Optimizing MPI-IO for Applications on Cray XT Systems

Abstract

Advances in multicore processor technology have enabled users to run MPI applications on tens of thousands of compute node cores. Such wide distribution of work can provide significant improvements in the performance of computationally intensive code but can also severely strain the I/O system.

The MPI-IO layer of the software stack provides a mechanism for optimizing I/O for the user. Cray recently added a new algorithm for collective buffering for writes. Work on I/O optimization by the MPI-IO layer is ongoing and some ideas for future work are described. Feedback from users is encouraged.

This paper describes the components of the Cray XT MPI-IO software stack, describes optimization techniques and their trade-offs, and documents the results of some benchmark tests.

Version 1.0 Published May 2009 Cray MPT 3.1 or later on Cray XT systems running CLE 2.1 or later.

Table of Contents

Introduction

The intent of this paper is to motivate users of Cray XT systems with applications that do a significant amount of I/O to:

- consider using MPI-IO collective I/O calls if not already doing so
- use MPI-IO optimization hints if not already doing so
- try the new collective buffering algorithm in the MPT 3.2 release

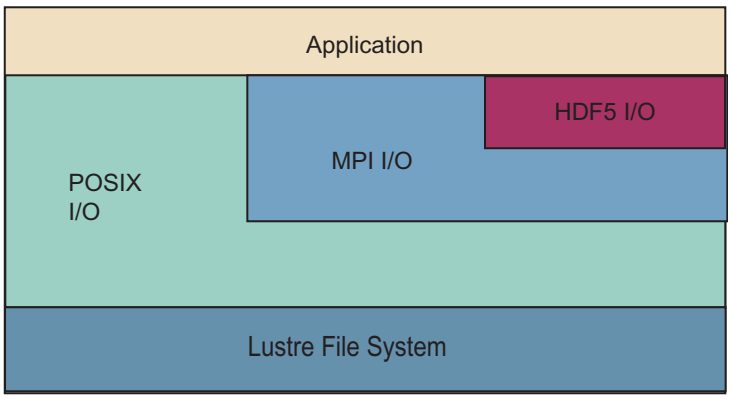
To provide context for the optimization techniques and benchmark test results described in this paper, we first need to look at the components of the Cray MPI-IO software stack and the basic I/O process flows. Then we give a simple example to help you understand the flow of collective buffering. Next, we describe the several collective buffering alignment algorithms and collective buffering optimization hints. Specific benchmark results show how collective buffering can improve I/O performance, and specific examples of how to set hints are given to help you get started.

In the Cray Message Passing Toolkit (MPT) 3.1 and 3.2 releases, Cray added algorithms for improved I/O workload distribution through the use of new collective buffering techniques.

Cray MPI-IO Software Stack

We intentionally simplified this description of the Cray MPI-IO software stack to focus on the key issues of collective buffering.

Figure 1. Cray MPI-IO Software Stack



The Cray MPI-IO software stack consists of the following layers:

- User's application using MPI. The application may or may not make calls to the MPI-IO library or the HDF5 library. If it does not, it may benefit from doing so.
- Optional Hierarchical Data Format Version 5 (HDF5) library. The HDF5 library provides an interface for the application between the application data structures and the data stored in a file. HDF5 I/O is implemented in terms of MPI-IO calls. The intent of HDF5 I/O is for the user to not have to worry about translating I/O calls for complex data structures into specific offsets into a file.
- Cray MPI-IO library. The Cray implementation of MPI-IO is contained in the MPI library `libmpich.a`. It is based on the ROMIO implementation of the I/O part of the MPI-2 Standard[1]. MPI-2 introduced MPI-IO functions, such as `MPI_File_write()` and `MPI_File_write_all()`. These MPI-IO functions eventually make POSIX I/O system calls to perform the desired I/O functions.
- POSIX I/O system calls. Any application doing I/O on a Cray XT system eventually, at some level of the application executable, makes POSIX I/O system calls: `open()`, `write()`, `read()`, `lseek()`, `close()`, and so on. These system calls are processed by the underlying Lustre file system. For the purposes of this paper, "POSIX I/O" refers to I/O done through this system call interface.
- Lustre File System. For the purposes of this paper, we are grouping a lot of software and hardware into the single concept of the Lustre file system.

Read and Write Process Flows

The read and write process flows are:

POSIX I/O

If the application makes standard C, C++, or Fortran I/O calls, these calls are translated into POSIX I/O calls. On writes, data flows from the user's data space to the Lustre file system, possibly being buffered in the CNL kernel. On reads, data flows from the Lustre file system, through the CNL kernel, and to the user's data space. There is nothing in any of these calls that inherently supports parallel I/O. The application must manage the parallel aspects of the I/O by itself. Similarly, if complex data structures are written to a file or read from a file, the application must break the non-contiguous data into separate contiguous segments and make separate calls for each segment.

One way to manage parallel I/O with POSIX I/O is for each processing element (PE) to write to or read from its own file, often referred to as "file-per-process" I/O. From a functional point of view, this may or may not work for an application. For example, if it is writing or reading a multidimensional array that all PEs are working on, the data is interleaved at some level. From a performance point of view, independent writes and reads perform well because, on the Lustre file system, multiple object storage targets (OSTs) can support parallel I/O to many separate files. However, when all these files are opened or closed, all these operations must go to a single Metadata Server (MDS), and this becomes a bottleneck with a large number of PEs.

[1] ROMIO is an implementation of the MPI-2 Standard by the Argonne National Laboratory Group.

For all three of these reasons (no inherent support for parallel I/O, no inherent support for I/O on complex data structures, and the inability to parallelize the metadata operations), you should consider converting to MPI-IO or HDF5 I/O.

MPI-IO independent I/O

The basic MPI-IO independent I/O calls for write and read are `MPI_File_write()` and `MPI_File_read()`. These are very similar to the POSIX I/O calls, with two important differences. First, the POSIX I/O calls support only contiguous segments of data, but the MPI-IO calls support derived data types, which the user defines according the application's data structures. The derived data types can contain non-contiguous data and non-uniform strides. This can simplify the application. A second difference is that, when a file is opened with `MPI_File_open()`, one of the arguments is an MPI *info* object which can contain information about the file, including hints for optimization.

As with POSIX I/O, MPI independent I/O can be done either with a single shared file or with file per process I/O.

From a performance point of view, POSIX I/O and MPI independent I/O usually perform almost identically. However, MPI-IO can also do some optimizations, such as data sieving (not discussed in this paper), that can help performance in some cases.

MPI-IO collective I/O

The basic MPI-IO collective I/O calls for write and read are `MPI_File_write_all()` and `MPI_File_read_all()`. These are similar to the MPI-IO independent I/O calls with two important differences. First, all PEs must make the collective I/O call. That is, the application cannot do something like:

```
if (myrank == 0) {
   MPI_File_write_all(...);
 }
```

because it will hang. Second, collective I/O optimizations are possible because of the inherent synchronization of all PEs. Most of the rest of this paper addresses the benefits of collective buffering optimization.

HDF5 I/O

HDF5 is a library that supports hierarchical data formats, that is, complex data structures. See http://www.hdfgroup.org/HDF5/ for more information. As with MPI-IO, HDF5 supports I/O with derived data types. However, HDF5 supports more levels of data abstraction than MPI, and HDF5 translates its I/O calls to the appropriate MPI-IO calls. Therefore, optimization of MPI-IO will generally have similar benefits for HDF5 I/O.

MPI-IO Collective Buffering

Perhaps the two most critical factors in I/O performance of a parallel application on Cray XT systems writing to or reading from a single shared file are:

- The use of MPI-IO functions instead of POSIX I/O
- The use of MPI-IO optimization techniques

If you are using POSIX I/O and if your code is not already structured for optimizing I/O, consider switching to MPI-IO.

Of all MPI-IO optimization techniques, collective buffering offers the greatest potential for improving I/O performance. To use collective buffering efficiently, it is important to understand the collective buffering (CB) modes, the CB alignment algorithms, and their trade-offs.

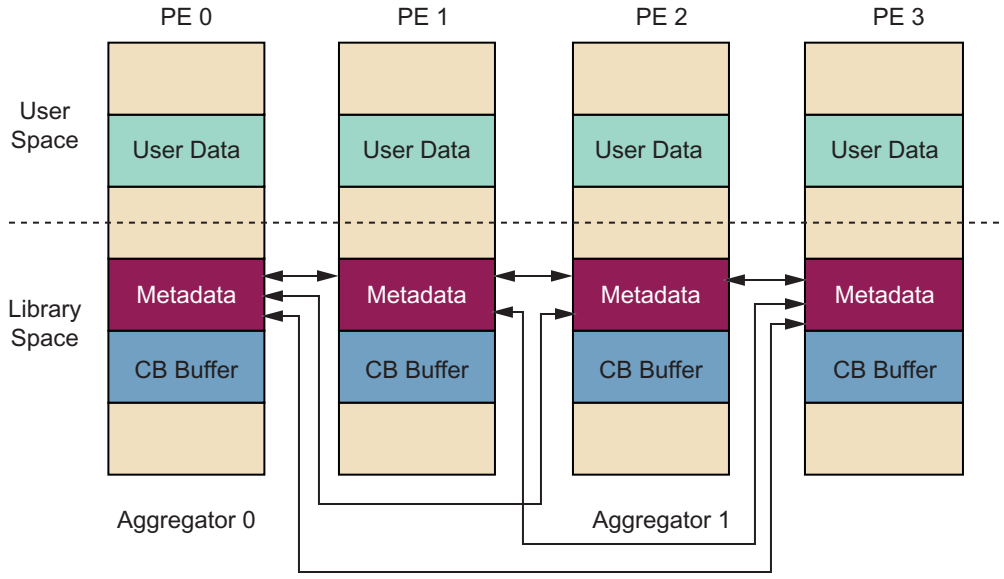
Collective buffering consolidates I/O requests for all processing elements PEs and redistributes the workload for more efficient Lustre file system I/O. Throughout this paper, for simplicity we use the phrase "all PEs" to mean all MPI ranks in the communicator associated with the file (that is, all the MPI ranks that have opened the file). After the consolidation, only a subset of the PEs performs the I/O. The subset is called the aggregators. The basic idea is to organize the file accesses so that different PEs are not competing for the same physical I/O block. More than one PE writing to the same physical I/O block (64K bytes in the case of Lustre on Cray XT systems) cannot be done in parallel, so having aggregation done first and then one PE write can be a big benefit.

The paper *An Extended Two-Phase Method for Accessing Sections of Out-of_core Arrays* by Rajeev Thakur and Alok Choudhary gives a very clear description of the problem to be solved by collective buffering and the ROMIO solution. For example, it can be beneficial for applications that have large out-of-core arrays, with sections of the arrays being read from and written to a file in non-contiguous sections for processing by the cooperating PEs.

Figure 2, Figure 3, and Figure 4 show the flow of data during a collective write call, `MPI_File_write_all()`, when collective buffering is enabled. These figures are for a simple example of four PEs with two aggregators.

In Figure 2, all PEs make an `MPI_File_write_all()` call at the same time, as required by the MPI-2 Standard. All PEs share information with all other PEs about the addresses of user data, the lengths of the data segments, and the offsets into the file where the data is to be written. Because MPI supports derived data types for describing complex data structures, the data does not have to be contiguous in memory or in the file. We refer to this data about the addresses and lengths of the user data and the offsets into the file as *metadata*.

Figure 2. Exchanging Metadata



After the metadata is exchanged, each PE can determine which aggregator will perform the system write of its data, and each aggregator can determine which PEs will provide the data. The data being sent by any PE might be to just one aggregator or to multiple aggregators, depending on the CB alignment algorithm (Collective Buffering Alignment Algorithms) being used and the size and file offsets of the data. As shown in Figure 3, all PEs transfer data to both aggregators.

Figure 3. Aggregating I/O Data

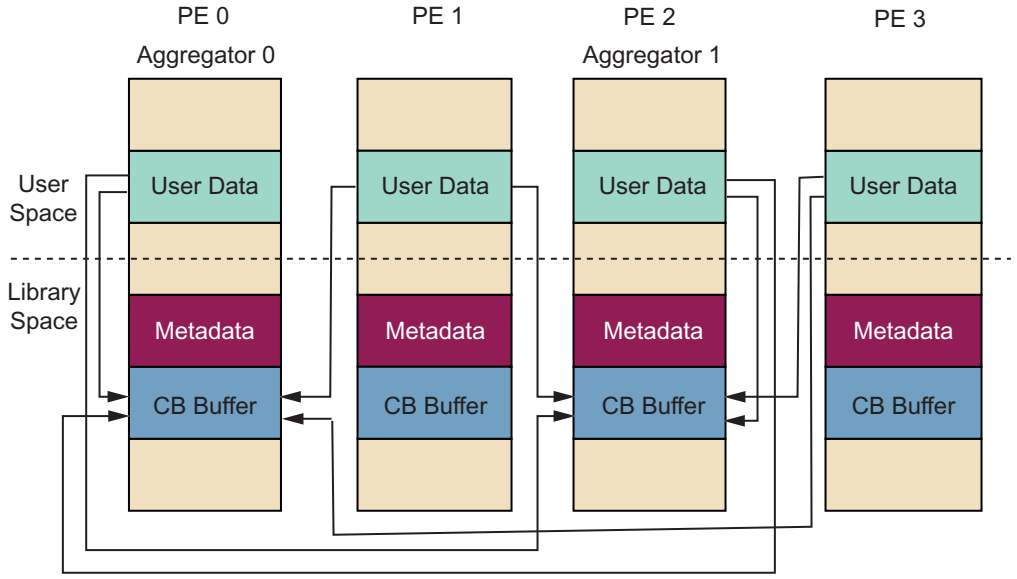


Figure 4 shows the aggregators, PEs 0 and 2, sending the data to the file system to be written to disk. PEs 1 and 3 are idle at this time.

Figure 4. Writing Data to a Lustre File

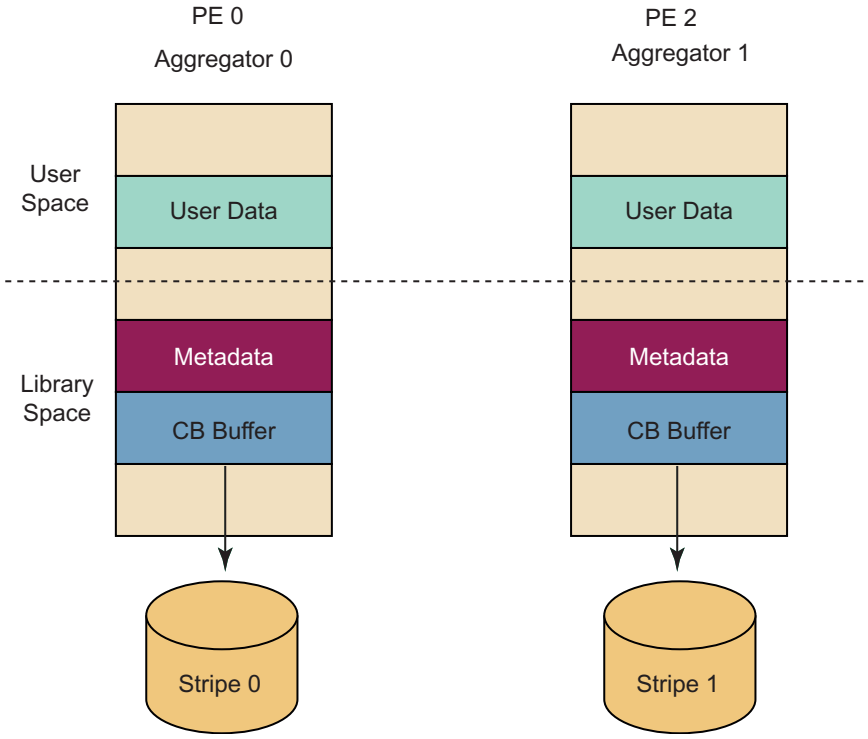


Figure 4 shows aggregator `0` writing to stripe `0` and aggregator `1` writing to stripe `1`. As described in the next section, this alignment is not always the case.

The process for reading a file is similar. As with the write process, metadata is exchanged to determine what data the aggregators will read from the file and which PEs to send it to. Then the aggregators read data from the file and send the data to the PEs that requested it.

Collective Buffering Alignment Algorithms

As described above, during collective buffering the I/O workload is divided among the aggregators. As a result of experience with and performance analysis of collective buffering, Cray added new algorithms in the MPT 3.1 and 3.2 releases for improved I/O workload distribution. In MPT 3.2, there are three algorithms to choose from.

Use the `MPICH_MPIIO_CB_ALIGN` environment variable (a Cray extension) to select the collective buffering alignment algorithm to be used.

The MPICH_MPIIO_CB_ALIGN environment variable sets the CB alignment algorithm:

MPICH_MPIIO_CB_ALIGN=0 or is not defined

An algorithm that divides the I/O workload equally among all aggregators without regard to physical I/O boundaries or Lustre stripes. This collective buffering method was used prior to MPT release 3.1. It is inefficient when the division of workload results in multiple aggregators referencing the same physical I/O block or when each aggregator has multiple segments of data with large holes between the segments.

With this algorithm, CB buffers larger than the 4 MiB default (for example, 10 MiB) may be beneficial if the user data needs it. The size of the CB buffer is controllable by the cb_buffer_size hint (see).

MPICH_MPIIO_CB_ALIGN=1

An algorithm that takes into account physical I/O boundaries and the size of I/O requests in order to determine how to divide the I/O workload when collective buffering is enabled. This can improve performance by causing the I/O requests of each aggregator to start and end on physical I/O boundaries and by preventing more than one aggregator from making reference to any given stripe on a single collective I/O call. However, unlike CB alignment algorithm 2, there is no fixed association between file stripe and aggregator from one call to the next.

MPICH_MPIIO_CB_ALIGN=2

An algorithm that divides the I/O workload into Lustre stripe-sized groups and assigns them to aggregators so that across all I/O calls each aggregator always accesses the same set of stripes and no other aggregator accesses those stripes. The file offset strictly determines across all the collective calls which aggregator will access the data at any given file offset. This fixed association between each aggregator and the portions of the file it accesses is sometimes called a *persistent file domain*[2].

This is generally the optimal collective buffering alignment algorithm on Lustre file systems [3] because it minimizes the Lustre file system extent lock contention on a single collective call and thus reduces system I/O time, and across all the collective calls, no data would reside in more than one aggregator's buffer. However, the overhead associated with this algorithm can in some cases exceed the system I/O time saved by using this method. This is the case if each PE is writing small (relative to stripe size) segments of data and the offsets for all PEs data are spread far apart (relative to stripe size).

---

2 This division of I/O workload was proposed by the Center for Ultra-Scale Computing and Information Security (CUCIS) team (http://cucis.ece.northwestern.edu/projects/MPIIO/). One of the strategies for domain assignment is to use the Lustre stripe size as a unit of domain and cyclically assign stripes to the set of aggregators.

3 In implementing this algorithm, Cray merged in some of the Lustre ADIO code from Sun Microsystems.

Collective Buffering Hints

The MPI 2.0 Standard provides for I/O hints. Hints are similar to compiler optimization directives in that they do not change the semantics of the program but can be used to guide optimizations. You specify hints either for selected portions of code in `MPI_Info_set()` calls (such as `MPI_Info_set(info, "romio_cb_write", "enable")` or for the entire application using the `MPICH_MPIIO_HINTS` environment variable (a Cray extension), such as `MPICH_MPIIO_HINTS=myfile:romio_cb_write=enable`.

Collective buffering is controlled by five hints:

`romio_cb_read`

Enables collective buffering on read when collective I/O operations are used. Valid values are `enable`, `disable`, and `automatic`. Default: `automatic`.

If you explicitly disable collective buffering by setting the hint to `disable`, collective buffering is not done, whether or not it might be helpful. Similarly, if you explicitly enable collective buffering by setting the hint to `enable`, collective buffering is done, whether or not it might be helpful. The hint setting `automatic` dynamically applies heuristics in determining if collective buffering is done.

`romio_cb_write`

Enables collective buffering on write when collective I/O operations are used. Valid values are `enable`, `disable`, and `automatic`. Default: `automatic`.

If you explicitly disable collective buffering by setting the hint to `disable`, collective buffering is not done, whether or not it might be helpful. Similarly, if you explicitly enable collective buffering by setting the hint to `enable`, collective buffering is done, whether or not it might be helpful. The hint setting `automatic` dynamically applies heuristics in determining if collective buffering is done.

`cb_buffer_size`

The buffer size in bytes. Default value is 4 MiB bytes. With CB alignment algorithm 2, this hint is ignored for writes.

`cb_nodes` Specifies the number of PEs that will serve as aggregators. Default: with CB alignment algorithms 0 and 1, default is 1 aggregator per Cray XT compute node. With algorithm 2, default is the stripe count if stripe count is less than or equal to the number of Cray XT compute nodes. If the number of compute nodes is less than the stripe count, you can increase the number of aggregators per compute node using the `cb_config_list` hint.

If, for example, a job is allocated 20 cores on a quad-core system and has specified that it is using all 4 cores on each node, then the default number of `cb_nodes` is 5. If you want more than one aggregator per Cray XT compute node, then you must change the `cb_config_list` hint.

With CB alignment algorithm 2, it is optimal to set `cb_nodes` to the same value as the `striping_factor` value (in other words, to the stripe count) to get the benefit of Lustre stripe alignment.

`cb_config_list`

Specifies by name which nodes are to serve as aggregators. The syntax for the value is: *#name1*`:`*maxPEs*`[,`*name2*`:`*maxPEs*`,.]#`, where *name* is either `*` (match all node names) or the name returned by `MPI_Get_processor_name()`, and *maxPEs* specifies the maximum number of PEs on that Cray XT compute node to serve as aggregators. Default: `*:1`.

You can use this hint to list nodes by node name, but there is rarely a need to do so on Cray XT systems because the set of compute nodes allocated to a job is usually homogeneous.

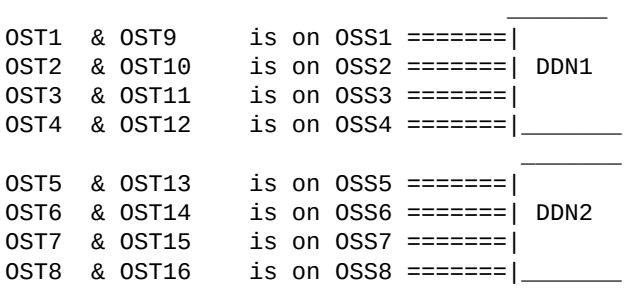
If you specify `*:2`, then, when aggregators are allocated, the first two aggregators are assigned to the first node in the alphabetically sorted list of node IDs, the second two aggregators are assigned to the second node, and so on until `cb_nodes` have been assigned. On Cray XT systems, one aggregator per node is usually best, but if you need more aggregators than nodes, you need to specify `*:2` or greater. If there are fewer nodes assigned the application than there are stripes in the file, something has to be adjusted. Either you need to change the `cb_config_list` value to `*:2` or more, or you need to reduce the stripe count to the number of nodes.

In addition, collective buffering is affected by two other hints. The `striping_factor` and `striping_unit` hints are specified by the MPI-2 Standard. These correspond directly to Lustre stripe count and stripe size, respectively. The MPT 3.2 release added support for these hints. Without using these hints, you could set the stripe count and stripe size of a file explicitly with the Lustre `lfs setstripe` command on a directory or with the command to create an empty file. With these hints, setting of the stripe count and stripe size can either be incorporated into the code with the `MPI_Info_set()` and `MPI_File_open()` calls or done with the `MPICH_MPIIO_HINTS` environment variable. See the `intro_mpi`(3) man page for details and syntax. For either of these methods to have any effect, the file cannot already exist when the `MPI_File_open()` call is made.

`striping_factor`

Specifies the number of Lustre file system stripes to assign to the file. Default: the site-configured default value for the Lustre file system. You can determine you system's default value by using the `lfs getstripe` command on a newly created Lustre file system directory.

You can use the `lfs setstripe` command to set the stripe count from `1` up to the number of OSTs on your file system. Bandwidth from the object storage servers (OSSs) to the disk controllers is a limiting factor. For example, if there are 2 DDN controllers and 4 OSSs attached to each controller and 2 OSTs on each OSS, there can be 16 stripes per file. This configuration can saturate the bandwidth to the controllers. Increasing to 4 OSTs per OSS will increase the maximum size of a file but will not increase the bandwidth to the controllers.



Generally the greater the number of stripes, the greater the performance. However, if there are more than 4 OSTs per OSS, this will not necessarily be the case. Having more than 4 OSTs per OSS is generally done only to support extremely large files. As a starting point for optimal performance, set the stripe count to 4 times the number of OSTs (assuming your file is large enough to need that many stripes) and adjust it up or down according to application-specific results.

`striping_unit`

Specifies in bytes the size of the Lustre file system stripes assigned to the file. This value also can be set using the `lfs setstripe` command. Default: the site-configured default value for the Lustre file system.

A stripe size of 1 MiB is generally optimal for Lustre file systems on Cray XT systems. A stripe size smaller than that gives less performance, and the performance curve is essentially flat from 1 MiB upward.

Collective Buffering and Data Access Patterns

Whether collective buffering helps and how much it helps depends in part on the data access patterns for the file. Here are some factors to consider:

- When the size of each record is less than the stripe size, I/O slows down. The smaller the record size, the slower it gets.
- With non-contiguous, small record I/O, do not expect high bandwidth, with or without collective buffering. However, if multiple PEs are writing to the same stripe, all three CB alignment algorithms can help significantly. If the access pattern is such that, on each collective call, each stripe is being accessed by at most one PE, then little is gained with collective buffering and it may be better to disable it, especially for writes.
- With contiguous, large (relative to file stripe size) record I/O, collective buffering generally does not help that much.

If you understand your application very well, the data access patterns might be quite clear to you. However, if you do not know the access patterns, you can use the `MPICH_MPIIO_XSTATS` environment variable (a Cray extension) to see the file offsets and record lengths

This undocumented environment variable in the MPT 3.2.0 release can be set to `0`, `1`, or `2` to give some level of statistics.

Note: The name `MPICH_MPIIO_XSTATS` with an `X` was chosen because this is experimental at this point without a clear API defined. The intent is that after some experience and user feedback, an `MPICH_MPIIO_STATS` (without the `X`) API will be defined to provide useful data.

These statistics are collected only for `MPI_File_write_all*` and `MPI_File_read_all*` calls. If `MPICH_MPIIO_STATS` is set to `1` and no statistics are reported, that means that no MPI-IO collective calls were made by the application.

Setting `MPICH_MPIIO_XSTATS` to `0`, the default, causes no statistics to be displayed. Setting it to `1` causes a summary by rank of how many POSIX system writes or reads were done, grouped into three size ranges: <= 1KiB, <= 1MiB, and > 1MiB. Ranks that did no writing or reading are not included. Also displayed are the number of clock ticks (as measured by the processor real time clock register) for the following three phases of MPI-IO:

1. `metadata` time - time spent within the MPI-IO routines deciding what hints to honor, what path to take, and, if collective buffering is enabled, the amount of time to exchange information among the ranks about what data will be sent to which aggregators.
2. `send` time - if collective buffering is enabled, time spent sending user data to the aggregators on write calls or time spent sending data from the aggregators on read calls.
3. `sysIO` time - time spent doing the POSIX `write()` or `read()` call.

Setting MPICH_MPIIO_XSTATS to 2 displays the MPICH_MPIIO_XSTATS=1 statistics and also issue a record to stdout for each MPI collective I/O call (if any) and for each POSIX system I/O call, as shown in this example:

```
<snip>
PE=00005  WCrec=00001  off=0101783616  len=0008478000  myfile
PE=00001  W_rec=00015  off=0111149056  len=0001048576  myfile
PE=00005  RCrec=00002  off=0080000000  len=0000040000  myfile
PE=00005  R_rec=00016  off=0080744448  len=0001048576  myfile
<snip>
```

Just a few lines have been selected. The first WCrec line is for the second (zero-based counting) write collective call (MPI_File_write_all()) made by PE 5 to myfile. The file offset is 0101783616 bytes, and the length of the transfer is 8478000 bytes. Some of the data for this write operation is sent to the aggregator on PE 1. The W_rec line is for the 16th system write() call made by PE 1. The file offset is 0111149056 bytes and the length of the transfer is 1048576 bytes, which is the stripe size in this example. Note that the ~8 MiB MPI_File_write_all() call gets distributed to various aggregators and written to the file system in up to 1 MiB segments, and a 1 MiB segment starts and ends on the boundaries for the stripe.

The report format is designed to make post processing with tools such as grep, perl, and sort easy. Note, however, that there is no set order to which PE writes its output first. It also happens in a small percentage of records that output from two PEs gets merged, so the output is not perfect. But there are several things you can glean from this data:

- If there are no WCrec or RCrec lines, then no collective write or collective read calls were made. If you want to take advantage of collective buffering, you need to modify the code to make the collective I/O calls.
- The off and len values in the WCrec and RCrec lines reflect the view of the data at the application level. These are usually not power-of-2 size or stripe aligned.
- If the len values in the W_rec and R_rec lines are the same as the stripe size, then you know that full stripes of data were being written or read, which is optimal. This shows that collective buffering has done its job of redistributing the data for more efficient Lustre file system I/O.
- If the len values in the W_rec and R_rec lines are less than the stripe size, the segment of data after the data redistribution does not fill a full stripe. This can be either because the original length of the transfer was less than a stripe or the data spanned a stripe boundary and was divided into two or more pieces and assigned to two or more aggregators.
- If the len in the W_rec and R_rec lines are greater than the len in the WCrec or RCrec lines, then collective buffering successfully merged two or more segments of data from separate PEs.

Note: Setting MPICH_MPIIO_XSTATS to 2 can generate a lot of output, so if you want to use this option, you might want to scale down your application for this test.

Benchmark Results

Here are some benchmark results comparing the I/O bandwidth in MiB/second for four modes: without collective buffering (that is, collective buffering disabled), and with collective buffering alignment algorithms 0, 1, and 2. For benchmarks 1 and 2, we used the publicly available [4] I/O benchmark IOR (see http://sourceforge.net/projects/ior-sio/).

For benchmark 1, we show explicit examples of setting the `MPICH_MPIIO_HINTS` and `MPICH_MPIIO_CB_ALIGN` environment variables and the `aprun` command. This should help those unfamiliar with setting the hints. For more information, see the `intro_mpi`(3) man page.

Benchmark 1: IOR using the MPI-IO API with non-power-of-2 blocks and transfers — in this case blocks and transfers of 1,000,000 bytes and a strided access pattern. The size of the file is large enough that the file cannot be held in cache, thus better testing the file system bandwidth.

For comparison, a POSIX shared-file case is also shown. The non-power-of-2 aspect of this benchmark, which would be more typical of real application access patterns, greatly benefits from collective buffering that aligns the POSIX I/O with the Lustre stripe boundaries. (POSIX file-per-process mode is not shown, but it would perform better than shared-file mode. However, the POSIX file-per-process mode has its own drawbacks, as described in Introduction.)

Configuration: 32 PEs with 8 PEs per node, 16 stripes, 16 aggregators, 3220 segments, and a 96 GiB file. IOR run on a Cray XT5 8-core system.

The results for benchmark 1:

Test	Mode	MiB/sec
1	POSIX shared-file	`420`
2	Without collective buffering	`421`
3	CB = 0	`262`
4	CB = 1	`341`
5	CB = 2	`1629`

The `aprun` arguments, IOR arguments, and MPI-IO hints used in each test are:

1. POSIX shared-file mode

   File striping: `stripe_count=16`, `stripe_size=1048576`

   MPI-IO hints: N/A for POSIX.

   The `aprun` command and IOR arguments:

   ```
   % aprun -n 32 -N 8 ./IOR -a POSIX -w -C -e -g -s 3220 -b 1000000
   -t 1000000 -i 8 -o myfile
   ```

[4] Under the terms of the GNU General Public License.

2. MPI-IO without collective buffering

   File striping: `stripe_count`=16, `stripe_size`=1048576

   MPI-IO hints, `aprun` command, and IOR arguments:

```
% export MPICH_MPIIO_HINTS=myfile:romio_cb_write=disable:romio_ds_write=disable
% aprun -n 32 -N 8 ./IOR -a MPIIO -w -c -C -g -s 3220 -b 1000000
-t 1000000 -i 8 -o myfile
```

3. MPI-IO with CB alignment algorithm = 0

   File striping: `stripe_count`=16, `stripe_size`=1048576

   MPI-IO hints:

```
% export MPICH_MPIIO_HINTS=myfile:romio_cb_write=enable:
romio_ds_write=disable:cb_buffer_size=2097152:cb_nodes=16:cb_config_list=*:
% export MPICH_MPIIO_CB_ALIGN=0
% aprun -n 32 -N 8 ./IOR -a MPIIO -w -c -C -g -s 3220 -b 1000000
-t 1000000 -i 8 -o myfile
```

4. MPI-IO with CB alignment algorithm = 1

   File striping: `stripe_count`=16, `stripe_size`=1048576

   MPI-IO hints, `aprun` command, and IOR arguments:

```
% export MPICH_MPIIO_HINTS=myfile:romio_cb_write=enable:
romio_ds_write=disable:cb_buffer_size=2097152:cb_nodes=16:cb_config_list=*:
% export MPICH_MPIIO_CB_ALIGN=1
% aprun -n 32 -N 8 ./IOR -a MPIIO -w -c -C -g -s 3220 -b 1000000
-t 1000000 -i 8 -o myfile
```

5. MPI-I/O with CB alignment algorithm = 2

   File striping: `stripe_count`=16, `stripe_size`=1048576

   MPI-IO hints, `aprun` command, and IOR arguments:

```
% export MPICH_MPIIO_HINTS=myfile:romio_cb_write=enable:
romio_ds_write=disable:cb_buffer_size=2097152:cb_nodes=16:cb_config_list=*:
% export MPICH_MPIIO_CB_ALIGN=2
% aprun -n 32 -N 8 ./IOR -a MPIIO -w -c -C -g -s 3220 -b 1000000
-t 1000000 -i 8 -o myfile
```

Benchmark 2: IOR using the MPI-IO API with non-power-of-2 blocks and transfers — same as benchmark 1 except, in this case, transfers of 10,000 bytes. The much smaller record size compared with that in benchmark 1 shows that smaller record I/O will perform less well but collective buffering with alignment algorithms 1 and 2 help. The overhead of exchanging the metadata for many small records is what keeps the benefit from being greater.

Configuration: 32 PEs with 8 PEs/node, 8 stripes, 3220 segments, and a 96 GiB file. IOR run on a Cray XT5 8-core system.

The `aprun` command and IOR arguments are:

```
% aprun -n 32 -N 8 ./IOR -a MPIIO -w -c -C -g -s 3220 -b 1000000
-t 10000 -i 8 -o myfile
```

The results for benchmark 2:

Test	Mode	MiB/sec
1	POSIX shared-file	75
2	Without collective buffering	78
3	CB = 0	23
4	CB = 1	125
5	CB = 2	148

Benchmark 3: HDF5-format dump file from all PEs, total file size 6.4 GiB. Mesh of 64,000,000 bytes and 32,000,000 elements, with work divided among all PEs. Original problem was scaling very poorly. For example, without collective buffering, 8,000 PEs took over 5 minutes to dump. All three CB alignment algorithms helped. Note that disabling data sieving was necessary.

Configuration: from 1,000 to 8,000 PEs, 8 stripes, 8 aggregators. IOR run on a Cray XT5 8-core system.

The results for benchmark 3:

Test	Mode	MiB/sec for 1,000 PEs	MiB/sec for 2,000 PEs	MiB/sec for 4,000 PEs	MiB/sec for 8,000 PEs
1	Without collective buffering	129	52	26	13
2	CB = 0	462	429	287	224
3	CB = 1	488	401	307	243
4	CB = 2	504	372	310	256

Benchmark 4: HYCOM MPI-2 I/O (HYbrid Coordinate Ocean Model) with 5,107 PEs and, by application design, a subset of the PEs (88) do the writes. With collective buffering, this is further reduced to 22 aggregators writing to 22 stripes. Run on a Cray XT5 8-core system.

The results for benchmark 4:

Test	Mode	MiB/sec
1	Without collective buffering	183
2	CB = 0	336
3	CB = 1	557
4	CB = 2	3742

Future Work

The evolution of collective buffering optimization, not unlike optimizations of computation by compilers, shows that optimization improvements can be a never-ending process. Here are some topics for future work on collective buffering on Cray XT systems. There is no commitment from Cray that any of these will be done or that any of these will in fact improve performance, but it is currently in our plans to work on at least some of them.

- Reduce the metadata overhead. As described earlier in the paper, metadata is exchanged among all PEs as a step in the collective buffering process. The overhead is significant, so any reduction in this overhead will help. Because future Cray systems will have a faster interconnect than the current Cray XT systems, the interconnect speedup will have a direct benefit for collective buffering, both in the metadata exchange phase and the data exchange phase.
- Allocate page-aligned collective buffers. Analysis of the size of the I/O from the aggregators with CB alignment algorithm 2 shows that for many applications, there are many full-stripe-sized records being written. If the buffer is allocated to be page aligned and if the buffer is full, direct I/O could be done, saving the extra copy to kernel space.
- Buffer successive I/O calls within the collective buffering buffers. As described earlier in the paper, on write, data is copied from user data space to the aggregators' buffers and then sent to the file system. If the data access pattern is such that on one `MPI_File_write_all()` call the buffer is only partially full but on subsequent calls, the buffer gets more full, performance would improve if the aggregator can safely hold on to the data until either the buffer is full or it must send it to the file system for some other reason, such as an `MPI_File_sync()` call or an `MPI_File_close()` call.
- Topologically aware assignment of aggregators. Because aggregators must send data through the interconnect to get to the file system, placing aggregators closest to the I/O nodes could reduce network traffic and improve overall performance.
- Improvements on collective reads. The CB alignment algorithm 2 only changed the algorithm for writes. Further analysis of applying this algorithm or similar for reads is needed.
- Better heuristics for selecting defaults for hints. The default values for the collective buffering and data sieving hints are all `automatic`. The intent is to let MPI-IO make its best guess as to the best mode to run. The heuristics currently used probably no longer picks the best values.

Conclusions

- For most applications, use MPI-IO with collective buffering. Collective buffering alignment algorithm 2 offers the greatest potential for improving I/O performance.
- More work on improving I/O performance is needed. This is an on-going project.

Authors

David Knaak Software Engineer Cray Inc. Programming Environment (651) 605-9014 knaak@cray.com

Dick Oswald Technical Writer Cray Inc. Customer Documentation (651) 605-9053 oswald@cray.com

References

H. Shan and J. Shalf, *Using IOR to Analyze the I/O Performance of XT3*, in the Cray User Group Conference, May 2007.

An Extended Two-Phase Method for Accessing Sections of Out-of_core Arrays by Rajeev Thakur and Alok Choudhary.

Jeff Larkin, Mark Fahey *Guidelines for Efficient Parallel I/O on the Cray XT3/XT4*.

HYCOM (http://hycom.rsmas.miami.edu/)

Cray XT Programming Environment User's Guide

`intro_mpi`(3) man page.