Chapter 13

MPI-IO

Wei-keng Liao

Northwestern University

Rajeev Thakur

Argonne National Laboratory

13.1 Introduction

MPI-IO is a standard, portable interface for parallel file I/O that was defined as part the MPI-2 (Message Passing Interface) Standard in 1997. It can be used either directly by application programmers or by writers of high-level libraries as an interface for portable, high-performance I/O in parallel programs. It has many features specifically designed to efficiently support the I/O needs of parallel scientific applications. MPI-IO is an interface that sits above a parallel file system and below an application or a high-level I/O library, as illustrated in Figure 13.1. Hence it is often referred to as "middleware" for parallel I/O.

MPI-IO is intended as an interface for multiple processes of a parallel (MPI) program that is writing or reading parts of a single common file. For this purpose, an implementation of MPI-IO is typically layered on top of a parallel file system that supports the notion of a single, common file shared

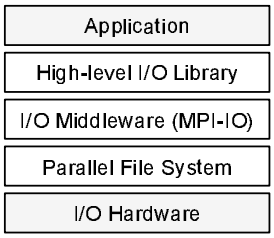


FIGURE 13.1: MPI-IO in the I/O software stack..

by multiple processes. Of course, MPI-IO can also be trivially used for the case where each process reads or writes a separate file.

13.1.1 MPI-IO Background

MPI-IO originated in an effort that began in 1994 at IBM Watson Research Center to investigate the impact of the then new MPI message-passing standard on parallel I/O. A group at IBM wrote an initial paper [7] that explored the analogy between MPI message passing and I/O. Roughly speaking, one can consider writing to file as sending a message, and reading from a file as receiving a message. This paper was the starting point of MPI-IO in that it was the first attempt to exploit this analogy by applying the (then relatively new) MPI concepts for message passing to the realm of parallel I/O.

The idea of using message-passing concepts in an I/O library appeared successful, and the effort was expanded into a collaboration with parallel I/O researchers from NASA Ames Research Center. The resulting specification appeared in an IBM technical report [1]. At this point a large email discussion group was formed, with participation from a wide variety of institutions. This group, calling itself the MPI-IO Committee, pushed the idea further in a series of proposals, culminating in the version 0.5 release of the MPI Standard [13].

During this time, the MPI Forum had resumed meeting to address a number of topics that had been deliberately left out of the original MPI Standard, including parallel I/O. The MPI-IO Committee eventually merged with the MPI Forum, and, from the summer of 1996, the MPI-IO design activities took place in the context of the MPI Forum meetings. The MPI Forum used the latest version of the existing MPI-IO specification (v 0.5) [13] as a starting point for the I/O chapter in MPI-2. The I/O chapter evolved over many meetings of the Forum and was released in its final form along with the rest of MPI-2 in July 1997 [6]. MPI-IO now refers to this I/O chapter in the MPI Standard.

13.1.2 Parallel I/O in Practice

There are three ways to do I/O from a parallel program perspective. In the first method, each process accesses a separate file, which is also known as

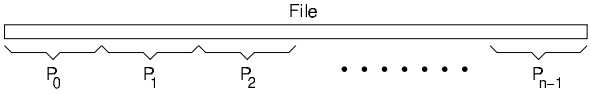
one-file-per-process I/O method. This method is easy to program and requires no MPI communication. However, it results in a large number of files that are hard to manage. Another drawback is that the same number of processes must be used to read the files as the parallel program that created the files. Otherwise, additional effort is required, such as determining the mapping of files to processes and the logical data layout from files to memory (most likely noncontiguous in file space). Due to these concerns, it is desirable to store data in a canonical order in files such that the mapping is independent of the number of processes. The second method of performing I/O addresses this limitation, and it involves funneling all I/O through one process of the program. All I/O requests are forwarded to that process and carried out there. Obviously, this approach can result in poor performance for large jobs due to communication congestion and limited memory space available in one process.

The third method, parallel I/O to a shared file, overcomes the limitations of both approaches. In this method, all processes open a common file and read or write different parts of the same file simultaneously. This approach maintains the logical view of a single file and can also result in high performance given sufficient I/O hardware, a parallel file system, and an efficient MPI-IO implementation. The following sections describe how MPI-IO can be used to perform this form of parallel I/O efficiently.

13.2 Using MPI for Simple I/O

Based on the third method described in Section 13.1.2, high performance parallel I/O can be achieved by enabling all processes to read or write to different parts of a single shared file. Figure 13.2 shows a simple example of such a program. Overall, this code is not much different from how one would do it using regular POSIX I/O. It simply uses the MPI-IO equivalents of POSIX `open`, `lseek`, `read`, and `close` functions.

`MPI_File_open` takes an MPI communicator as the first argument, which represents the group of processes that will access the file. The second argument is the file name. MPI does not specify the format of the file name; implementations are free to specify the format. For example, in some cases, the user may need to prefix the file name with an implementation-specified string (such as `nfs:`) to specify the type of file system on which the file is located. In many cases, implementations may be able to determine the type of file system automatically, without the prefix. The third argument indicates the mode of access, in this case read only. The fourth argument provides users with a way to pass "hints" to the file system that may improve performance. In this simple example, the default set of hints are used (`MPI_INFO_NULL`); Section 13.4 will describe how to pass hints. The file handle is returned as the last argument. It is used in all future accesses to the file. `MPI_File_open`



(a)

```
1.  MPI_File fh;
2.  MPI_Status status;

3.  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4.  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

5.  bufsize = FILESIZE / nprocs;
6.  nints = bufsize / sizeof(int);

7.  MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
8.  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
9.  MPI_File_read(fh, buf, nints, MPI_INT, &status);
10. MPI_File_close(&fh);
```

(b)

FIGURE 13.2: (a) A simple example in which each process of a parallel program needs to access a separate portion of a common file. (b) Sample MPI-IO code used to perform the I/O.

is a *collective* function: all processes in the communicator passed as the first argument must call the function.

After the file is successfully opened, each process can independently access and read its portion of data from the file. For this purpose, each process seeks to the right offset in the file by calling `MPI_File_seek`. Then, each process reads the amount of data it needs by calling `MPI_File_read`. Each process reads the number of integers, `nints`, into the buffer, `buf`, in its local memory. The `status` object is the same as in an `MPI_Recv` function; it can be used to query the amount of data actually read. Finally, all processes call `MPI_File_close` to close the file.

13.2.1 Three Ways of File Access

MPI supports three ways of specifying the location from which data is accessed in a file. The first method is by using *individual file pointers*, as in the above example. In this case, each process maintains its own file pointer, independent of other processes. The file pointer can be moved to a specific offset in the file by calling `MPI_File_seek`. The following call to `MPI_File_read` or `MPI_File_write` will access data starting from the current location of the individual file pointer. The file pointer will be incremented by the amount of data read or written by the read/write call.

An alternate way is by using *explicit offsets*. No seek function is needed in this case. Instead, the user calls the functions `MPI_File_read_at` or `MPI_File_write_at`. These functions take the file offset as an argument. The individual file pointer is not affected by reads or writes using these functions.

Using explicit offsets is also a thread-safe way of accessing the file, since there are no separate seek and read/write functions.

MPI-IO also supports a third way that involves using a *shared file pointer*. The shared file pointer is a common file pointer shared by all processes in the communicator passed to the file open function. This file pointer can be moved by calling `MPI_File_seek_shared`. The corresponding read/write functions are `MPI_File_read_shared` and `MPI_File_write_shared`. This method is useful for writing log files, for example. However, maintaining a shared file pointer involves some overhead for the implementation. Hence, for performance reasons, the use of these functions is generally discouraged.

13.2.2 Blocking and Nonblocking I/O

MPI-IO supports both blocking and nonblocking I/O functions. The read-/write functions mentioned above are all blocking functions, which block until the specified operation is completed. Each of these functions also has nonblocking variants: `MPI_File_iread`, `MPI_File_iwrite`, `MPI_File_iread_at`, `MPI_File_iwrite_at`, `MPI_File_iread_shared`, and `MPI_File_iwrite_shared`. They return an `MPI_Request` object immediately after the call, similar to MPI nonblocking communication functions. The user must call `MPI_Test`, `MPI_Wait`, or their variants to test or wait for completion of these operations. The nonblocking I/O functions offer the potential for overlapping I/O and computation or communication in the program.

13.3 File Access with User Intent

Besides the POSIX-equivalent basic I/O functions, MPI-IO contains additional functions that can better convey the user's I/O intent. In our terminology, a user's I/O intent refers to the user's expectation on how the I/O operation should be carried out, and a user's I/O requirement refers to the end result. Consider an example that describes and distinguishes between a user's I/O intent and requirement. Figure 13.3 shows a 5×8 two-dimensional integer array that is partitioned among four processes in a block-block pattern. Each of process ranks 0 and 1 is assigned a subarray of size 3×4. Each of process ranks 2 and 3 is assigned a subarray of size 2×4. (The use of a small array and small number of processes here is only for explanation purposes. In practice, MPI-IO is used for large datasets and large system sizes.) The 2D array can be considered as a representation of the problem domain to a parallel application, and the subarrays represent the sub-domains distributed among the MPI processes. It is assumed that the user's intent is to write the entire 2D array to a file in parallel and the data layout in the file follows the array's canonical order. Such I/O operations often occur during an application's checkpoint.

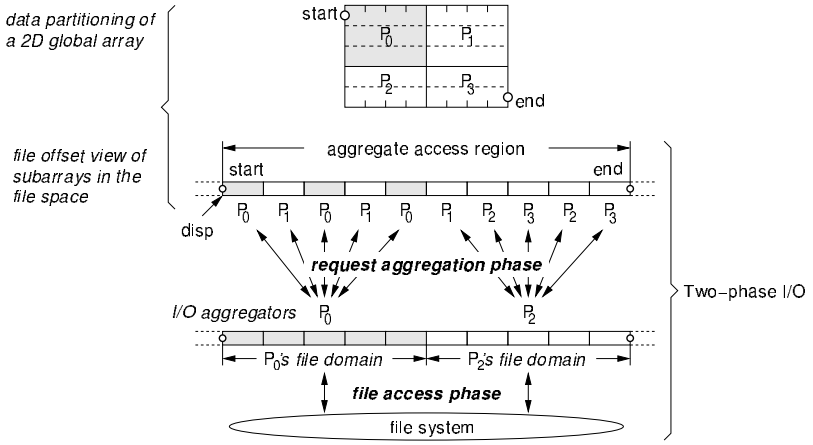


FIGURE 13.3: A 5×8 2D array partitioned among 4 MPI processes in a block-block pattern and its data layout in the file. The data layout in the file follows the array's canonical order. The bottom part describes the two-phase I/O operation carried out in MPI collective I/O.

At each checkpoint, the intention considers writing the 2D array as a single request. The outcome, and hence the user requirement, is that the 2D array is saved in the file starting from the given file offset, denoted as `disp`, and occupies a contiguous space of size equal to $5 \times 8 = 40$ integers. This section continues to describe three ways of using MPI-IO functions to carry out such I/O operations and discuss how they differ in their way of expressing the user intent.

13.3.1 Independent I/O

MPI-IO functions consist of two types: independent and collective. The MPI-IO read/write functions discussed so far belong to the independent I/O category, and their use is very similar to that of POSIX I/O functions. As for collective I/O functions, their appearance and syntax look the same as the independent ones, except that the collective functions have a suffix, `_all`, added to their names. The name "independent" implies the functions can be called by a process independently from another process. There is no restriction on the number of calls a process can make, and they need not match with calls on other processes. Collective functions, on the other hand, require the participation of all processes that collectively open the shared file. The participant processes are identified by the MPI communicator passed to `MPI_File_open`.

For example, Figure 13.4(a) shows an MPI code fragment that uses `MPI_File_write_at`, an independent I/O function, to write the 2D array in parallel. In this case, the loop at line 2 runs N iterations where N is equal to 3 for process ranks 0 and 1, and 2 for ranks 2 and 3. From this exam-

```
1.  offset = disp + (rank / 2) * 3 * 8 + (rank % 2) * 4
2.  for (i=0; i<N; i++, offset+=8)
3.      MPI_File_write_at(fh, offset, buf, 4, MPI_INT, &status);
```

(a) Write the 2D array using MPI independent I/O with explicit offsets.

```
1.  int   gsizes[2] = {5, 8};  /* global array size */
2.  int subsizes[2] = {N, 4};  /*  local array size */
3.  int starts[2];         /* starting file offsets */
4.  starts[0] = (rank / 2) * (gsizes[0] / 2 + 1);
5.  starts[1] = (rank % 2) * (gsizes[1] / 2);
6.  MPI_Type_create_subarray(2, gsizes, subsizes, starts,
7.                           MPI_ORDER_C, MPI_INT, &ftype);
8.  MPI_Type_commit(&ftype);
```

(b) Create an MPI derived data type that maps local subarray to global array.

```
1.  MPI_File_set_view(fh, disp, MPI_INT, ftype, "native", info);
2.  MPI_File_write(fh, buf, N*4, MPI_INT, &status);
```

(c) Write the 2D array using MPI independent I/O with file view.

```
1.  MPI_File_set_view(fh, disp, MPI_INT, ftype, "native", info);
2.  MPI_File_write_all(fh, buf, N*4, MPI_INT, &status);
```

(d) Write the 2D array using MPI collective I/O.

FIGURE 13.4: MPI program fragments to write the 2D array in parallel as illustrated in Figure 13.3.

ple, using independent functions allows MPI processes to make an unequal number of calls. Each call to `MPI_File_write_at` at line 3 is equivalent to a call to POSIX `lseek` with the given file offset, followed by a `write` with the same request amount. From an MPI-IO perspective, the program expresses the following I/O intent. At first, the loop at line 2 asks the MPI-IO library to handle the requests one after another, in that exact order. Secondly, the use of independent functions tells MPI-IO that requests from one process can arrive independently from other processes and expects MPI-IO to protect the data consistency for each individual request. Obviously, the above interpretation is not exactly the same as user's intent, in which the order does not matter and the consistency is for the whole 2D array. In order to prevent such misunderstanding, MPI-IO provides a feature named *file view* to let users better convey their I/O intent.

13.3.2 MPI File View

An MPI file view defines the portion of a file that is "visible" to a process. A process can only read and write the data that is located in its file view. When a file is first opened, the entire file is visible to the process. A process's file view can be changed with the function `MPI_File_set_view`. When using individual file pointers, a process's file view can be different from others. When using the shared file pointer, all processes must define the same view. A file view can

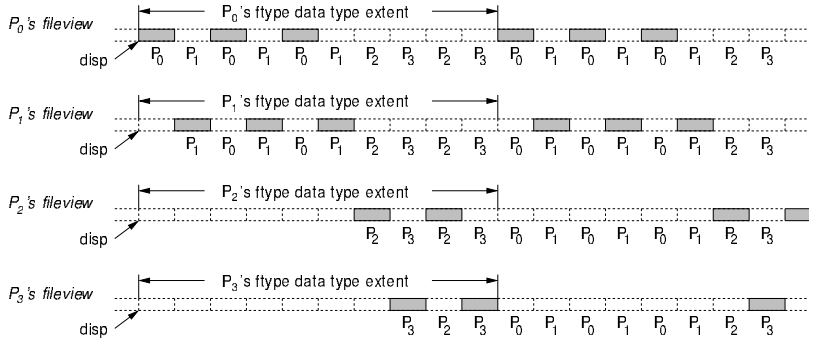


FIGURE 13.5: Individual process's file views for the 2D array example in Figure 13.3. The shaded area indicates the file portions visible to a process. A file view is constructed by repeatedly applying the filetype's extent, starting from the displacement argument, `disp`.

consist of multiple noncontiguous regions in the file. When this happens, the noncontiguous regions are logically stitched together into a contiguous region, which is read or written as if the view is a contiguous byte stream.

MPI datatypes, both basic and derived, are used to set file views. File views are specified by a triplet: *displacement*, *etype*, and *filetype*. The displacement is the number of bytes to be skipped from the beginning of the file. The etype (elementary datatype) is the smallest unit for file access and positioning, which can be an MPI basic or derived datatype. The file offset argument in all MPI-IO functions is specified in terms of the number of etypes. The filetype is an MPI basic or derived datatype that specifies the portions of the file that are "visible" to a process. The filetype must be either the same as the etype or a derived datatype constructed out of the etype. A process's file view starts at the displacement, followed by repeated copies of the filetype. When a call to `MPI_File_open` returns, the default file view sets the displacement to 0 and both etype and filetype to `MPI_BYTE`. The example programs shown in Figures 13.2 and 13.4(a) use the default file view.

The program fragment shown in Figure 13.4(b) constructs the derived datatype, `ftype`, through a call to `MPI_Type_create_subarray` to describe the file view for each of the four processes in the 2D array example. The arguments `gsizes` and `subsizes` are the sizes of global array and local subarray, respectively. The starting array offsets relative to the global array are set in the argument `starts`. Once the newly created filetype, `ftype`, is committed, it can be used in `MPI_File_set_view` to change a process's file view. This example uses `disp` for the file displacement, `MPI_INT` for etype, "native" for data representation, and `info` for MPI-IO hints (Section 13.4).

Figure 13.5 illustrates the file views of the four processes in this example. The file portions visible to individual processes are marked as shaded areas. Note that the filetype extents specified by `ftype` across four processes all

point to the same file location. The same filetype and its extent are applied recursively to the file space, and all visible portions together comprise a process's file view. The program shown in Figure 13.4(c) sets file views and calls an independent write function to write the 2D array in parallel. The write amount is of size $(N \times 4)$ integers, equal to the entire subarray size in each process. Note that the amount requested in an MPI-IO function call need not be the same as the size of the filetype.

From this example, by defining a process's file view, writing the entire subarray can be accomplished by making just one MPI-IO function call. This program fragment also tells a different user I/O intent from the one expressed in Figure 13.4(a). MPI-IO interprets the user's intent as to write the whole subarray as a single request. The restriction of write order for the individual subarray rows as intended by Figure 13.4(a) is eliminated. Instead of protecting data consistency for each individual row, the MPI-IO library enforces the consistency for the entire subarray. Given such interpretation, an MPI-IO implementation has more freedom to adopt certain optimizations to improve performance. One of the well-known optimizations for such I/O requests is called "data sieving" [12]. Since most of the modern file systems do not perform well for a large number of small, noncontiguous requests, data sieving is an I/O strategy that makes large I/O requests to the file system and extracts, in memory, the data that is actually needed. Data sieving has been incorporated into ROMIO, a very popular MPI-IO implementation developed at Argonne National Laboratory [10]. When applied to the discussed example, data sieving will first read a contiguous file segment that is large enough to cover a process's entire file view, the whole subarray, into a temporary buffer and then copy the requested data to the user's buffer. As a result, this read-modify-write strategy reduces the number of I/O requests to the file system. Significant performance improvement has been observed on noncontiguous I/O requests when data sieving is enabled.

13.3.3 Collective I/O

The MPI file view feature gets us one step closer to meet the user I/O intent for this example, which is to write a global 2D array in parallel from four processes as a single I/O operation. However, there are still four independent I/O requests, one from each process. Hence, the MPI-IO library must protect the data integrity for the four requests individually. When data sieving is enabled, the entire file region involved in a single read-modify-write operation must be protected from another. A common solution to provide such protection is to use an advisory file locking mechanism to guarantee the exclusive access for a file region from concurrent access. In this example, the region to be locked by process rank 0's request overlaps with rank 1's lock region. Similarly, process rank 1's lock region overlaps with the one requested by rank 2. As a result, only half of the four processes are actively writing their data to the file system while the other half are waiting for their lock requests to

be granted. Thus, the achievable performance is significantly limited by the conflicted file locks.

To overcome such problems, MPI collective I/O functions can convey the exact user intent in this example that considers the parallel write of the whole global array as a single request. The program fragment using a collective write function is shown in Figure 13.4(d). It appears almost the same as the independent case, except the name of the write function. MPI collective I/O requires the participation of all processes that open the shared file. This requirement provides a collective I/O implementation an opportunity to exchange access information and reorganize I/O requests among the processes. Several process-collaboration strategies have been proposed, such as two-phase I/O [2], disk directed I/O [4], and server-directed I/O [8].

Two-phase I/O is a representative collaborative I/O technique that runs in the user space [11]. It exchanges data among processes, so that the rearranged requests can be processed by the underlying file system with the best performance. Two-phase I/O conceptually consists of a *request aggregation phase* (or referred as the communication phase) and a *file access phase* (or simply the I/O phase). In the request aggregation phase, a subset of MPI processes is picked as I/O aggregators that act as I/O proxies for the rest of the processes. The aggregate file access region requested by all processes is divided among the aggregators into non-overlapping sections, called *file domains*. For collective writes, the non-aggregator processes send their requests to the aggregators based on their file domains. In the file access phase, each aggregator commits the aggregated requests to the file system. ROMIO adopts the two-phase I/O strategy for implementing the collective MPI-IO functions.

The bottom part of Figure 13.3 depicts the two-phase I/O operation for the 2D array example. Assuming P_0 and P_2 are chosen as I/O aggregators, the aggregate access region of the collective write operation is evenly divided into two file domains, one for each aggregator. I/O data are redistributed from all four processes to the two aggregators during the request aggregation phase. Specifically, aggregator process rank 0 receives data from both rank 0 and 1, while aggregator rank 2 receives data from rank 1, 2, and 3. In the file access phase, each aggregator aggregates the received data into a single, contiguous request and then makes a write call to the file system.

Recently, there were several optimizations that further improve the performance of collective I/O. Various file domain partitioning methods have been studied that can be adaptively determined based on the file locking policies of the underlying file systems in order to minimize lock conflicts for collective I/O [5]. In Sehrish et al.'s work [9], a pipelined strategy was developed to overlap the two phases in the two-phase I/O method. In this work, large requests are divided into smaller ones, each of size equal to the file stripe size, and redistributed to the I/O aggregators using MPI asynchronous communication in a pipeline fashion so that the asynchronous communication overlaps with the file access phase. In Venkatesan et al.'s study [14], I/O aggregator placement methods are proposed to minimize the communication cost

in the request aggregation phase. Based on the file views, it calculates the non-aggregator to aggregator communication matrix, divides processes with high-volume communication into groups, and maps the process groups to the underlying network topology.

13.4 MPI-IO Hints

MPI-IO hints are a mechanism for users to pass information, such as access patterns, to the implementation so that it can help optimize file access. This is done by setting an `MPI_Info` object and passing it to the functions `MPI_File_open`, `MPI_File_set_view`, or `MPI_File_set_info`. An MPI-IO implementation may choose to ignore all hints, and the program would still be functionally correct. Some of MPI predefined I/O hints are `cb_buffer_size` (size of allowable buffer to be used by collective I/O), `cb_nodes` (maximum number of aggregators), `striping_factor` (number of file system stripes), and `striping_unit` (file stripe size). An implementation may define additional hints to make use of file system specific features. Readers are referred to the ROMIO user guide for the full list of available hints [10].

13.5 Conclusions

This chapter describes several important MPI-IO features to allow users to describe the data access patterns and the intent of their applications. The MPI-IO functions convey such information to the implementation so that better I/O strategies can be used to achieve high performance for parallel applications. MPI-IO has become a building block for several high-level I/O libraries, such as Parallel NetCDF (Chapter 15) and HDF5 (Chapter 16), which are widely used in various scientific communities. There are other MPI-IO features not covered in this chapter, including data consistency control, portable data representation, and split collective I/O. For their detailed information and learning materials, readers are referred to the MPI-2 Standard [6] and MPI tutorial books, such as *Using MPI-2* [3].

Acknowledgment:
This work was supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.