Optimizing I/O

Ritu Arora, Wayne State University / Venra Tech Inc.

Email: ritu@wayne.edu / ritu@venratech.com

1. OVERVIEW

Every useful scientific application does some type of Input/Output (I/O). Such applications could be reading initial conditions or data sets for processing, or could be writing numerical data from simulations, or could be writing checkpoints every so often to be used in the event of any hardware failure. This article presents some of the techniques for optimizing I/O in scientific applications, including optimizing I/O when working with AI frameworks. Topics such as buffering, and memory-mapped files are discussed in the general context. The concepts of prefetching, parallelizing data extraction, data loaders, data generators, and buffer checkpointing are discussed at a high-level in the context of AI frameworks. The topic of reducing I/O bottlenecks in parallel applications is discussed as well. The FLASH astrophysics code is used to demonstrate the impact of optimizing I/O on its performance.

2. INTRODUCTION

There are two main reasons due to which we should consider optimizing I/O: (1) to reduce the overall execution time of our applications, and in doing so, reduce the associated computation costs, if any, and (2) to reduce the burden on the shared High Performance Computing (HPC) systems on which our jobs may be running.

The total execution time of a serial (or sequential) **application can be broken down into (1) computation time and (2) <u>the I/O time.</u>** In the case of **distributed or parallel applications**, **the total execution time can be broken down into** three components: **(1) time taken for communication between the processes engaged in the parallel computation, (2) time taken for the computation themselves and, (3) <u>time taken in I/O.</u>** Hence, in both serial and parallel applications it is important that we optimize I/O to reduce the overall execution time of the applications and enhance their performance. However, it is often seen that optimizing I/O is an afterthought.

The large-scale, HPC systems funded by the National Science Foundation (NSF) (e.g., Expanse at SDSC and Stampede2 at TACC) and other agencies are shared computing systems. When working on such shared computing systems, it is important to know that certain patterns of I/O from the compute jobs can stress out the filesystems. Due to the stressful I/O activities, depending upon the type of the filesystem and its failover mechanism, the filesystem can become unavailable for the entire user community. Hence, it is very important that we are aware of how we are doing the I/O from our applications and do it optimally for not only enhancing the applications' performance but also for maintaining good citizenship on the shared computing systems.

3. SOME GENERAL EXAMPLES OF INEFFICIENT I/O

Following are some examples of inefficient I/O in serial and parallel applications:

1. Opening and closing files inside loops
2. Large-scale reading and writing to the shared filesystems directly from the jobs that are running on HPC platforms
3. Calling an I/O function every time a value needs to be read from a file
4. Choice of suboptimal functions/methods/APIs/algorithms for the tasks at hand
5. Attempting to access large datasets on a different system (secondary or tertiary storage system) while the compute jobs are running

4. SOME GENERAL STRATEGIES FOR ADDRESSING INEFFICIENT I/O

Let us now review some solution strategies for each of the examples presented in Section 3. Instead of opening and closing files inside loops, one can open the files before the loops, read/write in the loops, and close the files after the loops. Note that when you have nested function calls, where a function gets called in the loop, it is important to find the right-level of nesting at which the files should be opened and closed, especially for features like logging.

It is important to review the user-guides of the large-scale HPC systems of interest to learn about their hardware architecture. If a system supports a sufficient size of storage on the compute servers/nodes, may be as a `/tmp` space, then instead of directly reading and writing to the shared filesystems from the compute jobs, one can evaluate if it is possible to do I/O in memory or in the `/tmp` space of the compute server/node. Reading and writing data from the storage space available on the node can be significantly faster than reading from a filesystem. If this approach is adopted, it is important to remember to copy data from the memory or `/tmp` space to the files on the filesystem before the compute jobs terminate.

Calling an I/O function every time a value needs to be read from a file can cause performance penalties. Hence, if possible, read the entire file into an array and then work with the array instead of opening and closing the file every time the data is needed. Additionally, instead of reading line-by-line from a file, evaluate if reading blocks of data from a file would be possible and efficient. As an example, one could

consider using `fread()` in C to read more than a line in a single I/O call as compared to using `fgets()` that reads only one line at a time.

It is important to **compare the performance of different functions/methods/APIs by writing simple use cases before using the functions/methods/APIs in the actual code.** The choice of suboptimal functions/methods/APIs for the tasks at hand can lead to a decrease in performance.

Even if the compute jobs running on an HPC system can access large-scale datasets on a separate storage system (secondary or tertiary storage), the network bandwidth and latency can negatively impact the performance of the jobs/applications. To avoid this situation, copy the datasets to the HPC system of interest prior to running the jobs that need those datasets. Once the processing is done, copy the results back to the storage systems/ of your choice. Also**, split large directories into smaller ones and compress those directories before data transfer**. Use `rsync` with appropriate flags to transfer data from source and destinations at geographically disparate locations.

In summary, **consider all the memory and storage hierarchies along with the chosen functions/methods/APIs at the time of application design to optimize I/O**.

5. USING THE BUFFERING OPTIONS FOR OPTIMAL I/O

Let us now review the concept of buffering in general. **A buffer is a temporary storage area associated with the input/output (I/O) devices**. It is important to understand the I/O buffering options supported by the programming languages and platforms that you are working with. For example, C supports buffered I/O.

Let us consider a simple example in C that is shown below (`testbuffering1.c`) for further understanding how buffering works. This code adds two numbers whose value is read from the standard input.

```
#include<stdio.h>
int main(){
 float a, b, c;
 printf("\nEnter the value of a: ");
scanf("%f",&a);
printf("\nEnter the value of b: ");
scanf("%f",&b);
c= a+b;
printf("The sum of a and b is: %f\n",c);
return 0;
}
```

To compile this code, run the following command:

```
$ gcc -o testbuffering1 testbuffering1.c
```

In the first attempt of running this code, we enter the two values at the time when we are prompted for those.

```
$ ./testbuffering1
Enter the value of a: 3.1
Enter the value of b: 4.2
The sum of a and b is: 7.300000
```

In the second attempt, we pass more than the required number of values when we are presented the first prompt.

```
$ ./testbuffering1
Enter the value of a: 3.1 4.2
Enter the value of b: The sum of a and b is: 7.300000
```

The output from running the code (`7.300000`) in both the first and second attempts is the same and correct. What happens here is, **when we pass more than the required number of values, they are held in the input buffer, and then passed on to the next prompt (if there is one) automatically as input.**

The standard I/O library in C, `stdio.h`, supports the buffering of data to reduce the number of `read()` and `write()` system calls to the OS kernel by calling `setbuf` and `setvbuf` functions with appropriate parameters. **The `setvbuf` function allows the user to control the buffer associated with a stream** (file pointer named stream in the signature shown on the slide), including changing the buffer size, flushing the buffer, changing the buffer type, deleting the default buffer in the stream, and opening the buffer for the stream without the buffer. Please note that the `setbuf` function is deprecated and hence new software should consider using `setvbuf` function.

The signature of the `setvbuf` function is as follows:

```
int setvbuf(FILE *stream, char *buffer, int mode, size_t size)
```

Three modes of buffering are available through `setbuf` or `setvbuf` functions:

1. **_IONBF or Unbuffered**: output stream is unbuffered and hence any data written to the output stream is immediately written to a file.
2. **_IOLBF or Line buffered:** characters written to the output stream are buffered till a new line character is found and at that point the data is written to a file, and for reading, characters are read till a new line character is found.
3. **_IOFBF or Fully buffered**: characters written to the output stream are buffered till the buffer becomes full and at that point the output is written to a file, and for reading, the characters are read till the buffer is full.

With respect to I/O related optimization, **consider profiling the code on the platform of interest and checking if full buffering and line buffering with appropriate buffer sizes are improving the performance** or not, and in some cases (with large contiguous writes) **turn off buffering to see if performance improves**. Please note that **you may call the `setvbuf()` function only after the file has been successfully opened, and before any file I/O operations have taken place.** Also, as per the C standard, by default, standard input and output should be fully buffered and standard error should be unbuffered.

6. USING MEMORY-MAPPED FILE I/O FOR OPTIMIZING PERFORMANCE

When a file is copied directly to the virtual memory or address space of a process **via functions such as `mmap()` in C, the number of system calls for reading and writing are reduced and this can enhance the application's performance**. A C code doing file I/O could be involving the data transfer from a user-defined buffer (e.g., from an array) to the `stdio.h` library buffer, and then from the `stdio.h` library buffer to a kernel buffer, and then from the kernel buffer to a file on the storage device. Hence, the data could be getting copied thrice between the program-level to the storage-level and the system calls involved could themselves be incurring latency. **By using memory-mapped files, the buffer copies can be eliminated, and the overheads of system calls and cache lookups (and hence the associated latency) can be reduced**. Therefore, using the `mmap()` function for large files can potentially improve the performance of applications, especially those involving random access, page reuse, and where data fits in the memory. Please note that for reading small files sequentially, using the read system call may be better than using `mmap` and experimentation to find the best option amongst the two should be done. **The memory mapped files can be accessed like arrays in programs and only regions of files that are needed by the programs are loaded in the memory at any given point in time**.

The signature of the `mmap` function is as follows and it is available through the `sys/mman.h` header file in C:

```
void * mmap (void *startingAddressforMapping, size_t
numOfBytesToBeMapped, int typeOfAccess_RWX, int flags, int
fileDescriptor, off_t offset)
```

Following is the description of the arguments of the `mmap` function:

1. `startingAddressforMapping`: This argument provides the preferred starting address used for mapping.
2. `numOfBytesToBeMapped`: This argument specifies the length of the mappings and must be greater than 0.
3. `typeOfAccess_RWX`: This argument controls the type of allowed access to the pages. The `PROT_READ` is for read access, the `PROT_WRITE` is for write access, the `PROT_EXEC` is for execute access, and `PROT_NONE` is for no access.
4. `Flags`: This argument determines whether or not the updates made to the mappings are visible to the other processes mapping to the same region. Some of the options or values that this flag can take are as follows:
   - `MAP_SHARED` - The mapping is shared with other processes.
   - `MAP_PRIVATE` - The mapping is private, and the updates made to the mapping are not visible to other processes.

    - `MAP_FIXED` – This mapping indicates that the address provided is not a hint and the mapping should be placed exactly at that address.
5. `fileDescriptor`: This argument is a file descriptor representing the file that is meant to be mapped.
6. `Offset`: The argument specifies the offset from which the file mapping should start.

The `mmap`() function returns 0 if it works successfully and returns `MAP_FAILED` in the event of an error.

7. AI FRAMEWORKS AND I/O OPTIMIZATION

Large-scale I/O during the training of AI models can consume a significant amount of time and can pose bottlenecks during the preprocessing step. PyTorch, Keras, and TensorFlow are popular AI/machine learning frameworks/libraries. Let us review some techniques for optimizing I/O when using PyTorch, Tensorflow, or Keras.

Prefetching: This is a technique that can be used to reduce the time in each training step by overlapping preprocessing (for reading data from input file) with the model training such that when the model is executing a training step *N*, the input data is being read for the *N+1th* step. The `tf.data.Dataset.prefetch` transformation provided by the `tf.data` API in TensorFlow can be used for prefetching the data ahead of its use, and `tf.data.AUTOTUNE` can be used to decide at runtime about the amount of data to prefetch.

Parallelizing data extraction: Overheads are involved in reading data from remote locations or deserializing/decrypting the data, and hence the data should be copied locally and then `tf.Data.Dataset.interleave` transformation in TensorFlow could be used to parallelize the data loading step.

Data Loaders and Data Generators: When working with large datasets (especially large datasets with images), it is not uncommon to see error messages related to **insufficient memory on the GPU/CPU at the data loading step**. In such situations, it can be useful to ingest data in batches and if possible, in parallel. If you are working with PyTorch, then you can **consider doing asynchronous data loading by using multiple worker processes simultaneously** to avoid the out of memory problem during model training. A PyTorch class named `torch.utils.data.DataLoader` can be used for this purpose. By default, the setting for using the number of workers in DataLoader is 0 (which means the main process is doing the data loading) and this should be changed for engaging multiple worker processors in loading the data while the main training process continues its execution. If you are using Keras, then you can consider writing/using **data generators** with the appropriate methods (viz., evaluate_generator, fit_generator, and predict_generator).

Buffer Checkpointing: PyTorch supports buffer checkpointing which is **a technique that favors recomputing over storing inputs** of all the layers of the AI model during the training step. By reducing the number of layers for which the input should be stored for computing upstream gradients during the backward propagation step and recomputing for others, **the memory requirements are reduced, which in turn enables increasing the batch size** (or the number of samples used for training a neural network in an epoch). The impact of increasing the batch size may vary on a case-by-case basis. The `torch.utils.checkpoint` API in PyTorch can be used for performing this type of checkpointing and recomputing.

8. PARALLEL SCIENTIFIC APPLICATIONS

There are multiple levels at which the I/O can be optimized for parallel applications. We can leverage the features/settings of the parallel distributed file systems that can provide parallel data paths to storage disks. We can consider using MPI I/O for C/C++/Fortran applications. We can also consider using high-level libraries like PHDF5 and pNetCDF that support including metadata for describing the data stored in the files. However, the applications using these libraries will have additional software dependencies and will not be self-contained. In this section, we will discuss parallel filesystems and MPI I/O further.

Parallel Filesystem

Lustre, GPFS (now rolled into IBM Spectrum Scale), and PanFS are some of the popular options for distributed parallel filesystems and here we will discuss Lustre.

Lustre has a client-server architecture, where the client-side runs on the compute nodes and login nodes of the HPC systems, and the servers are in proximity to the actual physical storage devices or volumes. Figure 1 shows an overview of the different components in Lustre. Let us review some of the key components with respect to the discussion on Optimizing I/O: Metadata Servers (MDSs), Metadata Targets (MDTs), Object Storage Servers (OSSs) and Object Storage Targets (OSTs).

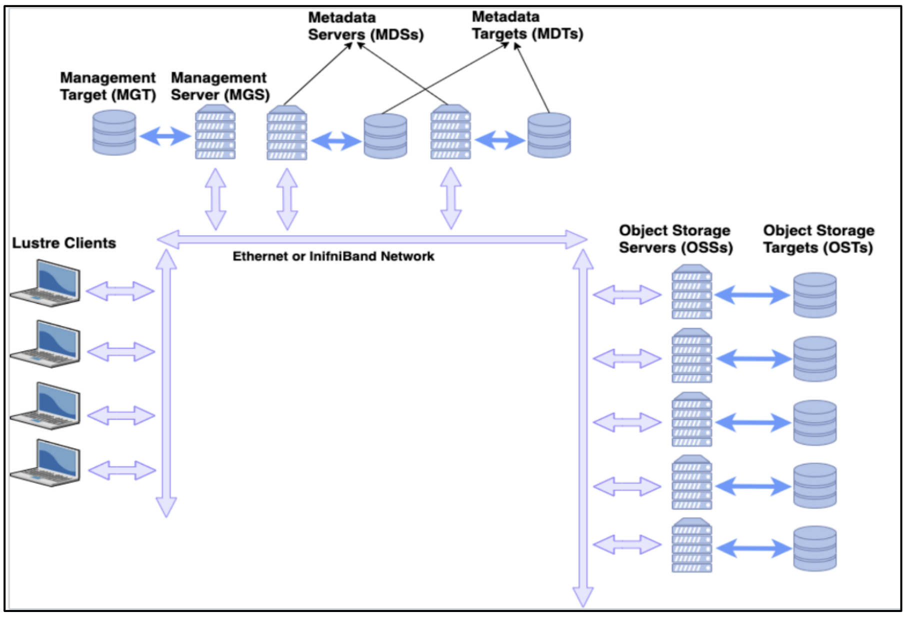


Figure 1. Lustre filesystem overview. *Source: https://wiki.lustre.org*

The MDS provides metadata services to the Lustre clients running on compute nodes or login nodes of an HPC system. It helps in looking up filenames and providing information on directories, file layouts, and access permissions. There can be one or more MDSs available in a filesystem. The MDTs are the storage devices used by an MDS to store metadata. An MDS can be connected to one or more MDTs. After receiving the file metadata from the MDS, the Lustre clients connect directly to the OSSs for file I/O.

The OSTs are devices used by an OSS node to store the contents of user files. An OSS node is often connected to multiple OSTs. The total storage capacity of a file system is the sum of the capacities of all the OSTs. Hence, the greater the number of OSTs the better the I/O capability of a filesystem. It should be notes that an HPC system can have one or more Lustre filesystems available on it and each could be having a different number of OSTs. To check the number of OSTs available on the filesystems connected to your HPC system, you may use the following command:

```
$ lfs osts
```

System Name	$HOME filesystem, # of OSTs	$WORK filesystem, # of OSTs	$SCRATCH filesystem, # of OSTs
Stampede 2.0 at TACC	4	24	66

Lustre supports the striping (or chunking) of files across several OSTs. A file can be split into multiple chunks (or blocks) of a fixed size such that each of these chunks can be stored on, and accessed from, different OSTs in parallel. Doing so increases the available bandwidth for I/O and provides the flexibility to overcome the storage limits associated with any single OST. Typically, RAID0 type of layout is adopted for a striped file wherein, a file is striped (or stored) across multiple OSTs in a round-robin manner. Figure 2 shows an overview of the concept of the stripe size, stripe count, chunking of the files, and the normal layout of the file across different OSTs.

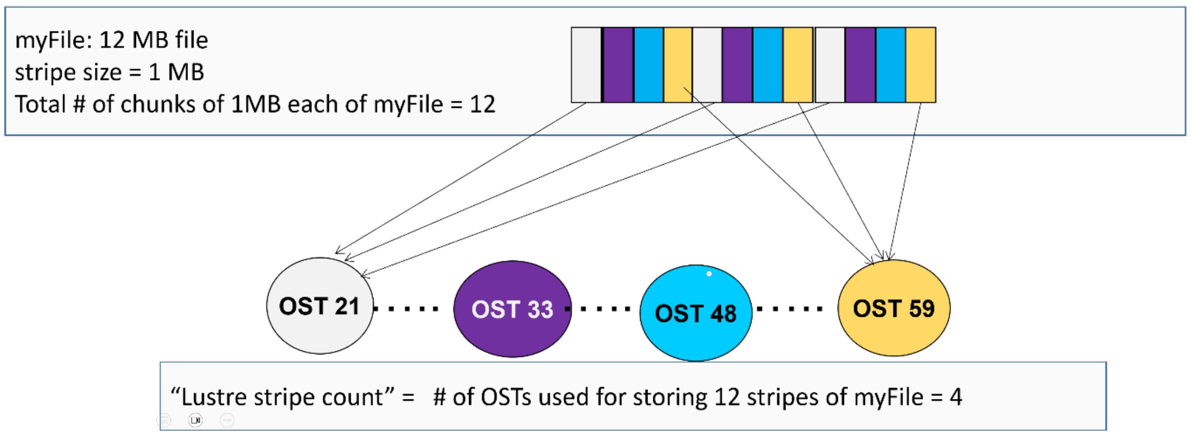
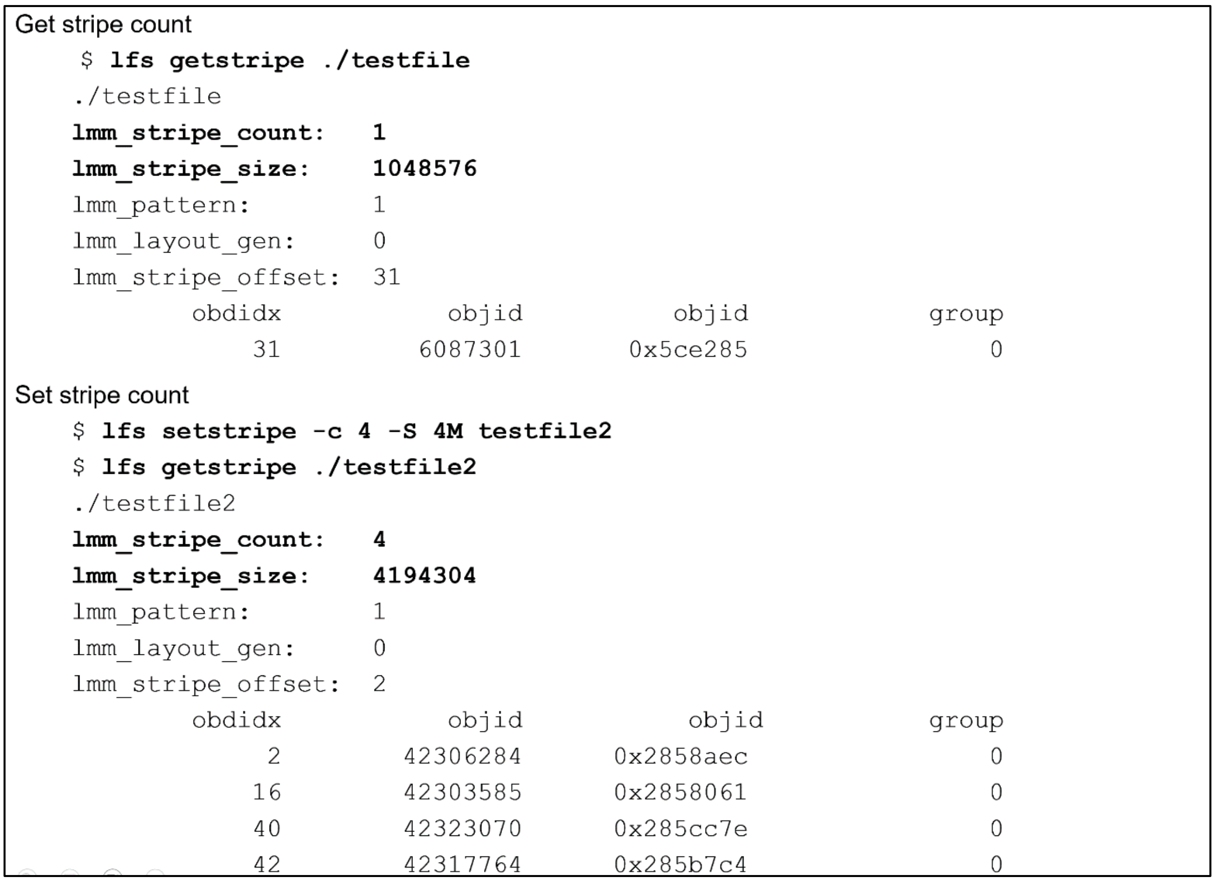

myFile: 12 MB file
stripe size = 1 MB
Total # of chunks of 1MB each of myFile = 12
OST 21
OST 33
OST 48
OST 59
"Lustre stripe count" = # of OSTs used for storing 12 stripes of myFile = 4

Figure 2. Illustration of the concept of Lustre stripe count and stripe size.

System administrators set a default stripe count (the number of OSTs across which a file is stored) and stripe size (the size of each chunk of the file that is stored on an OST) that applies to all newly created files on a filesystem. The users on an HPC system can check these default values by running the "`lfs getstripe`" command and can reset the default stripe count or stripe size on their files/directories by using the Lustre "`lfs setstripe`" command. The use of these commands is demonstrated in the example shown in Figure 3.



Get stripe count

```
 $ lfs getstripe ./testfile
./testfile
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  31
        obdidx          objid          objid            group
            31        6087301       0x5ce285                0
```

Set stripe count

```
$ lfs setstripe -c 4 -S 4M testfile2
$ lfs getstripe ./testfile2
./testfile2
lmm_stripe_count:   4
lmm_stripe_size:    4194304
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  2
        obdidx          objid           objid           group
             2       42306284       0x2858aec               0
            16       42303585       0x2858061               0
            40       42323070       0x285cc7e               0
            42       42317764       0x285b7c4               0
```

Figure 3. Lustre commands for setting stripe size and stripe count.

It should be noted that striping can be set at the directory level such that all the files created within the directory inherit the striping related settings on the directory. The desired striping related settings should be made before a file is created. Moving a file (with the `mv` command) does not change the striping related settings but by copying a file (with the `cp` command) to a new file, its striping related settings can be changed. Both under-striping and over-striping should be avoided as over-striping can result in writing small chunks to each OST thereby underutilizing it, and under-striping can lead to too much stress on a single OST (and the OSS associated with it). It should also be noted that there is an increased risk of a file becoming unavailable if any of the OSTs on which its chunks are stored go down. However, as the HPC systems are meant for computing and not storage, this risk should not cause a problem in the long-term as the users are expected to move their output files to a secondary or a tertiary storage system after their jobs complete running.

To maintain a good citizenship on the HPC system having the Lustre filesystem, avoid opening and closing the same file every few milliseconds from your compute jobs as it stresses the MDS. Also, avoid opening too many files too frequently as it stresses out both the MDS and OSTs. Also, avoid creating thousands of files in the same directory. Coordinating access to several thousand files simultaneously (or synchronizing their metadata) from a single job can stress out the MDS. Hence, it is recommended to break down large directories into subdirectories.

MPI I/O

MPI-I/O featured in MPI-2 which was released in 1997, and **it interoperates with the filesystem to enhance I/O performance for distributed-memory applications.** One of the advantages of using MPI I/O for supporting parallel I/O in applications is that it reduces the dependency on additional external libraries and hence can help in developing self-contained applications.

Given N number of processes participating in a computation, each process (or a subset of these processes) can participate in reading or writing a portion of a common file using MPI I/O. MPI provides three ways for positioning where the read or write takes place for each process:

- Using “individual file pointers”
- Using explicit byte offset - no file pointer is used nor updated
- Access a “shared file pointer”

While doing MPI I/O, we first need to open the file. Instead of using the regular file open command in C for example, we will need to call the `MPI_File_open` function and pass certain arguments to it such as the name of the file to be opened and the mode in which it should be opened (such as read only, write only, and read and write both). After opening the file, the data can be read from or written to files using appropriate read and/or write calls and appropriate number of MPI processes. Once the I/O is done, the file should be closed by calling the `MPI_File_close` function.

MPI provides multiple types of calls for reading and writing files in parallel. **There are API calls available with blocking or non-blocking synchronization mechanisms, having collective or a non-collective coordination pattern, and using individual file pointers, shared file pointers or explicit offsets.** For example, for reading a file in a blocking, non-collective coordination manner, either use `MPI_File_read` with `MPI_File_seek` or `MPI_File_set_view` (individual pointer), or use `MPI_File_read_at` that takes explicit byte offsets as argument. Likewise, for writing to a file, in a blocking, non-collective coordination manner, either use `MPI_File_write` with `MPI_File_seek` or `MPI_File_set_view` (individual pointer) or use `MPI_File_write_at` that takes explicit byte offsets as argument.

Using the MPI calls supporting collective I/O is a critical optimization strategy for reading from, and writing to, the parallel file system. **The MPI implementation optimizes the read/write requests based on the combined requests of all processes and can merge the requests of different processes for efficiently servicing the I/O requests (that is, the I/O requests from many processes can be aggregated for better performance).** This is particularly effective when the accesses of different processes are noncontiguous.

The blocking collective functions for reading and writing are as follows and their signatures are similar to their non-blocking counterparts: `MPI_File_read_all`, `MPI_File_write_all`, `MPI_File_read_at_all`, `MPI_File_write_at_all`.

The non-blocking collective functions can be useful for overlapping computations with I/O thereby improving the performance of the code. These are defined for data access routines with explicit offsets and individual file pointers but not with shared file pointers and need a call to `MPI_Wait`. An API call for reading a file in non-blocking, collective mode is `MPI_File_iread_at_all`.

When using MPI I/O calls, "Collective buffering" can be used for coordinating the I/O at the application-level such that the total number of disk operations for I/O are reduced and **the bottlenecks associated with thousands of processes writing to a shared file get mitigated**. With collective buffering, small data blocks are combined in the application - a subset of the total number of processes participating in the job act as buffers over which the small data blocks are aggregated into larger data blocks. The aggregated data is then written to the disks/storage targets by the subset of the total number of processes, thereby **reducing the total number of processes participating in the I/O at this stage and this improves the I/O performance**. Application developers can **pass MPI-I/O hints to the MPI library** on whether to use collective buffering or not, to select the number of processes acting as aggregators, and to set the optimal size for the buffers.

MPI-IO hints are extra information supplied to the MPI implementation through the following function calls for improving the I/O performance and the utilization of the hardware resources: `MPI_File_open`, `MPI_File_set_info`, and `MPI_File_set_view`. Please note that hints are optional and implementation dependent. You can specify hints, but the MPI implementation can choose to ignore them. `MPI_File_get_info` is used to get the list of hints associated with a file, examples of hints are Lustre `striping_unit`, and `striping_factor`. For these hints to work, an MPI library with Lustre support should be available (e.g., MVAPICH2 and OpenMPI). The **hints for collective buffering can also be passed** to the aforementioned calls.

A sample code written using C and MPI is shown below to demonstrate how hints for Lustre stripe counts and stripe size can be specified within the application.

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int i, rank, size, offset, N=160000;
   MPI_File fhw;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   int buf[N];
   for ( i=0;i<N;i++){
        buf[i] = i ;
   }
   offset = rank*(N/size)*sizeof(int);
   MPI_Info myinfo;
   MPI_Info_create (&myinfo);
   MPI_Info_set(myinfo,"striping_factor","4");
   MPI_Info_set(myinfo, "striping_unit","4194304");
   MPI_File_open(MPI_COMM_WORLD, "datafile_striped22",
          MPI_MODE_CREATE|MPI_MODE_WRONLY, myinfo, &fhw);
   printf("\nRank: %d, Offset: %d\n", rank, offset);
   MPI_File_write_at(fhw, offset, buf, (N/size), MPI_INT,
                                              &status);
   MPI_File_close(&fhw);
   MPI_Info_free(&myinfo);
   MPI_Finalize();
   return 0;
}
```

An info object is created and this will be passed to the MPI_Info_set calls

Using MPI_Info_set to set the Lustre stripe count to 4 on the output file

Using MPI_Info_set to set the Lustre stripe size to 4MB on the output file

An file named “datafile_striped22” is created and opened in write mode

This call will involve all the MPI processes in writing to the file and uses explicit offset value stored in variable “offset” for identifying the location at which each process will start writing its buffer values to the output file.

9. FLASH ASTROPHYSICS CODE – A CASE-STUDY

A few years ago, a user was running the popular FLASH astrophysics code (FLASH 4.2.2) on the Stampede system at TACC when their job stressed out the filesystem and the OSSs became unresponsive. Upon investigation, it was found that the job involved reading a restart file (or a checkpoint file) just once during running using 7000+ cores, after which checkpoints were written every 10 minutes or so, till the end of the job. **The restart file was 250 GB in size and was written with the default stripe count of two.** All 7000+ cores were trying to read this single file and this was inefficient and monopolized the two OSSs so much that they were unable to respond to other requests and became unresponsive. To find a remedy for this situation, performance testing was done using the different Lustre stripe count settings on different output directories. The code was run on 7620 CPU cores and the size of checkpoint file that was initially read was 189G.

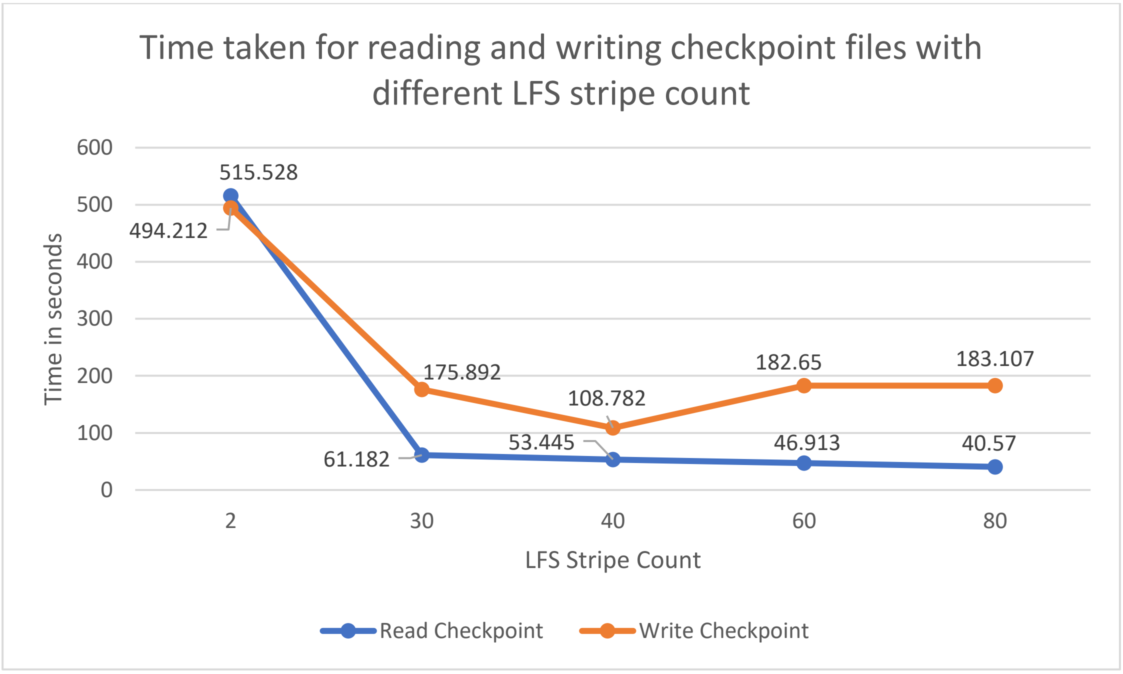


Figure 4. Running FLASH code with varying settings for Lustre (LFS) stripe count on the output directories.

As can be observed from Figure 4, with the stripe count of 80, the time taken for reading the file was lowest, while with stripe count 40, the time taken for writing was lowest. The stripe count of 40 looked best in this scenario as beyond that, even though the time for reading continued to decrease but it decreased at a slower rate. However, the time for writing the checkpoint file increases noticeably in going up from stripe count 40 to 60.

FLASH can do both serial and parallel I/O. As users may forget to set the appropriate stripe count on the directories in which output files from FLASH are written, it was important to update the FLASH code for serial and parallel I/O and set the appropriate stripe count on the files from within the code. In the code for doing parallel I/O (files `io_ncmpi_interface.c` & `io_h5file_interface.c`), `MPI_Info_set` was used to pass the hints for the "`striping_factor`" (stripe count). The stripe count was automatically set to the minimum amongst the following two values: number of MPI processes used to run the job, and 80. In the serial version of the code (`io_h5file_interface.c` file to be precise), the Lustre command for setting the stripe count of the output file was added as follows:

```
snprintf(buffer, sizeof(buffer), "lfs setstripe -c 10 %s",
local_filename);
```

10. CODE SAMPLE

The complete working code samples associated with the various topics covered in this article are available in the following GitHub repository under the LGPL-2.1 license: https://github.com/ritua2/bsswfellowship . The README files in each of the sub-directories in this repository include instructions on running and testing the code.

11. CONCLUSION

In this article, we reviewed the various techniques for optimizing I/O in serial, and parallel applications. We also reviewed some strategies for optimizing I/O in machine learning/AI applications developed using PyTorch, TensorFlow, and Keras. As underlying hardware architectures, middleware, frameworks, and libraries continue to evolve, some of the strategies mentioned in this article will need to be revisited and tested for their continued merits. However, the general message of devoting attention to how applications are doing I/O will continue to be relevant in future as well, especially, with the convergence of AI, HPC, and Big Data.

REFERENCES AND ADDITIONAL RESOURCES

1. NICS I/O guide: https://oit.utk.edu/hpsc/isaac-open/lustre-user-guide/
2. Introduction to Parallel I/O: http://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf
3. Introduction to Parallel I/O: https://sea.ucar.edu/sites/default/files/PIO-SEA2015.pdf
4. Introduction to Parallel I/O and MPI-IO by Rajeev Thakur: https://www.slideserve.com/yoshe/introduction-to-parallel-i
5. William Gropp, UIUC: http://wgropp.cs.illinois.edu/usingmpiweb/examples-advmpi/index.html
6. William Gropp, UIUC: https://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf
7. Rob Latham, ANL: https://www.mcs.anl.gov/~robl/tutorials/csmc/pio-architecture.pdf
8. Steve Pat, "UNIX Filesystems: Evolution, Design, and Implementation", chapters 3 and 4
9. Michael Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004, ISBN13: 978-0071232654, LC: QA76.73.C15.Q55.
10. William Gropp, Using MPI and Using Advanced MPI: http://wgropp.cs.illinois.edu/usingmpiweb/
11. PyTorch, performance tuning guide: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html
12. TensorFlow, optimize pipeline performance: https://www.tensorflow.org/guide/data_performance
13. Keras examples: https://keras.io/examples/vision/mnist_convnet/
14. Nutan Sharma, PyTorch CNN with MNIST: https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118

15. PyTorch, saving and loading general checkpoints: https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html
16. Lustre wiki: https://wiki.lustre.org
17. GitHub repository for code samples related to this article: https://github.com/ritua2/bsswfellowship