Introduction to Parallel I/O

John Cazes, Ritu Arora

Texas Advanced Computing Center

September 26th, 2013

Email: {cazes, rauta}@tacc.utexas.edu

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

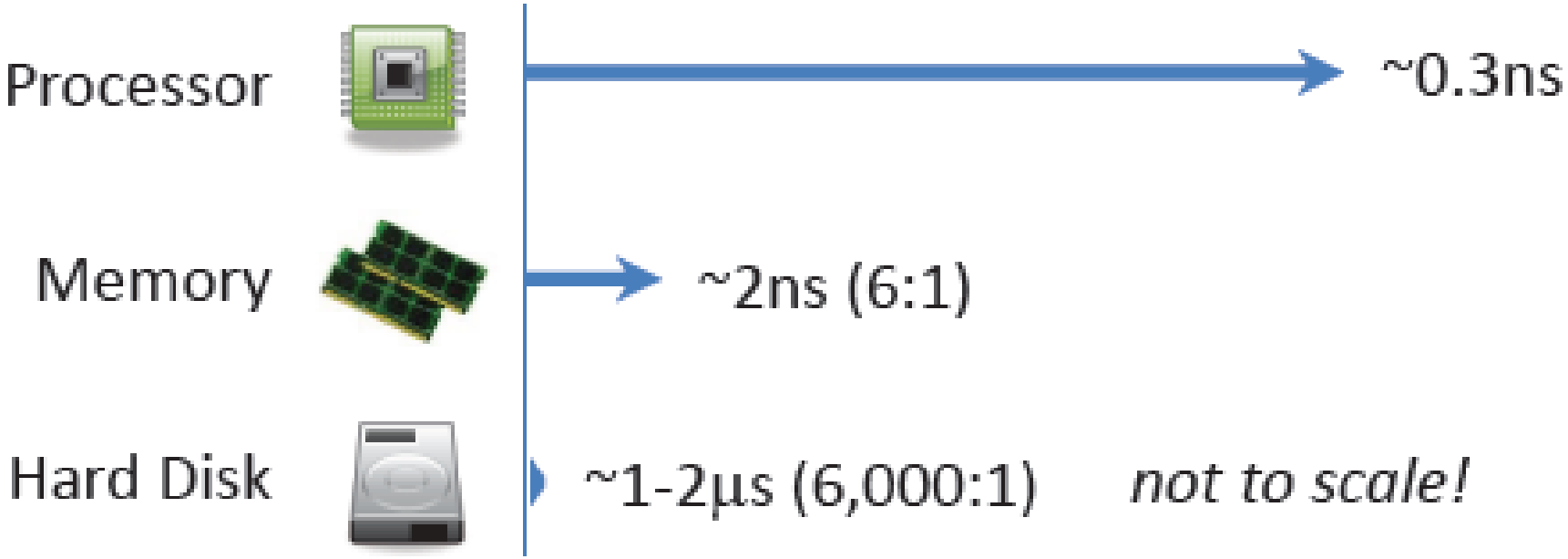
High Performance Computing & I/O

- High Performance Computing (HPC) applications often do I/O for
  - Reading initial conditions or datasets for processing
  - Writing numerical data from simulations
    - Parallel applications commonly need to write distributed arrays to disk
    - Saving application-level checkpoints
      - Application state is written to a file for restarting the application in case of a system failure

- Efficient I/O without stressing out the HPC system is challenging
  - Load and store operations are more time-consuming than multiply operations
  - **Total Execution Time = Computation Time + Communication Time + I/O time**
  - **Optimize all the components of the equation above to get best performance**

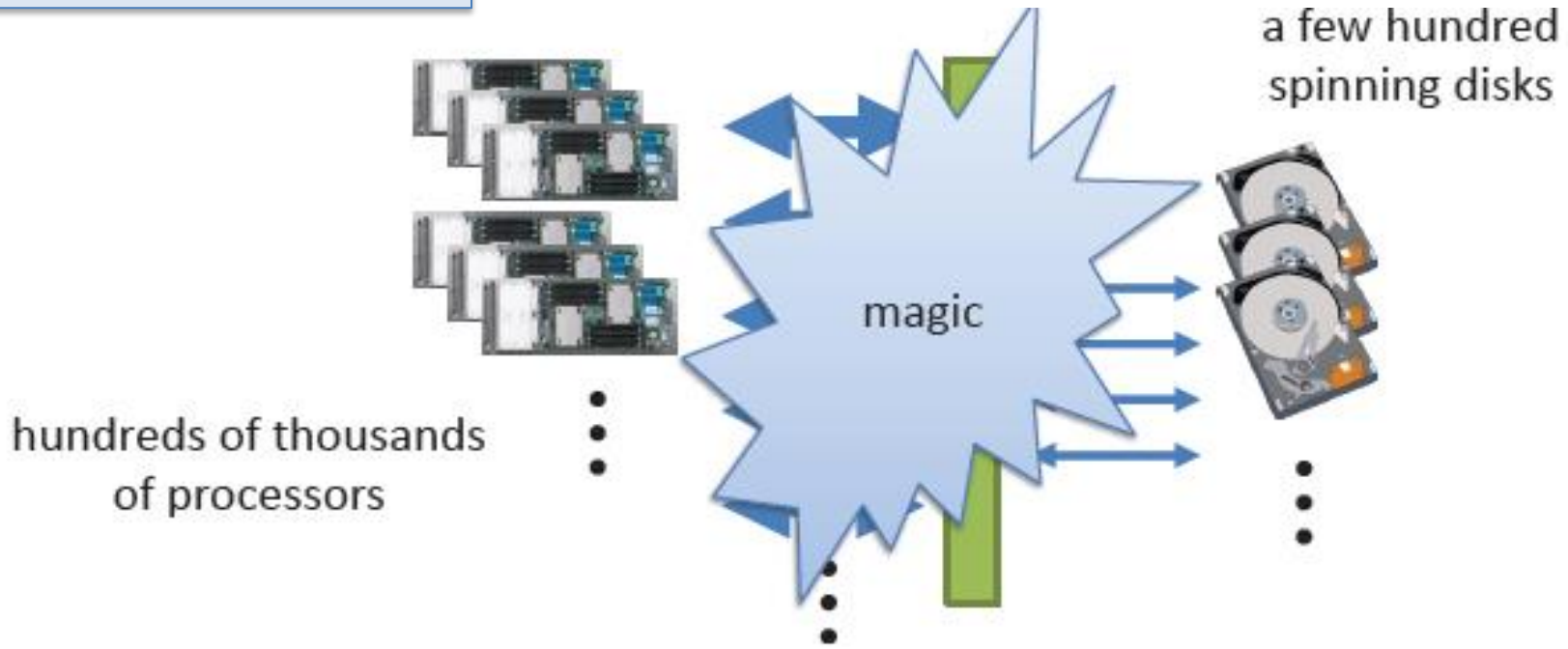
Relative Speed of Components in HPC Platform

- An HPC platform's I/O subsystems are typically slow as compared to its other parts.
- The I/O gap between memory speed and average disk access stands at roughly 10^{-3}

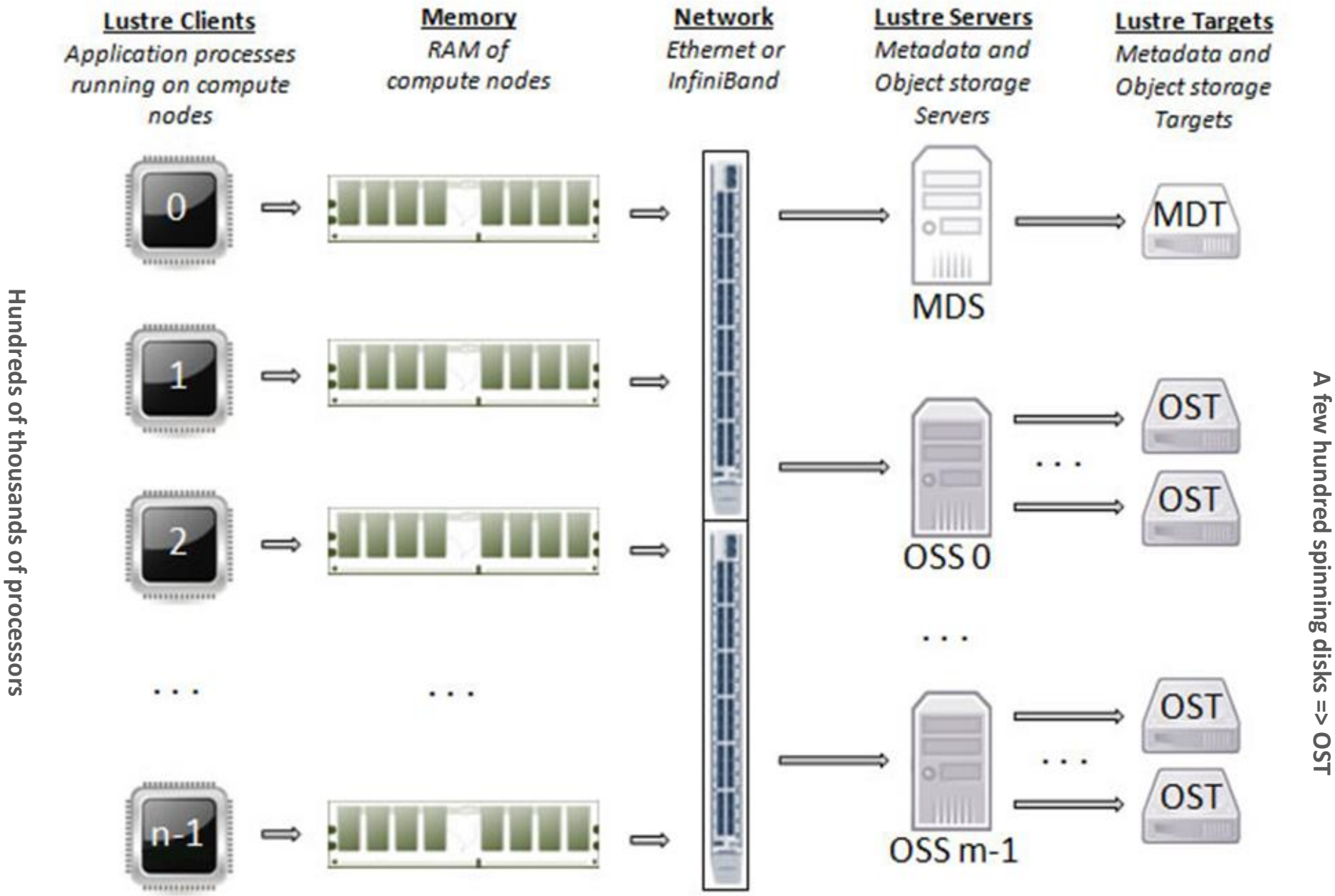


Managing Storage Hardware & Allowing Concurrent Data Access in a Cluster

…we need some magic to make the collection of spinning disks act like a single disk…



…Lustre File System Provides the Magic



Source: Reference 1

Parallel I/O – Why & How?

- Goal of Parallel I/O is to use parallelism to increase bandwidth
- Parallel I/O can be hard to coordinate and optimize if working directly at the level of Lustre API or POSIX I/O Interface (not discussed in this tutorial)
- Therefore, specialists implement a number of intermediate layers for coordination of data access and mapping from application layer to I/O layer
- Hence, application developers only have to deal with a high-level interface built on top of a software stack that in turn sits on top of the underlying hardware

Summary of the Discussion so Far

- Different hardware components in an I/O subsystem of an HPC platform operate at different speeds

- Optimizing the I/O time is as important as optimizing the computation and communication time in an application

- A number of intermediate layers (sitting between low-level hardware layer and the top-level application layer) have been implemented on top of which, parallel applications can be developed
  - MPI-I/O, parallel HDF5, parallel netCDF, T3PIO,…

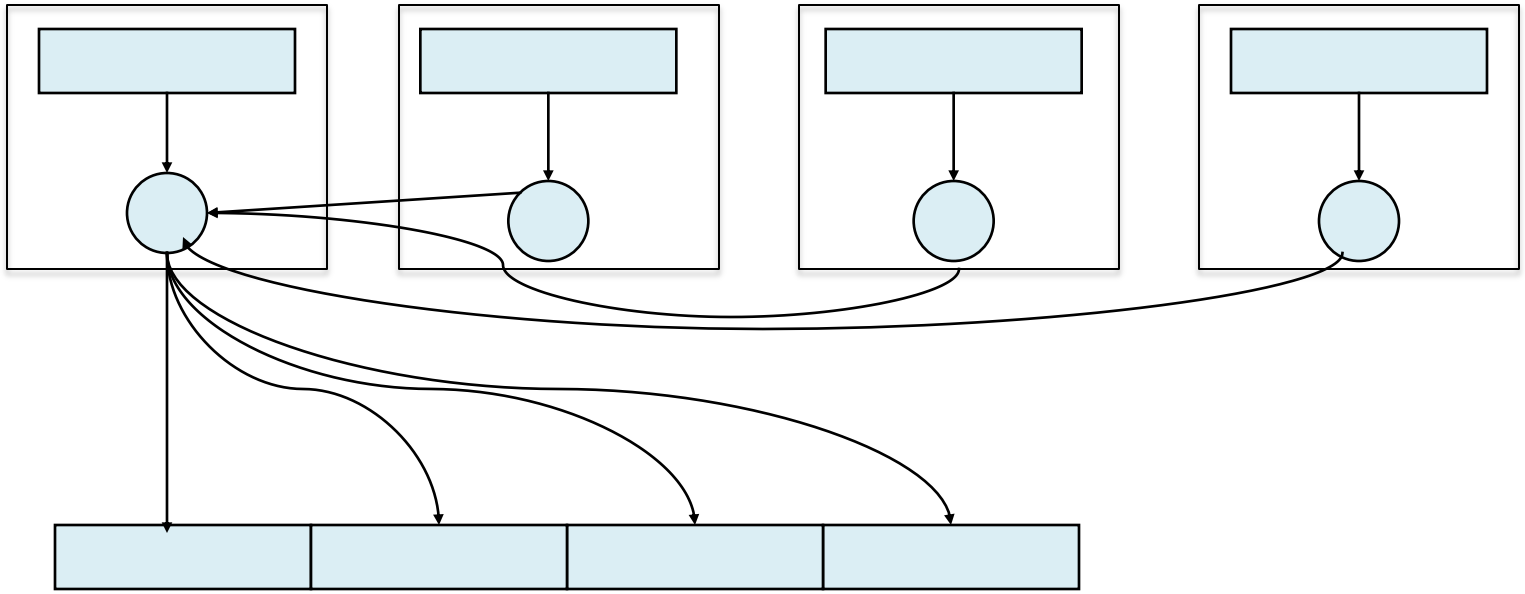
Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

Typical Pattern: Parallel Programs Doing Sequential I/O

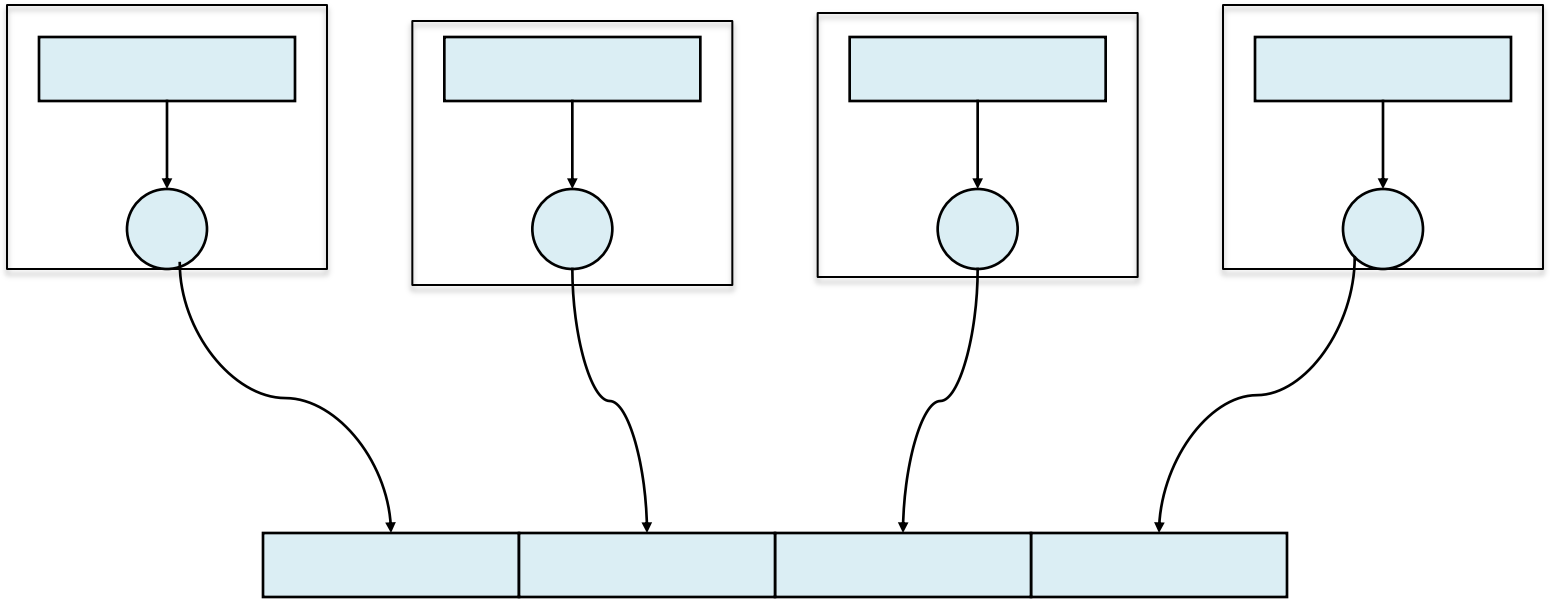
- All processes send data to master process, and then the process designated as master writes the collected data to the file

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

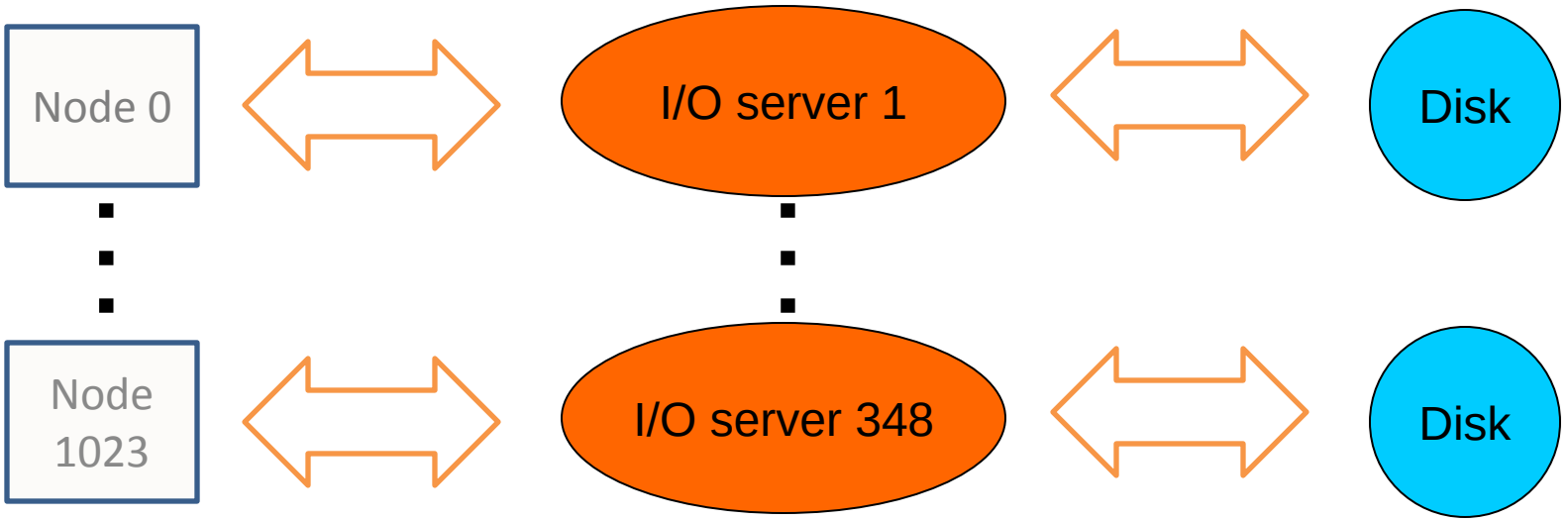


Desired Pattern: Parallel Programs Doing Parallel I/O

- 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

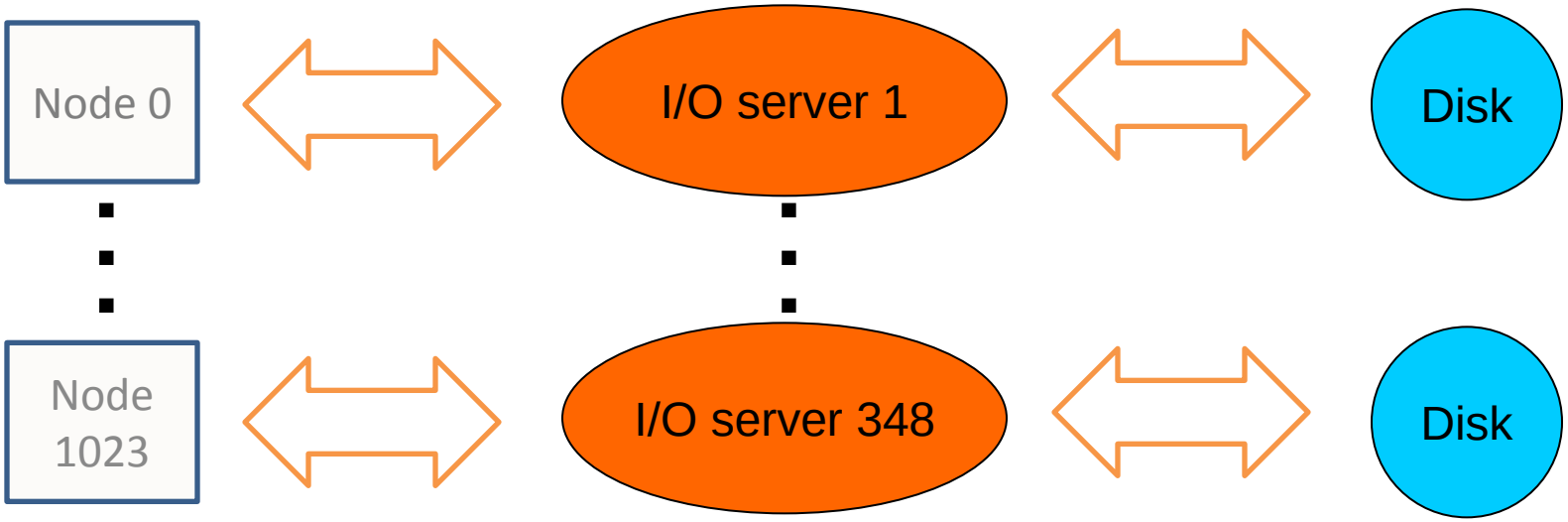


Balance Parallel I/O (1)



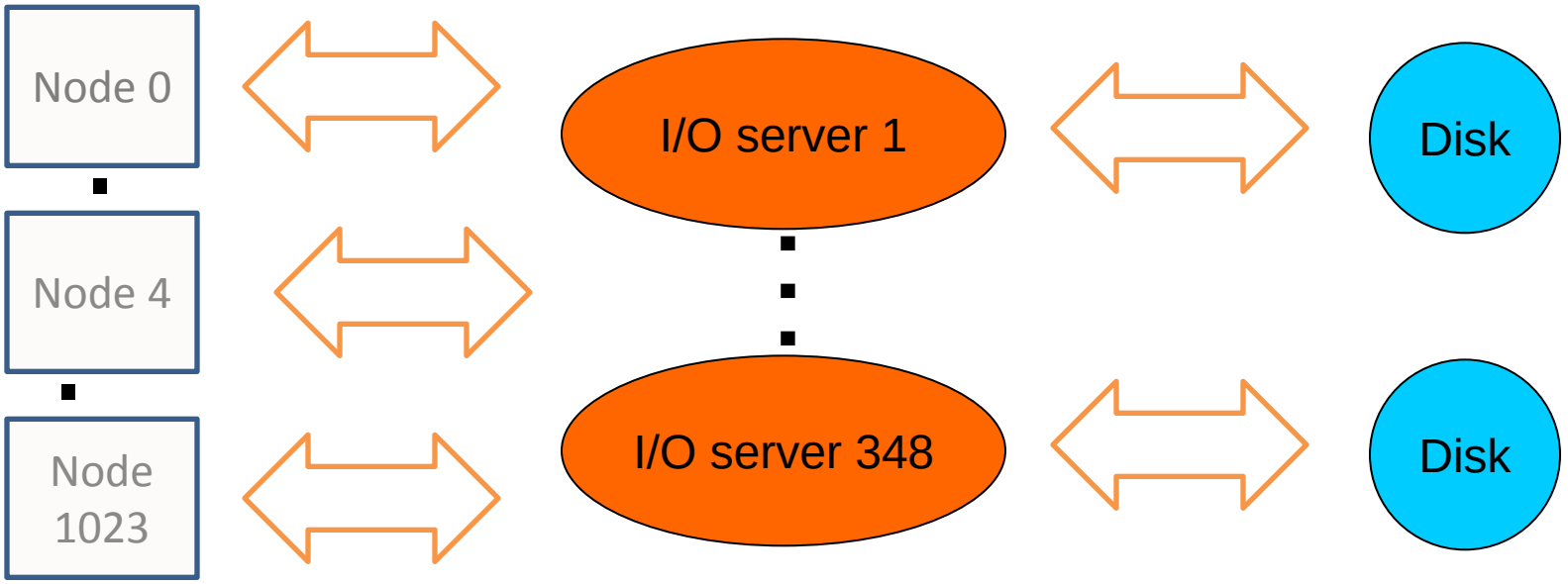
1024 nodes x 16 tasks = 16384 I/O clients
Terribly oversubscribed

Balance Parallel I/O (2)



1024 nodes x 1 task = 1024 I/O clients
Much better

Balance Parallel I/O (3)



160 nodes x 1 task = 160 I/O clients

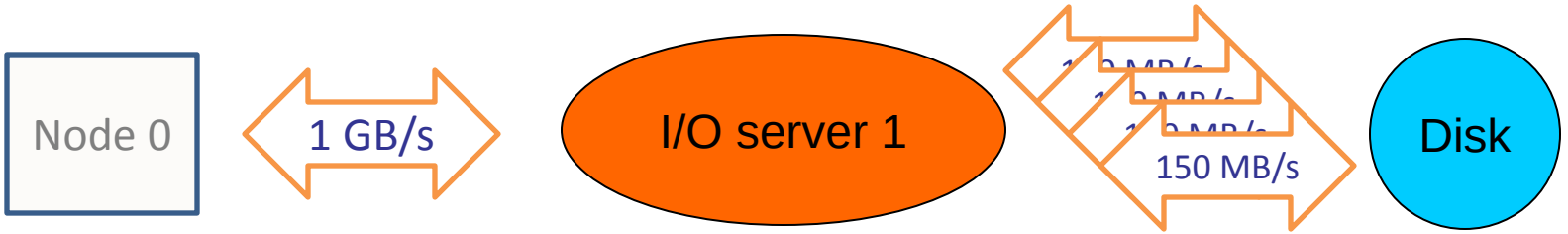
Best for I/O bandwidth

But, reality sets in

- I/O servers are shared
- Still have to move the data to other tasks

Balance Serial I/O -- If you must

– Match the I/O bandwidth with the switch bandwidth

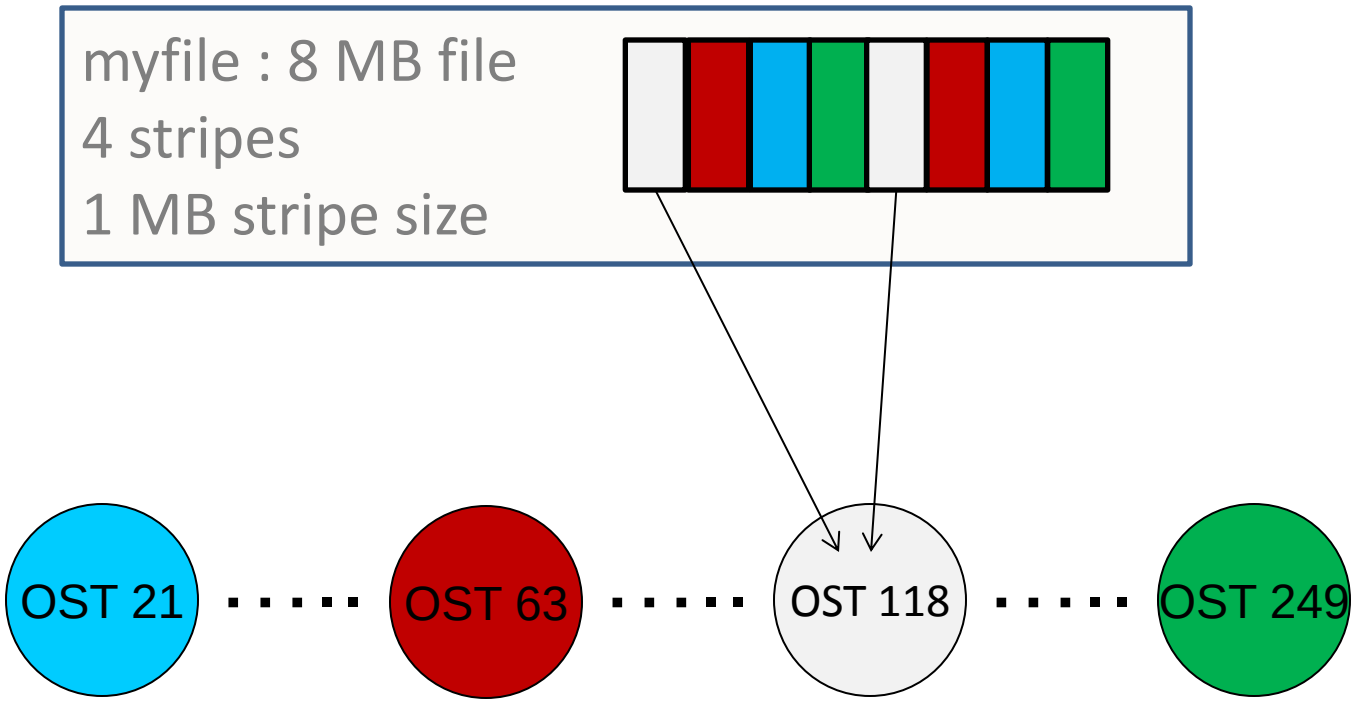


– Set the stripe count to 4 to saturate the IB bandwidth (Lustre only)

- `lfs setstripe -c 4 input_dir`

Lustre

- Lustre supports the “striping” of files across several I/O servers (similar to RAID 0)
- Each stripe is a fixed size block



Lustre

- Administrators set a default stripe count and stripe size that applies to all newly created files
  - Stampede: $SCRATCH: 2 stripes/1MB
    $WORK: 1 stripe /1MB
  - Lonestar: $SCRATCH: 2 stripes/1MB
    $WORK: 1 stripe /1MB
- However, users can reset the default stripe count or stripe size using the “lfs setstripe” command

Lustre

- Get stripe count

```
 % lfs getstripe ./testfile
./testfile
lmm_stripe_count:   2
lmm_stripe_size:    1048576
lmm_stripe_offset:  50
        obdidx           objid          objid              group
            50         8916056       0x880c58                  0
            38         8952827       0x889bfb                  0
```

- Set stripe count

```
% lfs setstripe -c 4 -s 4M testfile2
% lfs getstripe ./testfile2
./testfile2
lmm_stripe_count:   4
lmm_stripe_size:    4194304
lmm_stripe_offset:  21
        obdidx           objid          objid              group
            21         8891547       0x87ac9b                  0
            13         8946053       0x888185                  0
            57         8906813       0x87e83d                  0
            44         8945736       0x888048                  0
```

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

Lustre

- 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

  Fortran:

```
call mpi_info_set(myinfo,"striping_factor",stripe_count,mpierr)
call mpi_info_set(myinfo,"striping_unit",stripe_size,mpierr)
call mpi_info_set(myinfo,"cb_nodes",num_writers,mpierr)
```

  C:

```
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

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies

MPI for Parallel I/O

- A parallel I/O system for distributed memory architectures will need a mechanism to specify collective operations and specify noncontiguous data layout in memory and file

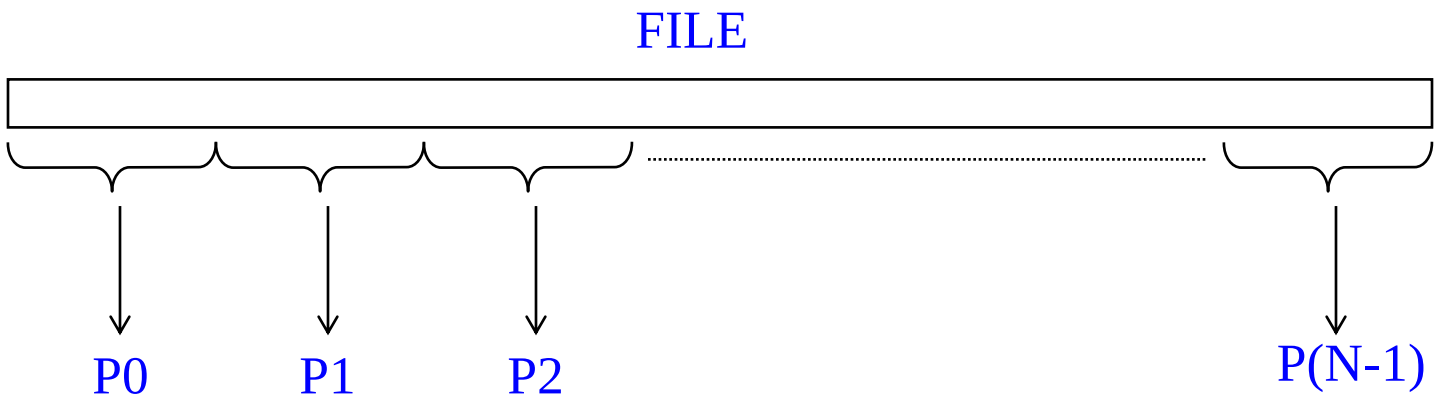
- Reading and writing in parallel is like receiving and sending messages

- 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

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 (e.g., `MPI_File_seek`/`MPI_File_read`)
  - Calculate byte offsets (e.g., `MPI_File_read_at`)
  - Access a shared file pointer (e.g., `MPI_File_seek_shared`, `MPI_File_read_shared`)



Source: Reference 3

MPI-I/O API 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 a`**`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, read data from files by either using `MPI_File_seek` & `MPI_File_read` Or `MPI_File_read_at`

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

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

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_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

- While 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, use **`MPI_File_set_view`** and **`MPI_File_write`** or **`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)

```
1.  #include<stdio.h>
2.  #include "mpi.h"
3.  int main(int argc, char **argv){
4.     int i, rank, size, offset, nints, N=16 ;
5.     MPI_File fhw;
6.     MPI_Status status;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    int buf[N];
11.    for ( i=0;i<N;i++){
12.          buf[i] = i ;
13.    }
14.  ...
```

Writing a File: writeFile1.c (2)

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

16. MPI_File_open(MPI_COMM_WORLD, "datafile",
    MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);

17. printf("\nRank: %d, Offset: %d\n", rank, offset);

18. MPI_File_write_at(fhw, offset, buf, (N/size),
    MPI_INT, &status);

19. MPI_File_close(&fhw);

20. MPI_Finalize();
21. return 0;
22.}
```

Compile & Run the Program on Compute Node

```
c401-204$ mpicc -o writeFile1 writeFile1.c
c401-204$ ibrun -np 4 ./writeFile1
```

TACC: Starting up job 1754636

TACC: Setting up parallel environment for MVAPICH2+mpispawn.

Rank: 0, Offset: 0

Rank: 1, Offset: 16

Rank: 3, Offset: 48

Rank: 2, Offset: 32

TACC: Shutdown complete. Exiting.

```
c401-204$ hexdump -v -e '7/4 "%10d "' -e '"\n"' datafile
```

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

File Views for Writing to a Shared File

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

- 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

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

- Data representation (datarep above) can be `native`, `internal`, or `exrternal32`
  ```

Writing a File: writeFile2.c (1)

```
1. #include<stdio.h>
2. #include "mpi.h"
3. int main(int argc, char **argv){
4.    int i, rank, size, offset, nints, N=16;
5.    MPI_File fhw;
6.    MPI_Status status;
7.    MPI_Init(&argc, &argv);
8.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.    MPI_Comm_size(MPI_COMM_WORLD, &size);
10.   int buf[N];
11.   for ( i=0;i<N;i++){
12.         buf[i] = i ;
13.   }
14.   offset = rank*(N/size)*sizeof(int);
15. ...
```

Writing a File: writeFile2.c (2)

```
16.MPI_File_open(MPI_COMM_WORLD, "datafile3",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL,
   &fhw);
17.  printf("\nRank: %d, Offset: %d\n", rank,
   offset);
18.  MPI_File_set_view(fhw, offset, MPI_INT,
   MPI_INT, "native", MPI_INFO_NULL);
19.  MPI_File_write(fhw, buf, (N/size), MPI_INT,
   &status);
20.  MPI_File_close(&fhw);
21.  MPI_Finalize();
22.  return 0;
23.}
```

Compile & Run the Program on Compute Node

```
c402-302$ mpicc -o writeFile2 writeFile2.c
c402-302$ ibrun -np 4 ./writeFile2
```

TACC: Starting up job 1755476

TACC: Setting up parallel environment for MVAPICH2+mpispawn.

Rank: 1, Offset: 16

Rank: 2, Offset: 32

Rank: 3, Offset: 48

Rank: 0, Offset: 0

TACC: Shutdown complete. Exiting.

```
c402-302$ 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 collective read and write calls force all processes in the communicator to read/write data simultaneously and to wait for each other

- 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

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

Collective I/O (2)

- The collective functions for reading and writing are:
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`

- Their signature is the same as for the non-collective versions

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
  - `MPI_File_open`
  - `MPI_File_set_info`
  - `MPI_File_set_view`

- Hints are optional and implementation-dependent
  - you may specify hints but the implementation can ignore them

- `MPI_File_get_info` used to get list of hints, examples of Hints: **`striping_unit, striping_factor`**

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

Lab-Sessions: Goals & Activities

- You will learn
  - How to do parallel I/O using MPI, HDF5, and T3PIO
  - How to compile and execute MPI code on Stampede

- What will you do
  - Compile and execute the code for the programs discussed in the lecture and exercises
  - Modify the code for the exercises to embed the required MPI routines, or calls to high-level libraries

Accessing Lab Files

Please see the hand-out for the username (login name) and password

- Log on to Stampede using **your_login_name**
- Uncompress the tar file, **trainingIO.tgz** , that is located in the **~train00** directory into your HOME directory.

```
ssh <your_login_name>@stampede.tacc.utexas.edu

tar -xvzf ~train00/trainingIO.tgz

cd trainingIO
```

Please Note

- The project number for this tutorial is 20130923CLUS2
- In the job submission script, provide the project number (replace “A-xxxxx” in “-A A-xxxxx”) mentioned above

Exercise 0

- **Objective**: practice compiling and running MPI code on Stampede
- 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 `ibrun` command
- Submit the job script to the SGE 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` in the subdirectory **`exercise`** within the directory **`trainingIO`**
  - 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 lecture slides for details on the MPI routines
- Compile the code and execute it via the job script (see Exercise 0 for the information related to compiling the code and the jobscript)

Outline

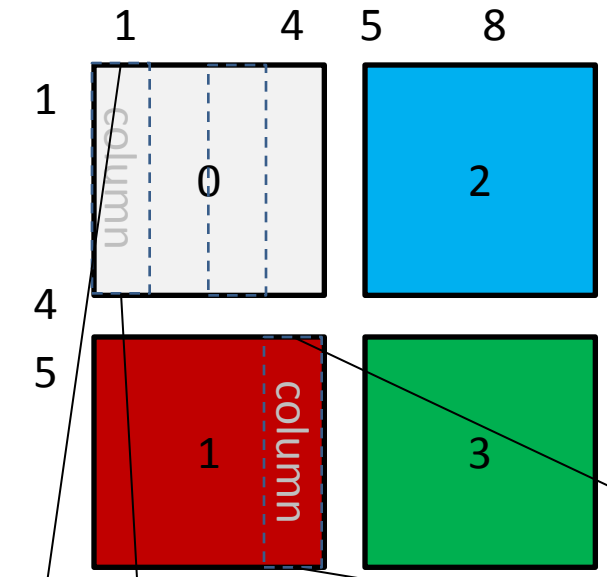
- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break – for 15 minutes
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

Outline

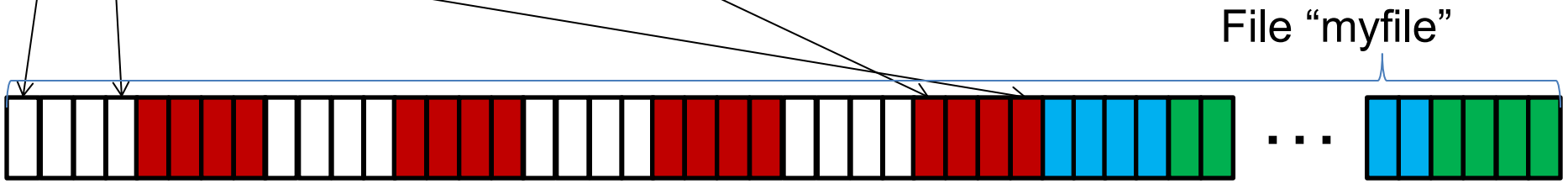
- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

MPI-IO

- Use MPI datatypes and a “view” on a file to perform a mapping of data to file



- Create MPI data types to correspond to the data layout of the global array
- Set the “view” of the file using the data type
- Write the data directly to the file in the proper sequential order
- Benefit from vendor optimizations to the collective write



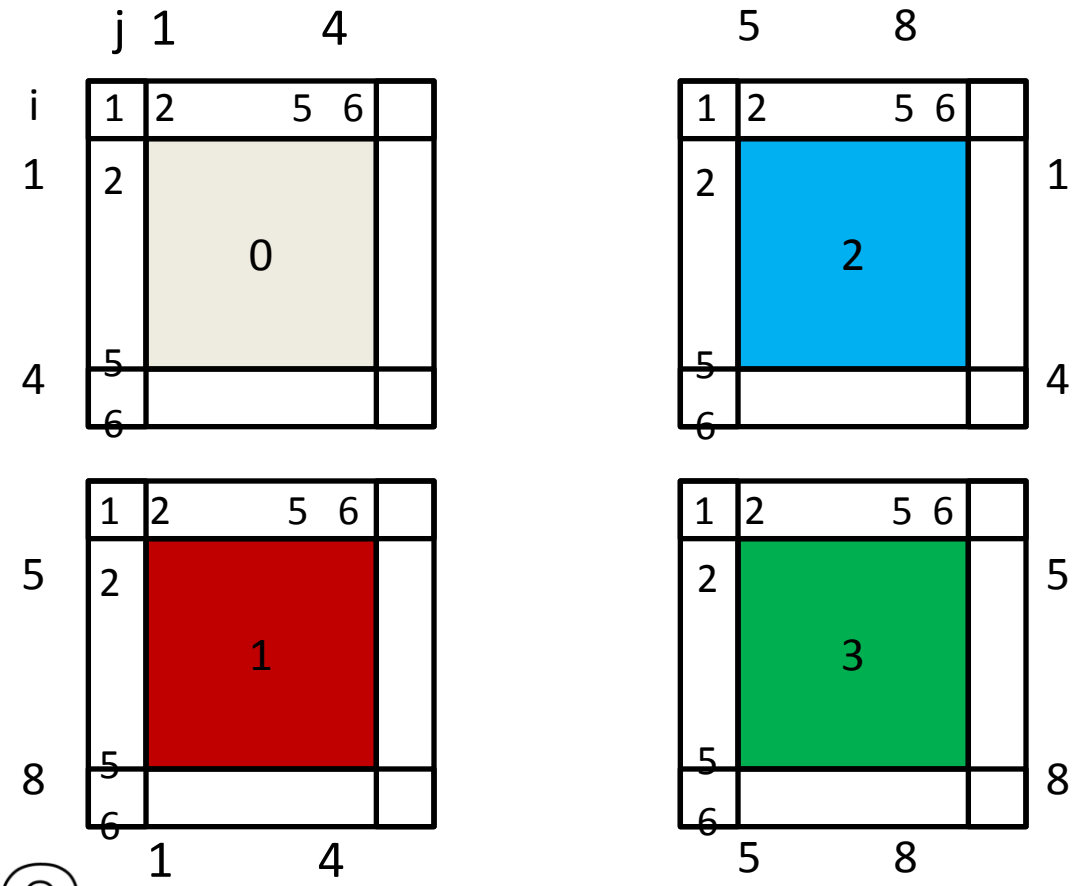
TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

MPI-IO

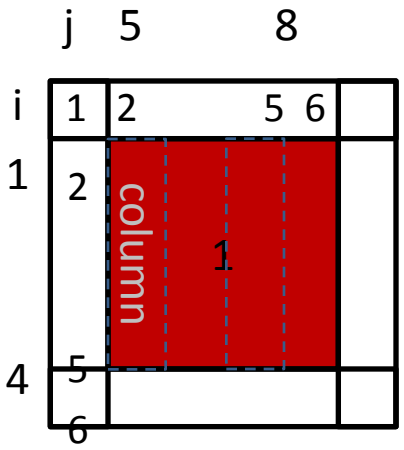
- Example: Write an 8x8 data array with a halo to a file in sequential order

```
allocate(data1(6,6) !4x4 local data array with halo
```



MPI-IO

- Create an MPI datatype to map the local data

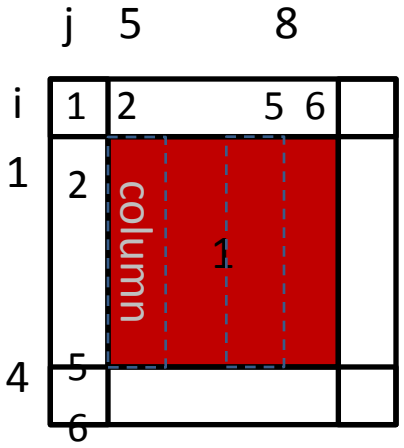



```
size(1) = 6; s_size(1)=4; start(1)=2 !local i index
size(2) = 6; s_size(2)=4; start(2)=2 !local j index
call mpi_type_create_subarray(2,size,s_size,start, &
   MPI_ORDER_FORTRAN,MPI_REAL8,core_data,mpierr)
call mpi_type_commit(core_data,mpierr)
```

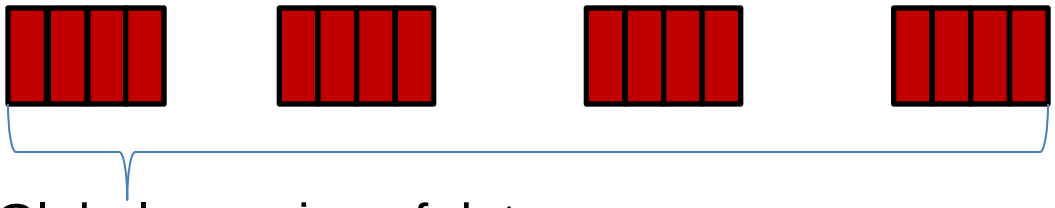
Local mapping of data

MPI-IO

- Create an MPI datatype to map the global data



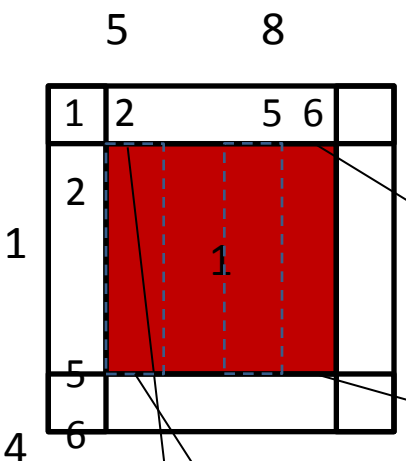

```
size(1) = 8; s_size(1)=4; start(1)=4 !global i index
size(2) = 8; s_size(2)=4; start(2)=0 !global j index
call mpi_type_create_subarray(2,size,s_size,start, &
   MPI_ORDER_FORTRAN,MPI_REAL8,global_data,mpierr)
call mpi_type_commit(global_data,mpierr)
```



MPI-IO

Open the file

```
call mpi_file_open(MPI_COMM_WORLD,"myfile",MPI_MODE_CREATE, &
     MPI_INFO_NULL,filehandle,mpierr)
```

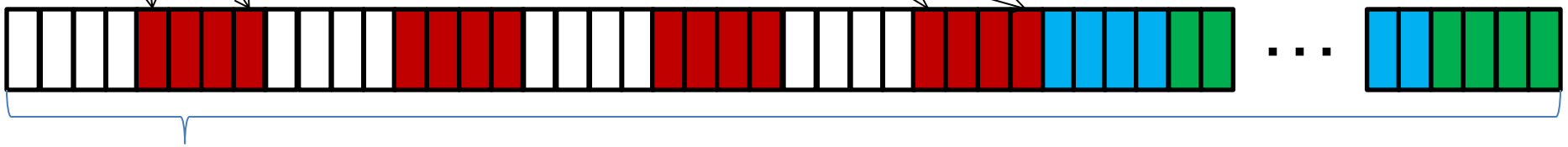


Set the view

```
file_offset=0
call mpi_file_set_view(filehandle,file_offset,&
  MPI_REAL8,global_data,"native",MPI_INFO_NULL,mpierr)
```

Write the file

```
call mpi_file_write_all(filehandle,data1,1, &
  core_data,mpistat,mpierr)
```



File "myfile" is arranged as if the data array was written sequentially

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

HDF5

- Hierarchical Data Format
  - Open source code base
  - Interface support for C, C++, Fortran, and Java
  - Supported by data analysis packages
    - Matlab, IDL, Mathematica, Octave, etc.
  - Machine independent data storage format
  - Supports user defined datatypes
  - Supports user defined metadata
  - Portability!

HDF5

- Can be viewed as a file system inside a file
- Unix style directory structure
- Mixture of groups, datasets, and attributes
- Any entity may be associated with descriptive attributes (metadata), e.g. physical units
- Currently used as the basis for NETCDF4

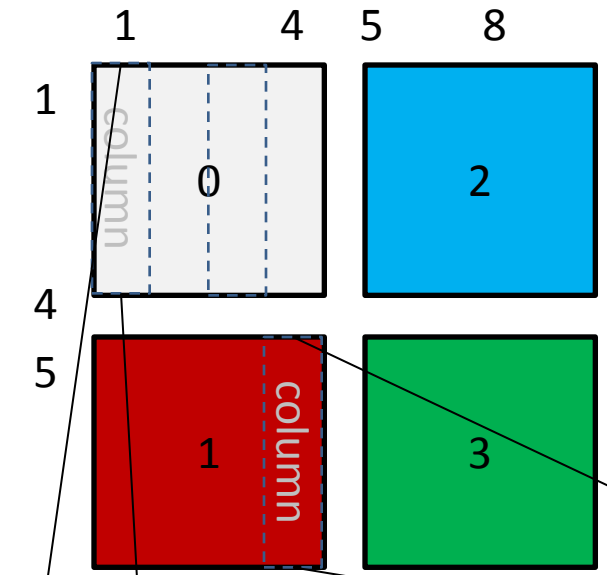
PHDF5

Parallel HDF5

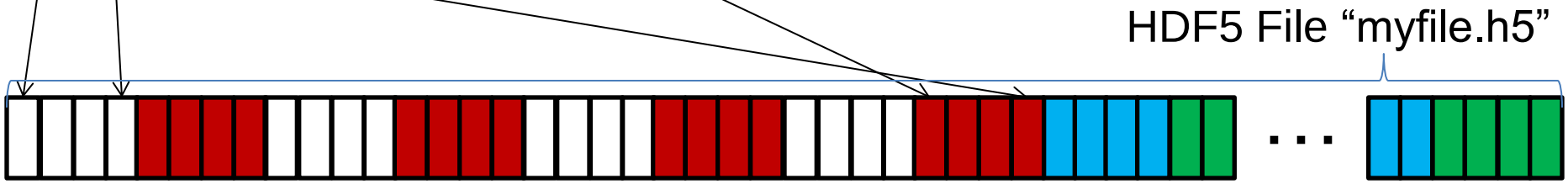
- Uses MPI I/O (Don’t reinvent the wheel)
- MPI I/O techniques apply to HDF5
- Use MPI_Info object to control # writers, # stripes(Lustre), stripe size(Lustre), etc.

PHDF5

- Use HDF5 datasets and hyperslabs to create task independent HDF5 files



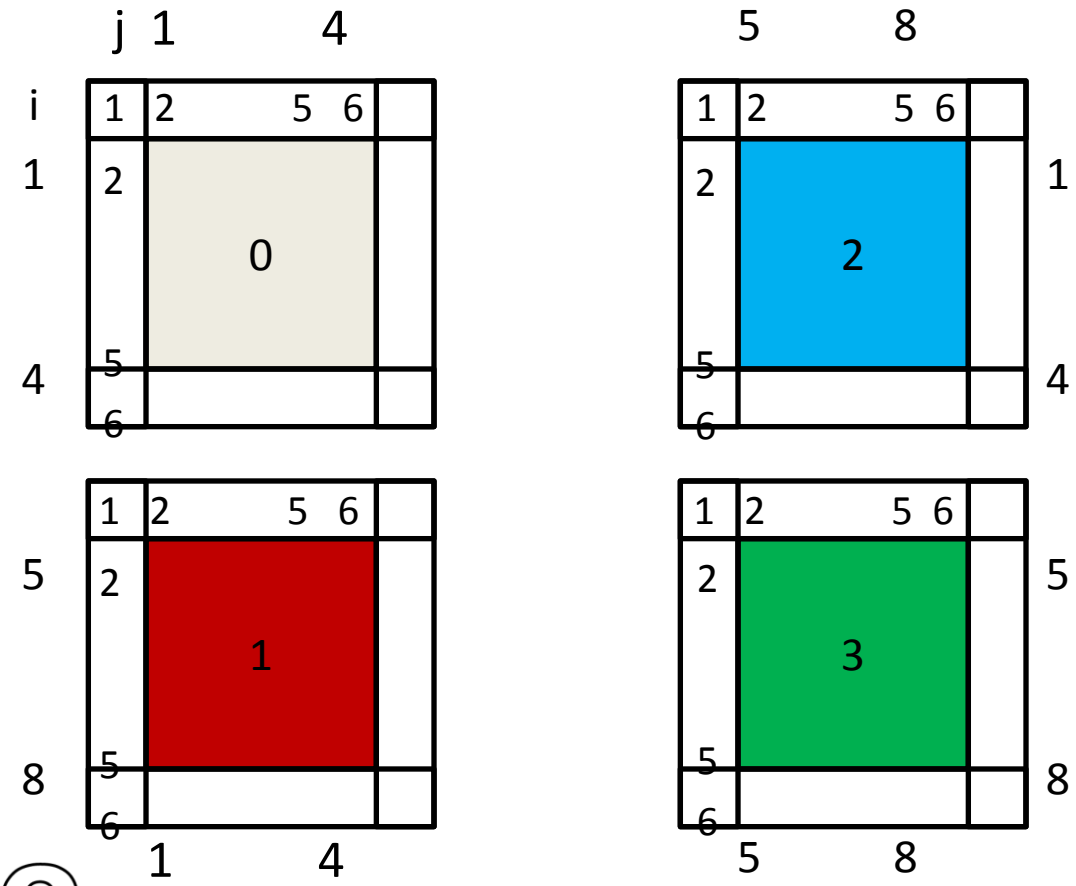
- Create local datasets for local arrays
- Create global datasets for file
- Use hyperslabs to control the mapping
- Use MPI I/O settings to tune performance



PHDF5

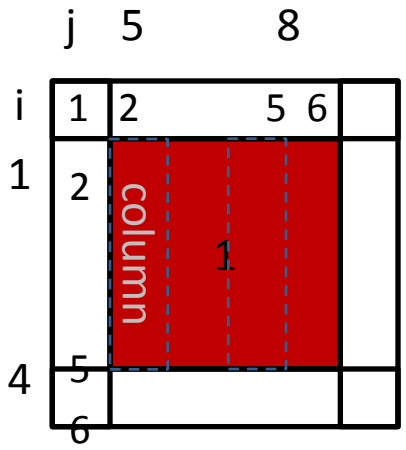
- Example: Write an 8x8 data array with a halo to an HDF5 file

```
allocate(data1(6,6) !4x4 local data array with halo
```

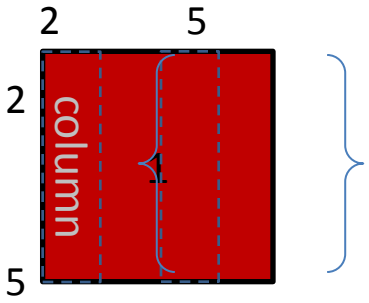


PHDF5

- Create local dataspace with hyperslab mapping



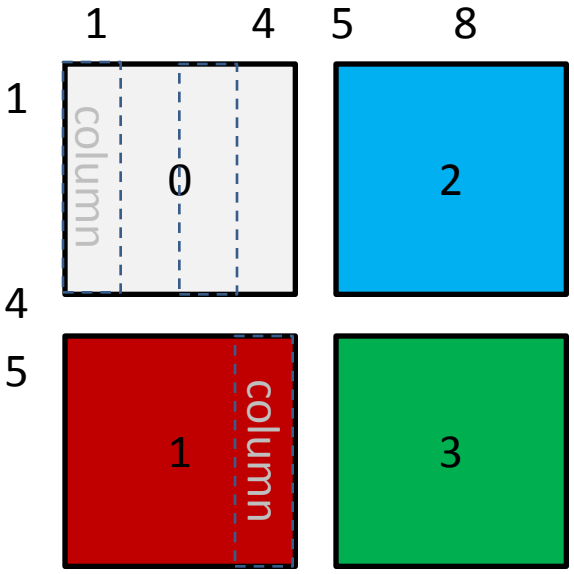

```
size(1)=6;   size(2)=6   !Local array size
halo(1)=1;   halo(2)=1   !Local offset
s_size(1)=4; s_size(2)=4 !Local subset
stride(1)=1; stride(2)=1 !Use defaults
block(1)=1;  block(1)=1  !Use defaults
call h5screate_simple_f(2,size,memspace,err)
call h5sselect_hyperslab_f(memspace,&
 H5SELECT_SET_F,halo,s_size,err,stride,block)
```



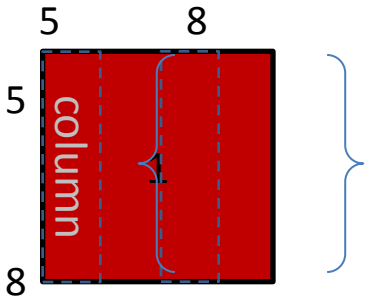
memspace dataspace

PHDF5

- Create global dataspace with hyperslab mapping




```
size(1)=8;   size(2)=8   !Global array size
start(1)=4;  start(2)=0  !Global offset
s_size(1)=4; s_size(2)=4 !Global subset
stride(1)=1; stride(2)=1 !Use defaults
block(1)=1;  block(1)=1  !Use defaults
call h5screate_simple_f(2,size,filespace,err)
call h5sselect_hyperslab_f(filespace,&
 H5SELECT_SET_F,start,s_size,err,stride,block)
```

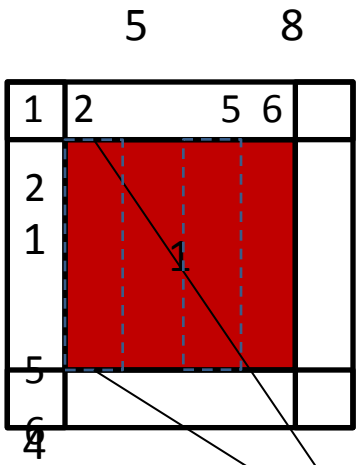


filespace dataspace

PHDF5

Set up file access

```
call h5open_f(err); call h5create_f(H5P_FILE_ACCESS_F,pl_id,err)
call h5pset_fapl_mpio_f(pl_id,comm,MPI_INFO_NULL,err)
```

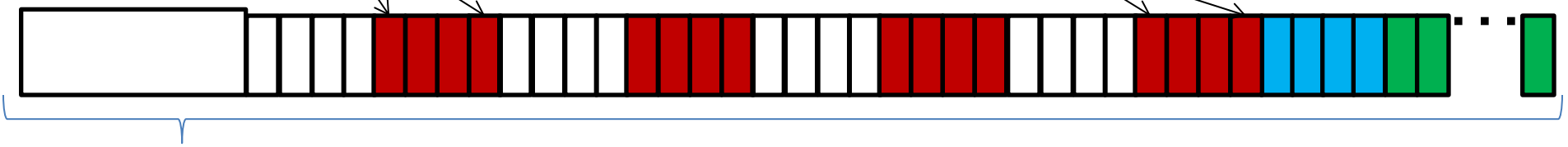


Create the file

```
call h5create_f("myfile.h5",H5F_ACC_TRUNC_F,file_id, &
                err,access_prp=pl_id)
call h5pclose_f(pl_id,err)
```

Write the file

```
call h5pcreate_f(H5P_DATASET_XFER_F,pl_id,err)
call h5pset_dxpl_mpio_f(pl_id,H5FD_MPIO_COLLECTIVE_F,err)
call h5dwrite_f(dset_id,H5T_NATIVE_DOUBLE,data1,sizes, &
    err,file_space_id=filespace,mem_space_id=memspace, &
    xfer_prp=pl_id)
```



File "myfile.h5" contains dataset with the full data1 array

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

T3PIO Library

- TACC's Terric Tool for Parallel I/O
- Lustre parallel I/O performance depends on
  - Number of Writers
  - Number of Stripes
  - Stripe Size
- This library will set these parameters for you.

How to use T3PIO library (F90)

```
subroutine hdf5 writer(....)
use hdf5
use t3pio
integer info                ! MPI Info object
integer comm                ! MPI Communicator
integer(hid t) :: plist id ! Property list identifier
...
comm = MPI COMM WORLD
! Initialize info object.
call MPI Info create(info,ierr)
! use library to fill info with nwriters, stripe
call t3pio set info(comm, info, "./", ierr, &
            GLOBAL SIZE=globalSize)
call H5open f(ierr)
call H5Pcreate f(H5P FILE ACCESS F,plist id,ierr)
call H5Pset fapl mpio f(plist id, comm, info, ierr)
call H5Fcreate f(fileName, H5F ACC TRUNC F, file id, ierr, &
            access prp = plist id)
```

How to use T3PIO library (C)

```
#include "t3pio.h"
#include "hdf5.h"
void hdf5 writer(....)
{
   MPI Info info = MPI INFO NULL;
   hid t plist id;
   ...
   MPI Info create(&info);
   ierr = t3pio set info(comm, info, "./",
   T3PIO GLOBAL SIZE, globalSize);
   plist id = H5Pcreate(H5P FILE ACCESS);
   ierr = H5Pset fapl mpio(plist id, comm, info);
   file id = H5Fcreate(fileName, H5F ACC TRUNC, H5P DEFAULT,
   plist id);
   ...
}
```

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

Strategies for I/O on Stampede

- Access data contiguously in memory and on disk
- Avoid “Too often, too many"
- Be gentle with the MDS (Meta Data Server)
- Write large files to $SCRATCH
- Write one global file instead of multiple files
- **Use parallel I/O**
  - **MPI I/O**
  - **Parallel HDF5**, parallel netCDF
- Set file attributes (stripe count, stripe size, #writers)
  - **T3PIO**

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Lab Session 1
- Break
- MPI I/O Example – Distributing Arrays
- Introduction to HDF5
- Introduction to T3PIO
- I/O Strategies
- Lab-Session2

Step 1

- Login to Stampede with the training account or your personal account
- Change your directory to $SCRATCH

```
cds
```

- Unpack the lab files

```
tar xvf ~train00/lab2_parallel_io.tar
```

- Change your directory to the mpiio lab

```
Cd lab2_parallel_io/mpiio
```

MPI I/O Lab

- Login to Stampede with the training account or your personal account
- Change your directory to $SCRATCH

```
cds
```

- Unpack the lab files

```
tar xvf ~train00/lab2_parallel_io.tar
```

- Change your directory to the mpiio lab

```
Cd lab2_parallel_io/mpiio
```

MPI I/O Lab

- The programs write_global_array.f90 and write_global_array.c are simple examples using MPI I/O to write a global array as if it were written sequentially.
- The makefile will produce two executables:

  f_write_global -- from write_global_array.f90
  c_write_global -- from write_global_array.c

- By default, each of these is configured to run on 4 tasks and create a file of 256x256 double precision numbers, 2KB in size.

MPI I/O Lab

To run the executables, follow these steps.

1. Build the executable:

```
make
```

2. Get an interactive session:

```
idev -n 16 -N 2
```

(This will start a session that supports 16 MPI tasks spread across 2 nodes.)

Exercise 1

Run the simple case using 4 tasks.

- Use the ibrun command from within an idev session to run the job:

  `ibrun -np 4 ./f_write_global` (Fortran)

  `ibrun -np 4 ./c_write_global` (C)

- Examine the binary output file:

  `od -t f8 data_0016x0016-procs_002x002.out`

You should see double precision numbers increasing from 1 to 256.

Exercise 2

Modify the code to run with 16 tasks instead of 4.

- Edit the C or Fortran code and change the "map" array to decompose the processor grid from a 2 tasks x 2 tasks to 4 tasks x 4 tasks:

  map = (4,4)

- Recompile the code and run using the ibrun command:

  `ibrun -np 16 ./f_write_global` (Fortran)

  `ibrun -np 16 ./c_write_global` (C)

Exercise 2(Cont.)

- Diff the new output file with the old one.

```
diff data_0016x0016-procs_004x004.out \
    data_0016x0016-procs_002x002.out
```

- Examine the binary output file:

```
od -t f8 data_0016x0016-procs_002x002.out
```

Are they the same?

Exercise 3

Modify the code to write out a larger array.

- Edit the C or Fortran code and change the nx and ny variables to create a 2048x2048 data array.

```
nx=2048; ny=2048
```

- Recompile the code and run using 16 tasks:

```
ibrun -np 16 ./f_write_global
```
(Fortran)

```
ibrun -np 16 ./c_write_global
```
(C)

Exercise 3(Cont.)

- Take note of the speed of the write. It should be much better than with the small 16x16 array.

- Larger fewer writes work best on parallel file systems.

Exercise 4

Repeat Exercise 3 using 4 tasks rather than 16.

- Edit the code again to reset the # of tasks to 4.

```
map = (2,2)
```

- Recompile the code and run using 16 tasks:

```
ibrun -np 4 ./f_write_global
```
(Fortran)

```
ibrun -np 4 ./c_write_global
```
(C)

- Did the write performance change? Why?
  Could the number of stripes of the output file affect the performance?

HDF5 Lab

- Login to Stampede with the training account or your personal account
- Change your directory to $SCRATCH

```
cds
```

- Unpack the lab files

```
tar xvf ~train00/lab2_parallel_io.tar
```

- Change your directory to the hdf5 lab

```
cd lab2_parallel_io/hdf5
```

HDF5 Lab

- The programs hdf_write_global_array.f90 and hdf_write_global_array.c are simple examples using HDF5 to write a distributed global array to a HDF5 file.

- The makefile will produce two executables:

  f_write_global -- from write_global_array.f90
  c_write_global -- from write_global_array.c

- By default, each of these is configured to run on 4 tasks and create an HDf5 file that contains a 256x256 double precision array.

HDF5 Lab

To run the executables, follow these steps.

Build the executable:

You must build from the login node. The required libz.a library is not available on regular compute nodes. So, if you're still on a compute node from the previous exercise, please logout.

1. Load the parallel hdf5 module before you build:

   ```
   module load phdf5
   ```

   Then build:

   ```
   make
   ```

2. Get an interactive session:

   ```
   idev
   ```

Exercise 1

Run the simple case using 4 tasks.

- Use the ibrun command from within an idev session to run the job:

  `ibrun -np 4 ./f_write_global` (Fortran)

  `ibrun -np 4 ./c_write_global` (C)

- Examine the hdf5 output file:

  `h5dump data_0016x0016-procs_002x002.h5`

You should see double precision numbers increasing from 1 to 256.

Note that this file is bigger than the MPI I/O file because it also contains metadata.

Exercise 2

Modify the code to run with 16 tasks instead of 4.

- Exit from the previous idev session
- Edit the C or Fortran code and change the "map" array to decompose the processor grid from a 2 tasks x 2 tasks to 4 tasks x 4 tasks:

  map = (4,4)
- Recompile the code and get a new idev session:

```
idev
```

Exercise 2(cont.)

- Run using the ibrun command:

  `ibrun -np 16 ./f_write_global` (Fortran)

  `ibrun -np 16 ./c_write_global` (C)

- Examine the hdf5 output:

  `h5dump data_0016x0016-procs_004x004.h5`

Although the actual file is different than the previous file, the data stored in them is the same.

Summary

- I/O can impact performance at large scale
  - Take advantage of the parallel file system
- Consider using MPI-IO, Parallel HDF5, or Parallel netCDF libraries
- Analyze your code to determine if you may benefit from parallel I/O
- Set stripe count and stripe size for optimal use if on a Lustre file system

Reference

1. http://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips
2. http://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf
3. http://www.arcos.inf.uc3m.es/~ii_ac2_en/dokuwiki/lib/exe/fetch.php?id=exercises&cache=cache&media=thakur-mpi-io.ppt

Fortran Code for Part 1

Example Code Snippet from Reference 3 (Reading a File)

```
  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
  call MPI_FILE_OPEN(MPI_COMM_WORLD,'/pfs/datafile',
MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
  nints = FILESIZE / (nprocs*INTSIZE)
  offset = rank * nints * INTSIZE
  call MPI_FILE_READ_AT(fh, offset, buf, nints,
MPI_INTEGER, status, ierr)
  call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
  print *, 'process ', rank, 'read ', count,'integers'
  call MPI_FILE_CLOSE(fh, ierr)
```

Example Code from Reference 3 (1) (Writing to a File)

```
PROGRAM main
use mpi
integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo
```

Code continued on next slide …

Example Code from Reference 3 (2) (Writing to a File)

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
 MPI_MODE_WRONLY + MPI_MODE_CREATE, &
 MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize, ierr)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                       MPI_INTEGER, 'native', &
                       MPI_INFO_NULL, ierr)

call MPI_FILE_WRITE(thefile, buf, BUFSIZE,
MPI_INTEGER,MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)
END PROGRAM main
```