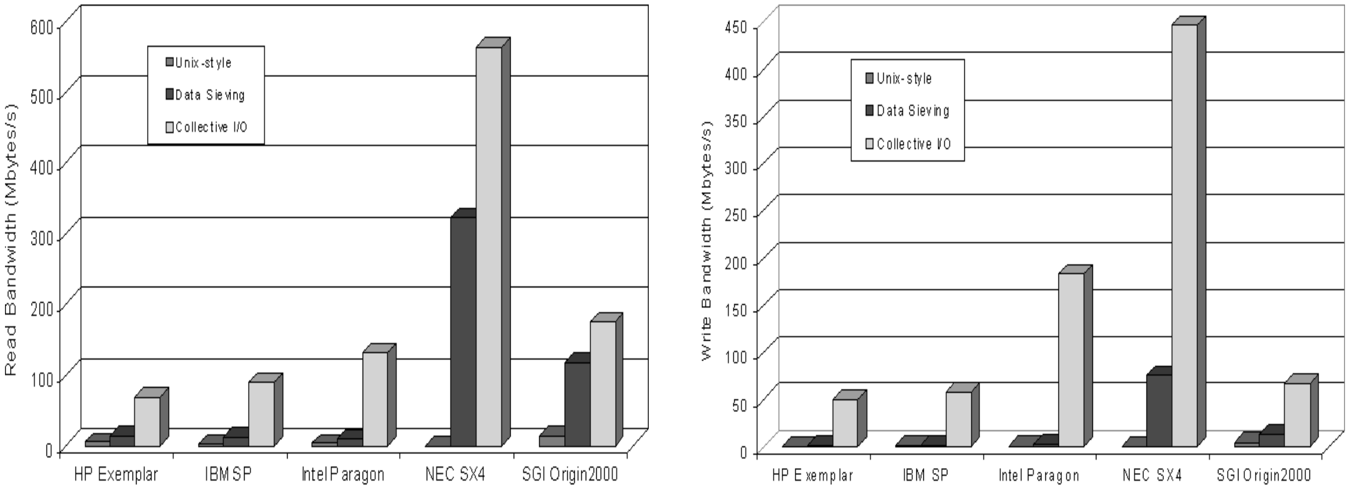


Figure 2: Performance of DIST3D using Unix-style independent I/O, data sieving, and collective I/O. The figure on the left shows read bandwidth, and the figure on the right shows write bandwidth.

### 3.6 Consistency Semantics

MPI-IO's consistency semantics (Section 9.6 of [19]) define the results users can expect with concurrent file accesses from multiple processes. MPI-IO's consistency semantics are actually weaker than the consistency semantics in Unix [29] or POSIX [12]. In Unix and POSIX, after a write function returns, the data is guaranteed to be visible to every other process in the system. MPI-IO guarantees that a write from one process is immediately visible only to processes that belong to the communicator with which the file was opened and only if atomic mode was enabled before the write. For any other case, the data is visible to another process only after both the writer and reader call `MPI_File_sync`.

MPI-IO's consistency semantics are therefore automatically guaranteed on file systems that support Unix consistency semantics.[4] NFS, by default, does not [28]. To obtain Unix consistency semantics on NFS, ROMIO uses byte-range locking (`fcntl`) across the reads and writes in order to turn off the noncoherent client-side caching that NFS otherwise performs. Turning off client-side caching reduces performance considerably but is, nonetheless, necessary for correctness. We believe that the other file systems on which ROMIO is implemented do support Unix consistency semantics correctly.

### 3.7 Atomicity Semantics

Atomicity semantics define the results when multiple processes issue concurrent requests to overlapping regions in the file, and one or more of those requests are write requests. MPI-IO supports two atomicity modes. The default mode is nonatomic, in which the results of such concurrent requests are undefined. The user can change the mode to atomic, in which case the overlapping region will contain data from any one process only.

The atomic mode is the only mode supported in Unix and POSIX. On file systems that support Unix atomicity semantics correctly, the atomic mode is therefore implemented by default, at least for contiguous MPI-IO requests. If the MPI-IO request is noncontiguous in the file, and the implementation writes it by making more than one write function call, then atomicity is not guaranteed for the entire noncontiguous MPI-IO request. To guarantee atomicity in such cases (when the user has set atomic mode and the request is noncontiguous), ROMIO locks the range of bytes being accessed in the file and then performs the necessary I/O.

On file systems that support only the atomic mode, the nonatomic mode is also implemented by default, since it has weaker semantics than the atomic mode. Some file systems, such as IBM PIOFS and Intel PFS, support both modes, because the nonatomic mode can result in higher performance. On PIOFS, the default mode is nonatomic (called `NORMAL`); the user can change the access mode to atomic (called `CAUTIOUS`) with the function `piofsioctl`. On PFS, the default mode is atomic (called `M_UNIX`); nonatomic mode (called `M_ASYNC`) can be selected by using either the function `gopen` or `setiomode`. Both `gopen` and `setiomode`, however, are "global" functions: *all* processes in the application must call them. In MPI-IO, users can create a communicator containing a subset of all processes and open the file with this communicator. In such cases, the MPI-IO implementation cannot use the nonatomic mode on PFS.

### 3.8 Hints

MPI-IO provides a mechanism for the user to pass hints to the implementation. Hints, such as access-pattern information, can help the implementation optimize file access [2, 22]. Hints do not change the semantics of the MPI-IO interface; an implementation may choose to ignore all hints, and the program would still be functionally correct. MPI-IO has some predefined hints for specifying file-striping parameters, access patterns, and so on. An implementation is free to define additional hints.

ROMIO supports some predefined hints and some additional hints. The predefined hints supported are the file-striping parameters (number of disks and striping unit) and the buffer size and number of processes to use for collective I/O. Additional hints supported by ROMIO are the disk number from which to begin striping the file, buffer sizes for data sieving, and, on Intel PFS only, a hint to turn on server buffering. ROMIO uses the file-striping hints only on the two file systems that allow the striping parameters to be varied, namely, Intel PFS and IBM PIOFS; they are ignored on other file systems. On PFS, ROMIO uses the `fcntl` function to vary file-striping parameters. On PIOFS, the function is `piofsioctl`.

[4] If the MPI-IO implementation performs its own buffering on top of the file system, it must take additional steps to ensure that MPI-IO consistency semantics are maintained.

MPI-IO also allows users to query the current value of a hint. With this feature, users can, for example, determine the default file-striping parameters or the buffer sizes ROMIO uses for data sieving and collective I/O.

### 3.9 Shared File Pointers

Most file systems, other than Intel PFS, do not support shared file pointers. On such file systems, the MPI-IO implementation must implement shared file pointers itself. Doing so requires some mechanism for maintaining the value of the shared file pointer for each file and for processes to access and atomically update this value. One method is to store the value of the shared file pointer in a file and have processes update the value atomically by using file locks. Another method is to have one process or thread own the shared file pointer and have other processes access the value from this process or thread. This method, however, requires that the MPI implementation support dynamic processes, or one-sided communication, or multiple threads, and none of these features are commonly supported by MPI implementations as yet. A third method, applicable only if all processes have access to shared memory, is to maintain the shared file pointer in shared memory and use some mechanism for atomically updating the value of the shared file pointer, such as semaphores.

ROMIO uses the first method because it works in all environments. ROMIO stores the value of the shared file pointer in a file in the same directory as the data file being accessed. When a process needs to access data using the shared file pointer, it locks the file containing the shared-file-pointer value, reads the value, increments it by the amount of data to be read or written, writes the new value back, releases the lock, and then performs the read or write of actual data. The shared-file-pointer file is created when the shared file pointer is first used in the program and is deleted when the user closes the data file.

### 3.10 Portable Data Representation

MPI-IO supports multiple data-storage representations: `native`, `internal`, `external32`, and also user-defined representations. `native` means that data is stored in the file as it is in memory; no data conversion is performed. `internal` is an implementation-defined data representation that may provide some (implementation-defined) degree of file portability. `external32` is a specific, portable data representation defined in MPI-IO. A file written in `external32` format on one machine is guaranteed to be readable on any machine with any MPI-IO implementation. MPI-IO also provides a mechanism for users to define a new data representation by providing data-conversion functions, which MPI-IO uses to convert data from file format to memory format and vice versa.

The `native` representation is implemented by default, and an implementation can use `external32` as its `internal` representation. One way to implement `external32` is to convert each datatype explicitly from/to the `external32` representation, which may require byte swapping, truncation, or padding, depending on the machine. Another way to implement `external32` is via the data-conversion functions: the implementation can provide the data-conversion functions to translate from `external32` to `native` representation (and vice versa) and use these functions to implement `external32`.

ROMIO currently supports only the `native` representation. We plan to implement `external32` via the data-conversion functions because this approach is modular, easily extensible to new platforms, and so that users can use the functions as a template to define other data representations.

### 3.11 File Preallocation

Only a few file systems provide a function to preallocate disk space for a file. Intel PFS has a function called `lsize`, on SGI XFS one can preallocate space via `fcntl`, and HP HFS has functions `prealloc` and `prealloc64`. On other file systems that do not support file preallocation, the MPI-IO implementation must allocate space by actually writing data to the file (which is expensive).

### 3.12 Miscellaneous Issues

Here we consider some miscellaneous issues in implementing MPI-IO.

#### 3.12.1 Library versus Client-Server Implementation

An MPI-IO implementer is faced with the choice of implementing it as a library or as a client-server implementation. We believe that if the underlying file system supports high-performance access from multiple processes to a common file, a library approach is sufficient. Any further optimizations needed, such as data sieving and collective I/O, can be implemented within the library. This is the case on parallel machines such as the IBM SP, Intel Paragon, SGI Origin2000, HP Exemplar, and NEC SX-4.

A client-server approach is needed if no common file system exists for all processes to access, for example, when the processes run on clusters of independent machines, each with their own local file system. In such a case, the MPI-IO implementation would need to have servers that implement a virtual shared file system on top of the individual file systems on these machines. Another example is when MPI-IO is used to access files from remote machines, as described in [8].

#### 3.12.2 Operating with Multiple MPI-1 Implementations

MPI-IO can be implemented in a way that it can operate with any MPI-1 implementation that also has a few functions from the MPI-2 external-interfaces chapter. These functions allow the MPI-IO implementation to access some of the internal data structures of the MPI implementation. The datatype-decoding functions, `MPI_Type_get_envelope` and `MPI_Type_get_contents`, are the ones most critically needed. Without them, the MPI-IO implementation cannot decipher what an MPI derived datatype represents. A complete MPI-IO implementation would also need a few more functions from the MPI-2 external-interfaces chapter, namely, functions for filling in the status object, generalized requests, adding new error codes and classes, attribute caching on datatypes, and duplicating datatypes.

The “info” functions from the MPI-2 miscellaneous chapter are needed for passing hints to MPI-IO, and the subarray and distributed-array datatype constructors are very useful to users of MPI-IO. These functions, however, can be implemented portably on top of any MPI-1 implementation.

ROMIO, at present, requires only that the MPI implementation support the two datatype-decoding functions from MPI-2; the other external-interface functions mentioned above are not used. The MPI-2 info functions and the subarray and distributed-array datatype constructors are implemented in ROMIO; however, if the MPI implementation also supports these functions, the ones provided by the MPI implementation are used instead.

ROMIO works with, and is included as part of, three MPI implementations, MPICH, HP MPI, and SGI MPI, all of which support the datatype-accessor functions that ROMIO needs. (ROMIO may also work with the LAM MPI implementation, as LAM also supports these functions now, but we have not yet tested ROMIO with LAM.)

#### 3.12.3 Automatic Detection of File-System Type

ROMIO allows users to access files on multiple file systems in the same program; therefore, it needs to know the type of file system on which a given file resides. Users can specify the type of file system explicitly by prefixing the filename with a string (like `nfs:`) or, on most machines, ROMIO can determine the type of file system on its own by using the function available for this purpose. On most file systems the function is `statvfs`, on some it is `statfs`, on Intel PFS it is `statpfs`, and on the NEC SX-4 it is `stat`.

#### 3.12.4 Automatic Configure and Build

Many parts of the ROMIO source code are conditionally compiled, depending on the features of the environment (machine, file system, MPI implementation). These features are detected automatically by using GNU's `autoconf` utility. We distribute ROMIO in the form of source code, and users can build it on any machine by simply doing

```
% configure
% make
```

We learned early on to have the configure script look for *features* of a particular environment and not for specific version numbers of the underlying operating system and other software. By following this approach, we are able to adapt easily to constantly changing version numbers and features. Users are also able to build ROMIO easily on new environments where we, the developers, have never before built or tested ROMIO.

## 4 Implications for File-System Design

File-system designers may want to know how they could design their file system to better support MPI-IO. We provide a list of features desired from a file system that would help in implementing MPI-IO correctly and with high performance.

1. **High-Performance Parallel File Access.** The file system must be designed to support *high-performance* access from *multiple* processes to a *common* file. This implies that concurrent requests (particularly writes) must not be serialized within the file system.

2. **Data-Consistency Semantics.** The data-consistency semantics in the presence of concurrent accesses from multiple processes must be clearly defined and correctly implemented. The file system must have a mode that supports byte-level consistency; it could support additional modes with weaker consistency semantics. (By byte-level consistency we mean that if a process writes some number of bytes starting from some location in the file, the data written must be visible to other processes immediately after the write from this process returns, without requiring an explicit cache flush.) Unix or POSIX consistency semantics, which support byte-level consistency, are sufficient for implementing MPI-IO.

3. **Atomicity Semantics.** File systems can deliver higher performance if they are not required to guarantee atomicity of accesses. Furthermore, most applications do not perform concurrent overlapping accesses and, consequently, do not need the stricter atomic mode. We therefore recommend that the file system support two modes: an atomic mode and a higher-performance nonatomic mode. Some file systems, such as IBM PIOFS and Intel PFS, already support both modes.

4. **File-Attribute Consistency.** The file system must also support consistency of file attributes, such as file size. For example, if two processes open a new (nonexisting) file, one process writes 100 bytes to the file, and the other process then calls a function that returns the size of the file, the function must return the file size as 100 bytes. We encountered problems with this feature on NFS, because NFS caches file attributes on each process noncoherently. As a result, the second process read the file size as zero bytes. We solved this problem by mounting the NFS directory with the "noac" option (no attribute caching).

5. **Interface Supporting Noncontiguous Accesses.** Although an MPI-IO implementation can perform data sieving to access noncontiguous data with high performance, we believe that the performance can be even better if data sieving is done within the file system. (Note that when data sieving is done within the file system, it is no different from regular caching; the extra data read/written can remain in the cache and need not be discarded.) For this purpose, the file system must provide an interface that supports noncontiguous accesses. A simple interface in which the user specifies a list of offsets and lengths is sufficient. (See Section 3.2 for reasons why POSIX `lio_listio` is not appropriate.) A simple interface, such as the following, is desired:

   ```
   int read_list(int mem_list_count,
                 long long *mem_offsets,
                 int *mem_lengths,
                 int file_list_count,
                 long long *file_offsets,
                 int *file_lengths)
   ```

   (similarly for `write_list`)

   where `mem_offsets` and `mem_lengths` are lists of offsets and lengths representing noncontiguous memory locations, `mem_list_count` is the number of entries in `mem_offsets` and `mem_lengths`, `file_offsets` and `file_lengths` are lists of offsets and lengths representing noncontiguous locations in the file, and `file_list_count` is the number of entries in `file_offsets` and `file_lengths`. This interface can be considered as a generalization of Unix `readv`/`writev` to allow noncontiguity in the file.

   In MPI-IO, noncontiguous data access with a single I/O function is allowed only to monotonically nondecreasing offsets in the file; memory offsets can be in any order. The `read_list`/`write_list` functions, therefore, need only allow monotonically nondecreasing offsets in `file_offsets`. This restriction can simplify the implementation of these functions.

6. **Support Files Larger than 2 Gbytes.** An increasing number of applications need to access files larger than 2 Gbytes. It is therefore critical that the file system be able to support large files. This means that the file-system interface and internal data structures must use 64-bit integers to represent file offsets.

7. **Byte-Range Locking.** The file system must support a locking facility equivalent to the advisory record-locking feature (`fcntl` locks) in Unix and POSIX. ROMIO uses this feature to implement MPI-IO's atomicity semantics for noncontiguous file accesses, to implement data sieving for write requests, and to implement shared file pointers.

8. **Control over File Striping.** Since the best values for file-striping parameters often depend on the application's access pattern, we recommend that the file system use a "good" set of values as the default and provide a facility for users to vary these parameters on a per-file basis.

9. **Variable Caching/Prefetching Policies.** Parallel applications exhibit such a wide variation in access patterns that any one caching/prefetching policy is unlikely to perform well for all applications [27]. The file system must therefore either detect and automatically adapt to changing access patterns [16, 17] or provide an interface for the user to specify the access pattern or caching/prefetching policy [2, 22].

10. **File Preallocation.** It is easy and inexpensive for a file system to provide a function to preallocate disk space for a file. If such a function is not provided, the MPI-IO function `MPI_File_preallocate` can be implemented only by actually writing data to the file, which is very expensive.

11. **Leave Collective I/O to the MPI-IO Implementation.** It is not entirely clear whether collective I/O is better if performed in the file system or as a library above the file system. Both techniques have been proposed in the literature [5, 14, 25]. Our opinion is that, for implementing MPI-IO's collective-I/O functionality, it is best if the file system focused on delivering the highest possible performance for independent (potentially noncontiguous) I/O requests from individual processes (as mentioned in item 5 above), and the MPI-IO implementation did the tasks of identifying the group of processes participating in the collective-I/O operation, efficiently shuffling data among the processes, and making large I/O requests from each process wherever possible. This approach keeps the file-system code simpler and, as ROMIO demonstrates [33], can also deliver high performance.

12. **No shared file pointers.** Implementing shared file pointers within the file system also requires the file system to know which processes share the shared file pointer; that is, the file system must support the notion of MPI communicators or process groups or their equivalent. We believe that it would be simpler if the MPI-IO implementation instead implements shared file pointers on top of the file system by using any of the three methods described in Section 3.9.

13. **Nonblocking (Asynchronous) I/O Optional.** It is not mandatory for the file system to provide nonblocking I/O functions. An MPI-IO implementation can perform nonblocking I/O by using threads that call the blocking I/O functions. This method, however, requires proper thread support from the machine and a thread-safe MPI implementation.

We note that the semantics and interface provided by a POSIX file system are sufficient for implementing MPI-IO *correctly* (as ROMIO demonstrates), but additional features would help an MPI-IO implementation achieve higher *performance*. (ROMIO compensates for the absence of these features by performing optimizations such as data sieving and collective I/O.) Among the features listed above, the following are not supported in POSIX: an interface for noncontiguous accesses, control over file striping, hints for caching/prefetching policies, and file preallocation. High-performance parallel file access and file sizes larger than 2 Gbytes are not mandated by POSIX but are considered "implementation-dependent" features.

## 5 Conclusions

ROMIO demonstrates that it is possible to implement MPI-IO portably on multiple machines and file systems and also achieve high performance. The ADIO framework is the key component that makes this all possible, as it enables us to perform file-system-specific optimizations within a largely portable implementation.

The discussion in this paper covers numerous file systems—almost all the file systems on commercially available machines. An important storage system that we did not discuss (mainly because ROMIO is not implemented on it) is HPSS [37]. HPSS is different from other file systems in its goals and design features; for example, it supports *third-party transfer*. A group at Lawrence Livermore National Laboratory has implemented MPI-IO on HPSS, and we refer interested readers to [13] for a discussion of issues related to implementing MPI-IO on HPSS.

By making MPI-IO available everywhere and also delivering high performance, we expect that it will be widely used and popular among application programmers. We believe it will solve some of the I/O performance and portability problems currently experienced in parallel applications.

### Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Defense Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

## References

[1] S. Baylor and C. Wu. Parallel I/O Workload Characteristics Using Vesta. In R. Jain, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 7, pages 167–185. Kluwer Academic Publishers, 1996.

[2] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] P. Corbett, J. Prost, C. Demetriou, G. Gibson, E. Reidel, J. Zelenka, Y. Chen, E. Felten, K. Li, J. Hartman, L. Peterson, B. Bershad, A. Wolman, and R. Aydt. Proposal for a Common Parallel File System Programming Interface, Version 1.0. On the World-Wide Web at `http://www.cs.arizona.edu/sio/api1.0.ps.gz`, September 1996.

[4] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[5] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993. Also published in *Computer Architecture News*, 21(5):31–38, December 1993.

[6] P. Dickens and R. Thakur. Improving Collective I/O Performance Using Threads. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999.

[7] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO—A Portable Implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[8] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25. ACM Press, November 1997.

[9] G. Gibson, D. Stodolsky, P. Chang, W. Courtwright II, C. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. Patterson, J. Su, R. Youssef, and J. Zelenka. The Scotch Parallel Storage Systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410. IEEE Computer Society Press, Spring 1995.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[11] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[12] IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language], 1996 edition.

[13] T. Jones, R. Mark, J. Martin, J. May, E. Pierce, and L. Stanberry. An MPI-IO Interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I:37–50, September 1996.

[14] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[15] O. Krieger and M. Stumm. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. In *Proceedings of Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108. ACM Press, May 1996.

[16] T. Madhyastha and D. Reed. Intelligent, Adaptive File System Policy Selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 172–179. IEEE Computer Society Press, October 1996.

[17] T. Madhyastha and D. Reed. Exploiting Global Input/Output Access Pattern Classification. In *Proceedings of SC97: High Performance Networking and Computing*. ACM Press, November 1997.

[18] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.1, June 1995. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[19] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 1997. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[20] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. *Parallel Computing*, 23(4):447–476, June 1997.

[21] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[22] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 79–95. ACM Press, December 1995.

[23] J. Prost. MPI-IO/PIOFS. World-Wide Web page at `http://www.research.ibm.com/people/p/prost/sections/mpiio.html`, 1996.

[24] D. Sanders, Y. Park, and M. Brodowicz. Implementation and Performance of MPI-IO File Access Using MPI Datatypes. Technical Report UH-CS-96-12, University of Houston, November 1996.

[25] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[26] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE Computer Society Press, 1996.

[27] E. Smirni and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[28] H. Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[29] W. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, Inc., 1992.

[30] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[31] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[32] R. Thakur, W. Gropp, and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag, September 1996.

[33] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, February 1999.

[34] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998.

[35] R. Thakur, E. Lusk, and W. Gropp. I/O in Parallel Applications: The Weakest Link. *International Journal of High Performance Computing Applications*, 12(4):389–395, Winter 1998.

[36] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. On the World-Wide Web at `http://parallel.nas.nasa.gov/MPI-IO`, April 1996.

[37] R. Watson and R. Coyne. The Parallel I/O Architecture of the High-Performance Storage System (HPSS). In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE Computer Society Press, September 1995.