Getting Started on MPI I/O

S–2490–40

Cray wishes to acknowledge the NERSC staff for their valuable insights and recommendations.

Version 4.0 Published December 2009 Initial publication.

Contents

Page

Introduction [1]

This book introduces Cray XT users to MPI I/O. It defines the basic components of the I/O system, explains various I/O techniques and their performance pros and cons, and describes optimization techniques and their use. The intended audience is application developers. Prerequisite knowledge is a familiarity with MPI, the I/O process, and basic Cray XT terminology. A brief introduction to the Parallel HDF5 (Hierarchical Data Format) and NetCDF-4 (Network Common Data Form) I/O libraries is also given. These libraries are layered on top of MPI I/O and make use of many of the MPI I/O features.

For many applications, I/O is a bottleneck that limits scalability. Write operations often do not perform well because an application's processes do not write data to Lustre in an efficient manner, resulting in file contention and reduced parallelism. This book focuses on write operations. Output files tend to be larger than input files, write operations tend to have more performance bottlenecks than reads, and read operations are usually restricted to application startup, whereas writing typically takes place throughout application execution.

The intent of this book is to provide a basic understanding of how write operations work in a Cray XT system, explain the causes of poor write performance, and encourage application developers to modernize I/O techniques through the use of MPI I/O or other libraries layered on top of MPI I/O. Subsequent revisions to this book will address read performance and give usage guidelines based on reader feedback and ongoing research.

Cray XT Parallel I/O [2]

In this chapter, we first describe some basics of parallel I/O on Cray XT systems—basics that apply to any I/O interface used in an MPI application—and then describe the I/O interfaces.

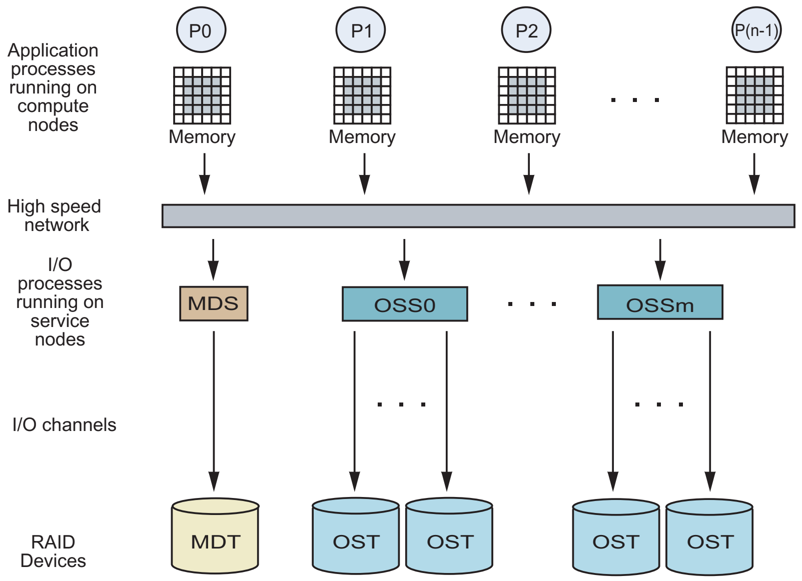
2.1 Overview of the File System Architecture

Figure 1 shows a simplified view of parallel I/O, highlighting only the components appropriate for this book. Parallel I/O is the simultaneous transfer of data by multiple processes between compute node memory and data files on disks. Data transfers to and from disks are managed by Lustre, the underlying parallel file system currently used on Cray XT systems.

A single Lustre file system consists of one metadata server (MDS) and one or more object storage servers (OSSs). The MDS opens and closes files and stores directory and file metadata such as file ownership, timestamps, and access permissions on the metadata target (MDT). Once a file is created, write operations take place directly between compute node processes (P0, P1, ...) and Lustre object storage targets (OSTs), going through the OSSs and bypassing the MDS. For read operations, file data flows from the OSTs to memory. Each OST and MDT maps to a distinct subset of the RAID devices.

Note: The meaning of the term OST can be confusing because it is sometimes stated that there are OSTs "on" an OSS, but this is not the case. The OSS is the physical node. OSTs are mount points on an OSSs. Throughout this book, we refer to OSTs as the physical destination of application data in a file. MDTs are the physical destination of application file metadata.

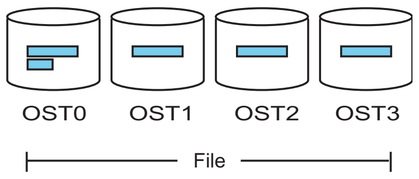
Figure 1. Overview of Parallel I/O



2.2 Physical and Logical Views of a File

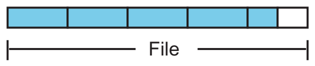
Physically, a file consists of data distributed across OSTs. Figure 2 shows an example of one file that is spread across four OSTs, in five distinct pieces. The next section will describe how the size and distribution of these pieces is determined and why you need to know about this.

Figure 2. A Physical View of a File



Logically, a file is a linear sequence of bytes. Continuing with the example shown in Figure 3, shows the five pieces lined up.

Figure 3. A Logical View of a File



Except for performance reasons, you do not need to know how and where the bytes are arranged physically as long as your application can reliably access the bytes. For performance reasons, the distribution of the file across OSTs does matter because of a process called *file striping*.

2.3 File Striping

The logical view of a file as a linear sequence of bytes is valid regardless of the file system used. However, on Lustre systems it is important to understand one aspect of the physical view of a file—file striping—because the effective use of striping can significantly improve I/O performance.

File striping is the process of physically separating a file's linear sequence of bytes into units called "stripes" so the I/O hardware can simultaneously write or read different stripes.

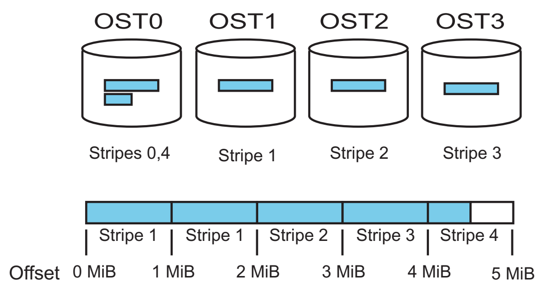
By default, Lustre stripes files across multiple disks (that is, OSTs). The number of OSTs across which a file is written is the stripe count. The number of bytes written on one OST before cycling to the next is the *stripe size*. To improve I/O performance, you may need to set the stripe count or stripe size to a value other than the default (see Setting the Stripe Count and Stripe Size on page 14).

You may also need to know how many OSTs are installed on your system because, in general, the more processes writing in parallel to a file, the better the I/O throughput.

Note: You cannot set the stripe count to greater than the number of OSTs present on your system. Alternatively, at some sites, the number of OSTs present may exceed the Lustre limit for the stripe count for a single file. The current Lustre stripe count limit is 160.

Figure 4 shows the example from the previous section, with the striping information added. In this example, the stripe count is 4 and the stripe size is 1 MiB.

Figure 4. Physical and Logical Views of Striping



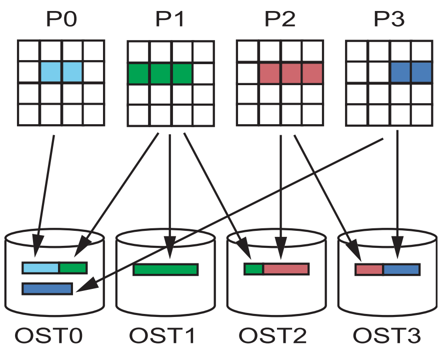
Note: The stripe count of a file is the same as the number of OSTs assigned to the file. The total number of stripes in the file will be more than the stripe count if the number of bytes in the file is greater than the stripe count times the stripe size.

When an application accesses bytes in the file, it does so from the logical view of a starting offset into the file and the number of bytes beyond the starting offset. Application code does not directly reference OSTs or physical I/O blocks.

The previous example shows how a file is striped across the OSTs but does not show I/O proceeding in parallel. Figure 5 shows the physical view of an example of four processes writing in parallel to a single shared file that is striped across four OSTs. The stripe size is 1 MiB. In this example, each process writes one contiguous record to distinct, non-overlapping regions of the file and of different lengths, with no gaps between records. The writes are done as follows:

- P0 writes a 600,000-byte record, starting at offset 0 to OST0.
- P1 writes a 1,800,000-byte record, starting at offset 600,000 to OSTs 0-2.
- P2 writes a 1,200,000-byte record, starting at offset 2,400,000 to OSTs 2 and 3.
- P3 writes a 1,400,000-byte record, starting at offset 3,600,000 to OSTs 3 and 0.

Figure 5. A Physical View of Striping

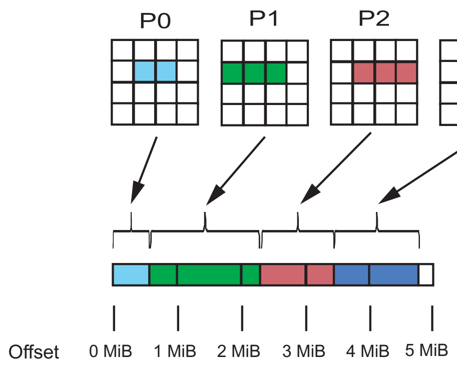


The records from processes 0-3 are each split into pieces by the Lustre software so that each piece gets sent to the appropriate destination OST: OST0 is simultaneously receiving data from processes 0, 1 and 3; OST2 is simultaneously receiving data from processes 1 and 2; and OST3 is simultaneously receiving data from processes 2 and 3.

When there are four OSTs receiving data in parallel, I/O performance can increase significantly—up to four times compared to all processes writing to one OST. Actual performance is limited by the effects of "stripe-aligned records" and "extent lock contention." Because the record lengths are not exact multiples of the stripe size and the starting and ending offsets are not exactly on stripe boundaries, the records in this example are not "stripe-aligned." And because some OSTs are simultaneously receiving data from more than one process, an extent lock must be put on a region of the file when more than one process is trying to write to the same disk block (the smallest writable unit) at a time. The processes are thus contending for the lock, which serializes access to these blocks and thus reduces overall parallel I/O performance.

The logical view of the application is normally presented in terms of file offsets and record lengths. Figure 6 shows the logical view of the example shown in Figure 5. The resulting file is a contiguous sequence of 5,000,000 bytes with no gaps, which is a little less than five full stripes.

Figure 6. A Logical View of Striping



Subsequent chapters will describe I/O interfaces that deliver good I/O performance without the application having to deal with physical layout of data on the OSTs or even physical offsets into the file. These interfaces allow you to think more in terms of the data models of the application and less in terms of I/O details.

2.3.1 Determining the Number of File System OSTs

To determine how many OSTs are on your file system, use the Lustre `lfs df -h` command. (The `-h` option presents the data in easy-to-understand units. This command also lists the MDT and lists the OSTs with indices starting with 0.)

Example 1. Determining the number of file system OSTs

```
% lfs df -h
UUID           bytes   Used Available Use% Mounted on
nid00008_mds_UUID    170.9G   1.2G  159.9G   0% /lus/nid00008[MDT:0]
ost0_UUID         1.0T  472.6G  545.7G  44% /lus/nid00008[OST:0]
ost1_UUID         1.0T  418.4G  599.8G  39% /lus/nid00008[OST:1]
ost2_UUID         1.0T  468.8G  549.5G  43% /lus/nid00008[OST:2]
ost3_UUID         1.0T  431.0G  587.3G  40% /lus/nid00008[OST:3]
ost4_UUID         1.0T  449.0G  569.2G  41% /lus/nid00008[OST:4]
ost5_UUID         1.0T  465.8G  552.5G  43% /lus/nid00008[OST:5]
ost6_UUID         1.0T  475.3G  542.9G  44% /lus/nid00008[OST:6]
ost7_UUID         1.0T  466.0G  552.2G  43% /lus/nid00008[OST:7]

filesystem summary:    8.4T   3.6T   4.4T  42% /lus/nid00008
```

In this example, there are eight OSTs mounted on file system `/lus/nid00008`.

2.3.2 Determining the Stripe Count and Stripe Size

To determine the system default stripe count and stripe size, use the `lfs getstripe` command on a newly created Lustre file system directory.

Example 2. Determining the stripe count and stripe size

```
% cd /lus/nid00008/user123
% mkdir newdir
% lfs getstripe newdir
OBDS:
0: ost0_UUID ACTIVE
1: ost1_UUID ACTIVE
2: ost2_UUID ACTIVE
3: ost3_UUID ACTIVE
4: ost4_UUID ACTIVE
5: ost5_UUID ACTIVE
6: ost6_UUID ACTIVE
7: ost7_UUID ACTIVE
newdir
(Default) stripe_count: 2 stripe_size: 1048576 stripe_offset: 0
```

In this example, the default stripe count is 2 and the default stripe size is 1,048,576 bytes (1 MiB).

Note: The system defaults are set by the system administrator when the file system is configured.

2.3.3 Setting the Stripe Count and Stripe Size

To set the stripe count and stripe size for a file, use the `lfs setstripe` command on the parent directory before the file is created.

Example 3. Setting the stripe count and stripe size

```
% cd /lus/nid00008/user123
% mkdir newdir
% lfs setstripe -c 8 -s 1M newdir
```

The above example sets the default stripe count to 8 and the stripe size to 1 MiB for all files subsequently created in `newdir`. By default, any file created in this directory will inherit the directory's stripe count and stripe size. The `lfs` command can also be used to create a zero length file with a specified stripe count and stripe size, but setting information on the parent directory is the more common option. See the `lfs`(1) man page for more information.

Note: There are other ways to set the stripe count and stripe size using the MPI I/O interface; see MPI I/O Optimization Hints on page 38.

2.3.4 Guidelines for File Striping

The following guidelines for setting the stripe count and stripe size are relevant for files that are many times larger than the stripe size. For much smaller files, the I/O time is probably not a significant part of the total application time.

- For single-shared-file I/O, striping across multiple OSTs usually improves performance significantly. When the number of processes is less than the number of OSTs available, set the stripe count to the number of processes. When the number of processes is greater than or equal to the number of OSTs available, set the stripe count to the number of OSTs available, or to the maximum stripe count allowed by Lustre (currently 160).

  For more information about single-shared-file I/O, see Single Shared File: One Writes on page 22, Single Shared File: All Write on page 23, and Single Shared File: Subset Writes on page 24. For details about I/O performance, see IOR Writes on page 44.

- For one-file-per-process I/O with more processes running than the number of OSTs on the file system, striping across multiple OSTs usually does not help performance because multiple processes will compete for the same set of OSTs. In this case, set the stripe count to 1.

  On the other hand, if the number of processes is less than half the number of OSTs available, setting the stripe count so that most or all of the OSTs are being used by the entire application can help performance.

  For more information about one-file-per-process I/O, see One File per Process: All Write on page 21.

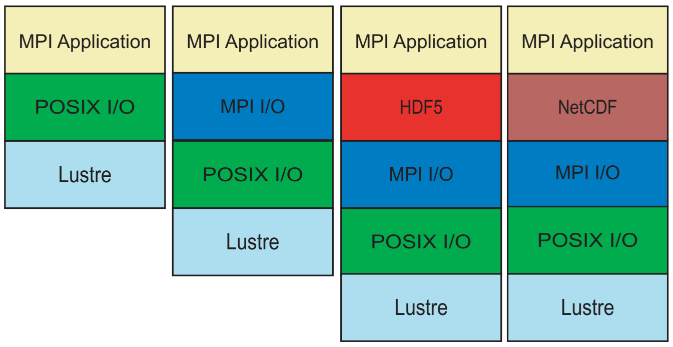
- In general, it is not necessary to change the default stripe size. Empirical evidence indicates that in most cases changing the default stripe size does not improve I/O performance. Throughout this book, we assume a stripe size of 1 MiB.

2.4 I/O Interfaces

There are four software interfaces to the Cray XT I/O system: POSIX I/O, MPI I/O, HDF5 I/O, and NetCDF-4 I/O.

As shown in Figure 7, the interfaces are layered. All MPI applications use one or more of the I/O interfaces and the Lustre parallel file system to transfer data between compute node memory and disk files. Cray recommends that you use one interface throughout an application. At a minimum, use one interface for I/O to and from a single file.

Figure 7. MPI I/O Software Layers



2.4.1 POSIX I/O Interface

A POSIX I/O file is simply a sequence of bytes. You use the POSIX I/O interface to transfer contiguous regions of bytes between the file and memory or to transfer noncontiguous regions of bytes from memory to a file.

MPI applications can perform POSIX I/O directly through the use of POSIX I/O calls or indirectly through C, C++, or Fortran I/O functions that are translated into POSIX calls.

The basic POSIX I/O calls are `read()`, which reads contiguous data from a file to memory, and `write()`, which writes contiguous data from memory to a file. The `readv()` call transfers contiguous data in a file to noncontiguous data in memory. The `writev()` call transfers noncontiguous data in memory to contiguous data in a file.

2.4.1.1 POSIX Benefits

POSIX I/O gives you full, low-level control of I/O operations. It is the standard interface for serial I/O on most systems. However, there is little in the interface that inherently supports parallel I/O. It is your responsibility to manage most aspects of parallel I/O, including the calculation of explicit offsets. POSIX I/O does not support collective access to files (that is, the simultaneous access to data by a group of processes). Therefore, the application must explicitly coordinate collective functions. In addition, if complex data structures are written to a file, you must break noncontiguous data into separate contiguous segments and make separate calls for each segment.

If I/O performance is important for an application using POSIX I/O, you may need to modify the application to move data between processes to accommodate process counts and the specifics of the parallel file system. An application tailored in this manner may not be very portable to environments where the application is run with different process counts and/or different file systems.

2.4.1.2 References

The POSIX I/O interface is documented in the *System Interfaces* volume of *The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition* (http://www.opengroup.org/onlinepubs/009695399/).

2.4.2 MPI I/O Interface

The MPI I/O interface provides two types of I/O calls: independent I/O and collective I/O.

Independent MPI I/O calls are referred to as *independent* because the calls can be made by any subset of the processes participating in I/O, with each process handling its own I/O independently. The basic independent MPI I/O calls are `MPI_File_read()` and `MPI_File_write()`.

These calls are similar to the MPI message passing send and receive calls, as shown in the syntax of the `MPI_File_write()` and `MPI_Send()` calls:

```
int MPI_File_write(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_Send(void *buf, int count,
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Collective MPI I/O calls are referred to as *collective* because I/O calls must be made by all processes participating in a particular I/O sequence. The basic collective calls are `MPI_File_write_all()` and `MPI_File_read_all()`. The only syntactic difference between independent and collective calls is the addition of `all` to the name of collective calls, as shown in the `MPI_File_write()` and `MPI_File_write_all()` calls:

```
int MPI_File_write(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_write_all(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

The `_all` portion of the call indicates that all processes in the group specified by the communicator passed to `MPI_File_open()` will call this function.

An MPI I/O file is an ordered collection of typed data items. You can define data models that are natural to your application through the use of typed data items. The MPI I/O interface provides calls for constructing and committing datatypes. After you commit a datatype, use it in a `MPI_File_set_view()` call.

With MPI I/O, each process has its own view of the file, relieving you of the burden of calculating explicit offsets for each I/O operation. Every process that is sharing the file must call `MPI_File_set_view()` to define its portion of the file. For an example program showing how to use datatypes and `MPI_File_set_view()`, see Example 4. Once the view is set, MPI I/O maintains the file offset for each process.

For both independent I/O and collective I/O, there are individual-file-pointer functions and explicit-offset functions, providing flexibility in programming paradigms.

2.4.2.1 MPI I/O Benefits

MPI I/O provides the following benefits for parallel I/O applications:

- A higher level of data abstraction than POSIX I/O. With MPI-IO, you can define complex data patterns for parallel writes, which allows Lustre and the MPI-IO library to optimize performance.
- Support of data coherence and atomicity
- Optimization of I/O functions

Independent MPI I/O calls are similar to POSIX I/O calls, with two important differences. POSIX I/O calls support only contiguous segments of data in files, but independent MPI I/O calls support derived data types, which can contain noncontiguous data and nonuniform strides.

Many parallel applications lend themselves naturally to collective I/O. When all processes participating in a sequence of parallel computations need to read and/or write data at the same time, they can do so efficiently by using collective MPI I/O calls.

With respect to performance, independent MPI I/O is very similar to POSIX I/O. With collective MPI I/O, however, the MPI I/O layer coordinates the I/O of participating processes, allowing the library to do optimizations that could not be done with independent I/O. MPI I/O can do some optimizations for independent I/O, but all of these and more can also be done for collective MPI I/O, so we will focus on collective MPI I/O, pointing out distinctions where appropriate.

2.4.2.2 References

- The MPI I/O interface is documented in Chapter 9 of the *MPI-2.0 Standard* (Chapter 13 of the 2.2 Standard). See http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html and http://www.mpi-forum.org/.
- *MPI: The Complete Reference* (2-volume set) by Marc Snir, William Gropp, and Bill Nitzberg.
- *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)* by William Gropp, Ewing Lusk, and Anthony Skjellum.
- *Parallel I/O for High Performance Computing* by John M. May.

2.4.3 HDF5 I/O Interface

HDF5 is a platform-independent I/O interface that simplifies the modeling, viewing, and analysis of complex data objects. HDF5 provides a higher level of data abstraction than MPI I/O.

The basic HDF5 I/O calls are `H5Dread()` and `H5Dwrite()`.

2.4.3.1 HDF5 Benefits

Whereas MPI I/O handles one I/O call at a time, HDF5 can treat multiple I/O calls as a logical entity and therefore can do optimizations that MPI I/O cannot or can provide optimization hints to MPI I/O appropriate for the larger context.

HDF5 files are "self-describing," meaning that the files contain information fully describing the typed data items. Self-describing files are portable between different file systems.

2.4.3.2 References

There are no HDF5 man pages, but you can find additional information, including example programs, at:

- The HDF5 home web at http://www.hdfgroup.org/.
- *Introduction to HDF5* at http://www.hdfgroup.org/HDF5/doc/H5.intro.html.
- For descriptions of HDF5 I/O calls, see *HDF5: API Specification Reference Manual* at http://www.hdfgroup.org/HDF5/doc1.6/RM_H5Front.html.

2.4.4 NetCDF-4 I/O Interface

NetCDF-4 is a platform-independent I/O interface that you can use to create, access, and share array-oriented data. NetCDF-4 provides a higher level of data abstraction than MPI I/O. NetCDF-4 views data as sets of related arrays. The NetCDF-4 data model comprises variables, dimensions, attributes, and user-defined types. Datasets can be stored in hierarchical groups.

The basic NetCDF-4 calls are:

- `nc_put_var_*()`, which writes a NetCDF-4 variable
- `nc_put_arr_*()`, which writes a NetCDF-4 attribute
- `nc_get_var_*()`, which reads a NetCDF-4 variable
- `nc_get_arr_*()`, which reads a NetCDF-4 attribute

NetCDF-4 supports collective and independent MPI I/O. The default is independent I/O. To make writes to a variable collective, call the `nc_var_par_access()` function.

Note: NetCDF-4 uses HDF5 1.8.1 as the data storage layer for netcdf, so technically HDF5 could be depicted as a separate layer in Figure 7. However, for the purposes of this discussion, we will treat HDF5 as a component of NetCDF.

2.4.4.1 NetCDF-4 Benefits

NetCDF-4 files are "self-describing," meaning that the files contain information fully describing the typed data items.

Use NetCDF-4 as an alternative to HDF5 if its simpler data model meets the needs of the application. NetCDF-4 provides access to Parallel HDF5 I/O features. In addition, HDF5 files produced by NetCDF-4 can be read by an HDF5 application.

2.4.4.2 References

NetCDF-4 information is available though the `netcdf`(3) man page and *The NetCDF Users Guide*. This guide and additional NetCDF-4 documentation, including example programs, is available at http://www.unidata.ucar.edu/software/netcdf/docs/.

I/O Strategies [3]

There are several strategies you can use for moving data between memory and files. In general, an application has one or more processes writing data to one or more files. While almost any combination of the number of processes doing writes and the number of application data files is possible, the four most common are:

- One File per Process: All Write
- Single Shared File: One Writes
- Single Shared File: All Write
- Single Shared File: Subset Writes

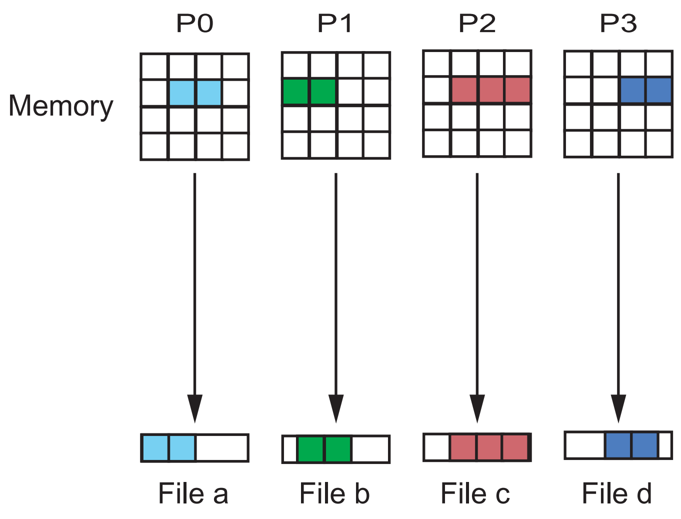
There is no best "one size fits all" solution. You may need to experiment with your I/O strategy to achieve good performance and scaling, and even after you do so, you may find that a strategy that works well under one set of circumstances may no longer be optimal if the number of processes or the amount of data changes significantly.

3.1 One File per Process: All Write

With this strategy, each process writes to its own file, as shown in Figure 8. This is a relatively simple I/O strategy to use, and the independent writes can perform well because multiple OSTs can support parallel I/O to many separate files.

However, when all of the files are opened or closed, all file metadata resides on a single MDT, creating a bottleneck, both for the application and for the entire file system. If the number of processes is small, this is probably not a serious problem, but this strategy does not scale well for a large number of processes, and if the ratio of processes to OSTs is large, this can also overwhelm the OSTs even after the files are opened. Also, if the entire set of files represents a single logical set of data, postprocessing is probably necessary, adding to the overall time and cost. For these reasons, we discourage the use of this strategy for large scale applications.

Figure 8. One File per Process: All Write



3.2 Single Shared File: One Writes

With this strategy, all processes involved in a write operation send data to one process, which then writes the data to a single shared file. In Figure 9, P1-P3 send data to P0. P0 sends all P0-P3 data to a single shared file.

This strategy uses data aggregation, a technique in which processes send their data to another process, where it is combined and sent to the file as an aggregate.

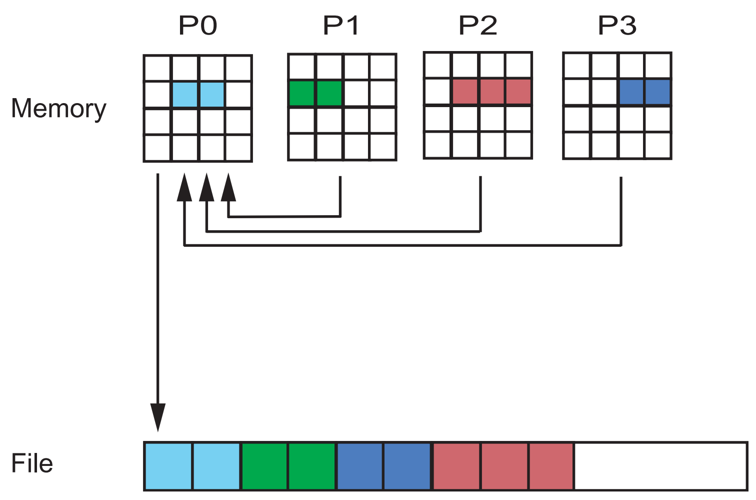
The strategy is relatively simple. The advantages are that there is no MDT bottleneck, and all the data is already consolidated into one file for subsequent use. There are potentially several disadvantages:

- The writes are sequential, not parallel. Performance is limited by the write bandwidth from one process. This strategy can perform better than all processes doing their own writes if process 0 can combine the data into fewer and larger contiguous pieces and if the aggregate size per write is approximately the size of one stripe or less. However, if the aggregate size of the data spans multiple stripes, better performance can be obtained with other strategies.

- The focus on system architecture has nothing to do with the problem the application is trying to solve.

- This strategy may provide good I/O performance for one system architecture but not another.

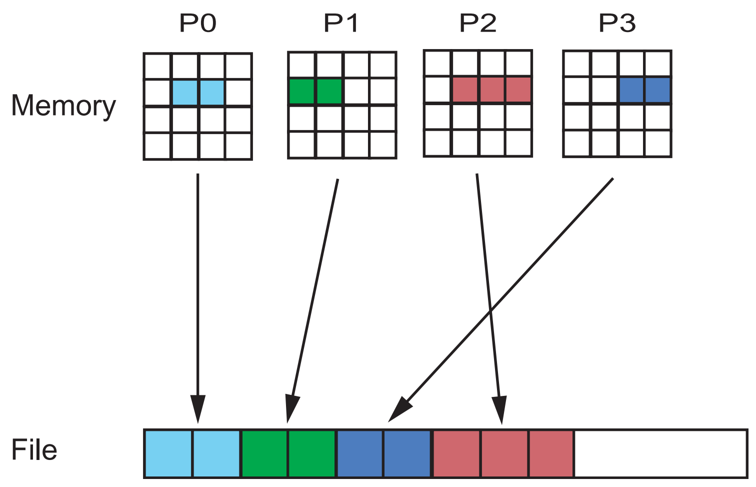
Figure 9. Single Shared File: One Writes



3.3 Single Shared File: All Write

With this strategy, all processes involved in a write operation send data to a single shared file, as shown in Figure 10. This strategy requires extra work by the application to maintain the separate file offsets for each process so one process does not overwrite another's data. Because all processes are writing, parallel I/O can be achieved, although with less-than-peak performance if the records from multiple processes are being written to the same OSTs. For records significantly larger than a stripe, performance can be very good.

Figure 10. Single Shared File: All Write



3.4 Single Shared File: Subset Writes

With this strategy, a subset of all processes involved in a write operation sends data to a single shared file, as shown in Figure 11.

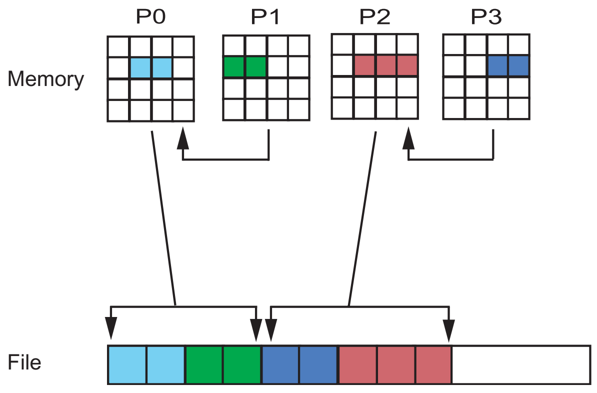
This strategy also uses data aggregation. The advantages are that there is no MDT bottleneck, and all the data is already consolidated into one file for subsequent use. There are potentially several disadvantages:

- Assuming that all the processes were involved in the application computation, this strategy is sometimes used to strike a balance between many processes writing at the same time (with resulting frequent extent lock contention) and only one process writing (with the resulting loss of parallel I/O).

  The biggest difficulty with this approach is finding the best balance, which probably changes with the problem size and almost certainly changes when running the application on different computer systems with different I/O system characteristics.

- The focus on system architecture has nothing to do with the problem the application is trying to solve.

Figure 11. Single Shared File: Subset Writes



3.5 Guidelines for I/O Strategies

As you develop your I/O strategy, consider the following guidelines.

- The "one file per process, all write" strategy can work well if you have a small number of processes and the amount of data per process is large.
- The "single shared file, one writes" strategy can work well if the application writes a small amount of data.
- The "single shared file, all write" strategy can work well if the amount of data per process is large and the ratio of processes to OSTs is small.
- The "single shared file, subset writes" strategy can work well if there are a large number of processes and there is a large amount of data, and if the appropriate number for the subset is selected and the data is sent to the subset in the appropriate way.

The difficulty with the above guidelines is that "small" and "large" are subjective terms and difficult to quantify, and even if you succeed in choosing and tuning a strategy for your application that works well under one set of circumstances, that strategy may no longer be optimal if the number of processes or the amount of data changes significantly. For these reasons we recommend using the "single shared file, all write" strategy and MPI collective I/O calls, as described in Chapter 4, Parallel I/O With MPI on page 27 and Chapter 5, Benchmarks on page 43, to let the MPI I/O library choose the appropriate subset of processes and optimize the I/O patterns.

Parallel I/O With MPI [4]

This chapter describes the steps to take to use collective MPI I/O. Collective MPI I/O is defined by the MPI Standard and has these characteristics:

- A file is opened collectively by a group of processes.
- A file is partitioned among processes of this group.
- All collective I/O calls on a file are collective over this group.
- An MPI file is an ordered collection of typed data items.
- Data access and positioning is done in terms of MPI datatypes, which can be any basic datatype or derived datatype.
- Derived datatypes can be constructed using any of the MPI datatype constructor routines.
- The physical layout of data in process memory or in a file is described in terms of the datatypes.
- A filetype defines a template for accessing data in a file and is specified in terms of basic or derived datatypes.
- A file view defines the current set of data visible to and accessible to each process.
- The file view can be changed during program execution to accommodate different access patterns.

See the MPI Standard chapter 9 for a more complete definition of MPI I/O terms.

Independent MPI I/O shares many of the above characteristics, but because we encourage the use of collective MPI I/O, we do not draw the distinctions here.

When you use collective I/O, you use the "shared file, all write" strategy. However, the library can optimize operations dynamically and use the "shared file, subset writes" or "shared file, one writes" strategy if it improves performance.

Because you can use derived datatypes to define the physical layout of data in both process memory and the file, data can be moved between memory and a file in terms of the datatype, relieving the program from directly dealing with the details of the data positions and contiguous data sizes on each I/O call. Once the derived datatypes are defined and committed, the library handles the details for the program.

Using datatypes and complementary file views, the processes doing I/O can achieve global data distribution in a single collective I/O call.

4.1 Basic Program Steps for Collective MPI I/O

Follow these steps for programming collective MPI I/O.

Note: The following steps and examples are provided in C. For comparable Fortran or C++ information, see the man pages.

1. Optionally, create an *info* object.

   The `MPI_Info_create()` call creates an opaque *info* object that is a means of passing a variety of information to the library when the file is opened. This is required if file hints will be passed to the library through MPI function calls.

   The synopsis is:

   `int MPI_Info_create(MPI_Info *`*info*`)`

2. Optionally, add information to the *info* object.

   The `MPI_Info_set()` call adds information to the *info* object created in step 1 in the form of *key,value* pairs. These information pairs are used by the library at various stages of the I/O process. Most of them are optimization hints, including striping information. Pairs not understood by the library are ignored. For more information about hints, see and the `intro_mpi`(3) man page.

   The synopsis is:

   `int MPI_Info_set(MPI_Info` *info*`, char *`*key*`, char *`*value*`)`

3. Optionally, delete the file.

   The `MPI_File_delete()` call deletes a named file if it exists. Striping information cannot be changed on an existing file, so to set the stripe count (and stripe size) for the amount of parallelism you want to achieve, the file must be deleted if it exists.

   The synopsis is:

   `int MPI_File_delete(char *`*filename*`, MPI_Info` *info*`)`

4. Open the file.

   The `MPI_File_open()` call opens the file. All processes in the collective I/O group (that is, all processes in the communicator group *comm*) must make this call. Often, this group would be `MPI_COMM_WORLD`. The file's name and the access modes (read only, write only, read/write, and so forth) are specified. If an *info* object has been created, it is also specified. Otherwise, the *info* argument is `MPI_INFO_NULL`.

The synopsis is:

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
MPI_Info info, MPI_File *fh)
```

5. Optionally, create a datatype to describe the physical layout of data in memory.

   The purpose of this step is to describe the data so that all the data to be transferred to or from memory at this point in the application can be done with one I/O call rather than a loop of I/O calls. Doing this in one call allows the library to optimize in ways not possible with many separate calls.

   MPI provides a set of datatype constructors that support describing any arbitrary physical layout of data, either in memory or a file. The format of these constructors is `MPI_Type_`*xxx*`()` and `MPI_Type_create_`*xxx*`()`. Choose Datatype Constructors to Match Your Data Model on page 31 describes many of them and shows how to use the appropriate constructor to match the data model of the application.

   A datatype constructor is not needed if the data to be transferred is a contiguous region of a basic datatype.

6. If a datatype was created, commit it.

   Datatype constructors can be applied recursively to build up arbitrarily complex datatypes, so there may be intermediate datatypes that are only a step toward the final datatype. Therefore, it is not until `MPI_Type_commit()` is called does the library record the information for use in I/O calls.

   The synopsis is:

   ```
   int MPI_Type_commit(MPI_Datatype *datatype)
   ```

7. Optionally, create a datatype to describe the physical layout of data in the file.

   This is similar to the previous create-datatype step, except it applies to the physical layout of data in the file. The layouts in memory and in the file may match, but they are not required to do so. If they do match, the same datatype can be used for both cases, and this step is not needed.

8. If a datatype was created for the file, commit it.

9. Set each process's view of the file.

   The `MPI_File_set_view()` call defines the current set of data in the file that is visible to and accessible to each process.

   The synopsis is:

   ```
   int MPI_File_set_view(MPI_File mpi_fh, MPI_Offset disp,
   MPI_Datatype etype, MPI_Datatype filetype,
   char *datarep, MPI_Info info)
   ```

The *disp* argument specifies the start of each process's view within the file and must be different for each process. The *etype* argument specifies the unit of data in the file, and the *filetype* argument specifies the datatype for the distribution of the *etype*s in the file. The *datarep* argument would normally be `native` if the data is created on and used on the same homogeneous system. The *info* argument can be the same *info* object used for the open call or can be different. Some of the information specified in the *info* object has no effect except at file open time.

10. Read data from a file or write data to a file.

    The `MPI_File_read_all()` call reads data from the specified file into process memory according to the call arguments and also the information previously passed to the library by the `MPI_File_open()` and `MPI_File_set_view()` calls.

    The synopsis is:

    ```
    int MPI_File_read_all(MPI_File mpi_fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
    ```

    Similarly, the `MPI_File_write_all()` call writes data from process memory to the specified file according to the arguments in this call and also the information previously passed to the library by the `MPI_File_open()` and `MPI_File_set_view()` calls.

    The synopsis is:

    ```
    int MPI_File_write_all(MPI_File mpi_fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
    ```

    The flow of data between memory and the file for a single read or write call can be quite complex. However, the MPI I/O library handles this complexity and can often optimize it in significant ways, making the earlier steps worthwhile.

11. Close the file.

    The `MPI_FIle_close()` call causes all previous writes to the file to be transferred to the storage devices and then closes the file.

    The synopsis is:

    ```
    int MPI_File_close(MPI_File *mpi_fh)
    ```

The example programs and code snippets that follow will show how this information is built up and made visible to the library. Discussion of the example code will describe some of the ways the library uses the information.

4.2 Choose Datatype Constructors to Match Your Data Model

Basic Program Steps for Collective MPI I/O on page 28 describes the basic steps for using collective MPI I/O. This section gives a brief description of most of the datatype constructors, more specific guidelines for choosing the datatype constructors that best match the data model of your application, and code examples for some of these.

Note: Descriptions and guidelines for using the `MPI_Type_create_darray()` and `MPI_Type_create_subarray()` datatype constructors are deferred to a future release of this guide.

An MPI application programmer probably already uses datatype constructors for moving data from one process's memory to another. The concept of using derived datatypes for moving data from process's memory to a file may be new, but there is no difference conceptually.

Datatypes describing multidimensional arrays of any size and shape and with elements of any datatype can be created using the new type created by one constructor call as an argument to another constructor call.

The variety of datatype-related calls suggests many possible combinations. In fact, with `MPI_Type_create_struct()`, any arbitrary layout can be described. The other calls are easier to use and can be used when the layout is simpler or more regular.

After the datatype is created, the `MPI_Type_commit()` routine must be called for the new datatype. This allows the implementation to store all the layout information for subsequent use in moving data and potentially for optimization of the movement.

See the MPI Standard or the other MPI references for more complete description.

4.2.1 Create a Contiguous Datatype

The `MPI_Type_contiguous()` datatype constructor can be used for a contiguous 1-D array that is distributed across the memory of all the processes. This constructor can also be used for building up a multidimensional array, as shown in a later example.

This first code example shows all of the basic collective MPI I/O steps, as described above, that would be used in a parallel application. Comments in the code describe each step. To make this example self-contained, an input data file is created first. You can copy from this example the parts that you need for your application.

Example 4. Distributed contiguous 1-D array

Source code of `basic.c`:

```
/* Include Files */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

/* For this self-contained example, the following macros are defined.
 * They can be defined with different values here. Or the example could
 * be expanded to take any of these as command line arguments. Of course
 * a real application would have its own set of variables and values. */
#define MY_INPUT_FILE "my_input"   /* name of input datafile */
#define MY_RESULTS_FILE "my_results" /* name of output results file */
#define MY_CHECKPOINT_FILE "my_checkpoint" /* name of checkpoint file */
#define LOCAL_SIZE 1000000L     /* size of local array in ints */
#define STRIPE_COUNT "4"       /* must be an ascii string */
#define STRIPE_SIZE "1048576"    /* must be an ascii string */
#define X_DIM 30000         /* size of 1st dimension */
#define Y_DIM 20000         /* size of 2nd dimension, if any */
#define Z_DIM 10000         /* size of 3rd dimension, if any */


int
create_my_input_file(char *file_name, int local_size, int init_value,
    int my_rank, int comm_size)
{
  MPI_File fh;
  MPI_Info info;

  int *local_array;
  int size;
  int rc;
  int i;
  double t0,t1;

  /* Delete any existing file so striping can be set. Striping cannot
   * be changed on an existing file. */
  rc = MPI_File_delete(file_name, MPI_INFO_NULL);

  /* Create a local array that will contain this process's data.
   * In this case, every local array is the same size, though that
   * is not necessary for collective I/O to work. */
  local_array = (int*)malloc((size_t)(local_size * sizeof(int)));
  if (local_array == NULL) {
    return -1;
  }

  /* Initialize the array with data that will serve as the input data. */
  for (i = 0; i < local_size; i++) {
    local_array[i] = init_value + i;
  }

  /* Set the stripe_count and stripe_size, that is, the striping_factor
   * and striping_unit. Both keys and values for MPI_Info_set must be
   * in the form of ascii strings. */
  MPI_Info_create(&info);
  MPI_Info_set(info, "striping_factor", STRIPE_COUNT);
  MPI_Info_set(info, "striping_unit", STRIPE_SIZE);
// MPI_Info_set(info, "romio_cb_write", "disable");

  /* All processes in the application open the file. The info object
   * sets the striping information for the file. */
  MPI_Barrier(MPI_COMM_WORLD);
  t0 = MPI_Wtime();
  rc = MPI_File_open(MPI_COMM_WORLD, file_name,
      MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

  /* The return code will be MPI_SUCCESS if the open was successful.
   * There are a number of options for handling unsuccessful calls.
   * We will return a negative status for the caller to handle. */
  if (rc != MPI_SUCCESS) {
    return -1;
  }

  /* Write the file as a collective call, with every process writing
   * a part of the file. */
  rc = MPI_File_set_view(fh, my_rank * (MPI_Offset)local_size * sizeof(int),
      MPI_INT, MPI_INT, "native", info);
  rc = MPI_File_write_all(fh, local_array, local_size, MPI_INT,
      MPI_STATUS_IGNORE);
  if (rc != MPI_SUCCESS) {
    return -1;
  }

  /* Close the file. */
  rc = MPI_File_close(&fh);

  MPI_Barrier(MPI_COMM_WORLD);
  t1 = MPI_Wtime();

  /* Print file creation information. */
  if (my_rank == 0) {
    MPI_Offset size = comm_size * (MPI_Offset)LOCAL_SIZE * sizeof(int);
    double time = t1-t0;
    double mibps = (size/time)/1048576.0;
    printf("input data file '%s' created;\n"
        " file_size=%ld create_time=%f6.2 MiB/sec=%f\n",
        MY_INPUT_FILE, size, time, mibps);
  }

  return 0;
}


int main(int argc, char **argv)
{
  MPI_Aint lb, extent;
  MPI_Datatype etype, memtype, filetype, contig;
  MPI_Offset disp;
  MPI_File in_fh;
  MPI_File out_fh;
  MPI_Info info;
  int buf[LOCAL_SIZE];
  int my_rank;
  int comm_size;
  double t0,t1;
  int stripe_count;
  int init_value;
  int rc;

  /* MPI Initialization */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

  /* A real application would likely already have an input data file
   * but for a self-contained example, we will create one and close it. */
  init_value = my_rank * LOCAL_SIZE;
  rc = create_my_input_file(MY_INPUT_FILE, LOCAL_SIZE, init_value,
      my_rank, comm_size);
  if (rc != 0) {
    fprintf(stderr, "could not create input file\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  /* Create an info object for hints. Note that striping can't be
   * changed on an existing file. */
  MPI_Info_create(&info);
  MPI_Info_set(info, "romio_cb_read", "enable");

  /* Open the input data file. */

  rc = MPI_File_open(MPI_COMM_WORLD, MY_INPUT_FILE, MPI_MODE_RDONLY,
      info, &in_fh);
  if (rc != MPI_SUCCESS) {
    fprintf(stderr, "could not open input file\n");
    MPI_Abort(MPI_COMM_WORLD, 2);
  }

  /* Construct a datatype for distributing the input data across all
   * processes. */
  MPI_Type_contiguous(LOCAL_SIZE, MPI_INT, &contig);
  MPI_Type_commit(&contig);

  /* Set the file view so that each process gets its portion of the
   * input data. */
  disp = my_rank * LOCAL_SIZE * sizeof(int);
  rc = MPI_File_set_view(in_fh, disp, contig, contig, "native", info);
  if (rc != MPI_SUCCESS) {
    fprintf(stderr, "error setting file view on input file\n");
    MPI_Abort(MPI_COMM_WORLD, 3);
  }

  /* Read the input data file. Since we created a contiguous datatype
   * the full size of each process's local data, the count is 1. */
  rc = MPI_File_read(in_fh, buf, 1, contig, MPI_STATUS_IGNORE);
  if (rc != MPI_SUCCESS) {
    fprintf(stderr, "error reading input file\n");
    MPI_Abort(MPI_COMM_WORLD, 3);
  }
  MPI_File_close(&in_fh);
```

```
/* Compute. This is where you do your science. In this simple
 * example, we will just check that the input was read correctly and
 * then multiply by 2. */

/* At some point, or perhaps at multiple time steps, calculated results
 * would be written out. And perhaps checkpoint files would periodically
 * written out. In this simple example, we will write results just
 * once. */

/* Delete the output file if it exists so that striping can be set
 * on the output file. */
rc = MPI_File_delete(MY_RESULTS_FILE, MPI_INFO_NULL);

/* Set the striping */

/* Open the results file. */

rc = MPI_File_open(MPI_COMM_WORLD, MY_RESULTS_FILE, MPI_MODE_WRONLY |
    MPI_MODE_CREATE, MPI_INFO_NULL, &out_fh);
if (rc != MPI_SUCCESS) {
  fprintf(stderr, "could not open results file\n");
  MPI_Abort(MPI_COMM_WORLD, 3);
}

/* Set the file view for the output file. In this example, we will
 * use the same contiguous datatype as we used for reading the data
 * into local memory. A better example would be to write out just
 * part of the data, say 4 contiguous elements followed by a gap of
 * 4 elements, and repeated. */

disp = my_rank * LOCAL_SIZE * sizeof(int);
MPI_File_set_view(out_fh, disp, contig, contig, "native", MPI_INFO_NULL);
if (rc != MPI_SUCCESS) {
  fprintf(stderr, "error setting view on results file\n");
  MPI_Abort(MPI_COMM_WORLD, 4);
}

/* MPI Collective Write */
t0 = MPI_Wtime();
rc = MPI_File_write_all(out_fh, buf, 1, contig, MPI_STATUS_IGNORE);
if (rc != MPI_SUCCESS) {
  fprintf(stderr, "error writing results file\n");
  MPI_Abort(MPI_COMM_WORLD, 5);
}

/* Close Files */
MPI_File_close(&out_fh);
t1 = MPI_Wtime();

/* Print time info. */
if (my_rank == 0) {
  MPI_Offset size = comm_size * (MPI_Offset)LOCAL_SIZE * sizeof(int);
  double time = t1-t0;
  double mibps = (size/time)/1048576.0;
  printf("results file '%s' written;\n"
      " file_size=%ld write_time=%f6.2 MiB/sec=%f\n",
      MY_RESULTS_FILE, size, time, mibps);
  }

  /* MPI Finalize */
  MPI_Finalize();

  return 0;
}
```

Compile and run program `basic`:

```
% cc -o basic basic.c
% aprun -n 4 ./basic
input data file 'my_input' created;
 file_size=16000000 create_time=0.0256586.2 MiB/sec=594.696055
results file 'my_results' written;
 file_size=16000000 write_time=0.0254666.2 MiB/sec=599.178003
Application 1030961 resources: utime 0, stime 0
```

Performance for this I/O pattern can be quite good with collective MPI I/O. Whatever the size of the 1-D array, the collective buffering optimization can reorder the data in memory to do very efficient I/O.

Large records, no gaps (see Large Records with No Gaps on page 45).

Small records, no gaps (see Small Records with No Gaps on page 47).

4.2.2 Create a Noncontiguous Datatype with Uniform Stride

The `MPI_Type_vector()` datatype constructor can be used for a distributed non-uniformly segmented 1-D array.

Small records, small gaps (see Small Records with Small Gaps on page 48).

4.2.3 Create a Noncontiguous Datatype with Non-uniform Stride

There are two noncontiguous datatype constructors for non-uniform stride:

- Datatypes with non-uniform stride. The `MPI_Type_indexed()` datatype constructor can be used for a distributed noncontiguous 2-D array with ghost cells (uniformly domain-decomposed multidimensional arrays).

  Large records, large gaps (see Large Records with Large Gaps on page 46).

  Small records, small gaps (see Small Records with Small Gaps on page 48).

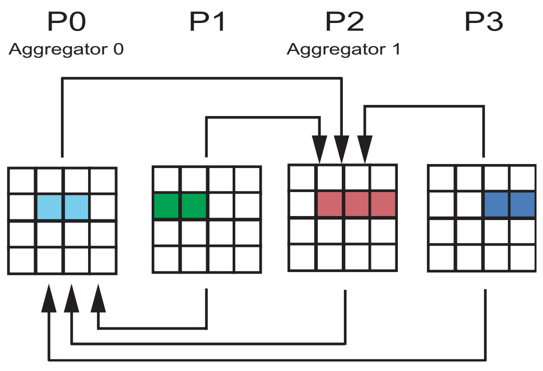
- Datatypes with non-uniform stride with an array of different datatypes as inputs. The `MPI_Type_create_struct()` can be used for a distributed noncontiguous 3-D array.

  Small records, large gaps (see Small Records with Large Gaps on page 49).

4.3 Collective Buffering

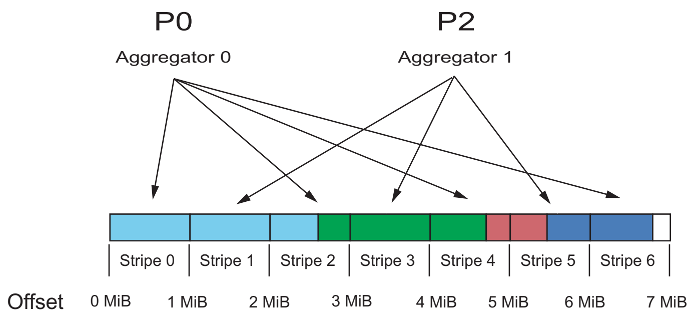
With collective MPI I/O, by default you use a technique called *collective buffering*. As shown in Figure 12, collective buffering consolidates I/O requests for all processes. In this example, the MPI I/O library chooses P0 and P2 as aggregators. All processes transfer data to the appropriate aggregator, based on the record lengths and offsets.

Figure 12. Aggregating Data



After the consolidation, only the aggregators perform I/O, as shown in Figure 13. P0) writes data to stripes 0, 2, 4, and 6. In parallel, P2 writes data to stripes 1, 3, and 5.

Figure 13. Aggregators Writing Data



You do not have to do anything to designate a process as an aggregator; the MPI I/O interface does that for you. The interface sets the number of aggregators to the stripe count. This allows the aggregators to use Lustre in an efficient manner, because writes to a shared file are stripe aligned and therefore do not compete for the same physical I/O block or OST.

4.4 Collective I/O Guidelines

For most large-file writes (1 GB or greater), MPI I/O collective buffering gives the best results. However, the overhead associated with dividing the I/O workload can in some cases exceed the time otherwise saved by using this method. Here are guidelines to help you decide when to use collective buffering:

- For large unaligned writes to sometimes the same stripes, use collective buffering.
- For small writes to often the same stripe or stripes, use collective buffering.
- For small writes to different stripes, do not use collective buffering.
- For reads, the only time collective buffering helps is if there is a significant amount of data still in cache from a previous write. This is not a common occurrence for the large scale jobs where I/O time is a significant issue.
- When scaling to a large number of processes (that is, more than 10,000), two or three aggregators per OST may improve performance.
- If the data access pattern for a file switches during application execution from a pattern that benefits from collective buffering to a pattern where collective buffering is detrimental, then change the hint value for collective buffering during execution between `enable` and `disable` as follows:

  ```
  % setenv MPICH_MPIIO_HINTS myfile:romio_cb_write=disable
  ```

  If you do this, also set `romio_no_indep_rw` to `false`.
- If an application's I/O strategy is to have a subset of the processes act as writers and only the writer processes open the file, then the maximum number of aggregators allowed is the number or writers. If there are more OSTs available than there are writers, set the stripe count to an integer multiple of the number of writers.

4.5 MPI I/O Optimization Hints

MPI I/O optimization hints can be passed in by two methods: either by calling `MPI_Info_set` in the application code, or by setting the `MPICH_MPIIO_HINTS` environment variable. If set, the environment variable overrides the default value of one or more MPI I/O hints, and also overrides any values set using `MPI_Info_set`. The new values apply to the file the next time it is opened using an `MPI_File_open()` call.

Note: For more information about MPI-related environment variables, see the `intro_mpi`(3) man page.

After the `MPI_File_open()` call, subsequent `MPI_Info_set` calls can be used to pass new MPI I/O hints that take precedence over some of the environment variable values. Other MPI I/O hints such as `striping_factor`, `striping_unit`, `cb_nodes`, and `cb_config_list` cannot be changed after the `MPI_File_open()` call, as these are evaluated and applied only during the file open process.

An `MPI_File_close` call followed by an `MPI_File_open` call can be used to restart the MPI I/O hint evaluation process.

The syntax for this environment variable is a comma-separated list of specifications. Each individual specification is a *pathname_pattern* followed by a colon-separated list of one or more `key=value` pairs. In each `key=value` pair, the `key` is the MPI-IO hint name, and the `value` is its value as it would be coded for an `MPI_Info_set` library call.

For example:

`MPICH_MPIIO_HINTS=`*spec1*`[,`*spec2*`,...]`

Where each specification has the syntax:

pathname_pattern`:`*key1*=*value1*`[:`*key2*=*value2*`:...]`

The *pathname_pattern* can be an exact match with the *filename* argument used in the `MPI_File_open()` call or it can be a pattern as described below.

When a file is opened with `MPI_File_open()`, the list of hint specifications in the `MPICH_MPIIO_HINTS` environment variable is scanned. The first *pathname_pattern* matching the *filename* argument in the `MPI_File_open()` call is selected. Any hints associated with the selected *pathname_pattern* are applied to the file being opened. If no pattern matches, no hints from this specification are applied to the file.

The *pathname_pattern* follows standard shell pattern-matching rules with these meta-characters:

Pattern	**Description**
`*`	Match any number of characters
`?`	Match any single character
`[`*a*`-`*b*`]`	Match any single character between *a* and *b*, inclusive
`\`	Interpret the meta-character that follows literally

The simplest *pathname_pattern* is `*`. Using this results in the specified hints being applied to all files opened with `MPI_File_open()`. Use of this wildcard is discouraged because of the possibility that a library linked with the application may also open a file for which the hints are not appropriate.

The following example shows how to set hints for a set of files. The final specification in this example, for file `/scratch/user/me/dump.*`, has two `key=value` pairs.

```
MPICH_MPIIO_HINTS=file1:direct_io=true,file2:romio_ds_write=disable,
/scratch/user/me/dump.*:romio_cb_write=enable:cb_nodes=8
```

The following MPI-IO *key* values are supported on Cray systems.

`striping_factor`

Specifies the number of Lustre file system stripes (stripe count) to assign to the file. This has no effect if the file already exists when the `MPI_File_open()` call is made. File striping cannot be changed after a file is created. Currently this hint applies only when `MPICH_MPIIO_CB_ALIGN` is set to `2`.

Default: the default value for the Lustre file system, or the value for the directory in which the file is created if the `lfs setstripe` command was used to set the stripe count of the directory to a value other than the system default.

`striping_unit`

Specifies in bytes the size of the Lustre file system stripes (stripe size) assigned to the file. This has no effect if the file already exists when the `MPI_File_open()` call is made. File striping cannot be changed after a file is created. Currently this hint applies only when `MPICH_MPIIO_CB_ALIGN` is set to `2`.

Default: the default value for the Lustre file system, or the value for the directory in which the file is created if the `lfs setstripe` command was used to set the stripe size of the directory to a value other than the system default.

`direct_io` Enables the `O_DIRECT` mode for the specified file. The user is responsible for aligning the write or read buffer on a `getpagesize()` boundary. MPI-IO checks for alignment and aborts if it is not aligned. Valid values are `true` or `false`.

Default: `false`.

`romio_cb_read`

Enables collective buffering on read when collective IO operations are used. Valid values are `enable`, `disable`, and `automatic`. In automatic mode, whether or not collective buffering is done is based on runtime heuristics. When `MPICH_MPIIO_CB_ALIGN` is set to `2`, the heuristics favor collective buffering.

Default: `automatic`.

`romio_cb_write`

Enables collective buffering on write when collective IO operations are used. Valid values are `enable`, `disable`, and `automatic`. In automatic mode, whether or not collective buffering is done is based on runtime heuristics. When `MPICH_MPIIO_CB_ALIGN` is set to `2`, the heuristics favor collective buffering.

Default: `automatic`.

`cb_buffer_size`

Sets the buffer size in bytes for collective buffering.

When `MPICH_MPIIO_CB_ALIGN` is set to 2, this hint has no effect because the buffer size is equal to the stripe size (`striping_unit`).

Default: `16777216`.

`cb_nodes` Specifies the number of aggregators used to perform the physical I/O for collective I/O operations when collective buffering is enabled. On multi-core nodes, all cores share the same node name.

When `MPICH_MPIIO_CB_ALIGN` is set to 2, `cb_nodes` should be set the same as *striping_factor* (in other words, to the stripe count) to get the maximum benefit of Lustre stripe alignment.

Default: *striping_factor* when `MPICH_MPIIO_CB_ALIGN` is set to `2`, or the number of XT compute nodes when `MPICH_MPIIO_CB_ALIGN` is set to `0` or `1`.

`cb_config_list`

Specifies by name which nodes are to serve as aggregators. The syntax for the value is:

#name1`:`*maxprocesses*`[,`*name2*`:`*maxprocesses*`,...]#`

Where *name* is either `*` (match all node names) or the name returned by `MPI_Get_processor_name`, and *maxprocesses* specifies the maximum number of processes on that node to serve as aggregators. If the value of the `cb_nodes` hint is greater than the number of XT compute nodes, the value of *maxprocesses* must be greater than `1` in order to assign the required number of aggregators. When `MPICH_MPIIO_CB_ALIGN` is set to `2`, the aggregators are assigned using a round-robin method across XT compute nodes. When `MPICH_MPIIO_CB_ALIGN` is set to `0` or `1`, up to *maxprocesses* aggregators are assigned to the first node, and so on for each node as needed.

The pair of # characters beginning and ending the list are not part of the normal MPIIO hint syntax but are required. Because colon (`:`) characters are used in both this list and in the `MPICH_MPIIO_HINTS` environment variable syntax, the # characters are required in order to determine the meaning of colon (`:`) character.

This value cannot be changed after the file is opened.

Default: `*:*` when `MPICH_MPIIO_CB_ALIGN` is set to `2`, or `*:1` when `MPICH_MPIIO_CB_ALIGN` is set to `0` or `1`.

`romio_no_indep_rw`

Specifies whether deferred open is used. Valid values are `true` and `false`.

Default: `false`.

`romio_ds_read`

Specifies if data sieving is to be done on read. Valid values are `enable`, `disable`, and `automatic`.

Default: `disable` when `MPICH_MPIIO_CB_ALIGN` is set to `2`, or `automatic` when `MPICH_MPIIO_CB_ALIGN` is set to `0` or `1`.

`romio_ds_write`

Specifies if data sieving is to be done on write. Valid values are `enable`, `disable`, and `automatic`.

Default: `disable` when `MPICH_MPIIO_CB_ALIGN` is set to `2`, or `automatic` when `MPICH_MPIIO_CB_ALIGN` is set to `0` or `1`.

`ind_rd_buffer_size`

Specifies in bytes the size of the buffer to be used for data sieving on read.

Default: `4194304`

`ind_wr_buffer_size`

Specifies in bytes the size of the buffer to be used for data sieving on write.

Default: `524288`

Benchmarks [5]

Cray has extensively tested the I/O methods described in this book using the IOR (Interleaved or Random) benchmark. IOR, developed by Lawrence Livermore National Laboratories, tests I/O performance using POSIX I/O and MPI I/O interfaces and various access patterns. The test creates a new file, writes data to it, then reads the data back.

We show IOR results here instead of results for real HPC applications because with IOR, data access patterns can be clearly defined by the input parameter values, and these patterns represent a wide range of real applications. These results show that the I/O strategy and data access patterns have a significant affect on I/O performance, sometimes by two orders of magnitude.

The system configuration and IOR parameter values are given for the various results. You can, if you wish, reproduce these IOR results on similar Cray XT systems—scaling up or down to larger or smaller configurations and parameter values—and then compare results with other computer systems.

We ran these benchmarks on a dedicated Cray XT5 system, using eight 8-core nodes (64 cores). The Lustre file system has one DDN 9550 controller and 16 OSTs. All of the OSTs were approximately 20% full.

We have also run benchmarks on larger and smaller Cray XT4 and Cray XT5 systems, and the relative results are approximately the same as shown in the graphs below for this configuration.

5.1 IOR Requirements

There are two requirements that need to be met in order to obtain meaningful bandwidths:

- There must be a sufficient number of processes writing data relative to the number of OSTs. Generally, 2-4 processes per OST will maximize the file system bandwidth. Adding more processes does not increase bandwidth and may decrease it somewhat. So a job running with many times more processors than OSTs may not attain peak bandwidth, but if you use collective buffering, performance stays near peak at even very large process counts.
- The total file size must be large enough to eliminate any benefits of caching in either the compute node kernels or the file system. A file that is larger than the total amount of memory on all the compute nodes used by the application meets this requirement.

5.2 IOR Writes

The following graphs show the results of the IOR benchmark write operations. The labels are:

- **POSIX**: results for IOR using POSIX I/O
- **MPI I/O Ind**: results for IOR using independent MPI I/O
- **MPI I/O Coll No Buf**: results for IOR using collective MPI I/O without collective buffering
- **MPI I/O Coll Buf**: results for IOR using collective MPI I/O with collective buffering

Record sizes are in power-of-10 units, KB and MB. We use these units because real applications rarely have power-of-2 sizes and file offsets, so power-of-10 units are more realistic for illustrating relative performance.

The absolute value of the performance results is not as significant as the relative values. These results are from a file system with just one disk controller. Results on file systems with multiple disk controllers usually have proportionately better performance.

5.2.1 Large Records with No Gaps

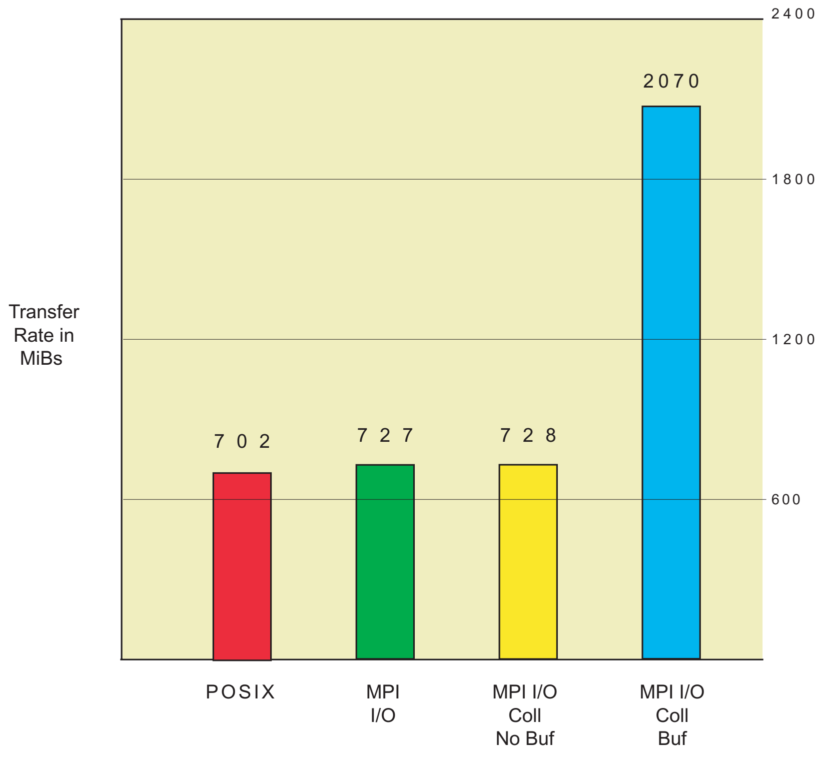
This set of results shows that collective buffering improves performance significantly for this data access pattern. The key factors are:

- The size of the records and the file offsets for the records are not set to nicely align with Lustre physical block boundaries. Therefore, Lustre has to serialize some of the write operations. But with collective buffering, the MPI I/O layer reorganizes the data access pattern so all the write operations are aligned.
- There are no gaps between the collective set of records from all the processes, so with collective buffering, chunks of data from different processes can be merged into larger regions, thus reducing the number of chunks of data being written.
- The records are large relative to stripe size, so for the total amount of data written, the overhead of doing collective buffering is quite small.

The POSIX, independent MPI I/O, and collective MPI I/O without collective buffering tests all performed about the same because the path through the MPI I/O layer is short for independent MPI I/O and collective MPI I/O without collective buffering.

The total file size for this test was 191 GB.

Figure 14. Benchmark: Large Records with No Gaps



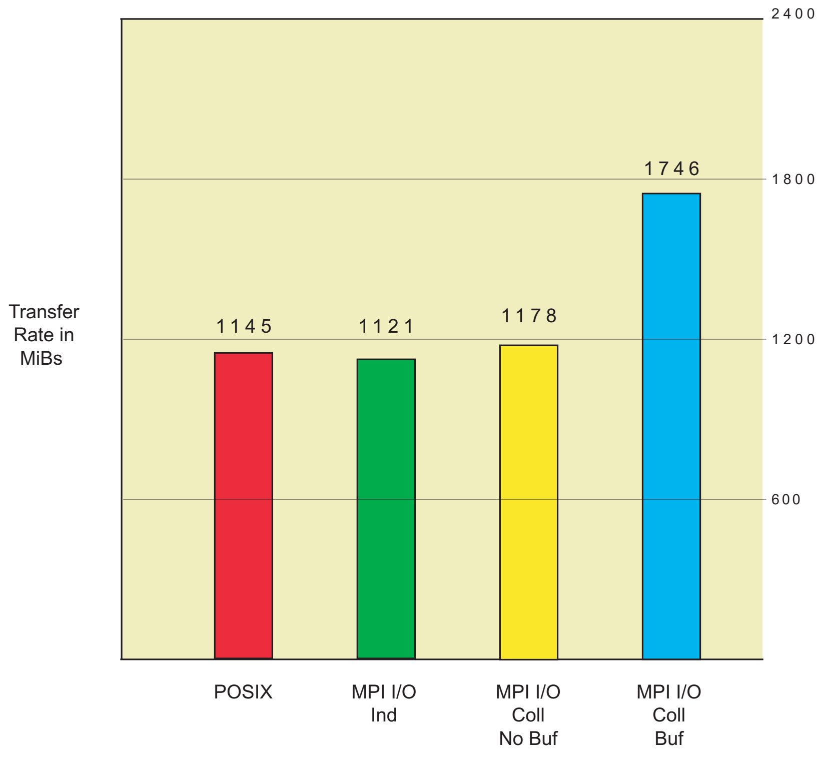
5.2.2 Large Records with Large Gaps

The performance of POSIX, independent MPI I/O, and collective MPI I/O without collective buffering is greater than the same modes in the previous "no gaps" case because the gaps between the records are large enough that there is significantly less contention for the same physical block at any given time.

The performance of collective MPI I/O with collective buffering is less than for the same mode with no gaps because the merging of adjacent records does not occur. As a result, after the data is reorganized, there are some chunks of data less than a full stripe to be written on any given call.

The total file size for this test was 191 GB.

Figure 15. Benchmark: Large Records with Large Gaps



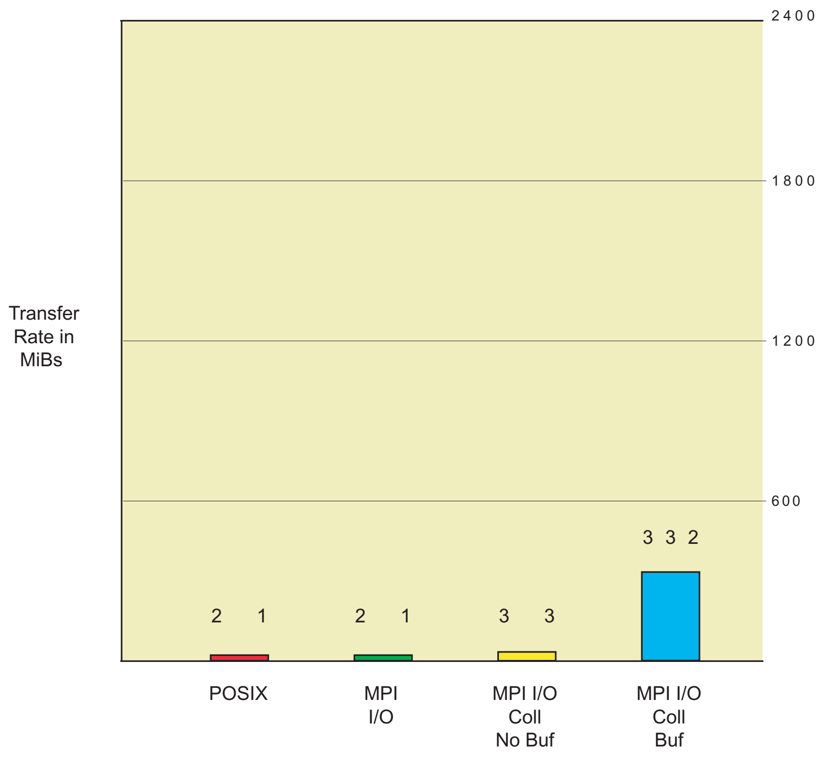
5.2.3 Small Records with No Gaps

This is the most dramatic difference between using collective buffering and not using collective buffering. Given this data access pattern, without collective buffering, many processes contend for the same stripe and often the same physical block at the same time. This contention makes for very low write bandwidth. With collective buffering, the contention is completely eliminated. Also, because there are no gaps, collective buffering can merge adjacent records.

The bandwidth for collective buffering mode is low compared to the "large record" cases because, with small record sizes and many I/O calls, the overhead on each I/O call for collective buffering becomes significant.

The total file size for this test was 19 GB. We used the smaller file size because the bandwidth is so low that run time is prohibitive at 191 GiB.

Figure 16. Benchmark: Small Records with No Gaps

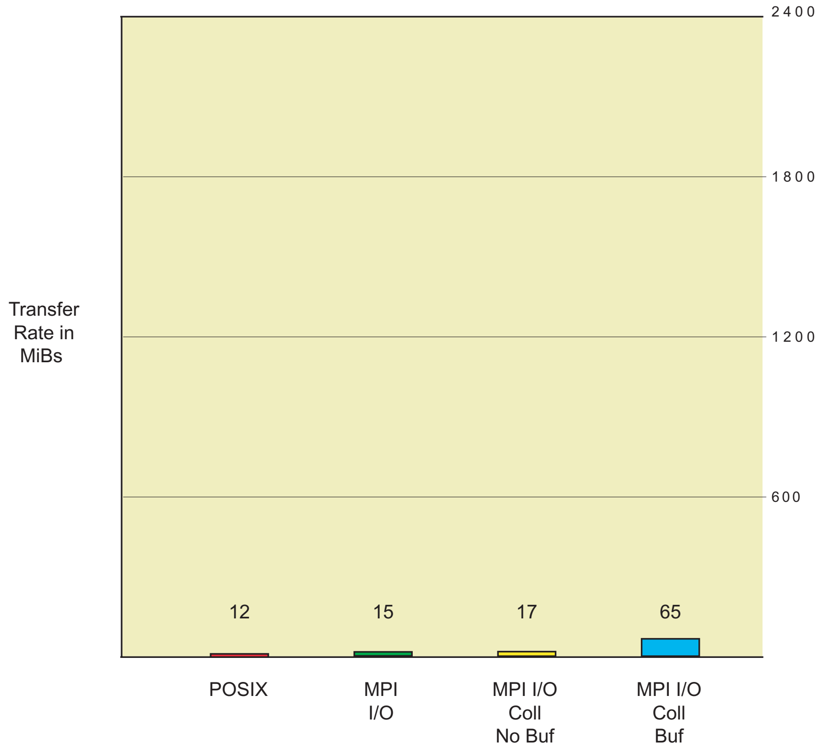


5.2.4 Small Records with Small Gaps

Without collective buffering, the large number of relatively small records results in much call overhead and lock contention. With collective buffering, the lock contention is eliminated, but the call overhead and the additional collective buffering overhead is significant. It is best to avoid this data access pattern if possible.

The total file size for this test was 19 GB. We used the smaller file size because the bandwidth is so low that run time is prohibitive at 191 GiB.

Figure 17. Benchmark: Small Records with Small Gaps

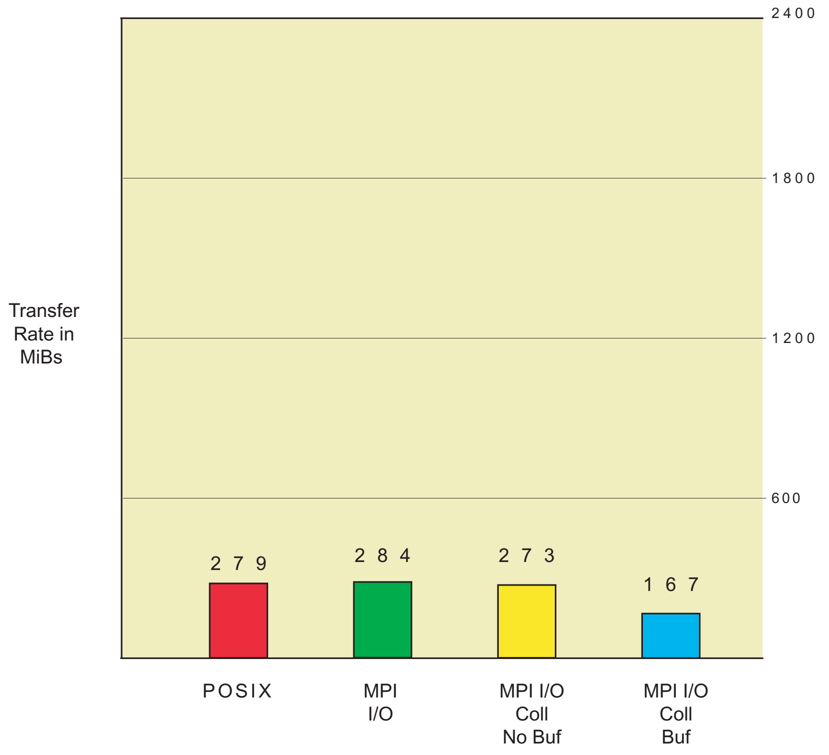


5.2.5 Small Records with Large Gaps

The large gaps between records means that there is little lock contention. With collective buffering, there is overhead for the many small calls, but little or no gain is realized by reducing lock contention or merging adjacent records.

The total file size for this test was 19 GB. We used the smaller file size because the bandwidth is so low that run time is prohibitive at 191 GiB.

Figure 18. Benchmark: Small Records with Large Gaps



Summary [6]

Remember, there is no "one size fits all" approach to improving MPI I/O. However, the following guidelines should give you a reasonable expectation of what is possible:

- Modifying an existing application or writing a new application to use collective I/O optimization techniques is not necessarily easy, but the payoff can be substantial. On real applications, I/O performance improvements of 2, 4, and even 20 times have been seen.
- The preferred I/O interface is MPI collective I/O with collective buffering.
- IOR benchmark results show significant performance advantages for collective buffering over other strategies. For all tested patterns except small records with large gaps, the MiBs/second transfer rate for collective buffering exceeds that of the other strategies by factors of 50-1000%.

References [7]

- *Cray XT System Overview*
- *Parallel I/O for High Performance Computing* by John M. May
- MPI I/O
  - MPI publications at http://www.mcs.anl.gov/research/projects/mpi/learning.html
  - MPI I/O man pages (read `intro_mpi`(3) first)
- POSIX I/O
  - POSIX.1-2008 at http://www.opengroup.org/onlinepubs/9699919799/
  - POSIX I/O man pages (`open`(2), `read`(P), `write`(P), `close`(2))
- HDF5
  - *Introduction to HDF5* at http://www.hdfgroup.org/HDF5/doc/H5.intro.html
  - *HDF5: API Specification Reference Manual* at http://www.hdfgroup.org/HDF5/doc1.6/RM_H5Front.html
- NetCDF-4
  - NetCDF-4 man pages (read `netcdf`(3) first)
  - *The NetCDF Users Guide* at http://www.unidata.ucar.edu/software/netcdf/docs/
- Lustre
  - Lustre publications at http://wiki.lustre.org/index.php/Lustre_Publications
  - Lustre `lfs`(1) man page
- *IOR User Guide* at http://sourceforge.net/projects/ior-sio/
- NERSC I/O performance documents from John Shalf and Katie Antipas at http://www.nersc.gov/nusers/systems/franklin/io.php

Glossary

compute node

A node that runs application programs. Compute nodes have one or more cores (also referred to as CPUs).

KB (Kilobyte)

A decimal multiple equal to 10^3 bytes (1,000 bytes).

KiB (Kibibyte)

A binary multiple equal to 2^{10} bytes (1,024 bytes).

MB (Megabyte)

A decimal multiple equal to 10^6 bytes (1,000,000 bytes).

metadata target (MDT)

A Lustre representation of a physical disk containing metadata about user files.

MiB (Mebibyte)

A binary multiple equal to 2^{20} bytes (1,048,576 bytes).

OST (object storage target)

A Lustre representation of a physical disk containing user files.

process

An instance of an application executable. Each process runs on a compute node core. A process is also referred to as a *processing element*.

striping_factor

The MPI term for stripe count.

striping_unit

The MPI term for stripe size.