1

High-Performance Techniques for Parallel I/O

Avery Ching
Northwestern University

Kenin Coloma
Northwestern University

Jianwei Li
Northwestern University

Alok Choudhary
Northwestern University

Wei-keng Liao
Northwestern University

1.1 Introduction

An important aspect of any large-scale scientific application is data storage and retrieval. I/O technology lags other computing components by several orders of magnitude with a performance gap that is still growing. In short, much of I/O research is dedicated to narrowing this gap.

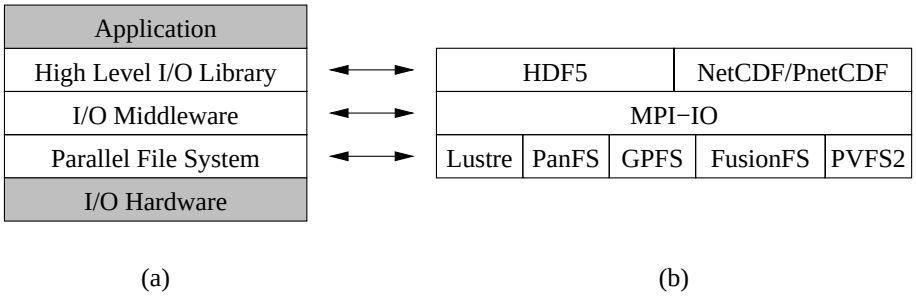

FIGURE 1.1: (a) Abstract I/O software stack for scientific computing. (b) Current components of the commonly used I/O software stack.

0-8493-8597-0/01/$0.00+$1.50

Applications that utilize high-performance I/O do so at a specific level in the parallel I/O software stack depicted in Figure 1.1. In the upper levels, file formats and libraries such as netCDF and HDF5 provide certain advantages for particular application groups. MPI-IO applications can leverage optimizations in the MPI specification [Mes] for various operations in the MPI-IO and file system layers. This chapter explains many powerful I/O techniques applied to each stratum of the parallel I/O software stack.

1.2 Portable File Formats and Data Libraries

Low level I/O interfaces, like UNIX I/O, treat files as sequences of bytes. Scientific applications manage data at a higher level of abstraction where users can directly read/write data as complex structures instead of byte streams and have all type information and other useful metadata automatically handled. Applications commonly run on multiple platforms also require portability of data so that the data generated from one platform can be used on another without transformation. As most scientific applications are programmed to run in parallel environments, parallel access to the data is desired. This section describes two popular scientific data libraries and their portable file formats, netCDF and HDF5.

1.2.1 File Access in Parallel Applications

Before presenting a detailed description of library design, general approaches for accessing portable files in parallel applications (in a message-passing environment) are analyzed. The first and most straightforward approach is described in the scenario of Figure 1.2a where one process is in charge of collecting/distributing data and performing I/O to a single file using a serial API. The I/O requests from other processes are carried out by shipping all the data through this single process. The drawback of this approach is that collecting all I/O data on a single process can easily create an I/O performance bottleneck and also overwhelm its memory capacity.

In order to avoid unnecessary data shipping, an alternative approach has all processes perform their I/O independently using the serial API, as shown in Figure 1.2b. In this way, all I/O operations can proceed concurrently, but over separate files (one for each process). Managing a dataset is more difficult, however, when it is spread across multiple files. This approach undermines the library design goal of easy data integration and management.

A third approach introduces a parallel API with parallel access semantics and an optimized parallel I/O implementation where all processes perform I/O operations to access a single file. This approach, as shown in Figure 1.2c, both frees the users from dealing with parallel I/O intricacies and provides more opportunities for various parallel I/O optimizations. As a result, this design principle is prevalent among modern scientific data libraries.

1.2.2 NetCDF and Parallel NetCDF

NetCDF [RD90], developed at the Unidata Program Center, provides applications with a common data access method for the storage of structured datasets. Atmospheric science applications, for example, use netCDF to store a variety of data types that include single-point observations, time series, regularly spaced grids, and satellite or radar images. Many organizations, such as much of the climate modeling community, rely on the netCDF data access standard for data storage.

NetCDF stores data in an array-oriented dataset which contains dimensions, variables,

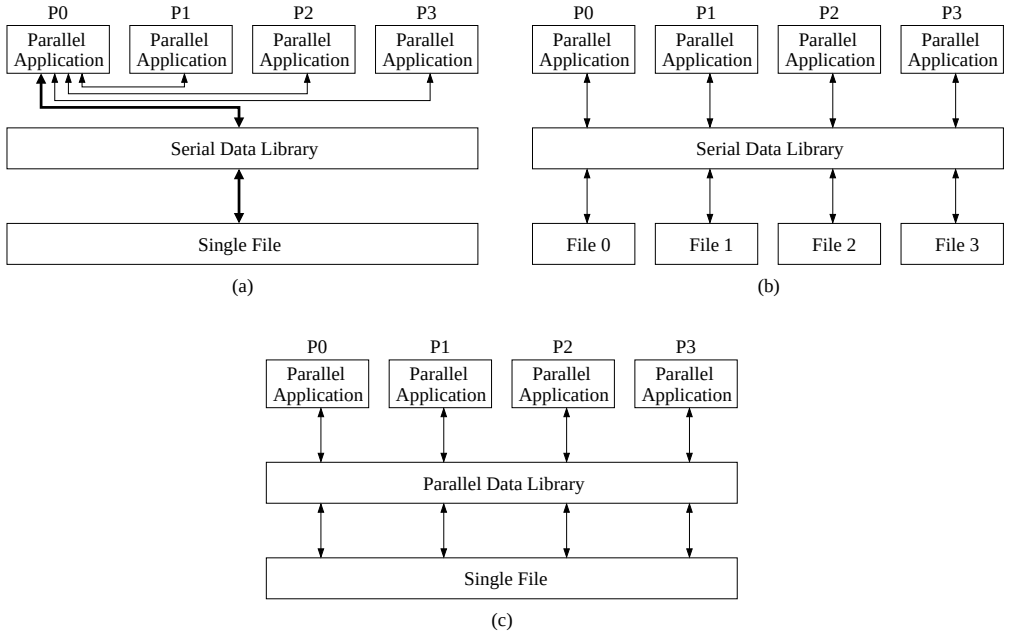

FIGURE 1.2: Using data libraries in parallel applications: (a) using a serial API to access single files through a single process; (b) using a serial API to access multiple files concurrently and independently; (c) using a parallel API to access single files cooperatively or collectively.

TABLE 1.1 NetCDF Library Functions

Function Type	Description
Dataset Functions	create/open/close a dataset, set the dataset to define/data mode, and synchronize dataset
Define Mode Functions	define dataset dimensions and variables
Attribute Functions	manage adding, changing, and reading attributes of datasets
Inquiry Functions	return dataset metadata: dim(id, name, len), var(name, ndims, shape, id)
Data Access Functions	provide the ability to read/write variable data in one of the five access methods: single value, whole array, subarray, subsampled array (strided subarray) and mapped strided subarray

and attributes. Physically, the dataset file is divided into two parts: file header and array data. The header contains all information (metadata) about dimensions, attributes, and variables except for the variable data itself, while the data section contains arrays of variable values (raw data). Fix-sized arrays are stored contiguously starting from given file offsets, while variable-sized arrays are stored at the end of the file as interleaved records that grow together along a shared unlimited dimension.

The netCDF operations can be divided into the five categories as summarized in Table 1.1. A typical sequence of operations to write a new netCDF dataset is to create the dataset; define the dimensions, variables, and attributes; write variable data; and close the dataset. Reading an existing netCDF dataset involves first opening the dataset; inquiring about dimensions, variables, and attributes; then reading variable data; and finally closing the dataset.

The original netCDF API was designed for serial data access, lacking parallel semantics and performance. Parallel netCDF (PnetCDF) [LLC+03], developed jointly between

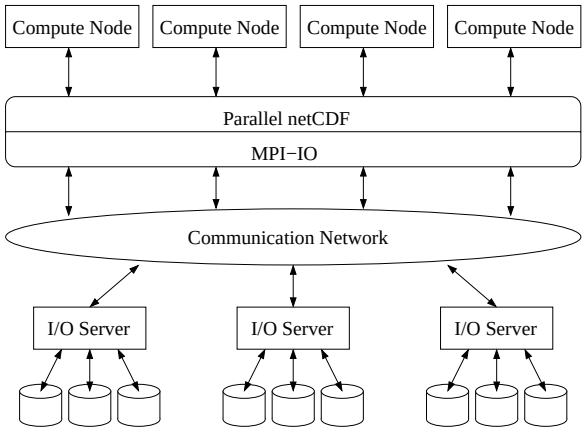

FIGURE 1.3: Design of PnetCDF on a parallel I/O architecture. PnetCDF runs as a library between the user application and file system. It processes parallel netCDF requests from user compute nodes and, after optimization, passes the parallel I/O requests down to MPI-IO library. The I/O servers receive the MPI-IO requests and perform I/O over the end storage on behalf of the user.

Northwestern University and Argonne National Laboratory (ANL), provides a parallel API to access netCDF files with significantly better performance. It is built on top of MPI-IO, allowing users to benefit from several well-known optimizations already used in existing MPI-IO implementations, namely the data sieving and two phase I/O strategies in ROMIO. MPI-IO is explained in further detail in Section 1.3. Figure 1.3 describes the overall architecture for PnetCDF design.

In PnetCDF, a file is opened, operated, and closed by the participating processes in an MPI communication group. Internally, the header is read/written only by a single process, although a copy is cached in local memory on each process. The *root* process fetches the file header, broadcasts it to all processes when opening a file, and writes the file header at the end of the define mode if any modifications occur in the header. The define mode functions, attribute functions, and inquiry functions all work on the local copy of the file header. All define mode and attribute functions are made collectively and require all the processes to provide the same arguments when adding, removing, or changing definitions so the local copies of the file header are guaranteed to be the same across all processes from the time the file is collectively opened until it is closed.

The parallelization of the data access functions is achieved with two subset APIs, the *high-level API* and the *flexible API*. The high-level API closely follows the original netCDF data access functions and serves as an easy path for original netCDF users to migrate to the parallel interface. These calls take a single pointer for a contiguous region in memory, just as the original netCDF calls did, and allow for the description of single elements (`var1`), whole arrays (`var`), subarrays (`vara`), strided subarrays (`vars`), and multiple noncontiguous regions (`varm`) in a file. The flexible API provides a more MPI-like style of access by providing the user with the ability to describe noncontiguous regions in memory. These regions are described using MPI datatypes. For application programmers that are already using MPI for message passing, this approach should be natural. The file regions are still described using the original parameters. For each of the five data access methods in the flexible data access functions, the corresponding data access pattern is presented as an

MPI file view (a set of data visible and accessible from an open file) constructed from the variable metadata (shape, size, offset, etc.) in the netCDF file header and user provided starts, counts, strides, and MPI datatype arguments. For parallel access, each process has a different file view. All processes can collectively make a single MPI-IO request to transfer large contiguous data as a whole, thereby preserving useful semantic information that would otherwise be lost if the transfer were expressed as per process noncontiguous requests.

1.2.3 HDF5

HDF (Hierarchical Data Format) is a portable file format and software, developed at the National Center for Supercomputing Applications (NCSA). It is designed for storing, retrieving, analyzing, visualizing, and converting scientific data. The current and most popular version is HDF5 [HDF], which stores multi-dimensional arrays together with ancillary data in a portable, self-describing file format. It uses a hierarchical structure that provides application programmers with a host of options for organizing how data is stored in HDF5 files. Parallel I/O is also supported.

HDF5 files are organized in a hierarchical structure, similar to a UNIX file system. Two types of primary objects, groups and datasets, are stored in this structure, respectively resembling directories and files in the UNIX file system. A group contains instances of zero or more groups or datasets while a dataset stores a multi-dimensional array of data elements. Both are accompanied by supporting metadata. Each group or dataset can have an associated attribute list to provide extra information related to the object.

A dataset is physically stored in two parts: a header and a data array. The header contains miscellaneous metadata describing the dataset as well as information that is needed to interpret the array portion of the dataset. Essentially, it includes the name, datatype, dataspace, and storage layout of the dataset. The name is a text string identifying the dataset. The datatype describes the type of the data array elements and can be a basic (atomic) type or a compound type (similar to a *struct* in C language). The dataspace defines the dimensionality of the dataset, i.e., the size and shape of the multi-dimensional array. The dimensions of a dataset can be either fixed or unlimited (extensible). Unlike netCDF, HDF5 supports more than one unlimited dimension in a dataspace. The storage layout specifies how the data arrays are arranged in the file.

The data array contains the values of the array elements and can be either stored together in contiguous file space or split into smaller *chunks* stored at any allocated location. Chunks are defined as equally-sized multi-dimensional subarrays (blocks) of the whole data array and each chunk is stored in a separate contiguous file space. The chunked layout is intended to allow performance optimizations for certain access patterns, as well as for storage flexibility. Using the chunked layout requires complicated metadata management to keep track of how the chunks fit together to form the whole array. Extensible datasets whose dimensions can grow are required to be stored in chunks. One dimension is increased by allocating new chunks at the end of the file to cover the extension.

The HDF5 library provides several interfaces that are categorized according to the type of information or operation the interface manages. Table 1.2 summarizes these interfaces.

To write a new HDF5 file, one needs to first create the file, adding groups if needed; create and define the datasets (including their datatypes, dataspaces, and lists of properties like the storage layout) under the desired groups; write the data along with attributes; and finally close the file. The general steps in reading an existing HDF5 file include opening the file; opening the dataset under certain groups; querying the dimensions to allocate enough memory to a read buffer; reading the data and attributes; and closing the file.

HDF5 also supports access to portions (or selections) of a dataset by *hyperslabs*, their

TABLE 1.2 HDF5 Interfaces

Interface		Function Name Prefix and Functionality
Library Functions	H5:	General HDF5 library management
Attribute Interface	H5A:	Read/write attributes
Dataset Interface	H5D:	Create/open/close and read/write datasets
Error Interface	H5E:	Handle HDF5 errors
File Interface	H5F:	Control HDF5 file access
Group Interface	H5G:	Manage the hierarchical group information
Identifier Interface	H5I:	Work with object identifiers
Property List Interface	H5P:	Manipulate various object properties
Reference Interface	H5R:	Create references to objects or data regions
Dataspace Interface	H5S:	Defining dataset dataspace
Datatype Interface	H5T:	Manage type information for dataset elements
Filters & Compression Interface	H5Z:	Inline data filters and data compression

unions, and lists of independent points. Basically, a hyperslab is a subarray or strided subarray of the multi-dimensional dataset. The selection is performed in the file dataspace for the dataset. Similar selections can be done in the memory dataspace so that data in one file pattern can be mapped to memory in another pattern as long as the total number of data elements is equal.

HDF5 supports both sequential and parallel I/O. Parallel access, supported in the MPI programming environment, is enabled by setting the file access property to use MPI-IO when the file is created or opened. The file and datasets are collectively created/opened by all participating processes. Each process accesses part of a dataset by defining its own file dataspace for that dataset. When accessing data, the data transfer property specifies whether each process will perform independent I/O or all processes will perform collective I/O.

1.3 General MPI-IO Usage and Optimizations

Before MPI, there were proprietary message passing libraries available on several computing platforms. Portability was a major issue for application designers and thus more than 80 people from 40 organizations representing universities, parallel system vendors, and both industrial and national research laboratories formed the Message Passing Interface (MPI) Forum. MPI-1 was established by the forum in 1994. A number of important topics (including parallel I/O) had been intentionally left out of the MPI-1 specification and were to be addressed by the MPI Forum in the coming years. In 1997, the MPI-2 standard was released by the MPI Forum which addressed parallel I/O among a number of other useful new features for portable parallel computing (remote memory operations and dynamic process management). The I/O goals of the MPI-2 standard were to provide developers with a portable parallel I/O interface that could richly describe even the most complex of access patterns. ROMIO [ROM] is the reference implementation distributed with ANL's MPICH library. ROMIO is included in other distributions and is often the basis for other MPI-IO implementations. Frequently, higher level libraries are built on top of MPI-IO, which leverage its portability across different I/O systems while providing features specific to a particular user community. Examples such as netCDF and HDF5 were discussed in Section 1.2.

1.3.1 MPI-IO Interface

The purposely rich MPI-IO interface has proven daunting to many. This is the main obstacle to developers using MPI-IO directly, and also one of the reasons most developers subsequently end up using MPI-IO through higher level interfaces like netCDF and HDF5.

TABLE 1.3 Commonly used MPI datatype constructor functions. Internal offsets can be described in terms of the base datatype or in bytes.

function	internal offsets	base types
MPI_Type_contiguous	none	single
MPI_Type_vector	regular (old types)	single
MPI_Type_hvector	regular (bytes)	single
MPI_Type_index	arbitrary (old types)	single
MPI_Type_hindex	arbitrary (bytes)	single
MPI_Type_struct	arbitrary (old types)	mixed

It is, however, worth learning a bit of advanced MPI-IO, if not to encourage more direct MPI-IO programming, then to at least increase general understanding of what goes on in the MPI-IO level beneath the other high level interfaces. A very simple execution order of the functions described in this section is as follows:

1. MPI_Info_create/MPI_Info_set (optional)
2. datatype creation (optional)
3. MPI_File_open
4. MPI_File_set_view (optional)
5. MPI_File_read/MPI_File_write
6. MPI_File_sync (optional)
7. MPI_File_close
8. datatype deletion (optional)
9. MPI_Info_free (optional)

Open, Close, and Hints

```
MPI_File_open(comm, filename, amode, info, fh)
MPI_File_close(fh)
MPI_Info_create(info)
MPI_Info_set(info, key, value)
MPI_Info_free(info)
```

While definitely far from "advanced" MPI-IO, `MPI_File_open` and `MPI_File_close` still warrant some examination. The `MPI_File_open` call is the typical point at which to pass optimization information to an MPI-IO implementation. `MPI_Info_create` should be used to instantiate and initialize an `MPI_Info` object, and then `MPI_Info_set` is used to set specific hints (*key*) in the info object. The info object should then be passed to `MPI_File_open` and later freed with `MPI_Info_free` after the file is closed. If an info object is not needed, `MPI_INFO_NULL` can be passed to open. The hints in the info object are used to either control optimizations directly in an MPI-IO implementation or to provide additional access information to the MPI-IO implementation so it can make better decisions on optimizations. Some specific hints are described in 1.3.2. To get the hints of the info object back from the MPI-IO library the user should call `MPI_File_get_info` and be sure to free the info object after use.

Derived Datatypes

Before delving into the rest of the I/O interface and capabilities of MPI-IO, it is essential to have a sound understanding of derived datatypes. Datatypes are what distinguish the

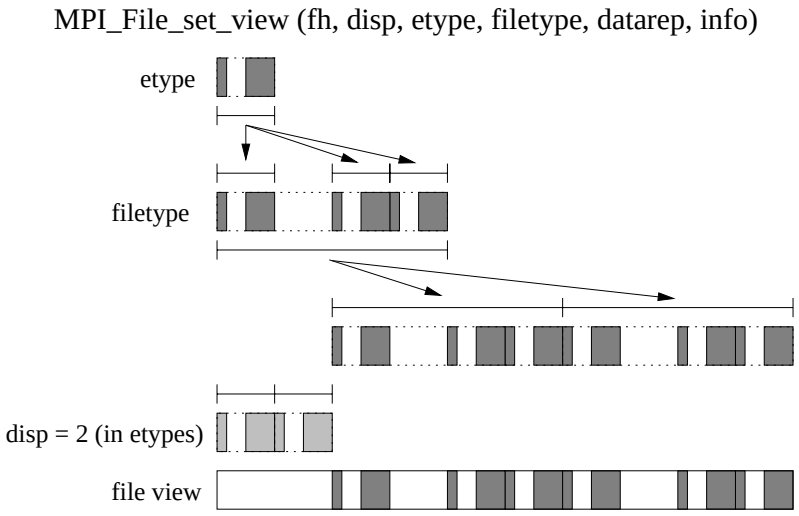

FIGURE 1.4: File views illustrated: filetypes are built from etypes. The filetype access pattern is implicitly iterated forward starting from the disp. An actual count for the filetype is not required as it conceptually repeats forever, and the amount of I/O done is dependent on the buffer datatype and count.

MPI-IO interface from the more familiar standard POSIX I/O interface.

One of the most powerful features of the MPI specification is user defined derived datatypes. MPI's derived datatypes allow a user to describe an arbitrary pattern in a memory space. This access pattern, possibly noncontiguous, can then be logically iterated over the memory space. Users may define derived datatypes based on elementary MPI predefined datatypes (`MPI_INT`, `MPI_CHAR`, *etc.*) as well as previously defined derived datatypes. A common and simple use of derived datatypes is to single out values for a specific subset of variables in multi-dimensional arrays.

After using one or more of the basic datatype creation functions in table 1.3, `MPI_Type_commit` is used to finalize the datatype and must be called before use in any MPI-IO calls. After the file is closed, the datatype can then be freed with `MPI_Type_free`.

Seeing as a derived datatype simply maps an access pattern in a logical space, while the discussion above has focused on memory space, it could also apply to file space.

File Views

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
```

File views specify accessible file regions using derived datatypes. This function should be called after the file is opened, if at all. Not setting a file view allows the entire file to be accessed. The defining datatype is referred to as the *filetype*, and the *etype* is a datatype used as an elementary unit for positioning. Figure 1.4 illustrates how the parameters in `MPI_File_set_view` are used to describe a "window" revealing only certain bytes in the file. The displacement (*disp*) dictates the start location of the initial filetype in terms of etypes. The file view is defined by both the displacement and filetype together. While this function is collective, it is important each process defines its own individual file view. All processes in the same communicator must use the same etype. The *datarep* argument is

typically set to "native," and has to do with file interoperability. If compatibility between MPI environments is needed or the environment is heterogeneous, then "external32" or "internal" should be used. File views allow an MPI-IO read or write to access complex noncontiguous regions in a single call. This is the first major departure from the POSIX I/O interface, and one of the most important features of MPI-IO.

Read and Write

```
MPI_File_read(fh, buf, count, datatype, status)
MPI_File_write(fh, buf, count, datatype, status)
MPI_File_read_at(fh, offset, buf, count, datatype, status)
MPI_File_write_at(fh, offset, buf, count, datatype, status)
MPI_File_sync(fh)
```

In addition to the typical MPI specific arguments like the MPI communicator, the datatype argument in these calls is the second important distinction of MPI-IO. Just as the file view allows one MPI-IO call to access multiple noncontiguous regions in file, the datatype argument allows a single MPI-IO call to access multiple memory regions in the user buffer with a single call. The *count* is the number of datatypes in memory being used.

The functions `MPI_File_read` and `MPI_File_write` use `MPI_File_seek` to set the position of the file pointer in terms of etypes. It is important to note that the file pointer position respects the file view, skipping over inaccessible regions in the file. Setting the file view resets the individual file pointer back to the first accessible byte.

The `MPI_File_read_at` and `MPI_File_write_at`, "_at" variations of the read and write functions, explicitly set out a starting position in the additional *offset* argument. Just as in the seek function, the offset is in terms of etypes and respects the file view.

Similar to MPI non-blocking communication, non-blocking versions of the I/O functions exist and simply prefix read and write with "i" so the calls look like `MPI_File_iread`. The I/O need not be completed before these functions return. Completion can be checked just as in non-blocking communication with completion functions like `MPI_Wait`.

The `MPI_File_sync` function is a collective operation used to ensure written data is pushed all the way to the storage device. Open and close also implicitly guarantee data for the associated file handle is on the storage device.

Collective Read and Write

```
MPI_File_read_all(fh, buf, count, datatype, status)
MPI_File_write_all(fh, buf, count, datatype, status)
```

The collective I/O functions are prototyped the same as the independent `MPI_File_read` and `MPI_File_write` functions and have "_at" equivalents as well. The difference is that the collective I/O functions must be called collectively among all the processes in the communicator associated with the particular file at open time. This explicit synchronization allows processes to actively communicate and coordinate their I/O efforts for the call. One major optimization for collective I/O is disk directed I/O [Kot94, Kot97]. Disk directed I/O allows I/O servers to optimize the order in which local blocks are accessed. Another optimization for collective I/O is the two phase method described in further detail in the next section.

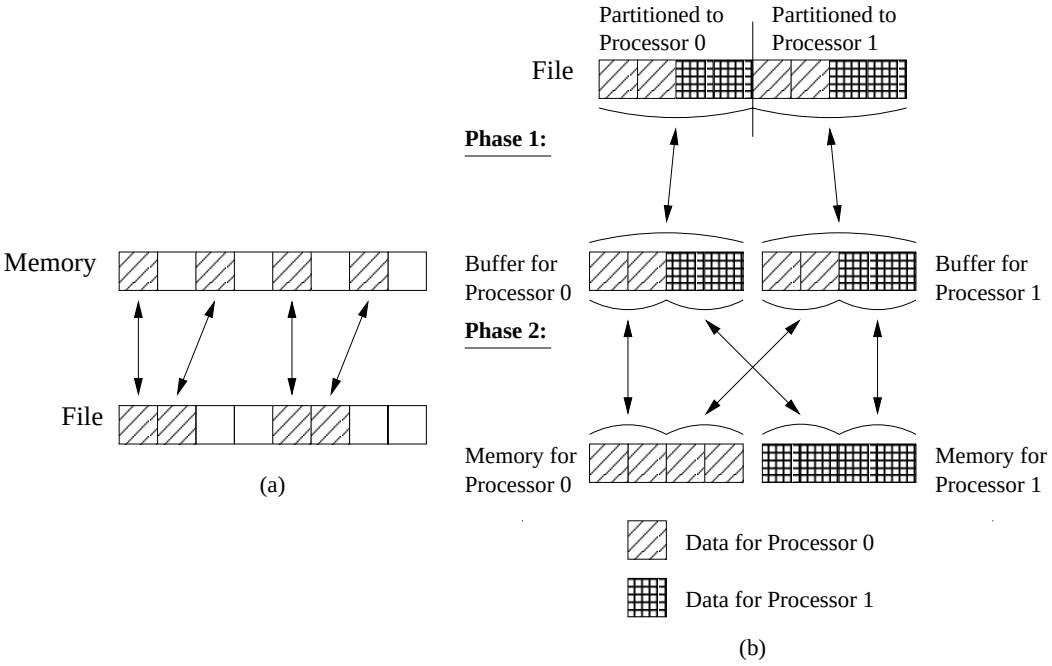

FIGURE 1.5: (a) Example POSIX I/O request. Using traditional POSIX interfaces for this access pattern cost four I/O requests, one per contiguous region. (b) Example two phase I/O request. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two phase method.

1.3.2 Significant Optimizations in ROMIO

The ROMIO implementation of MPI-IO contains several optimizations based on the POSIX I/O interface, making them portable across many file systems. It is possible, however, to implement a ROMIO driver with optimizations specific to a given file system. In fact, the current version of ROMIO as of this writing (2005-06-09) already includes optimizations for PVFS2 [The] and other file systems. The most convenient means for controlling these optimizations is through the MPI-IO hints infrastructure mentioned briefly above.

POSIX I/O

All parallel file systems support what is called the *POSIX I/O* interface, which relies on an offset and a length in both memory and file to service an I/O request. This method can service noncontiguous I/O access patterns by dividing them up into contiguous regions and then individually accessing these regions with corresponding POSIX I/O operations. While such use of POSIX I/O can fulfill any noncontiguous I/O request with this technique, it does incur several expensive overheads. The division of the I/O access pattern into smaller contiguous regions significantly increases the number of I/O requests processed by the underlying file system. Also, the division often forces more I/O requests than the actual number of noncontiguous regions in the access pattern as shown in Figure 1.5a. The serious overhead sustained from servicing so many individual I/O requests limits performance for noncontiguous I/O when using the POSIX interface. Fortunately for users which have access

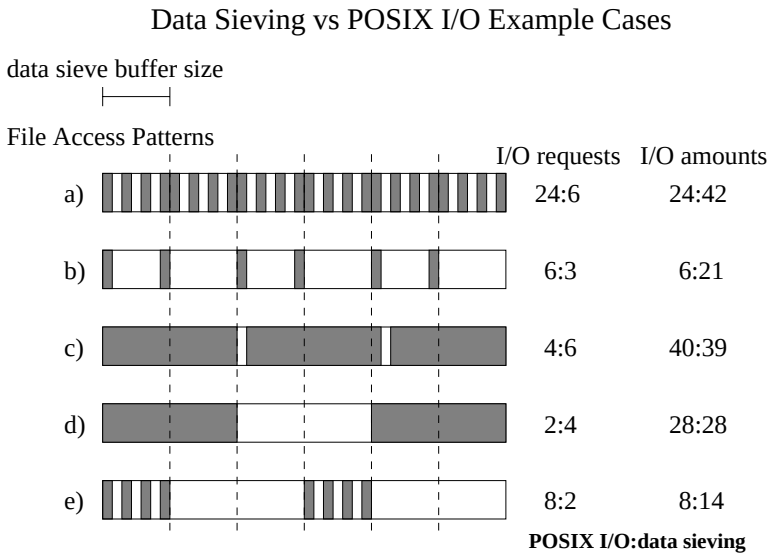

FIGURE 1.6: (a) Probably data sieve: Data sieving reduces I/O requests by a factor of 4, but almost doubles the I/O amount (b) Do not data sieve: Data sieving I/O requests are reduced by half, but almost 4 (8 if write) times more data is accessed (c) Do not data sieve: Data sieving increases I/O requests and only marginally reduces I/O amount. (d) Do not data sieve (Pareto optimal):Data sieving doubles I/O requests, but has no effect on I/O amount. (e) Probably data sieve: Data sieving reduced I/O requests by a factor of 4, but almost doubles I/O.

to file systems supporting only the POSIX interface, two important optimizations exist to more efficiently perform noncontiguous I/O while using only the POSIX I/O interface: data sieving I/O and two phase I/O.

Data Sieving

Since hard disk drives are inherently better at accessing large amounts of sequential data, the *data sieving* technique [TGL99a] tries to satisfy multiple small I/O requests with a larger contiguous I/O access and later "sifting" the requested data in or out of a temporary buffer. In the read case, a large contiguous region of file data is first read into a temporary data sieving buffer and then the requested data is copied out of the temporary buffer into the user buffer. For practical reasons, ROMIO uses a maximum data sieving buffer size so multiple data sieving I/O requests may be required to service an access patterns. ROMIO will always try to fill the entire data sieving buffer each time in order to maximize the number of file regions encompassed. In the write case, file data must first be read into the data sieving buffer unless the user define regions in that contiguous file region cover the entire data sieving region. User data can then be copied into the data sieving buffer and then the entire data sieving buffer is written to the file in a single I/O call. Data sieving writes require some concurrency control since data that one process does not intend to modify is still read and then written back with the potential of overwriting changes made by other processes.

Data sieving performance benefits come from reducing the number of head seeks on the

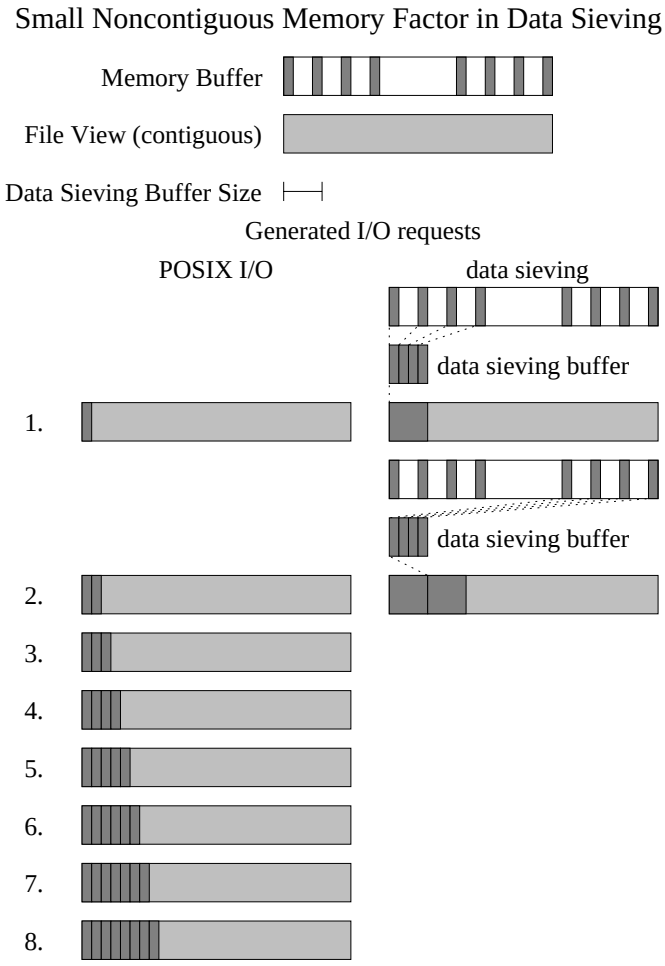

FIGURE 1.7: Evaluating the file access pattern alone in this case does not paint the entire I/O picture. The small noncontiguous memory pieces break up the large contiguous file access pattern into many small I/O requests. Since these small I/O requests end up next to each other, data sieving can reduce the number of I/O requests by a factor of 4 without accessing any extraneous data, making data sieving Pareto optimal, assuming it takes longer to read/write 1 unit of data 4 times than to copy 4 units of data into or out of the buffer and to read/write 4 units of data.

disk, the cut in the accrued overhead of individual I/O requests, and large I/O accesses. Figure 1.6a and Figure 1.6e illustrate specific cases where data sieving may do well. Data sieving is less efficient when data is either sparsely distributed or the access pattern consists of contiguous regions much larger than the data sieving buffer size (end case would be a completely contiguous access pattern). In the sparse case, as in Figure 1.6b and 1.6d, the large data sieving I/O request may only satisfy a few user requests, and in even worse, may be accessing much more data than will actually be used (Figure 1.6b). The number of I/O accesses may not be reduced by much, and the extra time spent accessing useless data may be more than the time taken to make more small I/O requests. In the case where the user's

General Guide to Using Data Sieving
vs POSIX I/O

Small noncontig memory regions

noncontig file region distribution

noncontig file regions

	sparse	dense
small	no	yes
large	yes	yes

Other memory regions and sizes

noncontig file regions

	sparse	dense
small	no	yes
large	no	no

FIGURE 1.8: The three main factors to consider in determining whether to use data sieving are whether the user buffer is noncontiguous with small pieces, the size of the noncontiguous file regions, and the distribution of the file accesses all with respect to the data sieving buffer size. If both memory and file descriptions are contiguous, do not use data sieving.

access pattern is made up of contiguous regions nearly the size of or greater than the data sieving buffer size, shown in Figure 1.6c and 1.6d, the number of I/O requests generated may actually be greater than the number of I/O requests generated had the user's I/O requests been passed directly to the file system. Additionally, data sieving will have been double buffering, and paid an extra memory copy penalty for each time the data sieve buffer was filled and emptied.

One factor not yet considered is the user memory buffer. If the user memory buffer is noncontiguous with small regions (relative to the data sieving buffer), it will have the effect of breaking up, but not separating what might have been large contiguous regions in file, thus creating an numerous I/O requests for POSIX I/O. This effect is illustrated in Figure 1.7, and presents an ideal opportunity for data sieving to reduce the overall number of I/O calls, as well as making efficient use of the data sieving buffer. Even if the original filetype consisted of large sparsely distributed regions, data sieving would still likely prove to be very beneficial.

So while data sieving could conceivably result in worse performance (the point at which would be sooner in the case of read-modify-write data sieving writes), some simple considerations can be kept in mind to determine whether data sieving will be a benefit or detriment. Assuming data is fairly uniformly spaced (no locally dense, overall sparse distributions), and the user access pattern is indeed noncontiguous, Figure 1.8 provides a quick table for determining when data sieving is most appropriate. Small, big, sparse, and dense metrics are all relative to the data sieving buffer size. An MPI-IO implementation ought to

preprocess the user's access pattern at least to some degree to determine the appropriateness of data sieving on its own. As mentioned earlier, however, less uniform access patterns may require some user intervention as an automated runtime determination may not catch certain cases. In the previous example (Figure 1.6e), an access pattern which consists of clusters of densely packed data will likely benefit from data sieving. Using only the data sieving technique for I/O will be referred to as *data sieving I/O*.

Two Phase I/O

Figure 1.5b illustrates the two phase method for collective I/O [TGL99b], which uses both POSIX I/O and data sieving. This method is referred to as *two phase I/O* throughout this chapter. The two phase method identifies a subset of the application processes that will actually do I/O; these processes are called *aggregators*. Each aggregator is responsible for I/O to a specific and disjoint portion of the file.

In an effort to heuristically balance I/O load on each aggregator, ROMIO calculates these *file realms* dynamically based on the aggregate size and location of the accesses in the collective operation. When performing a read operation, aggregators first read a contiguous region containing desired data from storage and put this data in a local temporary buffer. Next, data is redistributed from these temporary buffers to the final destination processes. Write operations are performed in a similar manner. First, data is gathered from all processes into temporary buffers on aggregators. Aggregators read data from storage to fill in the holes in the temporary buffers to make contiguous data regions. Next, this temporary buffer is written back to storage using POSIX I/O operations. An approach similar to data sieving is used to optimize this write back to storage when there are still gaps in the data. As mentioned earlier, data sieving is also used in the read case. Alternatively, other noncontiguous access methods, such as the ones described in Section 1.4.2, can be leveraged for further optimization.

The big advantage of two phase I/O is the consolidation by aggregators of the noncontiguous file accesses from all processes into only a few large I/O operations. One significant disadvantage of two phase I/O is that all processes must synchronize on the open, set view, read, and write calls. Synchronizing across large numbers of processes with different sized workloads can be a large overhead. Two phase I/O performance relies heavily on the particular MPI implementation's data movement performance. If the MPI implementation is not significantly faster than the aggregate I/O bandwidth in the system, the overhead of the additional data movement in two phase I/O will likely prevent two phase I/O from outperforming direct access optimizations like list I/O and datatype I/O discussed later.

Common ROMIO Hints

There are a few reserved hints in the MPI-IO specification and are therefore universal across MPI-IO implementations, but for the most part, MPI-IO implementers are free to make up hints. The MPI-IO specification also dictates that any unrecognized hints should just be ignored, leaving data unaffected. In this way user applications that specify hints relevant to either a certain file system or MPI-IO implementation should still be portable, though the hints may be disregarded.

Table 1.4 lists some hints that are typically used. The exact data sieving hints are specific to ROMIO, but the collective I/O hints are respected across MPI-IO implementations. While an MPI-IO developer may choose not to implement two phase collective I/O, if they do decide to, they should use the hints in the table for user configuration. The *striping_factor* and *striping_unit* are standardized MPI-IO hints used to dictate file distribution parameters to the underlying parallel file system.

TABLE 1.4 The *same* column indicates whether the hint passed needs to be the same across all processes in the communicator. The *Std.* column indicates official MPI reserved hints.

Hint	Value	Same	Std.	Basic Description
romio_cb_read	E/D/A	yes	no	ctrl 2-phase collective reads
romio_cb_write	E/D/A	yes	no	ctrl 2-phase collective writes
cb_buffer_size	integer	yes	yes	2-phase collective buffer size
cb_nodes	integer	yes	yes	no. of collective I/O aggregators
cb_config_list	string list	yes	yes	list of collective aggregator hosts
romio_ds_read	E/D/A	no	no	ctrl data sieving for indep. reads
romio_ds_write	E/D/A	no	no	ctrl data sieving for indep. writes
romio_no_indep_rw	bool	yes	no	no subsequent indep. I/O
ind_rd_buffer_size	integer	no	no	read data sieve buffer sz.
ind_wr_buffer_size	integer	no	no	write data sieve buffer sz.
striping_factor	integer	yes	yes	no. of I/O servers to stripe file across
striping_unit	integer	yes	yes	stripe sz distributed on I/O servers
				E/D/A = Enable/Disable/Auto

Although hints are an important means for applications and their developers to communicate with MPI implementations, it is usually more desirable for the MPI-IO implementation to automate the use and configuration of any optimizations.

1.3.3 Current Areas of Research in MPI-IO

Although data sieving I/O and two phase I/O are both significant optimizations, I/O remains a serious bottleneck in high-performance computing systems. MPI-IO remains an important level in the software stack for optimizing I/O.

Persistent File Realms

The Persistent File Realm (PFR) [CCL+04] technique modifies the two phase I/O behavior in order to ensure valid data in an incoherent client-side file system cache. Following MPI consistency semantics, non-overlapping file writes should be immediately visible to all processes within the I/O communicator. An underlying file system must provide coherent client-side caching if any at all.

Maintaining cache coherency over a distributed or parallel file system is no easy task, and the overhead introduced by the coherence mechanisms sometimes outweigh the performance benefits of providing a cache. This is the exact reason that PVFS [CLRT00] does not provide a client-side cache.

If an application can use all collective MPI I/O functions, PFRs can carefully manage the actual I/O fed to the file system in order to ensure access to only valid data in an incoherent client-side cache. As mentioned earlier, the ROMIO two phase I/O implementation heuristically load balances the I/O responsibilities of the I/O aggregators. Instead of rebalancing and reassigning the *file realms* according to the accesses of each collective I/O call, the file realms "persist" between collective I/O calls. The key to PFRs is recognizing that the data cached on a node is not based on its own MPI-IO request, but the combined I/O accesses of the communicator group. Two phase I/O adds a layer of I/O indirection. As long as each I/O aggregator is responsible for the same file realm in each collective I/O call, the data it accesses will always be the most recent version. With PFRs, the file system no longer needs to worry about the expensive task of cache coherency, and the cache can still safely be used. The alternative is to give up on client-side caches completely as well as the performance boost they offer.

Portable File Locking

The MPI-IO specification provides an *atomic mode*, which means file data should be sequentially consistent. As it is, even with a fully POSIX compliant file system, some extra work is required to implement atomicity in MPI-IO because of the potential for noncontiguous access patterns. MPI-IO functions must be sequentially consistent across noncontiguous file regions in atomic mode. In high-performance computing, it is typical for a file system to relax consistency semantics or for a file system that supports strict consistency to also support less strict consistency semantics. File locking is the easiest way to implement MPI's atomic mode. It can be done in three different ways based on traditional contiguous byte-range locks. The first is to lock the entire file being accessed during each MPI-IO call, the down side being potentially unneeded access serialization for all access to the file. The second is to lock a contiguous region starting from the first byte accessed ending at the last byte access. Again, since irrelevant bytes between a noncontiguous access pattern are locked, there is still potential for false sharing lock contention. The last locking method is two phase locking where byte range locks for the entire access (possibly noncontiguous) must be acquired before performing I/O [ACTC06]. While file locking is the most convenient way to enforce MPI's atomic mode, it is not always available.

Portable file locking at the MPI level [LGR+05] provides the necessary locks to implement the MPI atomic mode on any file system. This is accomplished by using MPI-2's Remote Memory Access (RMA) interface. The "lock" is a remote accessible boolean array the size of the number of processes *N*. Ideally, the array is a bit array, but it may depend on the granularity of a particular system. To obtain the lock, the process puts a *true* value to its element in the remote array and gets the rest of the array in one MPI-2 RMA epoch (in other words both the put and get happen simultaneously and atomically). If the array obtained is clear, the lock is obtained, otherwise the lock is already possessed by another process, and the waiting process sets up a `MPI_Recv` to receive the lock from another process. To release the lock, the locking process writes a *false* value to its element in the array and gets the rest of the array. If the array is clear, then no other process is waiting for the lock. If not, then should pass the lock with a `MPI_Send` call to the next waiting process in the array. Once lock contention manifests itself, the lock is passed around in the array sequentially (and circularly), and thus this algorithm does not provide fairness.

MPI-IO File Caching

It is important to remember that caching is a double-edged sword that can some times help and other times impede. Caching is not always desirable, and these situations should be recognized during either compile or run time [VSK+03]. Caching is ideally suited to applications performing relatively small writes that can be gathered into larger more efficient writes. Caching systems need to provide run time infrastructure for identifying patterns and monitoring cache statistics to make decisions such as whether to keep caching or bypass the cache. Ideally, the cache could self-tune other parameters like page sizes, cache sizes, and eviction thresholds as well.

Active Buffering [MWLY02] gathers writes on the client and uses a separate I/O thread on the client to actually write the data out to the I/O system. I/O is aggressively interleaved with computation, allowing computation to resume quickly.

DAChe [CCL+05] is a coherent cache system implemented using MPI-2 for communication. DAChe makes local client-side file caches remotely available to other processes in the communicator using MPI-2 RMA functions. The same effect can be achieved using threads on the clients to handle remote cache operations [LCC+05b, LCC+05a]. A threaded version of coherent caching provides additional functionality over DAChe to intermittently

write out the cache. Though some large-scale systems lack thread support, most clusters do support threads, and multi-core microprocessors are starting to become commonplace in high-performance computing. Modern high-performance computers also commonly use low latency networks with native support for one-sided RMA, which provides DAChe with optimized low level transfers. Cache coherency is achieved by allowing any given data to be cached on a single client at a time. Any peer accessing the same data passively accesses the data directly on the client caching the page, assuring a uniform view of the data by all processes.

1.4 Parallel File Systems

1.4.1 Summary of Current Parallel File Systems

Currently there are numerous parallel I/O solutions available. Some of the current major commercial efforts include Lustre [Lus], Panasas [Pan], GPFS [SH02], and IBRIX Fusion [IBR]. Some current and older research parallel file systems include PVFS [CLRT00, The], Clusterfile [IT01], Galley [NK96], PPFS [EKHM93], Scotch [GSC+95] and Vesta [CF96].

This section begins by describing some of the parallel file systems in use today. Next, various I/O methods for noncontiguous data access and how I/O access patterns and file layouts can significantly affect performance are discussed with a particular emphasis on structured scientific I/O parameters (region size, region spacing, and region count).

Lustre

Lustre is widely used at the U.S. National Laboratories, including Lawrence Livermore National Laboratory (LLNL), Pacific Northwest National Laboratory (PNNL), Sandia National Laboratories (SNL), the National Nuclear Security Administration (NNSA), Los Alamos National Laboratory (LANL), and NCSA. It is an open source parallel file system for Linux developed by Cluster File Systems, Inc. and HP.

Lustre is built on the concept of objects that encapsulate user data as well as attributes of that data. Lustre keeps unique inodes for every file, directory, symbolic link, and special file which holds references to objects on OSTs. Metadata and storage resources are split into metadata servers (MDSs) and object storage targets (OSTs), respectively. MDSs are replicated to handle failover and are responsible for keeping track of the transactional record of high level file system changes. OSTs handle actual file I/O directly to and from clients once a client has obtained knowledge of which OSTs contain the objects necessary for I/O. Since Lustre meets strong file system semantics through file locking, each OST handles locks for the objects that it stores. OSTs handle the interaction of client I/O requests and the underlying storage, which are called object-based disks (OBDs). While OBD drivers for accessing journaling file systems such as ext3, ReiserFS, and XFS are currently used in Lustre, manufacturers are working on putting OBD support directly into disk drive hardware.

PanFS

Panasas ActiveScale File System (PanFS) has customers in the areas of government sciences, life sciences, media, energy, and many others. It is a commercial product which, like Lustre, is also based on an object storage architecture.

The PanFS architecture consists of both metadata servers (MDSs) and object-based storage devices (OSDs). MDSs have numerous responsibilities in PanFS including authentication, file and directory access management, cache coherency, maintaining cache consistency

among clients, and capacity management. The OSD is very similar to Lustre's OST in that it is also a network-attached device smart enough to handle object storage, intelligent data layout, management of the metadata associated with objects it stores, and security. PanFS supports the POSIX file system interface, permissions, and ACLs. Caching is handled at multiple locations in PanFS. Caching is performed at the compute nodes and is managed with callbacks from the MDSs. The OBDs have a write data cache for efficient storage and a third cache is used for metadata and security tokens to allow secure commands to access objects on the OSDs.

GPFS

The General Parallel File System (GPFS) from IBM is a shared disk file system. It runs on both AIX and Linux and has been installed on numerous high-performance clusters such as ASCI Purple. In GPFS, compute nodes connect to file system nodes. The file system nodes are connected to shared disks through a switching fabric (such as fibre channel or iSCSI). The GPFS architecture uses distributed locking to guarantee POSIX semantics. Locks are acquired on a byte-range granularity (limited to the smallest granularity of a disk sector). A *data shipping* mode is used for fine-grain sharing for applications, where GPFS forwards read/write operations originating from other nodes to nodes responsible for a particular data block. Data shipping is mainly used in the MPI-IO library optimized for GPFS [PTH+01].

FusionFS

IBRIX, founded in 2000, has developed a commercial parallel file system called FusionFS. It was designed to have a variety of high-performance I/O needs in scientific computing and commercial spaces. Some of its customers include NCSA, the Texas Advanced Computing Center at the University of Austin at Texas, Purdue University, and Electromagnetic Geoservices.

FusionFS is a file system which is a collection of *segments*. Segments are simply a repository for files and directories with no implicit namespace relationships (for example, not necessarily a directory tree). Segments can be variable sizes and not necessarily the same size. In order to get parallel I/O access, files can be spread over a group of segments. Segments are managed by segment servers in FusionFS, where a segment server may "own" one or more segments. Segment servers maintain the metadata and lock the files stored in their segments. Since the file system is composed of segments, additional segments may be added for increasing capacity without adding more servers. Segment servers can be configured to handle failover responsibilities, where multiple segment servers have access to shared storage. A standby segment server would automatically take control of another server's segments if it were to fail. Segments may be taken offline for maintenance without disturbing the rest of the file system.

PVFS

The Parallel Virtual File System 1 (PVFS1) is a parallel file system for Linux clusters developed at Clemson University. It has been completely redesigned as PVFS2, a joint project between ANL and Clemson University. While PVFS1 was a research parallel file system, PVFS2 was designed as a production parallel file system made easy for adding/removing research modules. A typical PVFS2 system is composed of server processes which can handle metadata and/or file data responsibilities.

PVFS2 features several major distinctions over PVFS1. First of all, it has a modular

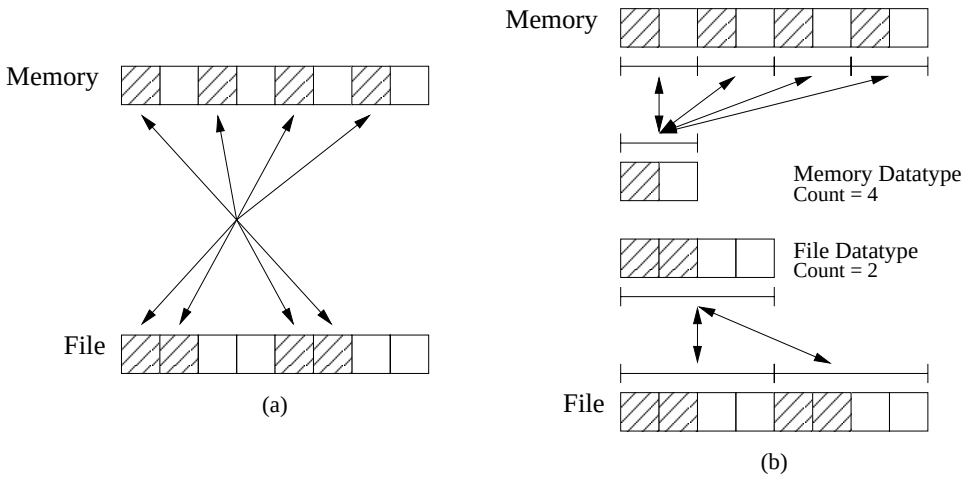

FIGURE 1.9: (a) Example list I/O request. All offsets and lengths are specified for both memory and file using offset-length pairs. (b) Example datatype I/O request. Memory and file datatypes are built for expressing structured I/O patterns and passed along with an I/O request.

design which makes it easy to change the storage subsystem or network interface. Clients and servers are stateless, allowing the file system to cleanly progress if a connected client crashes. Most important to this chapter, PVFS2 has strong built-in support for noncontiguous I/O and an optimized MPI-IO interface. PVFS2 also uses the concept of objects which are referred to by handles in the file system. Data objects in PVFS2 are stored in the servers with metadata information about the group of objects that make up a file as well as attributes local to a particular object. It is best to access PVFS2 through the MPI-IO interface, but a kernel driver provides access to PVFS2 though the typical UNIX I/O interface.

1.4.2 Noncontiguous I/O Methods for Parallel File Systems

Numerous scientific simulations compute on large, structured multi-dimensional datasets which must be stored at regular time steps. Data storage is necessary for visualization, snapshots, checkpointing, out-of-core computation, post processing [NSLD99], and numerous other reasons. Many studies have shown that the noncontiguous I/O access patterns evident in applications as IPARS [IPA] and FLASH [FOR+00] are common to most scientific applications [BW95, CACR95]. This section begins by describing the important noncontiguous I/O methods that can be leveraged through MPI-IO which require specific file system support (list I/O and datatype I/O). POSIX I/O and two phase I/O were discussed in depth in Section 1.3.2. In this section, noncontiguous I/O methods are described and compared.

List I/O

The list I/O interface is an enhanced file system interface designed to support noncontiguous accesses and is illustrated in Figure 1.9a. The list I/O interface describes accesses that are both noncontiguous in memory and file in a single I/O request by using offset-length pairs.

Using the list I/O interface, an MPI-IO implementation can flatten the memory and file datatypes (convert them into lists of contiguous regions) and then describe an MPI-IO operation with a single list I/O request. Given an efficient implementation of this interface in a file system, list I/O improves noncontiguous I/O performance by significantly reducing the number of I/O requests needed to service a noncontiguous I/O access pattern. Previous work [CCC+03] describes an implementation of list I/O in PVFS and support for list I/O under the ROMIO MPI-IO implementation. Drawbacks of list I/O are the creation and processing of these large lists and the transmission of the file offset-length pairs from client to server in the parallel file system. Additionally, list I/O request sizes should be limited when going over the network; only a fixed number of file regions can be described in one request. So while list I/O significantly reduces the number of I/O operations (in [CCC+03], by a factor of 64), a linear relationship still exists between the number of noncontiguous regions and the number of actual list I/O requests (within the file system layer). In the rest of this section, this maximum number of offset-length pairs allowed per list I/O request is addressed as *ol-max*.

Using POSIX I/O for noncontiguous I/O access patterns will often generate the same number of I/O requests as noncontiguous file regions. In that case, previous results have shown that list I/O performance runs parallel to the POSIX I/O bandwidth curves and shifted upward due to a constant reduction of total I/O requests by a factor of *ol-max*. List I/O is an important addition to the optimizations available in MPI-IO. It is most effective when the I/O access pattern is noncontiguous and irregular, since datatype I/O is more efficient for structured data access.

Datatype I/O

While list I/O provides a way for noncontiguous access patterns to be described in a single I/O request, it uses offset-length pairs. For structured access patterns, more concise solutions exist for describing the memory and file regions. Hence, datatype I/O (Figure 1.9b) borrows from the derived datatype concept used in both message passing and I/O for MPI applications. MPI derived datatype constructors allow for concise descriptions of the regular, noncontiguous data patterns seen in many scientific applications (such as extracting a row from a two-dimensional dataset). The datatype I/O interface replaces the lists of I/O regions in the list I/O interface with an address, count, and datatype for memory, and a displacement, datatype, and offset into the datatype for file. These parameters correspond directly to the address, count, datatype, and offset into the file view passed into an MPI-IO call and the displacement and file view datatype previously defined for the file. While the datatype I/O interface could be directly used by programmers, it is best used as an optimization under the MPI-IO interface. The file system must provide its own support for understanding and handling datatypes. A datatype I/O implementation in PVFS1 [CCL+03] demonstrates its usefulness in several I/O benchmarks.

Since datatype I/O can convert a MPI-IO operation directly into a file system request with a one-to-one correspondence, datatype I/O greatly reduces the amount of I/O requests necessary to service a structured noncontiguous request when compared to the other noncontiguous access methods. Datatype I/O is unique with respect to other methods in that increasing the number of noncontiguous regions that are regularly occurring does not require additional I/O access pattern description data traffic over the network. List I/O, for example, would have to pass more file offset-length pairs in such a case. When presented with an access pattern of no regularity, datatype I/O regresses into list I/O behavior.

1.4.3 I/O Suggestions for Application Developers

Application developers often create datasets using I/O calls in a simple to program manner. While convenient, the logical layout of a dataset in a file can have a significant impact on I/O performance. When creating I/O access patterns, application developers should consider three major I/O access pattern characteristics that seriously affect I/O performance. These suggestions focus on structured, interleaved, and noncontiguous I/O access, all of which are common for scientific computing (as mentioned earlier in Section 1.4.2). The following discussion addresses the effect of changing each parameter while holding the others constant.

- **Region Count** - Changing the region count (whether in memory or file) will cause some I/O methods to increase the amount of data sent from the clients to the I/O system over the network. For example, increasing the region count when using POSIX I/O will increase the number of I/O requests necessary to service a noncontiguous I/O call. Datatype I/O may be less affected by this parameter since changing the region count does not change the size of the access pattern representation in structured data access. List I/O could be affected by the region count since it can only handle so many offset-length pairs before splitting into multiple list I/O requests. Depending on the access pattern, two phase I/O may also be less affected by the region count, since aggregators only make large contiguous I/O calls.
- **Region Size** - Memory region size should not make a significant difference in overall performance if all other factors are constant. POSIX I/O and list I/O could be affected since smaller region sizes could create more noncontiguous regions, therefore requiring more I/O requests to service. File region size makes a large performance difference since hard disk drive technology provides better bandwidth to larger I/O operations. Since two phase I/O already uses large I/O operations, it will be less affected by file region size. The other I/O methods will see better bandwidth (up to the hard drive disk bandwidth limit) for larger file region sizes.
- **Region Spacing** - Memory region spacing should not make an impact on performance. However, file regions spacing changes the disk seek time. Even though this section considers region spacing as the logical distance to the next region, there is a some correlation with respect to actual disk distance due to the physical data layout many file systems choose. If the distance between file regions is small, two phase I/O will improve performance due to internal data sieving. Also, when the file region spacing small enough to fit multiple regions into a file system block, file system block operations may help with caching. Spacing between regions is usually different in memory and in file due to the interleaved data operation that is commonly seen in scientific datasets that are accessed by multiple processes. For example, in the FLASH code [FOR+00], the memory structure of the block is different that the file structure, since the file dataset structure takes into account multiple processes.

An I/O benchmark, *Noncontiguous I/O Test* (NCIO), was designed in [CCL+06] for studying I/O performance using various I/O methods, I/O characteristics, and noncontiguous I/O cases. This work validated many of the I/O suggestions listed here. Table 1.5 and Table 1.6 show a summary of how I/O parameters affect memory and file descriptions. If application developers understand how I/O access patterns affect overall performance, they can create optimized I/O access patterns which will both attain good performance as well as suit the needs of the application.

TABLE 1.5 The effect of changing the memory description parameters of an I/O access pattern assuming that all others stay the same.

I/O Method	Increase region count from (1 to c)	Increase region size from (1 to s)	Increase region spacing from (1 to p)
POSIX	increase I/O ops from 1 to c	no change	no change
List	increase I/O ops from 1 to $c/ol\text{-}max$	no change	no change
Datatype	increment datatype count from 1 to c	no change	no change
Data sieving	minor increase of local memory movement	surpassing buffer size requires more I/O ops	surpassing buffer size requires more I/O ops
Two phase	minor increase of memory movement across network	surpassing aggregate buffer requires more I/O ops	surpassing aggregate buffer requires more I/O ops

TABLE 1.6 The effect of changing the file description parameters of an I/O access pattern assuming that all others stay the same.

I/O Method	Increase region count from (1 to c)	Increase region size from (1 to s)	Increase region spacing from (1 to p)
POSIX	increase I/O ops from 1 to c	improves disk bandwidth	increases disk seek time
List	increase I/O ops from 1 to $c/ol\text{-}max$	improves disk bandwidth	increases disk seek time
Datatype	increment datatype count from 1 to c	improves disk bandwidth	increase disk seek time
Data sieving	minor increase of local memory movement	lessens buffered I/O advantages & surpassing buffer size requires more I/O ops	lessens buffered I/O advantages & surpassing buffer requires more I/O ops
Two phase	minor increase of memory movement across network	lessens buffered I/O advantages & surpassing aggregate buffer size requires more I/O ops	lessens buffered I/O advantages & surpassing aggregate buffer size requires more I/O ops

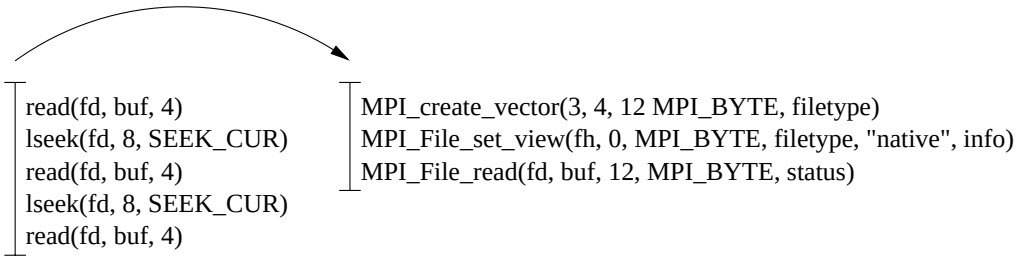

FIGURE 1.10: Example code conversion from the POSIX interface to the MPI-IO interface.

Possible I/O Improvements

- **All large-scale scientific applications should use the MPI-IO interface (either natively or through higher level I/O libraries).** MPI-IO is a portable parallel I/O interface that provides more performance and functionality over the POSIX I/O interface. Whether using MPI-IO directly or through a higher level I/O library which uses MPI-IO (such as PnetCDF or HDF5), ap-

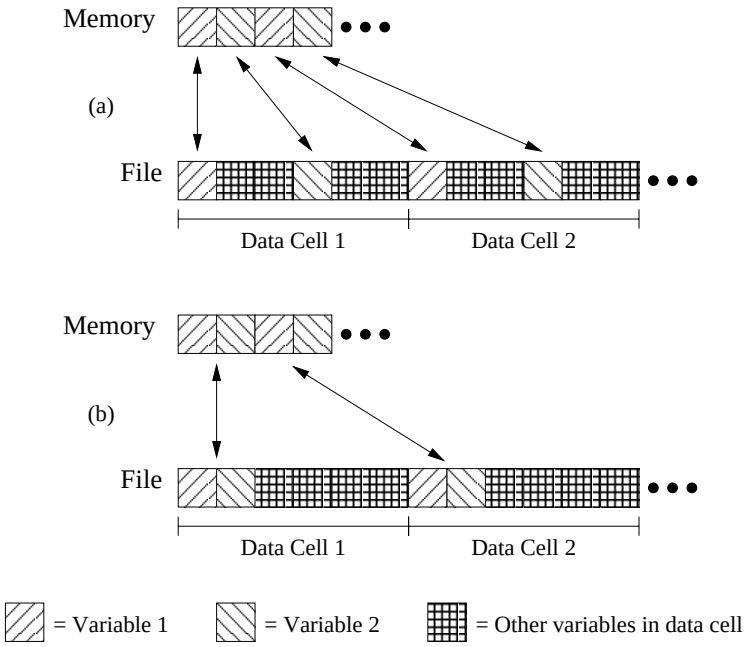

FIGURE 1.11: (a) Original layout of variables in data cells. (b) Reorganization of data to combine file regions during write operations increases I/O bandwidth.

plications can use numerous I/O optimizations such as collective I/O and data sieving I/O. MPI-IO provides a rich interface to build descriptive access patterns for noncontiguous I/O access. Most programmers will benefit from the relaxed semantics in MPI-IO when compared to the POSIX I/O interface. If a programmer chooses to use a particular file system's custom I/O interface (i.e. not POSIX or MPI-IO), portability will suffer.

- **Group individual I/O access to make large MPI-IO calls.** Even if an application programmer uses the MPI-IO interface, they need to group their read/write accesses together into larger MPI datatypes and then do a single MPI-IO read/write. Larger MPI-IO calls allow the file system to use optimizations such as data sieving I/O, list I/O and datatype I/O. It also provides the file system with more information about what the application is trying to do, allowing it to take advantage of data locality on the server side. A simple code conversion example in Figure 1.10 changes 3 POSIX read() calls into a single MPI_File_read() call, allowing it to use data sieving I/O, list I/O, or datatype I/O to improve performance.
- **Whenever possible, increase the file region size in an I/O access pattern.** After creating large MPI-IO calls which service noncontiguous I/O access patterns, try to manipulate the I/O access pattern such that the file regions are larger. One way to do this is data reorganization. Figure 1.11 shows how moving variables around in a data cell combined file regions for better performance. While not always possible, if a noncontiguous file access pattern can be made fully contiguous, performance can improve by up to 2 orders of magnitude [CCL+06]. When storing data cells, some programmers write one variable at a

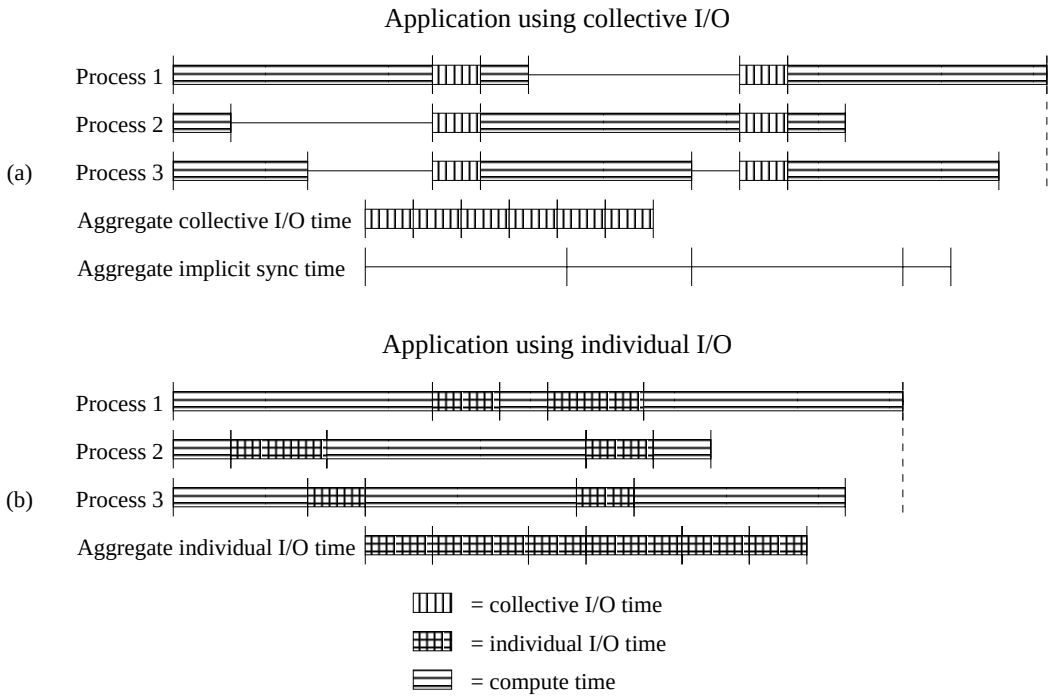

FIGURE 1.12: Cost of collective I/O synchronization. Even if collective I/O (a) can reduce the overall I/O times, individual I/O (b) outperforms it in this case because of no implicit synchronization costs.

time. Making a complex memory datatype to write this data contiguously in file in a single MPI-IO I/O call will be worth the effort.

- **Reduce the file region spacing in an I/O access pattern.** When using data sieving I/O and two phase I/O, this will improve buffered I/O performance by accessing less unused data. POSIX I/O, list I/O, and datatype I/O will suffer less disk seek penalties. Again, a couple of easy ways to do this is to reorganize the data layout or combine multiple I/O calls to make fewer, but larger I/O calls.
- **Consider individual versus collective (two phase I/O).** Two phase I/O provides good performance over the other I/O methods when the file regions are small (bytes or tens of bytes) and nearby since it can make large I/O calls, while the individual I/O methods (excluding data sieving I/O) have to make numerous small I/O accesses and disk seeks. The advantages of larger I/O calls outweigh the cost of passing network data around in that case. Similarly, the file system can process accesses in increasing order across all the clients with two phase I/O. If the clients are using individual I/O methods, the file system must process the interleaved I/O requests one at a time, which might require a lot of disk seeking. However, in many other cases, list I/O and datatype I/O outperform two phase I/O [CCL+06]. More importantly, two phase I/O has an implicit synchronization cost. All processes must synchronize before any I/O can be done. Depending on the application, this synchronization cost can be minimal or dominant. For instance, if the application is doing a checkpoint, since the processes will likely

synchronize after the checkpoint is written, the synchronization cost is minimal. However, if the application is continually processing and writing results in an embarrassingly parallel manner, the implicit synchronization costs of two phase I/O can dominate the overall application running time as shown in Figure 1.12.

- **When using individual I/O methods, choose datatype I/O.** In nearly all cases datatype I/O exceeds the performance of the other individual I/O methods. The biggest advantage of datatype I/O is it can compress the regularity of an I/O access pattern into datatypes, keeping a one-to-one mapping from MPI-IO calls to file system calls. In the worst case (unstructured I/O), datatype I/O breaks down to list I/O which is still much better than POSIX I/O.
- **Do not use data sieving I/O for interleaved writes.** Interleaved writes will have to be processed one at a time by the file system because the read-modify-write behavior in the write case requires concurrency control. Using data sieving I/O for writes is only supported by file systems which have concurrency control. Data sieving I/O is much more competitive with the other I/O methods when performing reads, but should still be used in limited cases.
- **Generally, there is no need to reorganize the noncontiguous memory description if file description is noncontiguous.** Some programmers might be tempted to copy noncontiguous memory data into a contiguous buffer before doing I/O, but recent results suggest that it will not make any difference in performance [CCL+06]. It would most likely just incur additional programming complexity and memory overhead.

References

References

[ACTC06] Peter Aarestad, Avery Ching, George Thiruvathukal, and Alok Choudhary. Scalable approaches for supporting MPI-IO atomicity. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[BW95] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[CACR95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[CCC+03] Avery Ching, Alok Choudhary, Kenin Coloma, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[CCL+03] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Efficient structured data access in parallel file systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, December 2003.

[CCL+04] Kenin Coloma, Alok Choudhary, Wei Keng Liao, Lee Ward, Eric Russell, and Neil Pundit. Scalable high-level caching for parallel I/O. In *Proceedings of the*

IEEE International Parallel and Distributed Processing Symposium, April 2004.

[CCL+05] Kenin Coloma, Alok Choudhary, Wei Keng Liao, Lee Ward, and Sonja Tideman. DACche: Direct access cache system for parallel I/O. In *Proceedings of the International Supercomputer Conference*, June 2005.

[CCL+06] Avery Ching, Alok Choudhary, Wei Keng Liao, Lee Ward, and Neil Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings of the International Conference of Parallel Processing*, April 2006.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[EKHM93] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Portable parallel file system detailed design. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[FOR+00] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[GSC+95] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[HDF] HDF5 home page. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[IBR] IBRIX FusionFS. `http://www.ibrix.com/`.

[IPA] IPARS: integrated parallel accurate reservoir simulation. `http://www.ticam.utexas.edu/CSM/ACTI/ipars.html`.

[IT01] Florin Isaila and Walter Tichy. Clusterfile: A flexible physical layout parallel file system. In *Proceedings of the IEEE International Conference on Cluster Computing*, October 2001.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kot97] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[LCC+05a] Wei Keng Liao, Alok Choudhary, Kenin Coloma, Lee Ward, and Sonja Tideman. Cooperative write-behind data buffering for MPI I/O. In *Proceedings of the Euro PVM/MPI Conference*, September 2005.

[LCC+05b] Wei Keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russell, and Sonja Tideman. Collective caching: Application-aware client-side file caching. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, July 2005.

[LGR+05] Robert Latham, William Gropp, Robert Ross, Rajeev Thakur, and Brian Toonen. Implementing MPI-IO atomic mode without file system support. In

Proceedings of the IEEE Conference on Cluster Computing Conference, September 2005.

[LLC+03] Jianwei Li, Wei Keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Sigel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific I/O interface. In *Proceedings of Supercomputing*, November 2003.

[Lus] Lustre. `http://www.lustre.org`.

[Mes] Message passing interface forum. `http://www.mpi-forum.org`.

[MWLY02] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, April 2002.

[NK96] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[NSLD99] Michael L. Norman, John Shalf, Stuart Levy, and Greg Daues. Diving deep: Data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science and Engg.*, 1(4):36–47, 1999.

[Pan] Panasas. `http://www.panasas.com`.

[PTH+01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of Supercomputing*, November 2001.

[RD90] Russ Rew and Glenn Davis. The unidata netcdf: Software for scientific data access. In *Proceedings of the 6th International Conference on Interactive Information and Processing Systems for Meterology, Oceanography and Hydrology*, February 1990.

[ROM] ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, San Jose, CA, January 2002. IBM Almaden Research Center.

[TGL99a] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[TGL99b] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[The] The parallel virtual file system 2 (PVFS2). `http://www.pvfs.org/pvfs2/`.

[VSK+03] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for I/O on clusters. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 96–103. IEEE Computer Society Press, May 2003.

Index