Optimizing I/O in Scientific Applications Part-1

Better Scientific Software Fellowship (BSSw Fellowship)

Ritu Arora

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

GitHub: https://github.com/ritua2/bsswfellowship

LinkedIn: https://www.linkedin.com/in/ritu-a-59b58ab/

Twitter: https://twitter.com/ritzaa2

YouTube: https://www.youtube.com/user/ritua2

Topics to be covered

1) **Why should we optimize I/O? (general review)**
2) **How can we optimize I/O? (general review plus one example code in C)**
3) **General I/O strategies for working on High Performance Computing (HPC) platforms (general review)**
4) **Introduction to the Lustre parallel file system (general review and demo of some Lustre commands)**
5) Introduction to parallel I/O patterns
6) Introduction to MPI I/O
7) I/O in the context of checkpointing and AI

Pre-Event Survey

https://tinyurl.com/278sap7x

Google Doc Link for Q&A

https://tinyurl.com/yjzakhf6

I/O in scientific applications

- Scientific applications often
  - Read initial conditions or datasets for processing
  - Write numerical data from simulations
    - Example: for saving application-level checkpoints or for developing scientific visualizations

- In case of serial applications, the total execution time can be broken down into the computation time and the I/O time (Tcomputation + TIO)

- In case of distributed/parallel applications, the total execution time can be broken down into the computation time, communication time, and the I/O time (Tcommunication + Tcomputation + TIO)

- Hence, optimizing the time spent in I/O is as important as the time spent in computation and communication to improve the overall application performance and reduce the time-to-solution

- However, it is observed that doing efficient I/O can be challenging and is often an afterthought, and therefore, it is important to discuss the topic of optimizing I/O

Some examples of inefficient I/O observed 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

Some ideas for addressing inefficient I/O (1)

1. Opening and closing files inside loops

   Solution: open the files before the loops, do 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.

2. Large-scale reading and writing to the shared filesystems directly from the jobs that are running on HPC platforms

   Solution: if possible, do I/O in memory or in `/tmp` space of the compute server/node and not directly from the files stored on a shared filesystem, and remember to copy data from the memory or `/tmp` space to a file on the shared filesystems before the job terminates

3. Calling an I/O function every time a value needs to be read from a file
   1. Solution: if possible, read the entire file into an array and then work with the array instead of opening and closing the file every time data is needed
   2. Solution: instead of reading line by line from a file, evaluate if reading blocks of data from file would be possible and optimal
      - 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

Some ideas for addressing inefficient I/O (2)

4. Choice of suboptimal functions/methods/APIs/algorithms for the tasks at hand

   Solution: compare the performance of different functions/methods/APIs/algorithms by writing simple use cases before using the functions/methods/APIs in the actual code

5. Attempting to access large datasets on a different system (secondary or tertiary storage system) while the compute jobs are running

   Solution: Copy the data to the system on which your jobs need to run and once the processing is done, copy the results back to the system 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

What is buffering? A simple C code to explain buffering.

$ cat testbuffering1.c

```
#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);
   //getch();
 return 0;
}
```

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

Enter the value of a: 3.1

Enter the value of b: 4.2

The sum of a and b is: 7.300000

```
$ ./testbuffering1
```

Enter the value of a: 3.1 4.2

Enter the value of b: The sum of a and b is: 7.300000

Understanding I/O buffering supported by programming languages – C language example

- In C, stdio.h is the standard I/O library and it supports buffering of data to reduce the number of read() and write() system calls to the OS kernel by calling `setbuf` (deprecated) or `setvbuf` functions with appropriate parameters:

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

  - Three **modes** of buffering are available through `setbuf` or `setvbuf` functions:
    - _IONBF or Unbuffered: output stream is unbuffered and hence any data written to the output stream is immediately written to a file
    - _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
    - _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

- As per the C standard, standard input and output should be fully buffered and standard error should be unbuffered by default

- 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

Using memory-mapped I/O to reduce the number of data copies and to improve performance – C language example

- 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
  - With large, 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
  - Using `mmap` can potentially improve performance of applications involving random access, page reuse, and where data fits in the memory
  - Note: for reading small files sequentially, using read system call may be better than mmap – please experiment
- Memory mapped files can be accessed like arrays in programs and only regions of files that are needed by the program are loaded in the memory at any given point in time
- Syntax:

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

Sample code showing how to use memory-mapped I/O in C – memorymapped.c

```
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>

int main(){
  int *mPtr;
  size_t pageSize = (size_t)sysconf(_SC_PAGESIZE);
  int fp = open("inputFile.txt", O_RDWR);
  ftruncate(fp, pageSize);
  mPtr = (int*)mmap(NULL, 65, PROT_READ|PROT_WRITE, MAP_SHARED, fp, 0);
  printf("\ncontents of mPtr are:\n%.*s\n", 65, mPtr);
  munmap(mPtr, 128);
  close(fp);
  return 0;
}
```

AI frameworks and I/O optimization

- TensorFlow
  - Prefetching: Overlap reading data from input file with computations by using prefetching - consider using the tf.data.Dataset.prefetch transformation provided by the tf.data API for prefetching data ahead of its use and use tf.data.AUTOTUNE 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 could be used to parallelize the data loading step
  - Consider writing/using data generators when working with large datasets
- PyTorch
  - Asynchronous data loading: PyTorch provides torch.utils.data.DataLoader to support asynchronous data loading in multiple worker processes but by default the setting for using the number of workers in DataLoader is 0 which should be changed for engaging the worker processors in loading the data while the main training process continues its execution
  - Buffer checkpointing: it is a technique that favors recomputing and thereby reduces the number of layers for which input should be stored for computing upstream gradients during backward propagation step

Reducing the I/O bottlenecks in parallel applications

- Leverage the software support for parallel I/O that is available in the form of
  - Parallel distributed file systems that provide parallel data paths to storage disks
  - MPI I/O
  - High-level libraries like PHDF5, pNetCDF
    - Files written using these libraries include metadata for describing the data stored
    - However, note that there will be a dependency created on the presence of external libraries

- Understand the I/O strategies for efficiently leveraging the underlying HPC platform

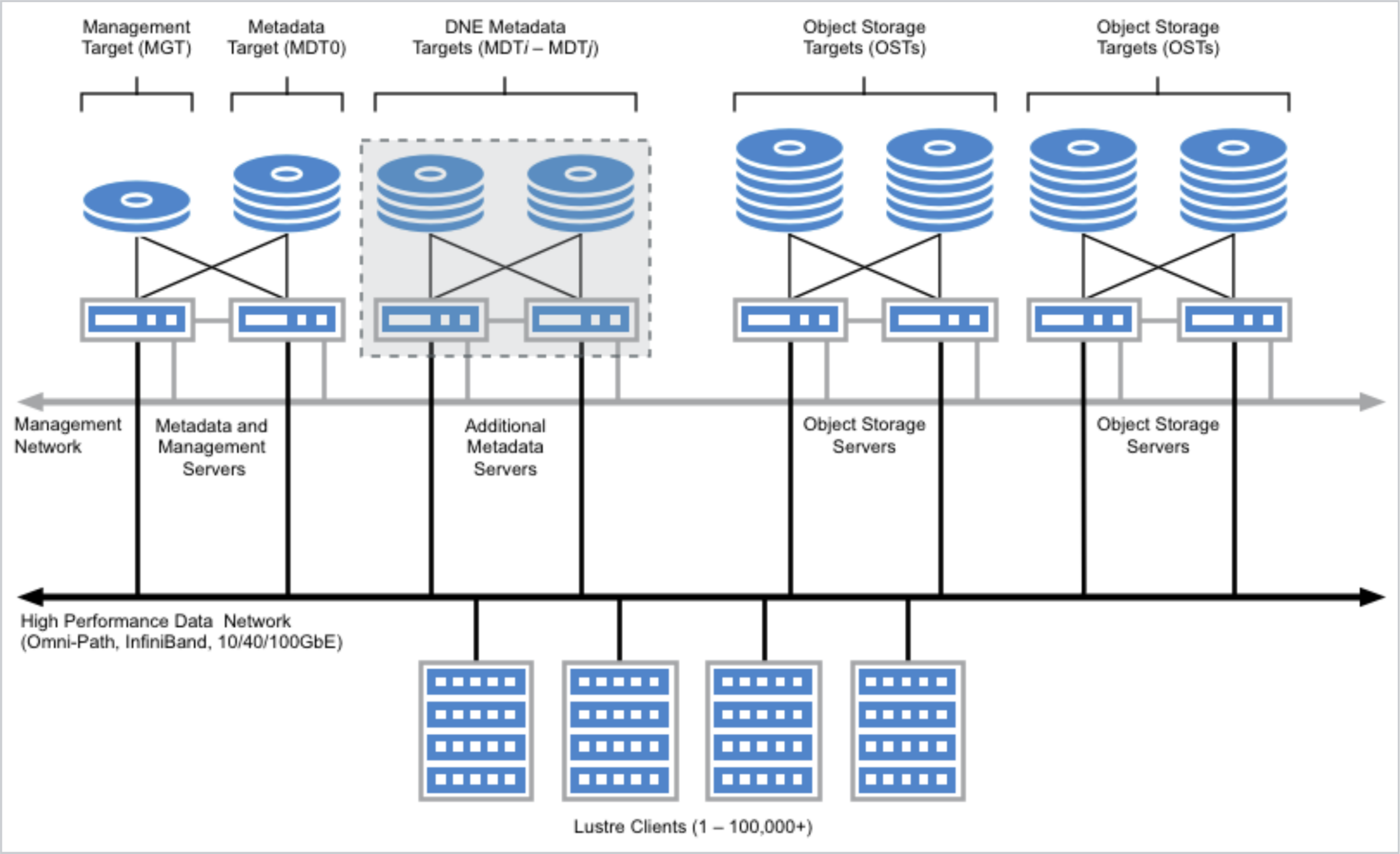
Some examples of parallel file systems

- Lustre File System

- General Parallel File System (GPFS)
  - Now rolled into IBM’s Spectrum Scale product
  - Multiple topologies: direct-attached storage, network-attached storage, and hybrid

- Other Parallel File Systems
  - Panasas Parallel File system (PanFS)
  - Parallel Virtual File System (PVFS)

Lustre File System - overview



Source: https://wiki.lustre.org/Introduction_to_Lustre

Lustre File System - OSTs

- An HPC system could have one or more Lustre filesystems available on it and each could be having a different number of OSTs

- The greater the number of OSTs the better the I/O capability

- To check the number of OSTs available on the filesystems, you may use the command:

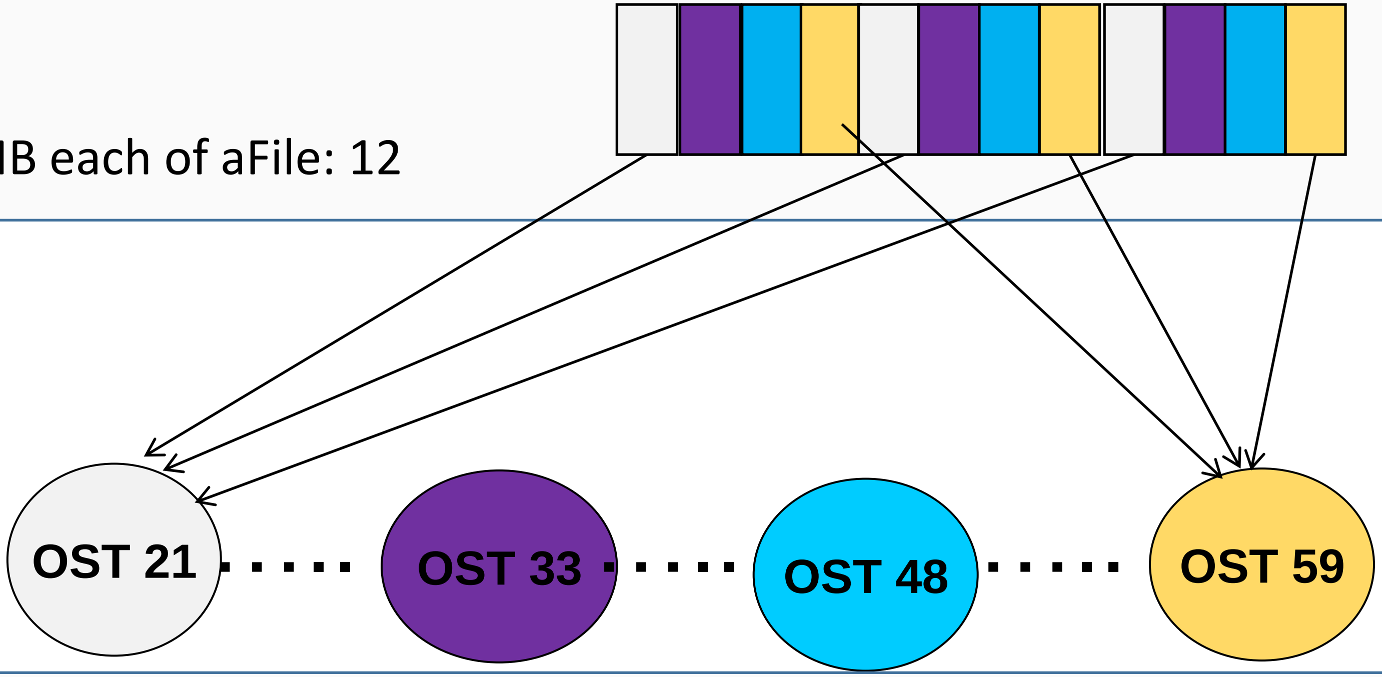
```
$ lfs osts
```

System Name	$HOME	$WORK	$SCRATCH
Stampede 2.0 at TACC	4	24	66

Lustre File System - striping

- Lustre supports the striping of files across several I/O servers
  - Example: a file can be split into multiple stripes of a fixed size each such that these stripes can be stored on and accessed from different OSTs in parallel
- Each stripe is a fixed size block of a file

aFile : 12 MB file
stripe size: 1 MB
Total # of chunks of 1MB each of aFile: 12



“Lustre stripe count” = # of OSTs used for storing 12 stripes of aFile = 4

Lustre File System: default stripe count and size

- Administrators set a default stripe count and stripe size that applies to all newly created files
  - You can check the default stripe count and stripe size by running the “lfs getstripe” command – please see the next slide for an example
- However, users can reset the default stripe count or stripe size using the Lustre commands
  - You can set the desired stripe count and stripe sizes on your files/directories using the “lfs setstripe” command - please see the next slide for an example
- Striping can be set at the directory level as well 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

Lustre commands - examples

- Get stripe count

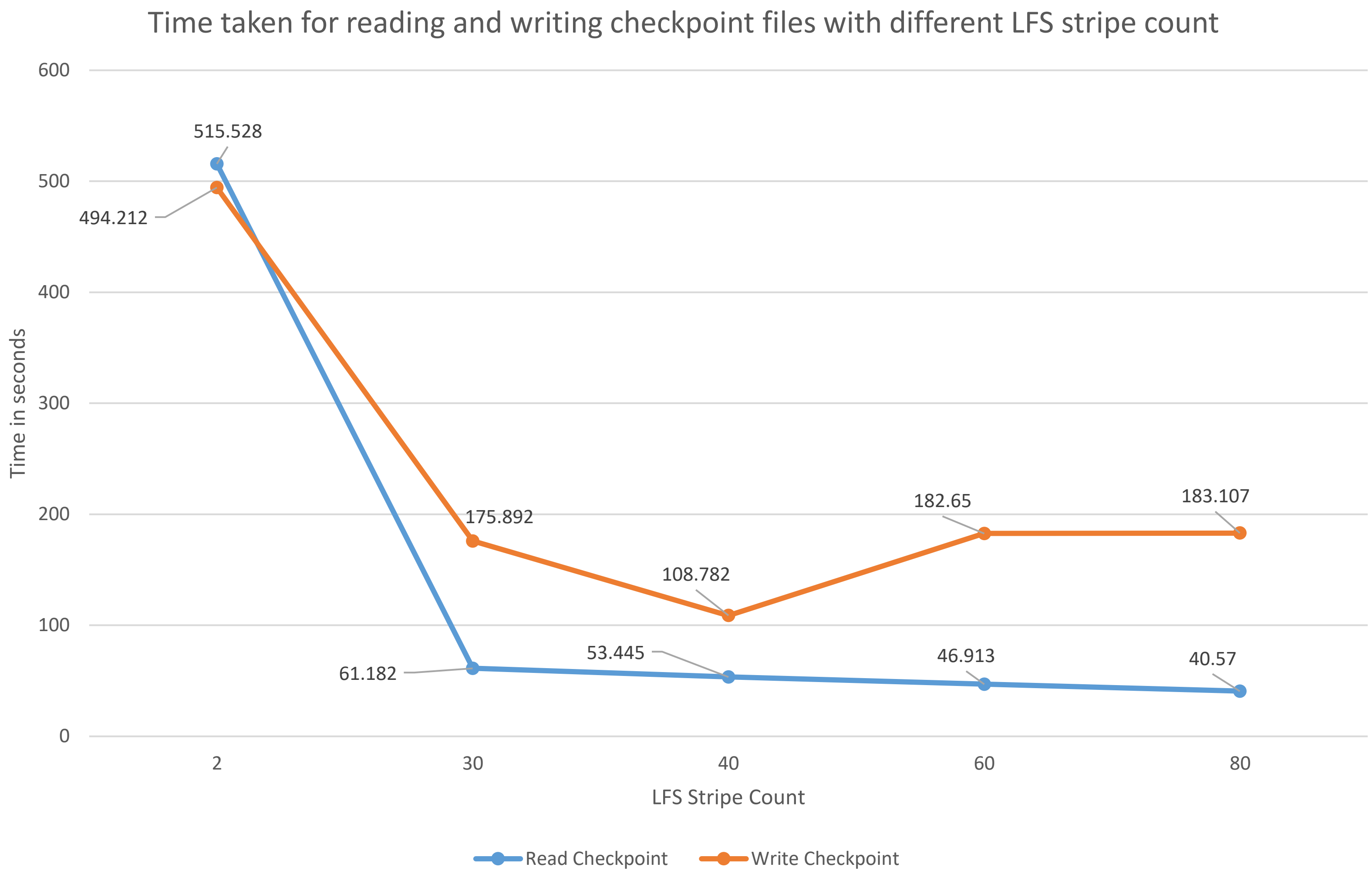
```
 $ lfs getstripe ./testfile
./testfile
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  31
        obdidx           objid          objid            group
            31         6087301      0x5ce285                0
```

- 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
```

Impact of file striping on I/O - FLASH astrophysics code

LFS stripe count #	Time taken to read a checkpoint file (in seconds)	Time taken to write the first checkpoint file (in seconds)
2	515.528	494.212
30	61.182	175.892
40	**53.445**	**108.782**
60	46.913	182.65
80	40.57	183.107



Notes: Size of the restart file: 189 GB, total number of cores used in the run: 7680

Advantages and disadvantages of file striping

- Advantages of Striping a File Across Multiple OSTs
  - A file's size is not limited to the space available on a single OST because by placing strips of a file on multiple OSTs the space required by the file is spread over those multiple OSTs.
  - The I/O bandwidth can be spread over multiple OSTs by placing strips of the file on multiple OSTs. In this manner, a file's I/O bandwidth is not limited to a single OST.

- Disadvantages of Striping a File Across Multiple OSTs
  - There is an increase in overhead due to the need to manage the file chunking/striping, network connections, and multiple OSTs.
  - There is an increased risk of a file becoming unavailable if any of the OSTs on which its chunks are stored go down.

Lustre storage scalability information from: https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf

	Value using LDISKFS backend	Value using ZFS backend	Notes
Maximum stripe count	2000	2000	Limit is 160 for ldiskfs if "ea_inode" feature is not enabled on MDT
Maximum stripe size	< 4GB	< 4GB	
Minimum stripe size	64KB	64KB	
Maximum object size	16TB	256TB	
Maximum file size	31.25PB	512PB*	
Maximum file system size	512PB	8EB*	
Maximum number of files or subdirectories per directory	10M for 48-byte filenames. 5M for 128-byte filenames.	2^{48}	
Maximum number of files in the file system	4 billion per MDT	256 trillion per MDT	
Maximum filename length	255 bytes	255 bytes	
Maximum pathname length	4096 bytes	4096 bytes	Limited by Linux VFS

Lustre can be easily stressed out by certain activities

- Opening and closing the same file every few milliseconds
  - Stresses the MDS

- Opening too many files too frequently
  - Stresses the MDS and OSTs

- Writing large files to filesystems that can be shared across multiple filesystems

- Creating thousands of files in the same directory
  - Note: a directory too is a file managed by the MDS
  - Coordinating access to several thousand files simultaneously (or synchronizing their metadata) from a single job can stress out the MDS
  - Advisable to break down large directories into subdirectories

References

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. John Burkardt: https://people.sc.fsu.edu/~jburkardt/c_src/poisson_mpi/poisson_mpi.html

Optimizing I/O in Scientific Applications Part-2

Better Scientific Software Fellowship (BSSw Fellowship)

Ritu Arora
Email: ritu@wayne.edu / ritu@venratech.com
GitHub: https://github.com/ritua2/bsswfellowship
LinkedIn: https://www.linkedin.com/in/ritu-a-59b58ab/
Twitter: https://twitter.com/ritzaa2
YouTube: https://www.youtube.com/user/ritua2

Topics to be covered

1) Why should we optimize I/O? (general review)

2) How can we optimize I/O? (general review plus one example code in C)

3) General I/O strategies for working on High Performance Computing (HPC) platforms (general review)

4) Introduction to the Lustre parallel file system (general review and demo of some Lustre commands)

5) **Introduction to parallel I/O patterns**

6) **Introduction to MPI I/O**

7) I/O in the context of checkpointing and AI

Pre-Event Survey

https://tinyurl.com/278sap7x

https://forms.gle/qKZJ8dtPQKGUP3rx6

Google Doc Link for Q&A

https://tinyurl.com/yjzakhf6

https://docs.google.com/document/d/1epOWgJg4ffcibUbAFlchXg0SvccxMCb8zlK6NwZzRT8/edit?usp=sharing

Introduction to Parallel IO

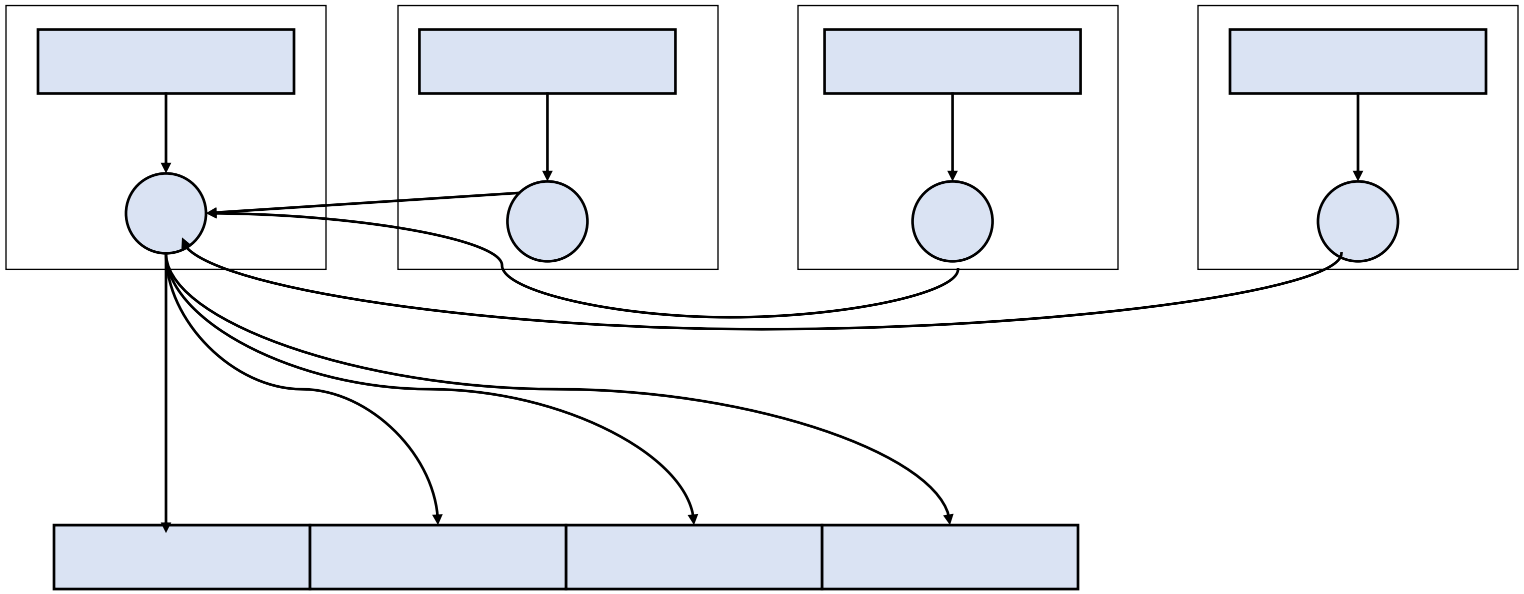
- I/O patterns in parallel programs
  - *Source: references # 4 and # 5*

- MPI I/O

Typical pattern: parallel programs doing sequential I/O

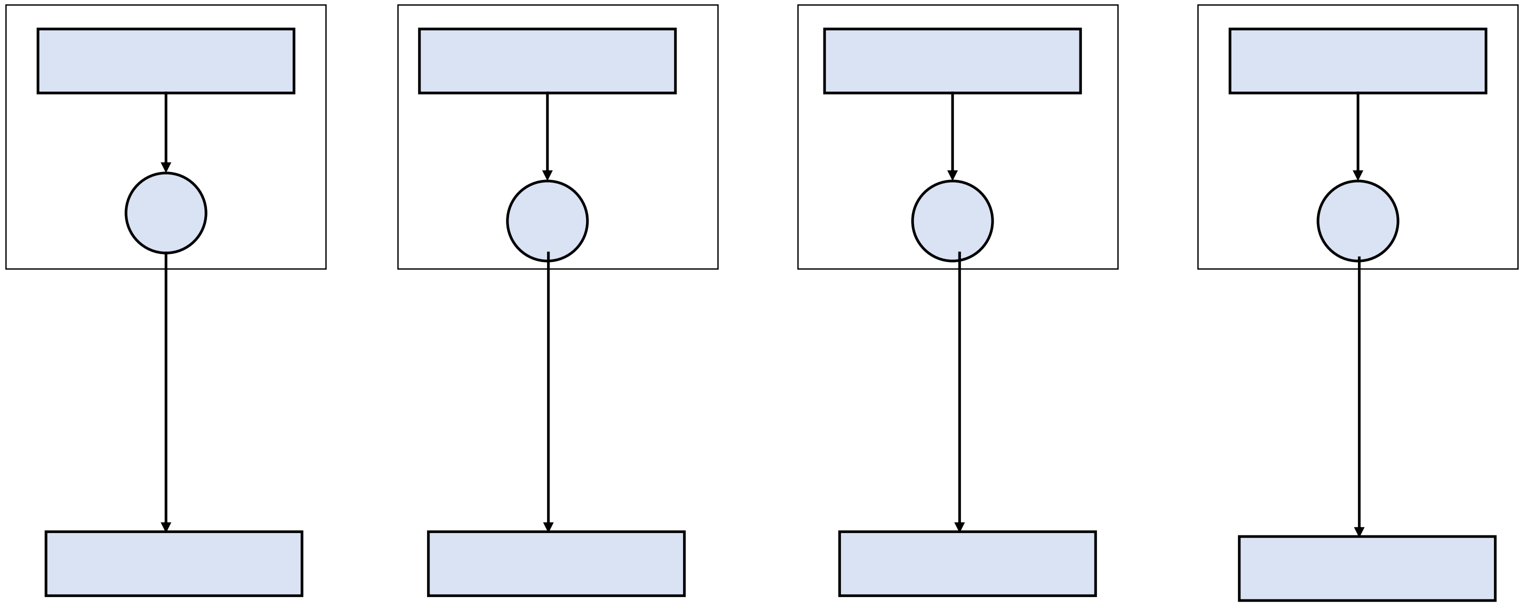
- All processes send data to the root process, and then the process designated as root writes the collected data to the file

- This sequential nature of I/O can limit performance and scalability of many applications



Adapted from: Reference 2, 5, 6

Another pattern:
each process writing to a separate file

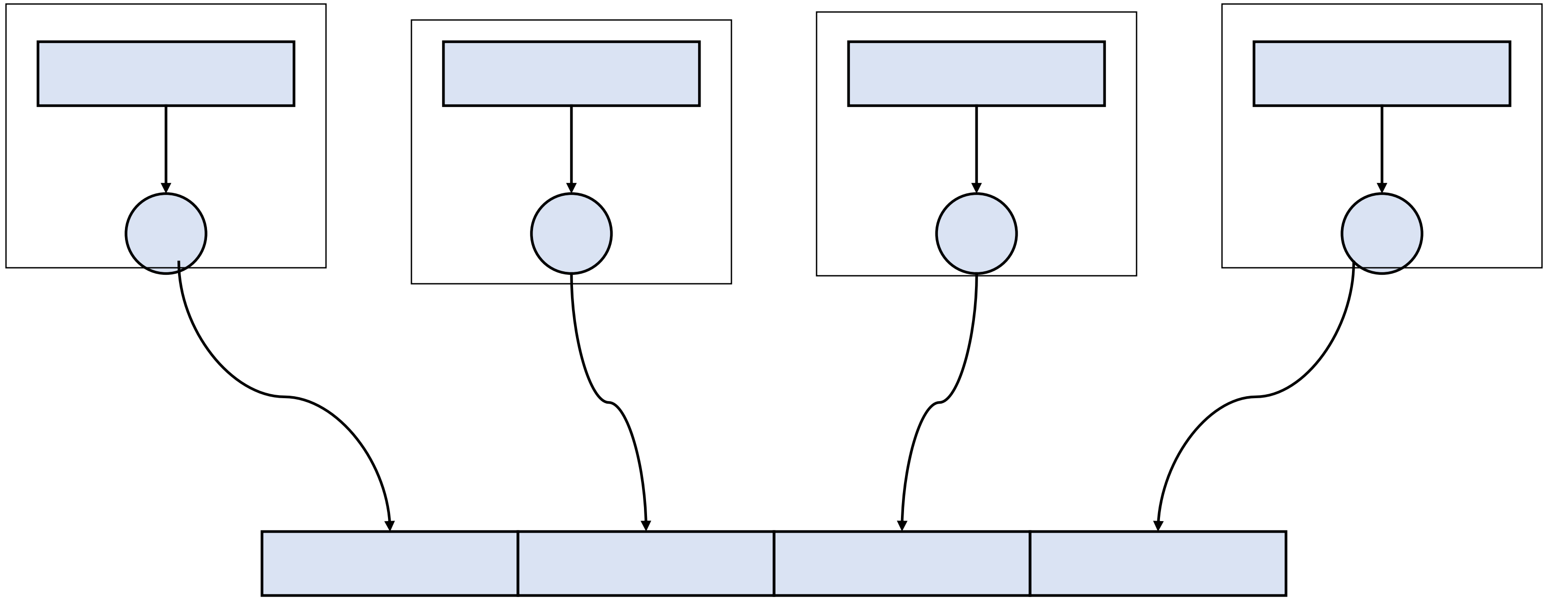


- If several thousand processes are involved in writing several thousand files, the MDS of the Lustre file system can get stressed out while managing/coordinating the metadata associated with the files

Adapted from: Reference 2, 5, 6

Desired pattern: parallel programs doing parallel I/O

- Multiple processes participating in reading data from or writing data to a common file in parallel
- This strategy improves performance and provides a single file for storage and transfer purposes, however, when multiple processes attempt to write to the same region of the file, locks are needed for serializing access, and this can degrade performance



Adapted from: Reference 2, 5, 6

MPI for parallel I/O

- Reading and writing in parallel is like receiving and sending messages

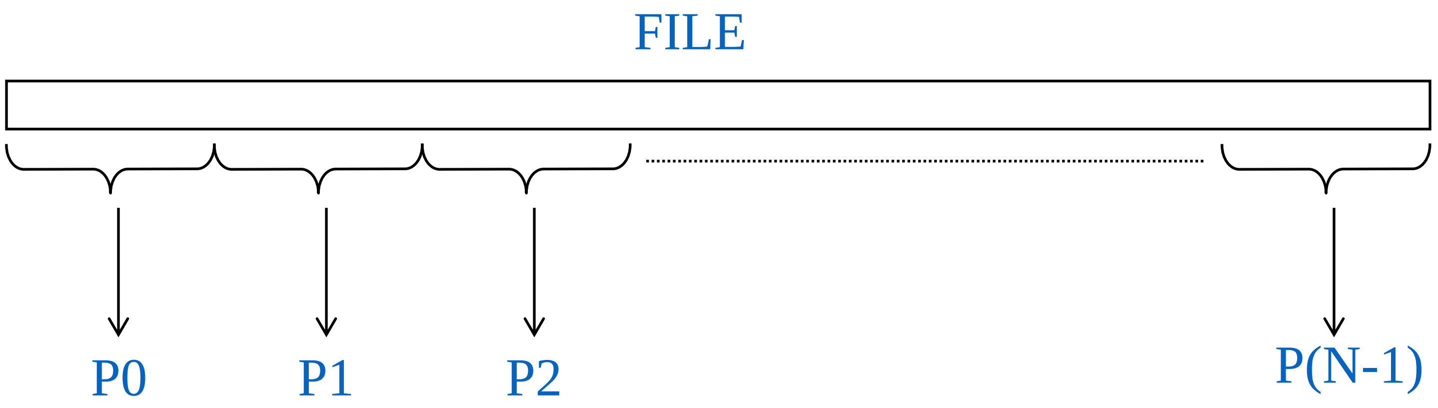
- Hence, an MPI-like machinery is a good setting for parallel I/O (think MPI communicators and MPI datatypes)

- MPI-I/O featured in MPI-2 which was released in 1997, and it interoperates with the file system 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

Using MPI-I/O

- Given N number of processes, each process participates in reading or writing a portion of a common file
- There are three ways of positioning where the read or write takes place for each process:
  - Use “individual file pointers”
  - Calculate byte offsets
    - Explicit offset operations perform data access at the file position given directly as an argument — **no file pointer** is used nor updated
  - Access a “shared file pointer”



Source: Reference 4

MPI-I/O API for opening and closing a file

- Calls to the MPI functions for reading or writing must be preceded by a call to `MPI_File_open`
  - `int` **`MPI_File_open`**`(MPI_Comm comm, char *filename, int` **`mode`**`, MPI_Info info, MPI_File *fh)`
- The parameters below are used to indicate how the file is to be opened

`MPI_File_open` mode	**Description**
`MPI_MODE_RDONLY`	read only
`MPI_MODE_WRONLY`	write only
`MPI_MODE_RDWR`	read and write
`MPI_MODE_CREATE`	create file if it doesn’t exist

- To combine multiple flags, use bitwise-or “`|`” in C, or addition “`+`” in Fortran
- Close the file using: **`MPI_File_close`**`(MPI_File fh)`

MPI-I/O API for reading files

After opening the file, the data can be read from files using a blocking/non-blocking synchronization mechanism, or a collective or a non-collective coordination. Let us consider two ways of reading a file using blocking and non-collective operations: (1) `MPI_File_read` or (2) `MPI_File_read_at`

`MPI_File_read` function is used to read the file using the individual file pointer and the individual file pointer is updated using **`MPI_File_seek`** .

```
int MPI_File_seek( MPI_File fh, MPI_Offset offset, int whence )
```

The *`whence`* in **`MPI_File_seek`** updates the individual file pointer according to

`MPI_SEEK_SET`: the pointer is set to offset

`MPI_SEEK_CUR`: the pointer is set to the current pointer position plus offset

`MPI_SEEK_END`: the pointer is set to the end of file plus offset

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

`MPI_File_read_at` function is used to read a file beginning at the position specified by the offset value.

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

Reading a file: readFile2.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
  MPI_File_read(fh, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Reading a file: readFile2.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
  MPI_File_read(fh, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Declaring a File Pointer

Calculating Buffer Size

Opening a File

File seek & Read

Closing a File

Reading a file: readFile1.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Combining file seek & read in one step for thread safety in **`MPI_File_read_at`**

MPI-I/O API for writing files

When opening the file in the write mode, use the appropriate flag/s in **`MPI_File_open`**: **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** and if needed, **`MPI_MODE_CREATE`**

For writing, we will use (1) **`MPI_File_set_view`** and **`MPI_File_write`** and (2) **`MPI_File_write_at`**

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

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

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

Writing a file: writeFile1.c (1)

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  int i, rank, size, offset, N=16 ;
  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 ;
  }
 //additional code on next slide
```

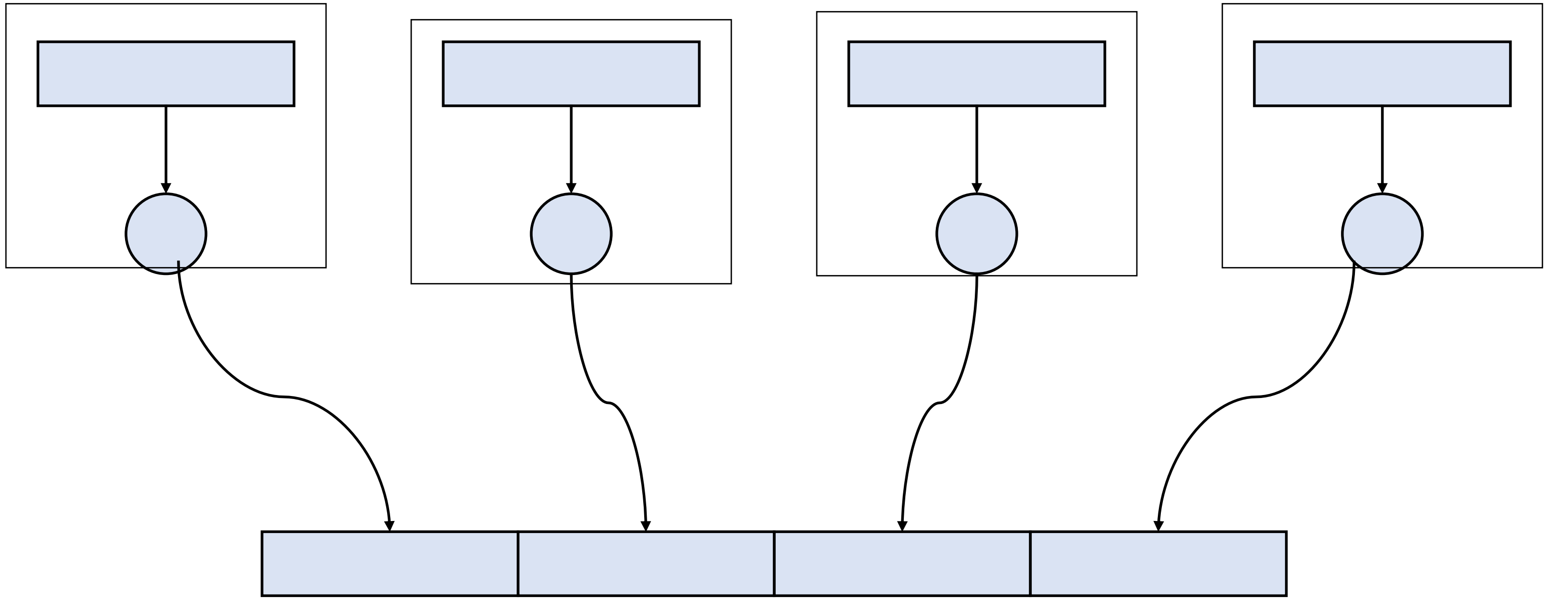
Writing a file: writeFile1.c (2)

```c
offset = rank*(N/size)*sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "datafile",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &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_Finalize();
return 0;
}
```

File views for writing to a shared file (1)

When processes need to write to a shared file, assign regions of the file to separate processes using **`MPI_File_set_view`**

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



Adapted from: Reference 2, 5, 6

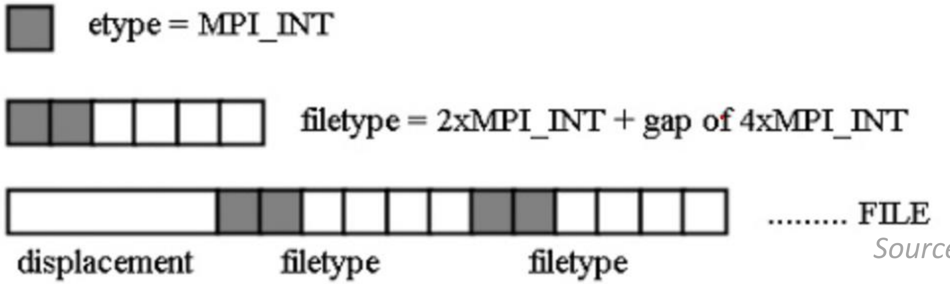
File views for writing to a shared file (2)

- File views are specified using a triplet - (*displacement*, *etype*, and *filetype*) – that is passed to **`MPI_File_set_view`**

  displacement = number of bytes to skip from the start of the file

  etype = unit of data access (can be any basic or derived datatype)

  filetype = specifies which portion of the file is visible to the process



Source: https://www.chpc.utah.edu/images/news/sp2002_Martin07.jpg

- Data representation (datarep on previous slide) can be `native, internal, external32`
  - User-defined representations supported too: `MPI_REGISTER_DATAREP`

Writing a file: writeFile2.c (1)

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  int i, rank, size, offset, N=16;
  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);
//Additional code on the next slide
```

Writing a file: writeFile2.c (2)

```
MPI_File_open(MPI_COMM_WORLD, "datafile3",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);
printf("\nRank: %d, Offset: %d\n", rank, offset);
MPI_File_set_view(fhw, offset, MPI_INT, MPI_INT, "native",
   MPI_INFO_NULL);
MPI_File_write(fhw, buf, (N/size), MPI_INT, &status);
MPI_File_close(&fhw);
MPI_Finalize();
return 0;
}
```

Compile & run the program on a compute node

```
$ mpicc -o writeFile2 writeFile2.c
$ mpirun -np 4 ./writeFile2
```

Rank: 1, Offset: 16

Rank: 2, Offset: 32

Rank: 3, Offset: 48

Rank: 0, Offset: 0

```
$ hexdump -v -e '7/4 "%10d "' -e '"\n"' datafile3
```

```
         0          1          2          3          0          1          2
         3          0          1          2          3          0          1
         2          3
```

Note about atomicity read/write

```
int MPI_File_set_atomicity ( MPI_File mpi_fh, int flag );
```

- Use this API to set the atomicity mode – 1 for true and 0 for false – so that only one process can access the file at a time

- When atomic mode is enabled, MPI-IO will guarantee sequential consistency and this can result in significant performance drop

- This is a collective function

Collective I/O (1)

- 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 request based on the combined requests of all processes and can merge the requests of different processes for efficiently servicing the requests
  - I/O requests from many processes can be aggregated

- This is particularly effective when the accesses of different processes are noncontiguous

Collective I/O (2)

- The blocking collective functions for reading and writing are as follows and their signature is similar to their non-blocking counter-parts:

```
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
  - Defined for data access routines with explicit offsets and individual file pointers but not with shared file pointers

```
MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void
*buf, int count, MPI_Datatype datatype, MPI_Request *request)
```

    - `MPI_Wait` needed

Collective non-blocking I/O example: collective_iwriteall2.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define SIZE 64
int main(int argc, char **argv){
    int *buf, *buf2, i, rank, size, nints, len, offset;
    MPI_File fh;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    buf = (int *) malloc(SIZE);
    buf2= (int *) malloc(SIZE);
    nints = SIZE/sizeof(int);
    for (i=0; i<nints; i++){
      buf[i] = rank*200 + i;
    }
    offset = rank*(SIZE/size)*sizeof(int);
    //open a file for writing the contents of the buffer named buf
    MPI_File_open(MPI_COMM_WORLD,"testing_async3.out", MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, offset, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
    MPI_File_iwrite_all(fh, buf, nints, MPI_INT , &request);
    MPI_Wait( &request, &status );
    MPI_File_close(&fh);
    //rest of the code on next slide
```

Collective non-blocking I/O example: collective_iwriteall2.c (2)

```
    //reopen the file and read the data into buf2
    for (i=0; i<nints; i++){
      buf2[i] = 0;
    }
    MPI_File_open(MPI_COMM_WORLD, "testing_async3.out", MPI_MODE_CREATE | MPI_MODE_RDWR,
MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, offset, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
    MPI_File_iread_all(fh, buf2, nints, MPI_INT, &request);
    MPI_Wait( &request, &status );
    MPI_File_close(&fh);

    //check the data read into buf2
    for (i=0; i<nints; i++) {
       printf("\nProcess %d, read: %d, stored: %d\n", rank, buf2[i], rank*200+i);
    }
    free(buf);
    free(buf2);
    free(fh);
    MPI_Finalize();
    return 0;
}
```

Compile & run the program on a compute node

```
$ mpicc -o collective_iwriteall2 collective_iwriteall2.c
$ mpirun -np 4 ./collective_iwriteall2
$ hexdump -v -e '7/4 "%10d "' -e '"\n"' testing_async3.out
         0          1          2          3          4          5          6
         7          8          9         10         11         12         13
        14         15        200        201        202        203        204
       205        206        207        208        209        210        211
       212        213        214        215        400        401        402
       403        404        405        406        407        408        409
       410        411        412        413        414        415        600
       601        602        603        604        605        606        607
       608        609        610        611        612        613        614
```

Collective I/O (3)

- “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 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-I/O hints

- 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`**
  - **`MPI_File_set_view`**

- Hints are optional and implementation-dependent
  - You can specify hints but the implementation can choose to ignore them
  - Some MPI implementations support setting the environment variables for hints

- `MPI_File_get_info` is used to get the list of hints associated with a file, examples of hints are **`striping_unit`**, and **`striping_factor`**

- Note:

`striping_factor` controls the number of OSTs across which the file should be striped

`striping_unit` controls the striping unit (in bytes)

Lustre – setting stripe count in MPI Code

- MPI may be built with Lustre support
  - MVAPICH2 & OpenMPI support Lustre

- Set stripe count in MPI code

  Use MPI I/O hints to set Lustre stripe count, stripe size, and # of writers

```
MPI_Info_set(myinfo, "striping_factor",stripe_count);
MPI_Info_set(myinfo, "striping_unit", stripe_size);
MPI_Info_set(myinfo, "cb_nodes", num_writers);
```

- Default:

  # of writers = # Lustre stripes

MPI-I/O hints example: provideHints.c

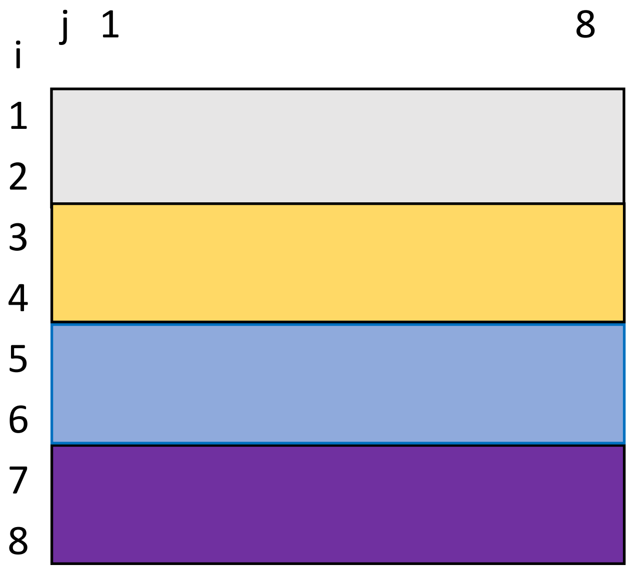
```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int i, rank, size, offset, N=160000;
   MPI_File fhw;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   int buf[N];
   for ( i=0;i<N;i++){
        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;
}
```

Checking the striping count and size of the output file written using the provideHints.c code

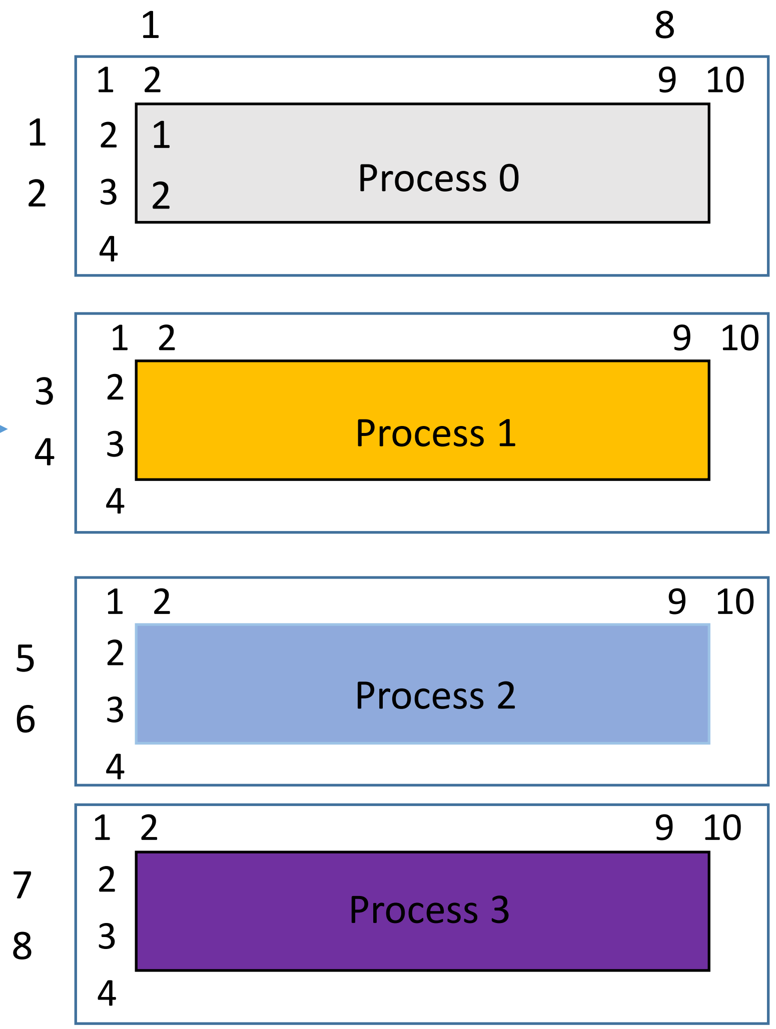
```
$ lfs getstripe datafile_striped22
datafile_striped22
lmm_stripe_count:  4
lmm_stripe_size:   4194304
lmm_pattern:       raid0
lmm_layout_gen:    0
lmm_stripe_offset: 61
        obdidx           objid          objid            group
            61       135101978      0x80d7e1a                0
            55       135019292      0x80c3b1c                0
            65       134933353      0x80aeb69                0
             9       134367978      0x8024aea                0
```

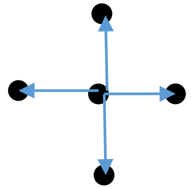
Stencil code – write the array minus halo to a file using parallel I/O (1)

- Consider a stencil code involving an 8X8 array such that the array can be broken down into 2X8 subarrays having ghost cells on top, bottom, left, right, thereby, making the subarrays of size 4X10



An 8X8 array distributed into 2X8 subarrays such that the subarrays have halo regions added to their top, bottom, left and right sides, making the size of the subarrays with halo region as 4X10 subarrays





A 5-point stencil is considered in the code. The code solves a 2D Poisson's equation using Jacobi iterative method. The original code is from Reference # 11.

Stencil code – write the array minus halo to a file using parallel I/O (2)

- After the solution has converged, each process should write its subarray to a shared file without including the halo region

- We will create a temporary array for copying the values from the required indices of the subarray with the halo region

- We will calculate the offset for each process for beginning its writing in the shared file

- We will open a file, set the view for each process, and use non-blocking collective write call

- We will also read the contents of the file back into another array and print it to verify the results

Stencil code – write the array minus halo to a file using parallel I/O (3) – snippet from poisson2D.c

```
// Here is where you can copy the solution to an array named buf and will write this array to a shared file
  int i2, j2;
  double buf[2][8];
  for ( i = i_min[my_rank], i2=0;  i <= i_max[my_rank], i2<2;  i++, i2++ ){
    for ( j = 1, j2=0; j <= N, j2<8; j++, j2++ ){
       buf [i2] [j2] = u_new[INDEX(i,j)];
    }
  }
  offset = my_rank*16*sizeof(double);
  //open a file for writing the contents of the buffer named buf which contains the copy of u_new without halo region
  MPI_File_open(MPI_COMM_WORLD,file_name, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh,offset, MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL);
  MPI_File_iwrite_all(fh, buf, 16, MPI_DOUBLE , &request);
  MPI_Wait( &request, &status );
  MPI_File_close(&fh);
```

Stencil code – write the array minus halo to a file using parallel I/O (4) – snippet from poisson2D.c

```
double buf2[2][8];
//reopen the file and read the data into buf2
for (i=0; i<2; i++){
  for (j=0; j<8; j++){
     buf2[i][j] = 0;
  }
}
MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, offset, MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL);
MPI_File_iread_all(fh, buf2, 16, MPI_DOUBLE, &request);
MPI_Wait( &request, &status );
MPI_File_close(&fh);
for (i=0; i<2; i++) {
   printf("\nFrom process %d\n", my_rank);
   for(j=0; j<8;j++){
     printf(" %lf ", buf2[i][j]);
   } printf("\n");
}
```

Hexdump of binary file written by poisson2D.c

```
$ hexdump -v -e '8/8 "%20f "' -e '"\n"' poisson_out.out
```

0.003786	0.007572	0.010202	0.008208	0.005329	0.003007	0.001470	0.000566
0.007572	0.016299	0.025028	0.017301	0.010103	0.005228	0.002311	0.000792
0.010202	0.025028	0.056311	0.025868	0.012553	0.005495	0.001752	0.000292
0.008208	0.017301	0.025868	0.017308	0.008750	0.002448	-0.001085	-0.001376
0.005329	0.010103	0.012553	0.008750	0.002691	-0.003364	-0.007165	-0.004709
0.003007	0.005228	0.005495	0.002448	-0.003364	-0.011432	-0.019497	-0.010294
0.001470	0.002311	0.001752	-0.001085	-0.007165	-0.019497	-0.049098	-0.016969
0.000566	0.000792	0.000292	-0.001376	-0.004709	-0.010294	-0.016969	-0.008485

References

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. John Burkardt: https://people.sc.fsu.edu/~jburkardt/c_src/poisson_mpi/poisson_mpi.html

Optimizing I/O in Scientific Applications Part-3

Better Scientific Software Fellowship (BSSw Fellowship)

Ritu Arora

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

GitHub: https://github.com/ritua2/bsswfellowship

LinkedIn: https://www.linkedin.com/in/ritu-a-59b58ab/

Twitter: https://twitter.com/ritzaa2

YouTube: https://www.youtube.com/user/ritua2

Topics to be covered

1) Why should we optimize I/O? (general review)

2) How can we optimize I/O? (general review plus one example code in C)

3) General I/O strategies for working on High Performance Computing (HPC) platforms (general review)

4) Introduction to the Lustre parallel file system (general review and demo of some Lustre commands)

5) Introduction to parallel I/O patterns

6) Introduction to MPI I/O

7) I/O in the context of checkpointing and AI

Pre-Event Survey

https://tinyurl.com/278sap7x

https://forms.gle/qKZJ8dtPQKGUP3rx6

Google Doc Link for Q&A

https://tinyurl.com/yjzakhf6

https://docs.google.com/document/d/1epOWgJg4ffcibUbAFlchXg0SvccxMCb8zlK6NwZzRT8/edit?usp=sharing

Situations that may arise while working on shared computing platforms (1)

Greetings,

We have a job to run on one node that is hitting the 72-hour limit of the compute_1_job queue. Is it possible to extend the limit to 144 hours for this one job?

Thanks,

AWC (A Worried Customer)

****

Hello Support Team,

I was running some simulations on the Curie system and found that some simulations ended before converging due to the 72-hour limit on the queued jobs in the compute_1_job queue. I was wondering whether there is a way for me to run simulations without these limits.

Thanks,

YAWC (Yet Another Worried Customer)

Situations that may arise while working on shared computing platforms (2)

Lustre filesystem issue

We are currently seeing issues with one of the XYZ Lustre filesystem's object storage servers (OSS). This is leading to access problems for some files and directories on /xyz/lustre/scratch and /xyz/lustre/projects. We will update once the issue is resolved.

Thanks

User Services

$ABC FILE SYSTEM STATUS

From: Operations Team

The /abc filesystem experienced a hardware problem with one of the storage controllers that began around 7:30 Central this morning and impacted a portion of the filesystem. The problem was resolved at 9:00AM and the filesystem has resumed normal operation. User processes may have experienced hangs or pauses trying to access files in /abc during this time.

To extend or not to extend the job run-time beyond default values

- Certain research and exploratory work may require running software applications for several days or weeks and despite allowing them to run for longer than normally allowed durations, it is not uncommon to see the applications getting interrupted due to unexpected reasons
- On certain occasions, we have seen such long-running applications crash due to unexpected hardware issues that are triggered by the applications themselves
- On some other occasions we had to urgently apply some firmware updates or do security related system patching due to which, we had to announce emergency maintenance and take the HPC system offline, thereby, terminating all the applications running on the system at that time
- On yet another occasion, we saw some long-running applications getting terminated due to the actions of other customers on the shared HPC platform - just one customer can cause heavy, suboptimal IO that can bring down the filesystem for the entire customer base
- Due to such reasons, it is important to consider checkpointing

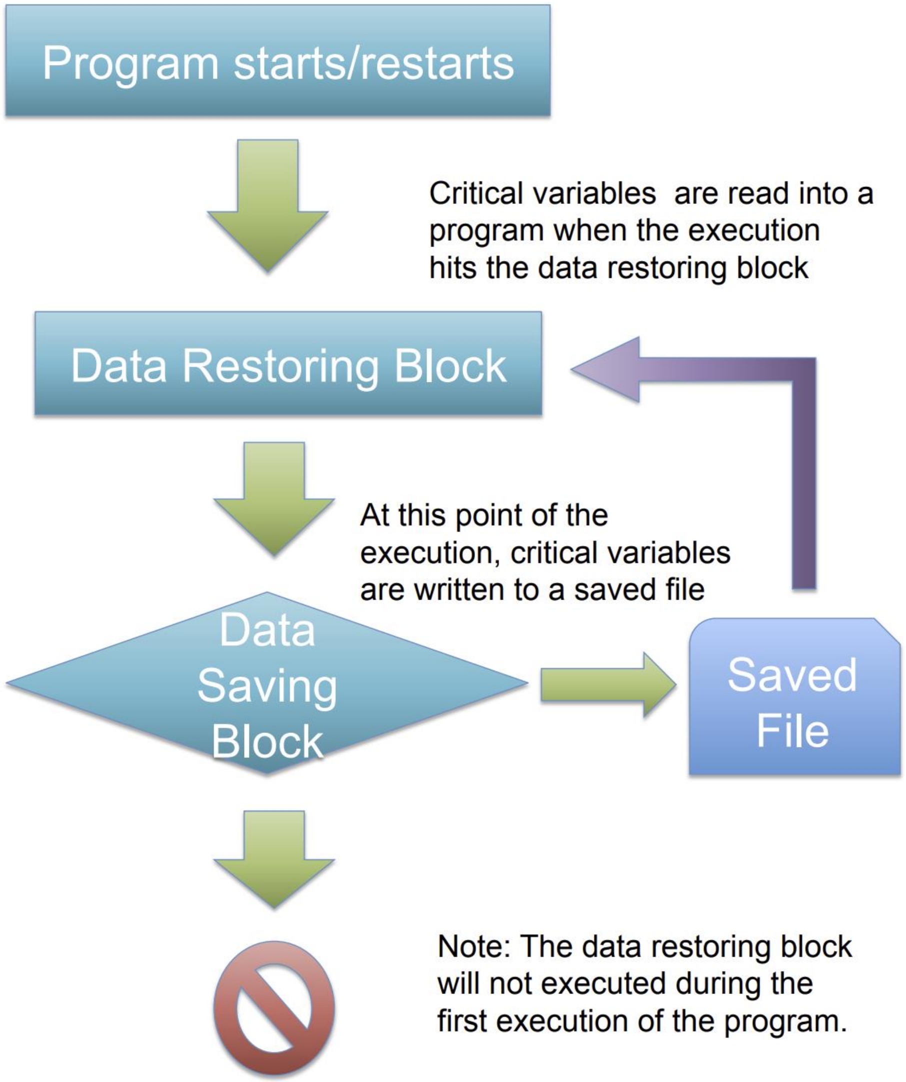
I/O and Checkpointing (1)

- Checkpointing is the process of periodically saving (or writing) the execution state of an application such that in the event of an interruption in the execution of an application, this saved state can be used to continue the execution at a later time

- Typically, the execution state is written to a file

- Resuming the execution of an application using a previously saved state or checkpoint (instead of starting it from scratch) is referred to as the Restart phase

- Checkpointing not only saves time by offering the capability to resume the execution of an application in case of a hardware failure in the underlying computing platform (e.g., network interconnect failure) or if the computing platform becomes unavailable due to emergency maintenance, but it also helps in overcoming the time-limits associated with the different job queues/partitions
  - Additionally, for AI models that can take very long to train, checkpointing and restart can help in inspecting the intermediate results during the inferencing and model training process



I/O and Checkpointing (2)

- Types of checkpointing
  - **System-level**: involves taking core-dumps of the computational state of the machine or system on which the application is running, example: Berkeley Lab Checkpointing and Restart (BLCR)
    - Pros: convenient to use, no code changes needed, user only specifies the checkpointing frequency
    - Cons: involves large memory-footprint of checkpoints as the entire execution state of the application and the operating system processes are saved, and system administrator level privileges are needed for installing additional code

  - **Library-level or user-level**: involves the use of libraries for taking checkpoints while being agnostic to kernel-level information such as process IDs, example: DMTCP
    - Pros: useful for checkpointing applications without requiring any changes to the source-code or the operating system kernel
    - Cons: The users may need to load the checkpointing library before starting their applications, and then, would need to dynamically link the loaded library to their applications, the checkpoints can have a large memory-footprint

I/O and Checkpointing (3)

- Types of checkpointing
  - **Application-level**: involves implementing the checkpoint-and-restart mechanism within the application itself
    - **Pros**: An efficient implementation of application-level checkpointing would require saving and reading the state of only those variables or data that are necessary for recreating the state of the entire application and such variables or data are referred to as critical variables/data, it does not rely on the availability of any external libraries or tools, and hence, is useful for writing portable code
    - **Cons**: While an efficient implementation of this technique will generate checkpoints with smaller memory footprint and incur lesser I/O overheads as compared to other types of checkpointing, the onus is on the user (or the developer) to manually implement it on a per application basis
    - Some of the popular applications that support the feature of checkpointing and restart are FLASH astrophysics code and Abaqus

I/O and Checkpointing (4)

- Depending upon the frequency of checkpointing, type of checkpointing, and the amount of data to be written in the checkpoint file, <u>checkpointing can become an I/O intensive activity</u>

- Hence, it is important to consider optimization of the I/O involved in checkpointing and restart process to ensure that the applications do not incur large I/O overheads or memory footprints

- In this session, we will cover some examples of implementing application-level checkpointing in serial and parallel applications written using different programming languages/frameworks

Application-Level Checkpointing and Critical Variables

```
int main(){
 int x = 4;
 int y = sqrt(x);
 int z, i;
 int j = x*y;
 for (i =0; i< 100; i++){
    z += j* myFct (randomNumber * i);
 }
 return 0;
 }
```

Note: Definition of function *myFct* is not included here.

In this code, "i" and "z" are critical variables as their values are updated in the code and cannot be derived easily to recreate the execution state of the code once it is interrupted.

Hence, in order to implement application-level checkpointing in this code, we can save the values of these critical variables in a file, and read them back into the code during the restart phase.

Checkpointing in serial applications – C++ - sample_code_with_ckpt.cpp

```
if(rose_count_in_file==rose_count_b){
        rose_start = rose_start_from_file;
        v = rose_v_from_file;
}
for(int i = rose_start ; i < 10; i++) {
        if(i==rose_start){
                printf("\nstarting out from i= %d\n", i);
        }
        if(i%5==0){
                fp = fopen ("chkpt_b.txt", "w");
                fprintf(fp, "%d\n", rose_count_b);
                fprintf(fp, "%d\n", i);
                fprintf(fp, "%d\n", v);
                fclose(fp);
          }
          v+=i;
          printf("Value changed to %d\n", v );
          sleep(1)
}
```

Reading the value of i from the file – if these values do not exist the program will run normally

Reading the value of v from the file

These are the critical variables – i and v - that are being written to the checkpoint file

Complete code is available at: https://github.com/ritua2/bsswfellowship

Checkpointing in Python with the Pickle package: chkpt.py (1)

```
import os
import time
import pickle
saved_dump = "ckptfile.pickle"
def main(start=0):
    #some useful code before this line
    global saved_dump
    a = start
    while 1:
        time.sleep(1)
        a += 1
        print(a)
        with open(saved_dump, 'wb') as f:
            pickle.dump(a, f)

if __name__ == '__main__':
    print("For testing, interrupt the running code by typing ctrl+c")
    while 1:
        if os.path.exists(saved_dump):
            with open(saved_dump, "rb") as f:
                start = pickle.load(f)
        else:
            start = 0
        try:
            main(start=start)
        except KeyboardInterrupt:
            resume = raw_input('Would you like to continue running the code? Type the letter y for yes.')
            if resume != 'y':
                break
```

```
$ python chkpt.py
For testing, interrupt the running code by typing ctrl+c
1
2
3
^CWould you like to continue running the code? Type the letter y for yes.y
4
5
6
7
8
9
10
11
12
13
^CWould you like to continue running the code? Type the letter y for yes
```

Checkpointing in Python with the Pickle package: chkpt.py (2)

```
$ python chkpt.py
For testing, interrupt the running code by typing ctrl+c
1
2
3
4
5
6
7
^CWould you like to continue running the code? Type the letter y for yes.
```

#After running the code for the first time, you will see the `chptfile.pickle` file generated

```
$ ls
chkpt.py ckptfile.pickle
```

#Let us remove the chptfile.pickle file

```
$ rm ckptfile.pickle
$ ls chkpt.py
```

#Let us run the code again. It will start from the beginning as we removed the file chptfile.pickle. We will interrupt the code while it is running using ctrl+c, and then restart

Checkpointing in Python with the Pickle package: chkpt.py (3)

```
$ python chkpt.py
For testing, interrupt the running code by typing ctrl+c
1
2
3
4
5
6
7
8
9
^CWould you like to continue running the code? Type the letter y for yes.y
10
11
12
13
14
15
^CWould you like to continue running the code? Type the letter y for yes.y
16
17
18
^CWould you like to continue running the code? Type the letter y for yes
```

Checkpointing in Python with the Pickle package: chkpt.py (4)

```
$ ls
chkpt.py ckptfile.pickle
```

Now let us run the code again. Because the `chptfile.pickle` is present, the series in the example below begins from 19 and not 1. Before the code was interrupted as shown in the previous slide, the last number that was printed was 18

```
$ python chkpt.py
For testing, interrupt the running code by typing ctrl+c
19
20
21
22
23
^CWould you like to continue running the code? Type the letter y for yes.
```

Checkpointing in TensorFlow: chkpt_tensorflow.py

- TensorFlow supports the feature of checkpointing and restart by offering APIs for saving and loading the model states, thereby reducing the training time during the restart phase in the event of an interruption

```
# the code for creating a checkpoint
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filename, monitor='val_loss',
verbose=1, save_best_only=True, mode='min', save_freq="epoch")

model.fit(x_train, y_train, epochs=5, batch_size = 100, validation_split = 0.1,
callbacks=[checkpoint_callback])

# the code for loading the saved model/checkpoint from a file
model = tf.keras.models.load_model(filename)
```

- There are multiple options supported for the ModelCheckpoint call and details are at the following link: https://www.tensorflow.org/tutorials/keras/save_and_load

- Complete code is available through the GitHub repo at the following link: https://tinyurl.com/ymz5e8xm

There are different APIs for checkpointing files and saving files

Tensorflow provides separate APIs for (1) saving and loading the models and (2) for checkpointing

While the checkpoints in Tensorflow include the values of the trained parameters used by the models (e.g., model weights and biases), they do not include any description of the computations that are done with the models, and hence, the same source code that was used for writing a checkpoint and reading it back is required for using a checkpoint in future. In situations where only model weights should be saved (perhaps for the inference step), checkpointing is recommended.

In contrast to checkpointing, saving a model in Tensorflow includes a serialized description of the computations defined by the model along with the trained parameter values, thereby making the saved models independent of the original source code that was used to build them.

The **tf.saved_model API or the tf.keras.Model API can be used for saving models in Tensorflow**, where as the **tf.train.checkpoint API in Tensorflow or ModelCheckpoint method in Keras can be used for checkpointing.**

Checkpointing in PyTorch: chkpt_pytorch.py

- PyTorch is an open-source deep learning framework that supports checkpointing by saving and loading dictionaries called `state_dict` objects for models and optimizers in addition to saving and loading data such as epoch number, and training loss. It also supports saving and loading models from multiple processes using the torch.distributed.checkpoint API.

  - During the saving stage, the data is serialized and saved using `torch.save`

```
torch.save({
            'epoch': EPOCH,
            'model_state_dict’: cnn.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': LOSS,
            }, PATH)
```

  - Saved data can be loaded using `torch.load`

```
restart = torch.load(PATH)
model.load_state_dict(restart['model_state_dict'])
optimizer.load_state_dict(restart['optimizer_state_dict'])
epoch = restart['epoch']
loss = restart['loss']
```

Checkpointing in R: chkpt_Rscript.R

```
args = commandArgs(trailingOnly = TRUE)
if (length(args)==0) {
  print("code is running normally")
  dd = read.csv("input.csv",header = FALSE)
  mat = as.matrix(dd)
  mat = mat +1
  print(mat)
  save(mat, file="chkpt.Rdata")
  Sys.sleep(100)
  mat = mat +1
  print(mat)
  save(mat, file="chkpt.Rdata")
 } else if(length(args) == 4 & args[3] == "restart") {
  load(args[2])
  mat = mat+1
  print(mat)
  save(mat, file=args[4])
 }else{
  print("Correct usage of this code in restart mode is: Rscript testRscript.R --args chkpt.Rdata restart newchkput.Rdata")
  print("The restart flag is passed through the command-line")
 }
```

Reading an input CSV file

The matrix is being saved to a file using the save method in R – this step will create a binary file

The matrix is being loaded from a file using the load method in R

Creating SLURM job dependencies for restarting jobs (1)

- If an HPC system uses SLURM, job dependencies can be created to automatically restart from the latest checkpoint after any interruption or time-out from the job queue
- Let us use the previous R script for this example and submit a SLURM job with it, interrupt the job by using `scancel`, and resume the job from the restart file
- Below is the job script named myJob.sh

```
#!/bin/bash
#SBATCH -J myRJob
#SBATCH -o myRJob.o%J
#SBATCH -p normal
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 00:05:00

module load Rstats
Rscript testRscript.R
```

Creating SLURM job dependencies for restarting jobs (2)

- Here is the job script for named myRestartJob.sh

```
#!/bin/bash
#SBATCH -J myRJob
#SBATCH -o myRJob.o%J
#SBATCH -p normal
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 00:05:00

module load Rstats
Rscript testRscript.R --args chkpt.Rdata restart newchkpt.Rdata
```

Creating SLURM job dependencies for restarting jobs (3)

- Submit the first job

```
$ sbatch myJob.sh
Submitted batch job 11138117
```

- Use the job id of the first job and submit the second job after creating a dependency on the previous job using afterok/afternotok option as appropriate

```
$ sbatch --dependency=afternotok:11138117 myRestartJob.sh
```

- Interrupt the first job

```
$ scancel 1113817
```

- The second job starts and resumes further after reading data form the checkpoint file

Creating SLURM job dependencies for restarting jobs (4)

- Inspect the output files from the first and second jobs

```
$ cat myRJob.o11138117
[1] "code is running normally"
     V1 V2 V3 V4 V5
[1,] 52 52 52 52 52
[2,] 52 52 52 52 52
[3,] 52 52 52 52 52
[4,] 52 52 52 52 52
[5,] 52 52 52 52 52

$ cat myRJob.o11138120
     V1 V2 V3 V4 V5
[1,] 53 53 53 53 53
[2,] 53 53 53 53 53
[3,] 53 53 53 53 53
[4,] 53 53 53 53 53
[5,] 53 53 53 53 53
```

Note: data transfer related inefficiencies can be addressed at both pre-processing and post-processing stages

- During the pre-processing stage
  - Transferring large amounts of uncompressed data from source to the destination
  - Not selecting the optimal data transfer protocol/method

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

- During the post-processing stage
  - Transferring large amounts of uncompressed data from source to the destination
  - Not selecting the optimal data transfer protocol/method

Exercises: goals & activities

- You will learn
  - How to do parallel I/O using MPI

- What will you do
  - Compile and execute MPI code
  - Modify the MPI code for the exercises to embed the required MPI routines

Accessing files for the exercises

- Log on to the HPC system of your choice using **`your_login_name`**
- Download, copy, and uncompress the file, **`mpiiotut.zip`** that is available in the following GitHub repo: https://github.com/ritua2/bsswfellowship/tree/main/mpiio
- Copy the downloaded zip file – assuming its name is mpiiotut.zip - to the remote HPC system

```
scp mpiio.zip <username>@< your HPC system host name>:<path to the directory where the file should be copied>
```

- Steps to connect to the remote HPC system

```
ssh <your_login_name>@<your HPC system host name>

<switch to the desired directory>

unzip mpiio.zip

cd mpiio
```

Exercise 0 (to familiarize yourself with the HPC system)

- **Objective**: practice compiling and running MPI code on the HPC system of your choice
- Note: the commands below will work on HPC systems having the SLURM job scheduler. Please refer to the user-guide of your HPC system to find the right commands to use if another job scheduler is used there.
- Switch to the MPI directory

  `login3$ cd basicmpi`

- Compile the sample code `mpiExample4.c`

  `login3$` **`mpicc -o mpiExample4 mpiExample4.c`**

- Modify the job script, `myJob.sh`, to provide the name of the executable to the `mpirun` command
- Submit the job script to the SLURM queue and check it's status

  `login3$` **`sbatch myJob.sh`** (you will get a `job id`)

  `login3$` **`squeue`** (check the status of your job)

- When your job has finished executing, check the output in the file

  `myMPI.o<job id>`

Exercise 1

- **Objective**: **Learn to use MPI I/O calls**

- Modify the code in file `exercise1.c`

    - Read the comments in the file for modifying the code

        - Extend the variable declaration section as instructed
        - You have to add MPI routines to open a file named "`datafile_written`", and to close the file
        - You have to fill the missing arguments of the routine **`MPI_File_write_at`**

    - See the slide # 37 for details on the MPI routines for writing files

- Compile the code and execute it via the job script using 4 MPI processes (see Exercise 0 for the information related to compiling the code and the jobscript)

Viewing the output file

To view the output file, use hexdump

```
$ hexdump -v -e ’8/4 "%10d "' -e '"\n"' datafile_written
 0         1         2         3         0         1         2         3
 0         1         2         3         0         1         2         3
```

Exercise 2

- **Objective**: **Learn to use collective I/O calls**

- Modify the code in file `exercise2.c`
  - Read the comments in the file for modifying the code
    - Use the `MPI_File_write_all` function in the specified place in the program
    - Remember, `MPI_File_write_all` is the collective version of `MPI_File_write` and uses the same arguments
  - Compile the code

```
$ mpicc -o exercise2 exercise2.c
```

  - Run interactively on 4 tasks using srun and mpirun

```
$ srun –n 4 –N 1 –p normal

$ mpirun –np 4 ./exercise2
```

Viewing the output file

To view the output file, use hexdump

```
$ hexdump -v -e ’8/4 "%10d "' -e '"\n"' datafile_written
 0         1         2         3         0         1         2         3
 0         1         2         3         0         1         2         3
```

Post-Event Survey

https://forms.gle/732HrgAVLHoy8cNo6

References

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. John Burkardt: https://people.sc.fsu.edu/~jburkardt/c_src/poisson_mpi/poisson_mpi.html
12. PyTorch, performance tuning guide: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html
13. TensorFlow, optimize pipeline performance: https://www.tensorflow.org/guide/data_performance
14. Keras examples: https://keras.io/examples/vision/mnist_convnet/
15. Nutan Sharma, PyTorch CNN with MNIST: https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118
16. PyTorch, saving and loading general checkpoints: https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html