Practical Examples for Efficient I/O on Cray XT Systems

Jeff Larkin, Cray Inc. <larkin@cray.com>

With graphs from:

Lonnie Crosby, University of Tennessee and Galen Shipman, ORNL

ABSTRACT: The purpose of this paper is to provide practical examples on how to perform efficient I/O at scale on Cray XT systems. Although this paper will provide some data from recognized benchmarks, it will focus primarily on providing actual code excerpts as specific examples of how to write efficient I/O into an HPC application. This will include explanations of what the example code does and why it is necessary or useful. This paper is intended to educate by example how to perform application checkpointing in an efficient manner at scale.

KEYWORDS: I/O, Lustre, MPI-IO, XT5, XT4, XT3, Parallel, IOR

Motivation

Over the past few years I have presented on the subject of application I/O at several Cray customer sites and at the Cray Users Group meeting. In each of these presentations I have shown data collected from I/O micro-benchmarks and interpreted at a high level in regards to application performance. Often the audience of these presentations lacks experience writing effective I/O code and, therefore, resorts to simpler, less effective methods. The purpose of this paper is to provide examples of I/O code that can be effectively used on Cray XT systems as a supplement to the previously mentioned, higher level presentations.

It should be noted that benchmarking I/O is extremely difficult. Because I/O operations make use of two shared resources, the network and the filesystem, I/O operations are probably the most difficult to accurately measure and predict. The fact that each application has different needs and I/O implementations also makes it difficult to accurately predict how well an application's I/O will perform in relation to I/O benchmarks. I/O benchmarks and kernels, like the ones used in this paper, are written with I/O in mind, while applications are written with their calculations in mind. Readers should not assume that their application will perform like an I/O kernel, but rather apply the ideas from these benchmarks and kernels to improve the I/O operations of their applications.

Example I/O Data

Before discussing I/O code examples, it is useful to establish a baseline for I/O performance comparisons. The IOR benchmark is very commonly used for establishing a high water mark for filesystem performance. In many ways, IOR does everything right to maximize I/O performance, so it is often the basis for

filesystem comparisons. The graphs in Figure 1 (Crosby) (Shipman) show IOR File Per Process performance on two separate large Cray XT5 systems. Since filesystem performance varies greatly from machine to machine and even between filesystems on the same machine, we will ignore the exact I/O rates of these graphs and instead concentrate on the shape of the graphs. It should be noted that these two graphs do not include open times, which scales particularly poorly with file-per-process operations and would be included in the time necessary to checkpoint an application.

The most striking note about the above graphs is the consistent shape, where performance increases rapidly as clients are added up to a certain threshold, after which performance degrades to a low asymptotic value. This should illustrate that below some threshold a program can utilize simplistic I/O techniques and achieve reasonable I/O performance, but as programs scale to many thousands of processors they will need to intelligently scale their I/O. Intelligently scaling application I/O may include using I/O aggregators, multiple shared files, or other techniques that computational scientists may lack the skills to implement. The sections below will demonstrate these techniques and explain their benefits and limitations.

Striping and Data Alignment

Before discussing different I/O methods, I feel it is important to discuss two simple, and commonly over-looked, ways of improving I/O performance: adjusting stripe count and stripe size. Because Lustre is a parallel filesystem, it is critical to exploit lustre striping intelligently. Default striping values may differ from filesystem to filesystem, but it is unlikely to be the best values for your application. The default stripe count for a lustre filesystem is commonly configured to 4, which is safer than 1 or all and performs better for serial operations than either, but is not ideal for most parallel jobs. At a high-level, large shared files generally need a large stripe count in order to perform optimally and files written on a per-process basis generally require a small stripe count. Simply changing lustre striping from the default value will often have a significant effect on the performance of parallel I/O operations. Figure 4 shows the difference in performance writing from just 1440 processor to a shared file when striped to the default count (4) and widest stripe (144).

The simplest way to change striping is via the `lfs` command, but setting the striping programmatically is an often-requested feature. Unfortunately the Lustre filesystem does not include a simple user API, but the code in Figure 2 demonstrates one way to set the striping of a new file[1] from within an application. This code should only be executed from one processor to avoid overwhelming the metadata server. Since version 3.2.0 of the xt-mpt library, users may also set the striping for MPI-IO files via the `striping_factor` and `striping_unit` hints.

[1] Lustre striping cannot be changed on an existing file.

It is also important for users to realize that adjusting the size of lustre stripes may affect the performance of I/O operations. As one would expect, larger stripe sizes often perform better than smaller, although increasing stripe size too much can negatively affect performance. Users should also be careful to ensure that multiple writers do not write to the same stripe. The simplest way to avoid this misalignment is to ensure that the size of I/O buffers is a multiple of the lustre stripe size. Additionally, lustre appears to prefer I/O operations on power of 2 data buffers.

In addition to properly striping files, properly aligning data along page boundaries can have a noticeable effect on I/O performance. Figure 5 shows the results of the same benchmark code running with and without page-aligned buffers. In general, page-aligning I/O buffers made operations no worse and often improved bandwidth by varying amounts. Data alignment in C/C++ can be achieved via the `posix_memalign` command and in FORTRAN by over-allocating an array and using the `loc` function to find the start of a new page.

POSIX File-per-process

Writing individual files from each processor is likely the simplest I/O method to implement and is among the most commonly used methods. When evaluating peak filesystem performance, this method generally has the best peak performance, but has numerous pitfalls at scale, including high stress on the metadata server and user inconvenience from file count. One significant benefit to this method is that it easily allows more OSTs to be used on filesystems with greater than 160 OSTs available[2].

As noted, FPP I/O operations are very stressful on the Lustre metadata server (MDS). Figure 6 shows the cost of opening a file from every processor versus opening a single, shared file. Notice that open time increases with processor count when using either technique, but much worse with file-per-process. Since a single checkpoint operation will almost certainly include opening one or more files, the poor scaling behavior or open operations should not be disregarded. Figure 11 gives a simplified example of how to perform a buffered, POSIX FPP write.

POSIX Shared Files

Using a single, shared file among all processes addresses some, but not all of the limitations of file-per-process. While using a single file may be more convenient for the user, it does not completely eliminate MDS load or reduce the number of clients simultaneously writing to the OSTs. Figure 6 compares open times when using POSIX FPP or POSIX shared file. Using a shared file generally performs slightly worse than individual files, but this may be a worthwhile trade-off for user convenience. Implementing a shared file in C using POSIX I/O functions is fairly straightforward, but does require each processor to calculate its own offset in the file. In applications with uniform I/O for each processor, this step is trivial, but it may be more difficult when each processor must write a different

[2] Lustre currently restricts the number of OSTs for a given file to 160

amount of data. Figure 7 shows a simplified example of how to implement a POSIX shared file between all processes. Each process will open the same file, calculate and seek to its portion of the file, write, and then close. One may wish to use `open_striped` from Figure 2 on one process before opening from the remaining processes.

FORTRAN Direct I/O

Shared file I/O may also be implemented in FORTRAN using direct access files. Just as with POSIX shared files, each process must calculate its own offset in the file. Figure 8 shows a simplified example of how to implement a FORTRAN direct access file. At time of publication, I am still refining this code to improve performance, but the code is functional. Similar to POSIX shared file, each process must determine its file offset. Unlike with the POSIX example, record size must be provided at open time and each write must be done in terms of record numbers within the file. As with POSIX I/O, one should try to write large, contiguous records of data for best performance.

MPI-IO

The MPI2 standard includes an extension for performing parallel I/O (MPI-IO). While one may find implementing MPI-IO operations somewhat confusing, this may be an appropriate I/O library for applications already implemented using MPI for communication.

One particular area of confusion for MPI-IO users is that operations may be handled using an explicit file offset, individual file pointers, or shared file pointers. The choice of which of these methods does not appear to affect I/O performance, so users should choose whichever method fits best with their code. Briefly, explicit offsets require a file offset be provided at write or read time, while individual pointers allow the offsets to be set when a file is opened. Shared file pointers are less ideal for parallel applications, as they inherently serialize I/O operations for applications that cannot pre-calculate file offsets, leading to very poor performance.

On the surface, MPI-IO does not appear to be better than simply writing a shared file via other means, but MPI-IO provides several options behind the scenes in the form of *hints* to improve on other methods. Just as the name implies hints guide the library to make better decisions about I/O operations. One family of hints that prove extremely useful at scale are those involved in collective buffering.

Collective buffering is a behind-the-scenes approach to aggregating I/O operations to a subset of the total processors. One may choose to write one of many different subsetting approaches by hand, but MPI-IO collective buffering provides an automatic solution. The choice between MPI-IO collective buffering and hand-coded aggregation is one of simplicity versus control. Users can enable collective buffering by setting the `cb_nodes` hint, which tells the library how many I/O aggregators to use, and the `romio_cb_{read,write}` hint to enable. One may also choose to adjust the size of the collective buffer on each aggregator via the `cb_buffer_size`

hint. It should be noted that as of XT-MPT/3.1, collective buffering is used by default with the number of aggregators set to one per node. In very large jobs, users may wish to adjust the value of `cb_nodes` to fewer than the total compute nodes. When adjusting the value of cb_nodes, it is suggested that the value be divisible by the stripe count of the file. Collective buffering and subsetting may allow applications run at large scales perform I/O closer to the peak of the I/O scaling curve.

In (Pagel) the author shows results comparing collective I/O versus direct POSIX I/O. He also summarizes several new enhancements to the Cray MPI-IO implementation that can provide significant benefits over other solutions.

For those unfamiliar with the MPI-IO interface, Figure 9 shows sample MPI-IO code for opening a shared file, setting a file view, performing a collective write, and closing the file. It is suggested that one use collective versions of MPI-IO write and read routines so to take advantage of the collective buffering capabilities within the library. Figure 9 is a very simple example that will use the library defaults for hint. The code will also be affected by the Cray-specific, MPICH_MPIIO_HINTS environment variable. If one would like to set the hints directly within the code, Figure 10 gives an example of how to do that.

HDF5 and NetCDF

The HDF5 and NetCDF libraries provide additional metadata and functionality that many users desire. Beginning with version 4.0, NetCDF is built over top of the HDF5 library, so I will focus primarily on HDF5 in this section.

For many, the additional metadata and portability provided by HDF5 makes using the library desirable. HDF5 allows the user to represent the data structures within the code, rather than just the raw data. This additional feature means that each application uses HDF5 very differently, making it difficult to build example kernels for using HDF5 effectively. I can, however, make the following observations about using HDF5.

HDF5 may be built to support serial or parallel I/O, which is layered over POSIX or MPI-IO routines respectively. Use of serial HDF5 to write or read large amounts of data in a large parallel job simply does not make sense and is analogous to any other serial method. The performance of such I/O operations will be fully limited by the performance of the single client, unless multiple files are used. Since the parallel HDF5 interface is built on top of MPI-IO, the same hints are available to users and the same guidance applies. Users should perform large I/O operations and explore collective buffering options. Figure 3 (Crosby) shows results from IOR and demonstrates that HDF5 is capable of I/O rates comparable to other methods when used efficiently. As stated previously, these results carry the caveat that IOR is written specifically to test I/O rates, so it may not be representative of how a given application may wish to use HDF5.

Multiple Shared Files

It should be clear from the previous sections that both per-process and shared

files both have scaling limitations. As discussed, a single shared file is limited to 160 OSTs in the current version of lustre. Using individual files per process solves this limitation, but with additional metadata overhead and inconvenience. Combining both of these techniques by using multiple shared files, via any of the APIs featured above, may allow an application to mitigate the limitations of both schemes. On a filesystem consisting of greater than 160 OSTs, using multiple shared files would allow an application to utilize more OSTs without the metadata or convenience issues of file-per-process. When exploiting a multiple-shared-file technique, one should look for existing groups within the application, as the data within these groups is likely related, so it will likely be contiguous within a file.

Future Work

As I was completing work for this paper some recent changes in the XT-MPT library related to the performance of MPI-IO collective buffering operations. These changes are enabled using the MPICH_CB_ALIGN environment variable. I did not have sufficient time to investigate the performance implications of this environment variable, but intend to do so. Preliminary results related to this feature are available in (Pagel). I also intend to further improve the FORTRAN direct I/O example and implement a multiple shared files example kernel.

The code written for this paper was intended to be sharable and modifiable for the education and use of the CUG community. At the time of writing, this code has not yet been made available, but it is my intention to make the source code available. When it has been made available, a link will be posted at http://users.nccs.gov/~larkin/.

About the Author

Jeff Larkin is a member of the Cray Supercomputing Center of Excellence located at Oak Ridge National Laboratory. He has Bachelor and Master of Science degrees in Computer Science and has worked for Cray since 2005. His specialties include performance analysis and optimization, application porting debugging, and scaling, and I/O issues. He can be contacted via email at larkin@cray.com.

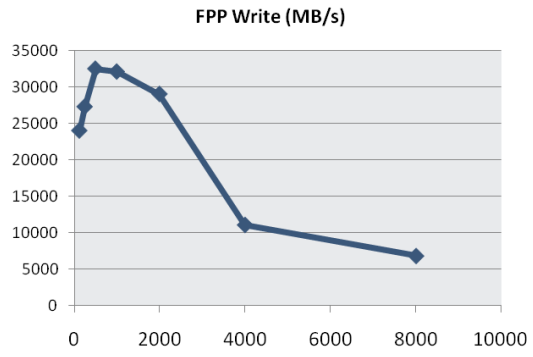
References

Crosby, Lonnie. "Performance Characteristics of the Lustre File System on the Cray XT5 with Regard to Application I/O Patterns." Cray User Group. Atlanta, GA, 2009.

Pagel, Mark. "Scaling the MPT Software on the XT5 and Other New Features." Cray Users Group. Atlanta, GA, 2009.

Shipman, Galen. "Introduction to Lustre and NCCS Spider Parallel File Systems for XT5." 2009. <http://www.nccs.gov/wp-content/training/2009_crayxt_workshop/apr15/GalenShipman.pdf>.

Figures and Charts



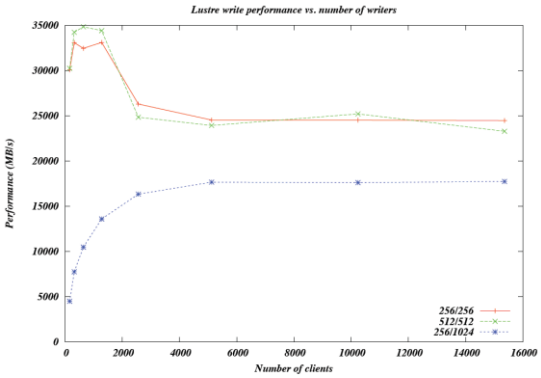


Figure 1 IOR File Per Process Baseline

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <lustre/lustre_user.h>

int open_striped(
    char *filename,
    int  mode,
    int  stripe_size,
    int  stripe_offset,
    int  stripe_count)
{
  int fd;
  struct lov_user_md opts = {0};
  opts.lmm_magic = LOV_USER_MAGIC;
  opts.lmm_stripe_size = stripe_size;
  opts.lmm_stripe_offset = stripe_offset;
  opts.lmm_stripe_count = stripe_count;

  /*
  ** Use O_LOV_DELAY_CREATE to delay file creation until
  ** after LL_IO_LOV_SETSTRIPE is used to set striping
  */
  fd = open64(filename, O_CREAT | O_EXCL |
              O_LOV_DELAY_CREATE | mode, 0644);
  if ( fd >= 0 ) ioctl(fd, LL_IOC_LOV_SETSTRIPE, &opts);

  return fd;
}
```

Figure 2 Setting Lustre Striping Programatically

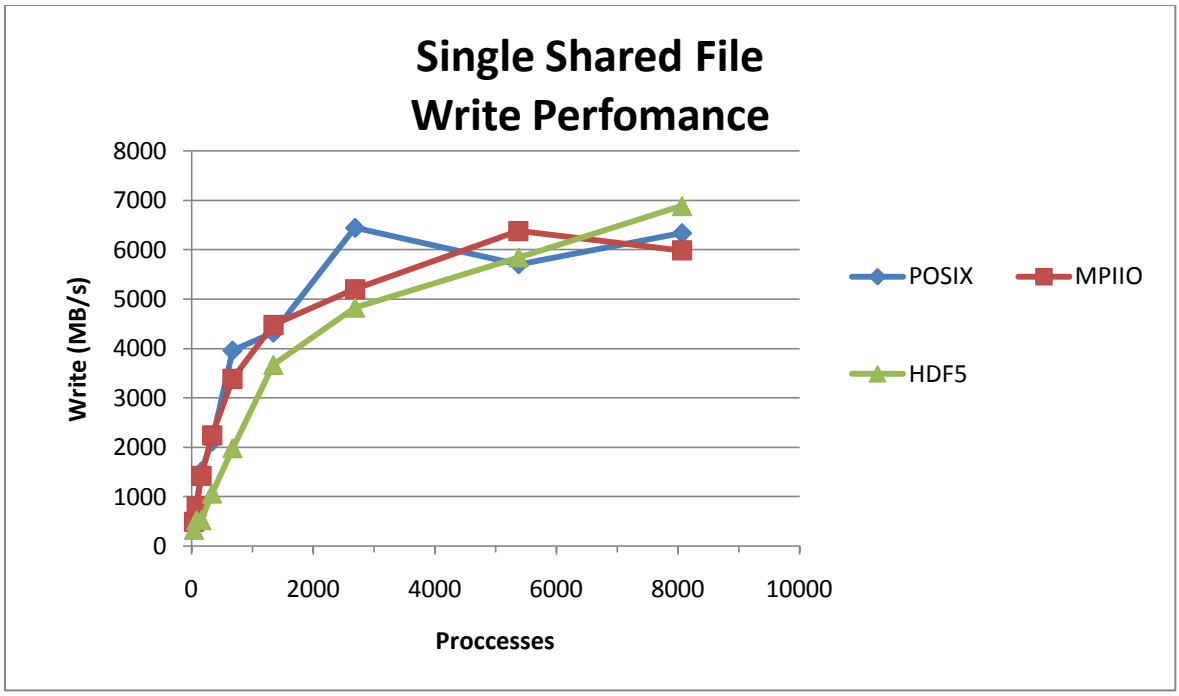


Figure 3 Shared File Write Performance Comparisons

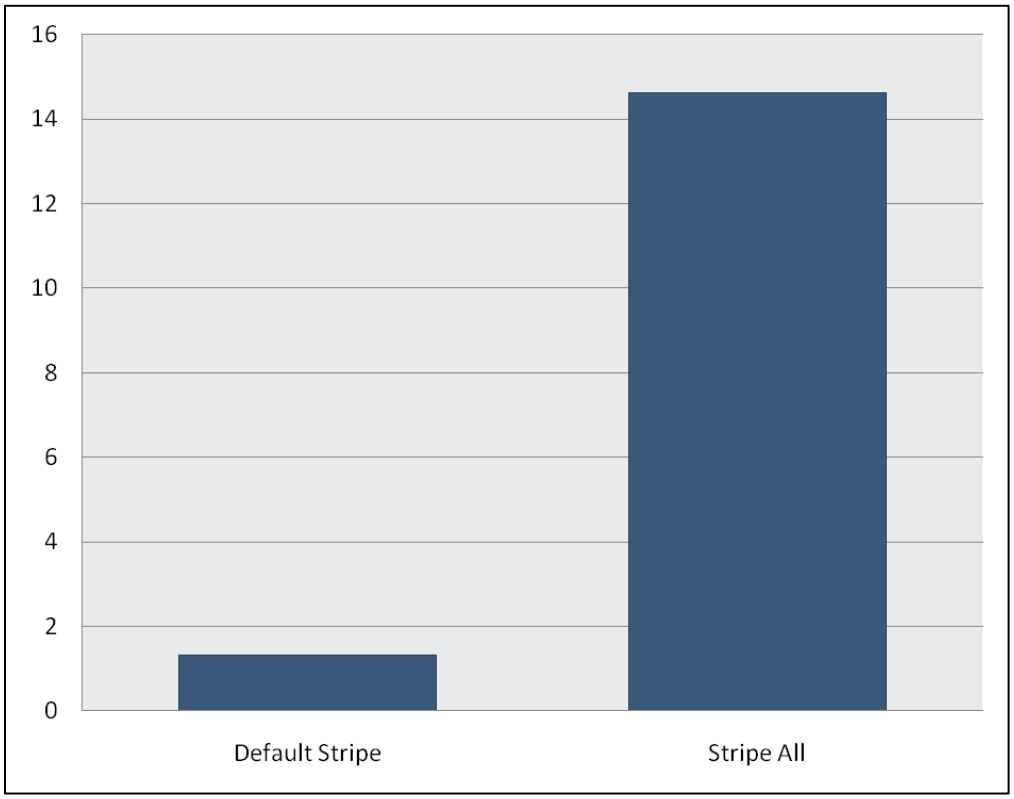


Figure 4 Single File parallel write performance: Default Striping vs. Wide Striping

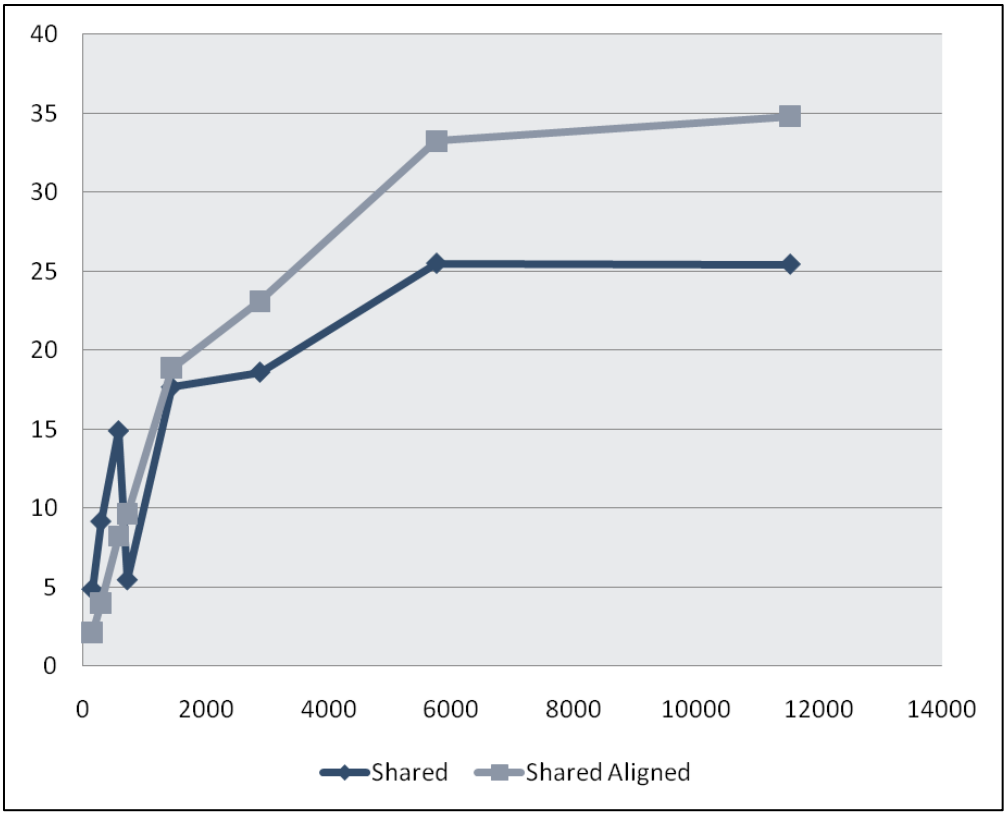


Figure 5 POSIX shared file with and without page-aligned buffers.

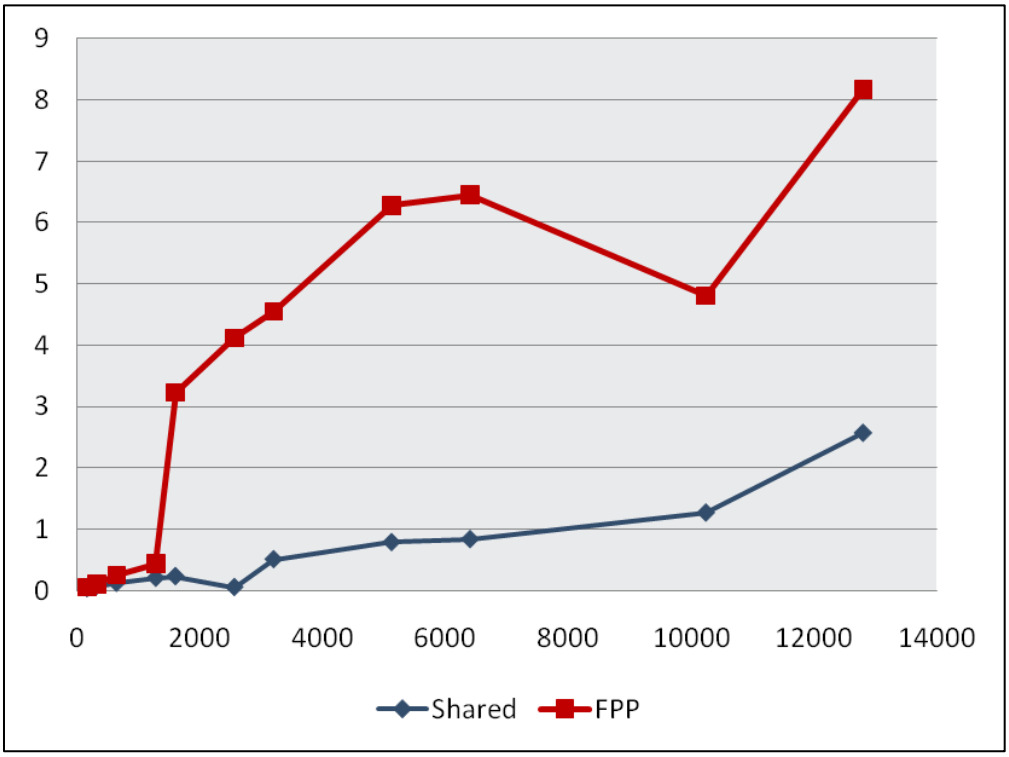


Figure 6 Comparison of file open time for file-per-process and single shared file.

```
fd = open64("test.dat", mode, 0644);
/* Seek to start place for rank */
ierr64 = lseek64(fd, commrank*iosize, SEEK_SET);
remaining = iosize;
/* Write by buffers to the file */
while (remaining > 0)
{
  i = (remaining < buffersize ) ?
      remaining : buffersize;
  /* Copy from data to buffer */
  memcpy(tmpbuf, dbuf, i);
  ierr = write(fd, tmpbuf, i);
  if (ierr >= 0) {
    remaining -= ierr;
    dbuf += ierr;
  } else
  {
    MPI_Abort(MPI_COMM_WORLD, ierr);
  }
}
close(fd);
```

Figure 7 Simplified POSIX Shared File code example

```
! Establish Sizes
reclength = 8*1024*1024
iosize = reclength * 10
! Starting Record For Rank
recnum = (iosize * myrank)/reclength
recs = iosize/8
numwords = recs/10
open(fid, file='output/test.dat',
   status='replace', form='unformatted',
   access='direct', recl=reclength,
   iostat=ierr)
! Write a record at a time to the file
do i=1,recs,numwords
  write(fid, rec=recnum, iostat=ierr)
    writebuf(i:i+numwords-1)
    recnum = recnum + 1
end do
close(fid)
```

Figure 8 Simplified FORTRAN shared direct-access file code example

```
int mode;
MPI_File fh;
MPI_Status status;

/* Open the file */
mode = MPI_MODE_CREATE|MPI_MODE_RDWR;
MPI_File_open(comm, "output/test.dat", mode, info, &fh);

/* Set the view of the file */
MPI_File_set_view(fh, commrank*iosize, MPI_DOUBLE,
  MPI_DOUBLE, "native", MPI_INFO_NULL);

/* Perform collective write from all. */
MPI_File_write_all(fh, dbuf, iosize/sizeof(double),
  MPI_DOUBLE, &status);

/* Close the file */
MPI_File_close(&fh);
```

Figure 9 Performing MPI-IO collective write.

```
char tmps[24];
MPI_Info info;

MPI_Info_create(&info);

/* Adjust buffer size */
snprintf(tmps,24, "%d",buffersize);
MPI_Info_set(info, "cb_buffer_size", tmps);

/* Adjust number of I/O aggregators */
snprintf(tmps,24,"%d",numagg);
MPI_Info_set(info, "cb_nodes", tmps);

/* Enable collective writing */
MPI_Info_set(info, "romio_cb_write", "enable");

/* Open the file */
MPI_File_open(comm, "output/test.dat", mode, info, &fh);
```

Figure 10 Setting MPI-IO hints

```
double *tmpbuf;
int mode, fd, remaining, ierr;
char tmps[24];

snprintf(tmps,24,"test%05d.dat", commrank);
fd = open_striped(tmps, mode, 0, -1, 1);
remaining = iosize;
while (remaining > 0)
{
  i = (remaining < buffersize ) ? remaining : buffersize;
  memcpy(tmpbuf, dbuf, i);
  ierr = write(fd, tmpbuf, i);
  if (ierr >= 0) {
    remaining -= ierr;
    dbuf += ierr;
  } else
  {
    fprintf(stderr,    "[%d]    write:    %s\n",    commrank,
strerror(errno));
    MPI_Abort(MPI_COMM_WORLD, ierr);
  }
}
close(fd);
```

Figure 11 POSIX File-per-process write