# Using Parallel I/O


Go to Menu

# Acknowledgements


- This document is based on the material originally presented by
  - Rajeev Thakur. *Mathematics and Computer Science Division Argonne National Laboratory*
    - MPI-2 Tutorial
  - Lonnie Crosby and Mark Fahey. *National Institute for Computational Sciences (NICS)*
    - 2009 Cray XT5 Quad-core Workshop

# Outline

# Outline: Introduction

- Factors which affect I/O
- Typical application I/O Patterns
- I/O Parallelism
- Types of Parallelism
- Limits of I/O
- I/O for Computational Science
  - High Level Libraries
  - I/O Middleware
  - Parallel File System

# Factors Which Affect I/O

- I/O is simply data migration.
  - Memory ⟷ Disk
    - Cache (L1, L2, L3)
    - RAM
    - Disk
- Size of write/read operations
  - Bandwidth vs. Latency
- Data continuity and locality on disk
  - Bandwidth vs. Latency
- Number of processes performing I/O
- Characteristics of the file system
  - Distributed or Shared

# Typical Application I/O Patterns

## Serial I/O

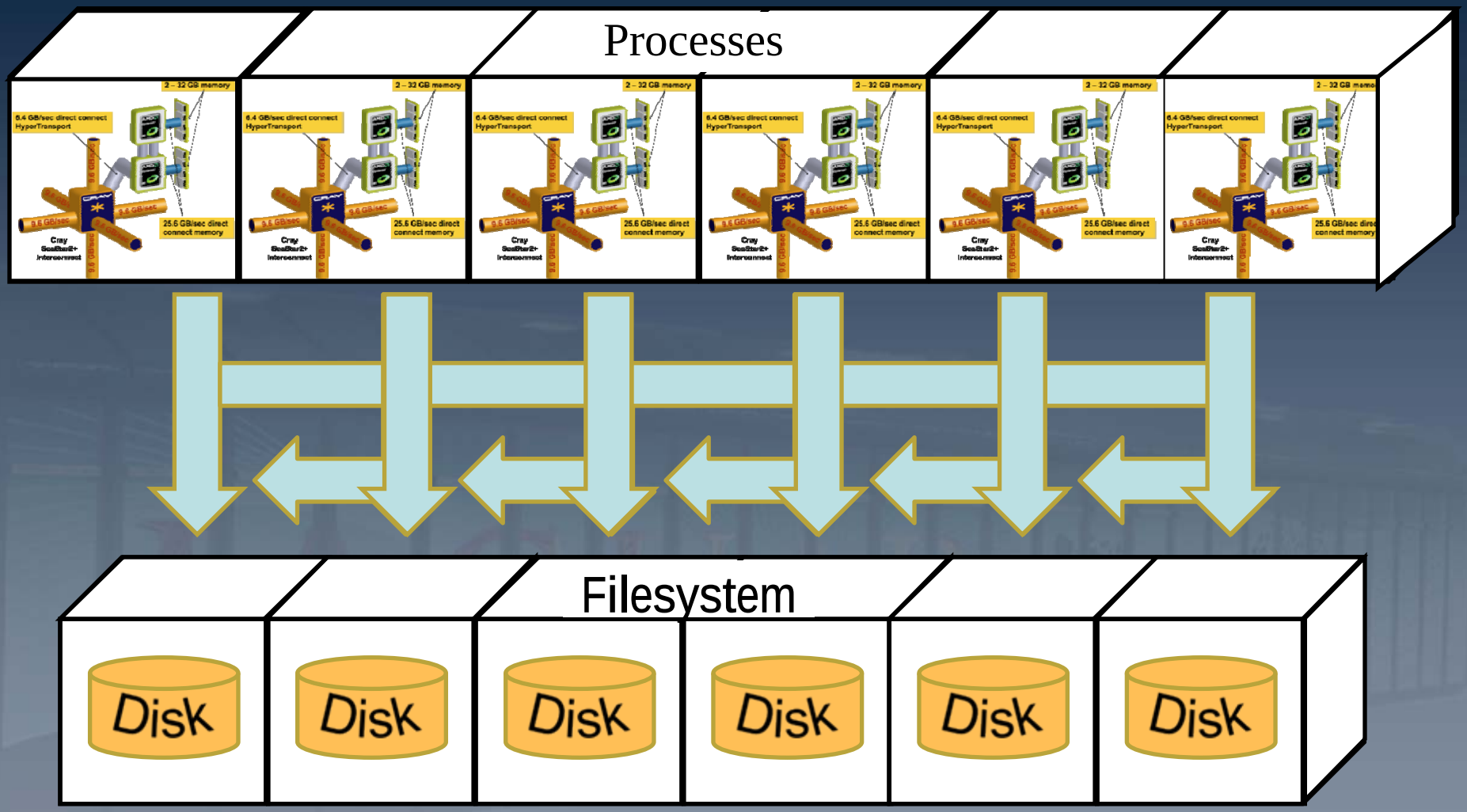
- Spokesperson
  - One process performs I/O.

## Parallel I/O

- File per Process
  - Each process performs I/O to a single file.
- Single Shared File
  - Each process collectively performs I/O to a single shared file.
- Multiple Shared Files
  - Groups of processes perform I/O to a single shared file.

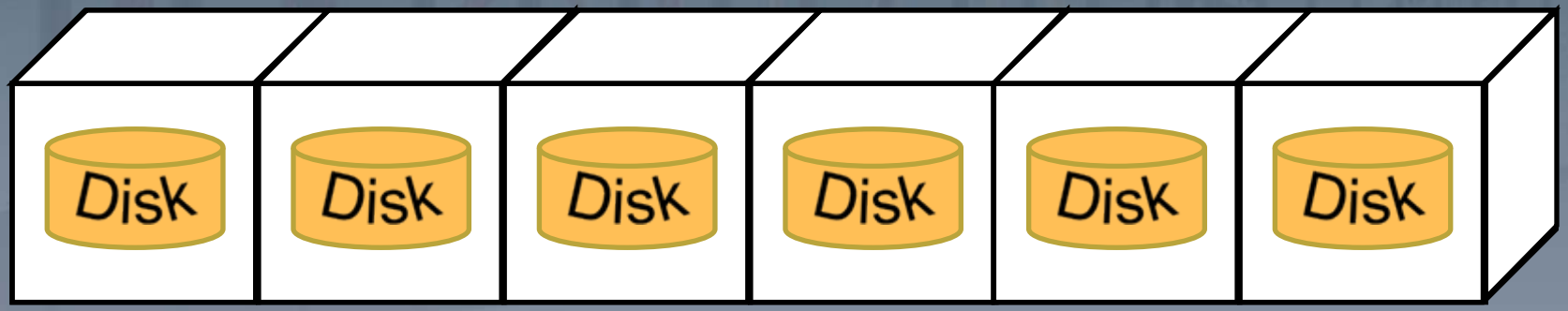
Go to Menu

# I/O Parallelism



# Types of Parallelism

- Process level parallelism
  - MPI
  - IO Libraries (HDF5, MPI-IO, p-netCDF)
- File System parallelism
  - Distributed File System
  - Shared Parallel File System (GPFS, Lustre)

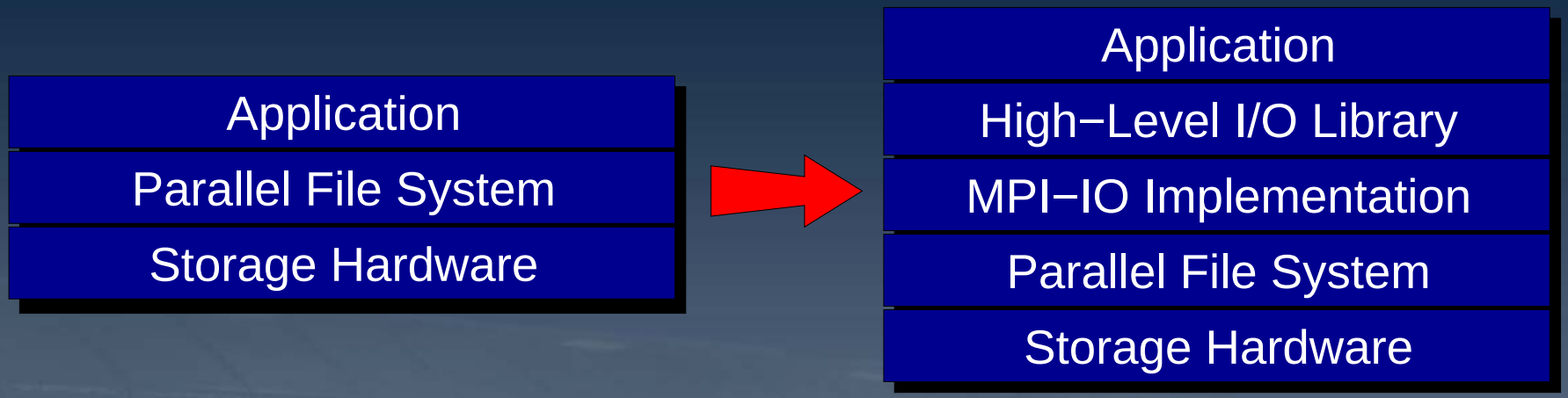


# Limits of I/O

- Serial I/O
  - is limited by the single process which performs I/O.
- Parallel Process I/O
  - is limited by the number of disks which are concurrently utilized.
  - Contention for file system resources.
- Distributed File System
  - Files are localized on a single disk.
- Parallel File System
  - Files are localized on a single disk.
  - Files are striped across multiple disks.

Go to Menu

# I/O for Computational Science

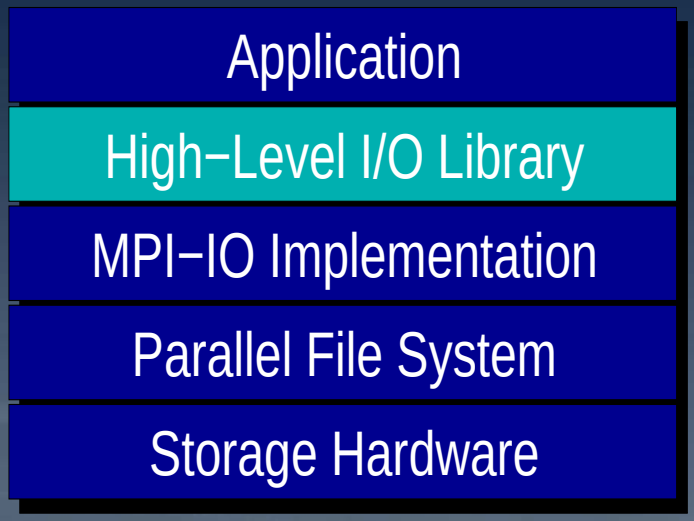



- Break up support into multiple layers:
  - High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
  - Middleware layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
  - Parallel file system maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)

Go to Menu

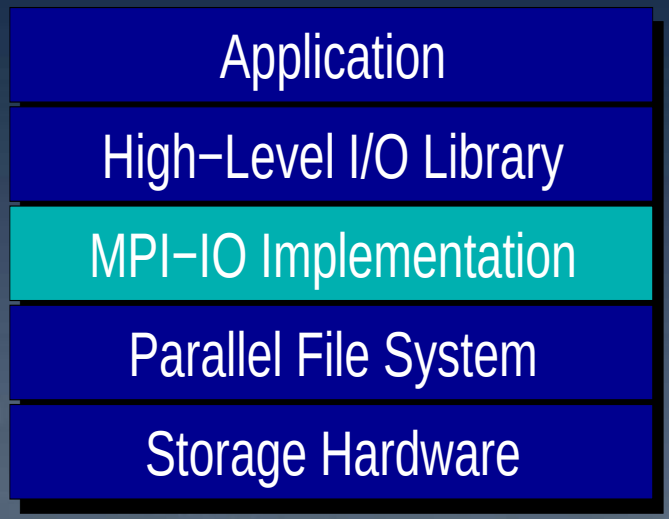
# High Level Libraries

- Provide an appropriate abstraction for domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Self-describing, structured file format
- Map to middleware interface
  - Encourage collective I/O
- Provide optimizations that middleware cannot
- Examples: HDF5, Parallel netCDF, ADI05



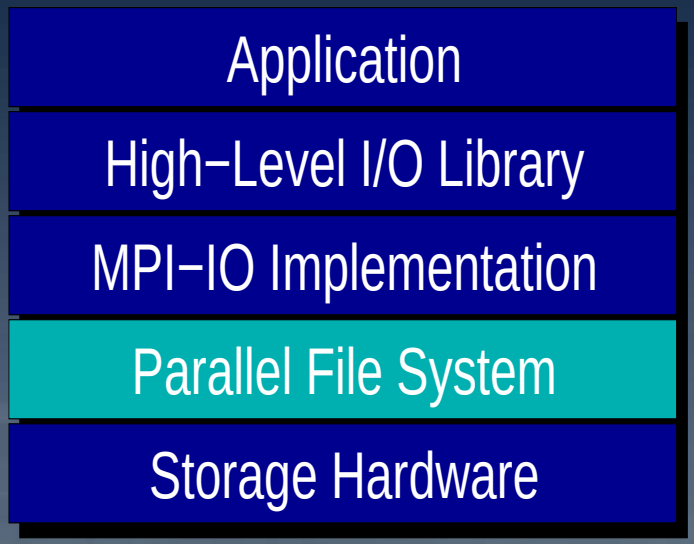
# I/O Middleware

- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Match the underlying programming model (e.g. MPI)
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs



# Parallel File System

- Manage storage hardware
  - Present single view
  - Focus on concurrent, independent access
  - Knowledge of collective I/O usually very limited
- Publish an interface that middleware can use effectively
  - Rich I/O language
  - Relaxed but sufficient semantics



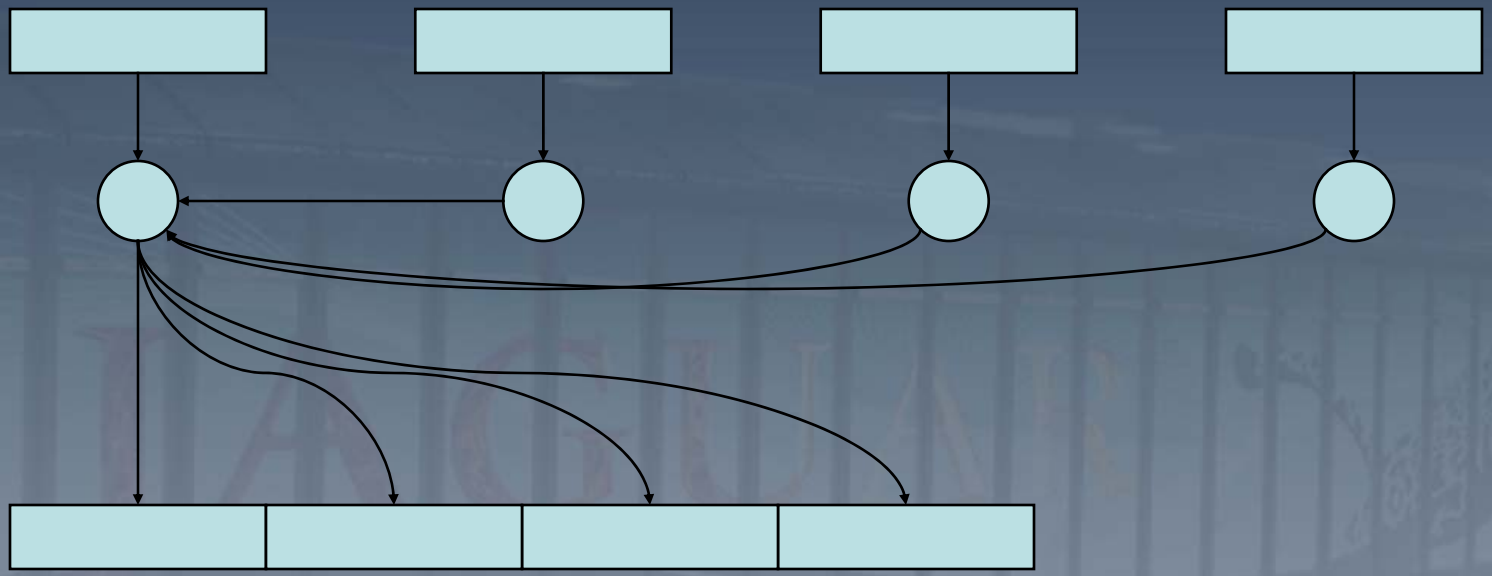
# Outline: MPI-IO

# Introduction

- Goals of this section
  - introduce the important features of MPI-IO in the form of example programs, following the outline of the Parallel I/O chapter in *Using MPI-2*
  - focus on how to achieve high performance
  - learn how to use MPI-IO
  - be able to immediately use MPI-IO in your applications
  - get much higher I/O performance than what you have been getting so far using other techniques

# Common Ways of Doing I/O in Parallel Programs

- Sequential I/O:
  - All processes send data to rank 0, and 0 writes it to the file

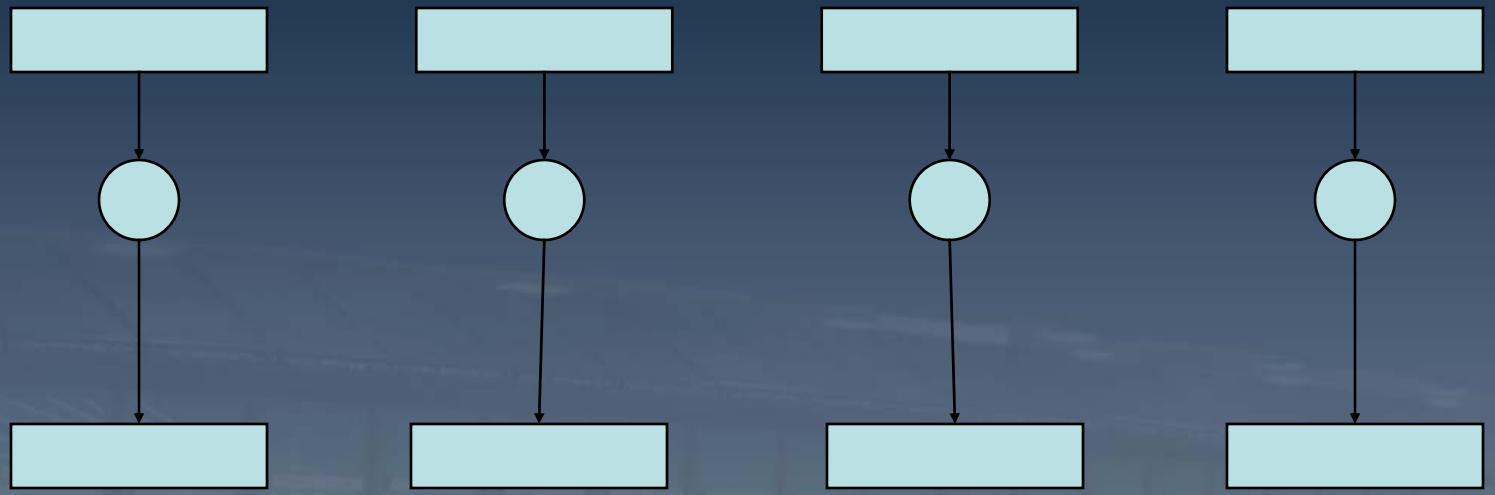


# Pros and Cons of Sequential I/O

- Pros:
  - parallel machine may not support parallel file system (e.g., no common file)
  - some I/O libraries (e.g. HDF-4, NetCDF) not parallel
  - resulting single file is handy for local file system utilities: `ftp`, `mv`
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance (single node bottleneck)

# Another Way

- Each process writes to a separate file

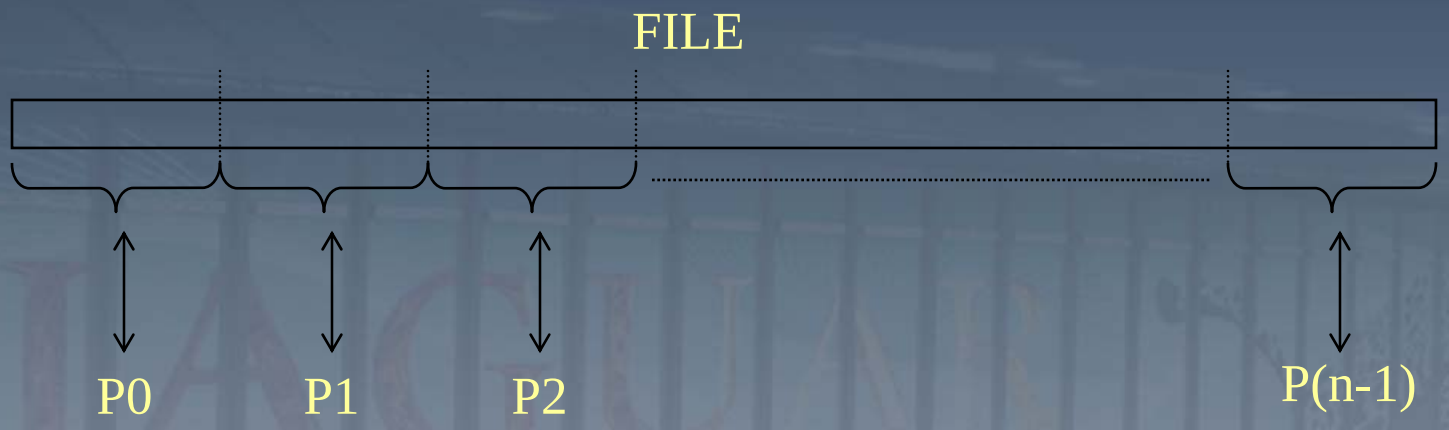


  - Pros:
    - parallelism, high performance
  - Cons:
    - potentially lots of files to manage – bottleneck with large process counts
    - difficult to read back data from different number of processes

Go to Menu

# What is Parallel I/O?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



# Why Parallel I/O?

- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

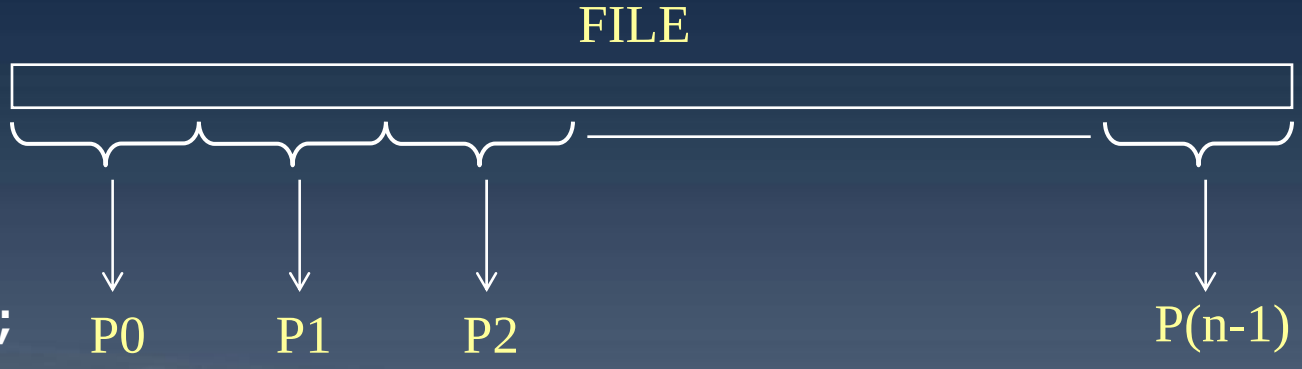
# Why is MPI a Good Setting for Parallel I/O?

- Writing is like sending a message and reading is like receiving
- Any parallel I/O system will need a mechanism to
  - define collective operations (*MPI communicators*)
  - define noncontiguous data layout in memory and file (*MPI datatypes*)
  - Test completion of nonblocking operations (*MPI request objects*)
- Lots of MPI-like machinery

# Using MPI for Simple IO

# Using MPI for Simple IO: Individual File Pointers

Each process needs to read a chunk of data from a common file




```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              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);
MPI_File_close(&fh);
```

Go to Menu

# Using MPI for Simple IO: Explicit Offsets

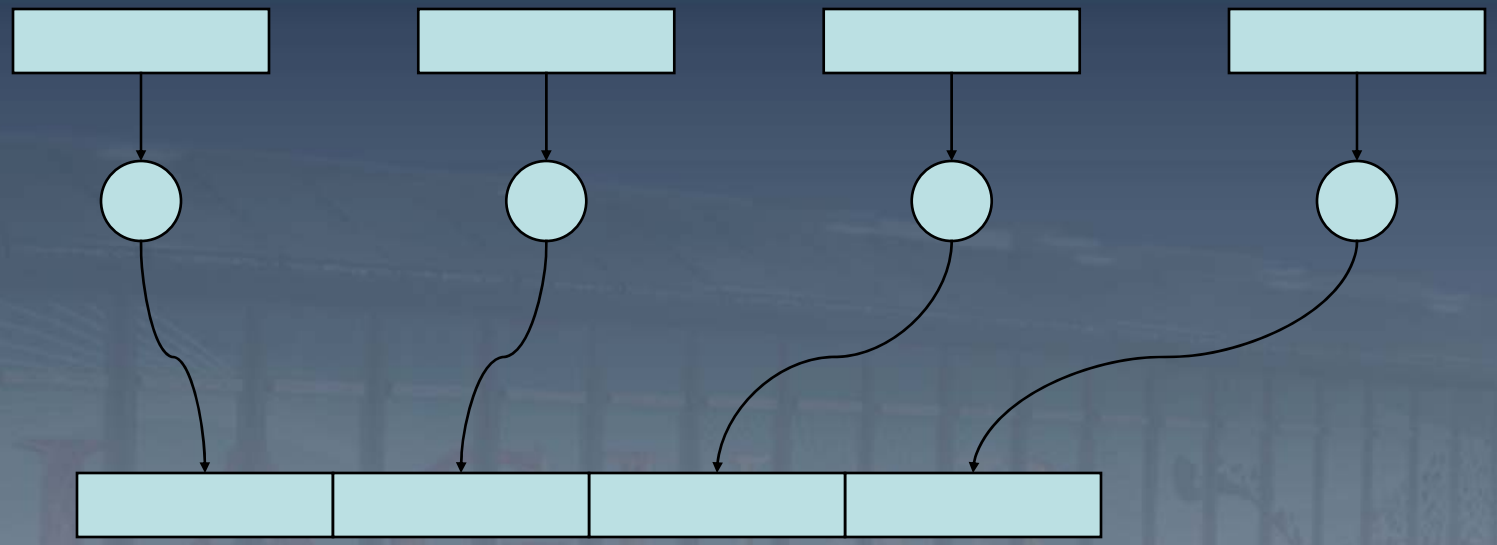
```
  include 'mpif.h'

  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
C in F77, see implementation notes (might be integer*8)

  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)
```

# Using MPI for Simple IO: Writing to a File

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

# Using MPI for Simple IO: Using File Views

- Processes write to shared file



- **`MPI_File_set_view`** assigns regions of the file to separate processes

Go to Menu

# Using MPI for Simple IO: File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

# Using MPI for Simple IO: File View Example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE *
                  sizeof(int), MPI_INT,
                  MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

# Using MPI for Simple IO: MPI_File_set_view

- Describes that part of the file accessed by a single MPI process.
- Arguments to **`MPI_File_set_view`**:
  - **`MPI_File file`**
  - **`MPI_Offset disp`**
  - **`MPI_Datatype etype`**
  - **`MPI_Datatype filetype`**
  - **`char *datarep`**
  - **`MPI_Info info`**

# Using MPI for Simple IO: Fortran Example

```
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


* in F77, see implementation notes (might be integer*8)
```

Go to Menu

# Using MPI for Simple IO: Fortran Example (continued)

```
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)
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
```

# Using MPI for Simple IO: C++ Example

```
// example of parallel MPI read from single file
#include <iostream.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int bufsize, *buf, count;
    char filename[128];
    MPI::Status status;

    MPI::Init();
    int myrank = MPI::COMM_WORLD.Get_rank();
    int numprocs = MPI::COMM_WORLD.Get_size();
    MPI::File thefile = MPI::File::Open(MPI::COMM_WORLD,
                                        "testfile",
                                        MPI::MODE_RDONLY,
                                        MPI::INFO_NULL);
```

Go to Menu

# Using MPI for Simple IO: C++ Example (continued)

```
    MPI::Offset filesize = thefile.Get_size();
    filesize    = filesize / sizeof(int);
    bufsize     = filesize / numprocs + 1;
    buf = new int[bufsize];
    thefile.Set_view(myrank * bufsize * sizeof(int),
                     MPI_INT, MPI_INT, "native",
                     MPI::INFO_NULL);
    thefile.Read(buf, bufsize, MPI_INT, &status);
    count = status.Get_count(MPI_INT);
    cout << "process " << myrank << " read " << count
         << " ints" << endl;
    thefile.Close();
    delete [] buf;
    MPI::Finalize();
    return 0;
}
```

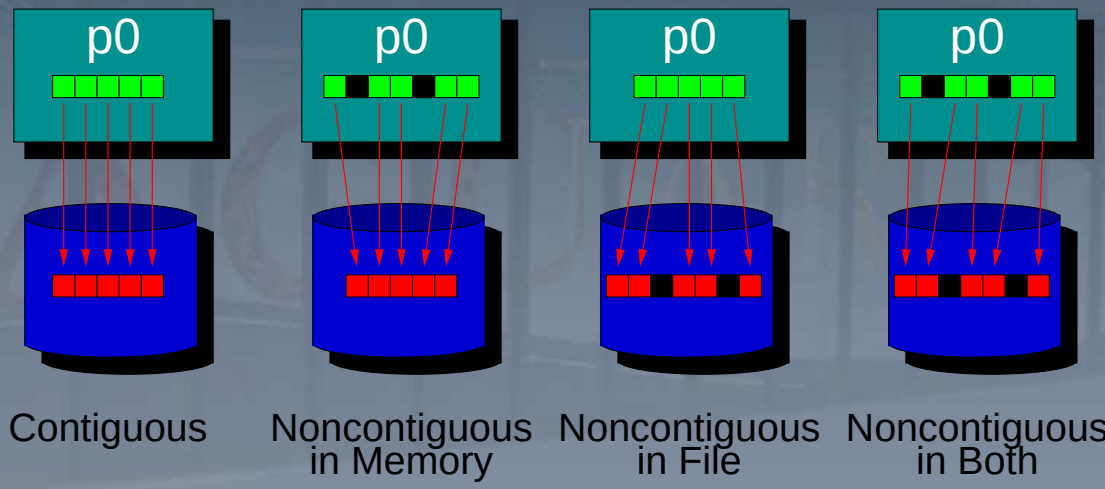
# Using MPI for Simple IO: Other Ways to Write to a Shared File

- **`MPI_File_seek`** } like Unix seek
- **`MPI_File_read_at`** } combine seek and I/O for thread safety
- **`MPI_File_write_at`**
- **`MPI_File_read_shared`** } use shared file pointer
- **`MPI_File_write_shared`**

- Collective operations

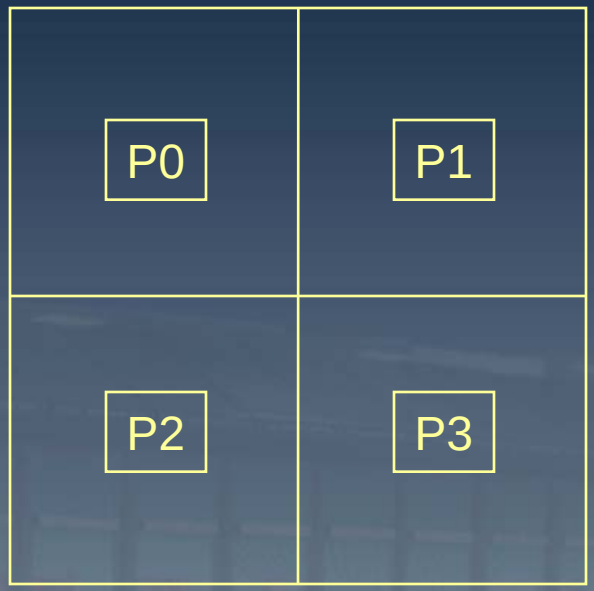
# Noncontiguous I/O

- Contiguous I/O moves data from a single block in memory into a single region of storage
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O



# Example: Distributed Array Access

2D array distributed among four processes

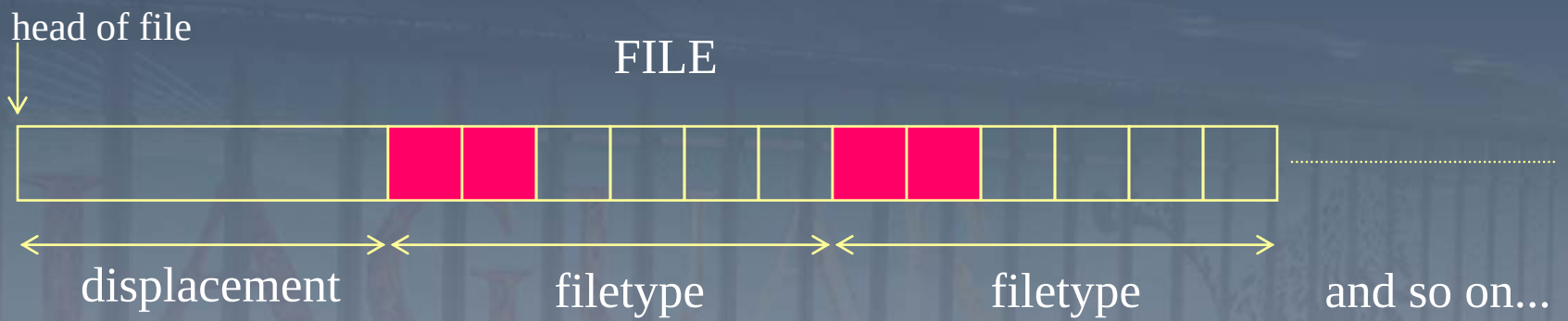



File containing the global array in row-major order

# A Simple Noncontiguous File View Example

etype = MPI_INT

filetype = two MPI_INTs followed by
a gap of four MPI_INTs

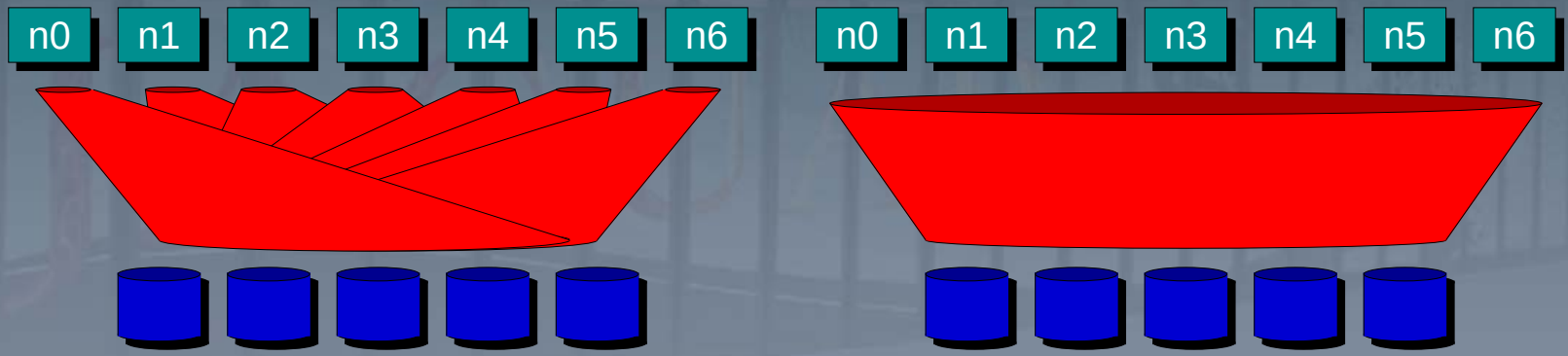


# File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

# Collective I/O

- Many applications have phases of computation and I/O
- During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions must be called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole
- Independent I/O is not organized in this way
- No apparent order or structure to accesses




Independent I/O

Collective I/O

Go to Menu

# Collective I/O (continued)

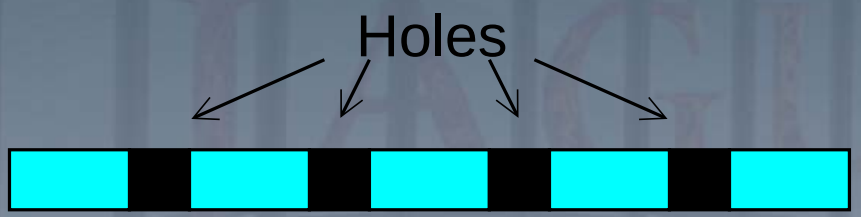
- **`MPI_File_read_all`**, **`MPI_File_read_at_all`**, etc
- **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

# Under the Covers of MPI-IO

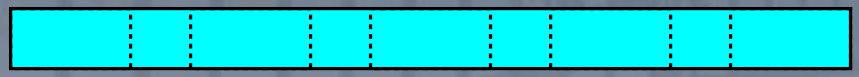
- MPI-IO implementation is given a lot of information in this case:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to obtain this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

# Data Sieving

- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Generally very effective, but not as good as having a PFS that supports noncontiguous access



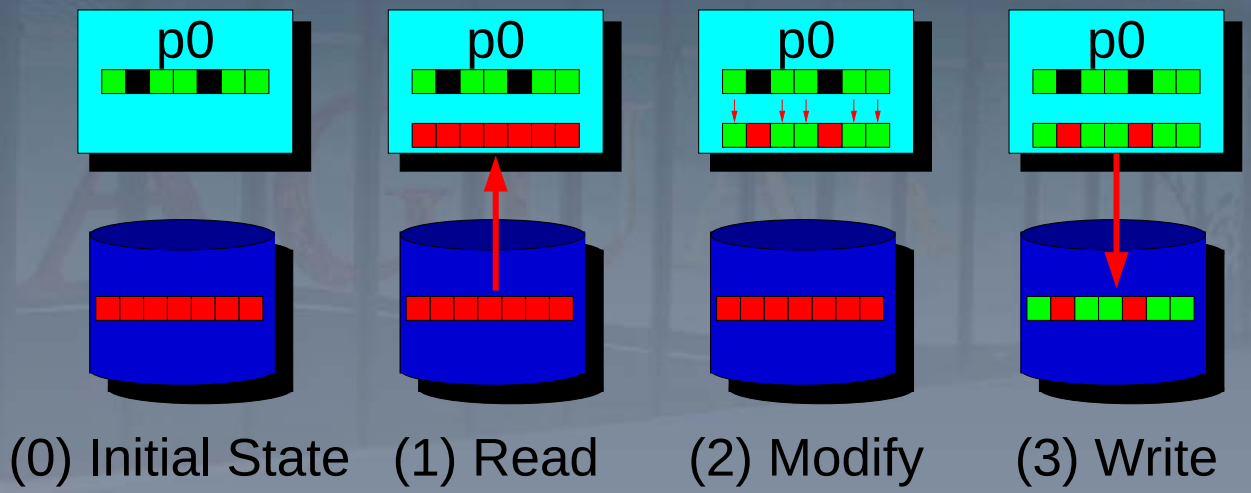

Region desired by application



Region accessed with data sieving

# Data Sieving Writes

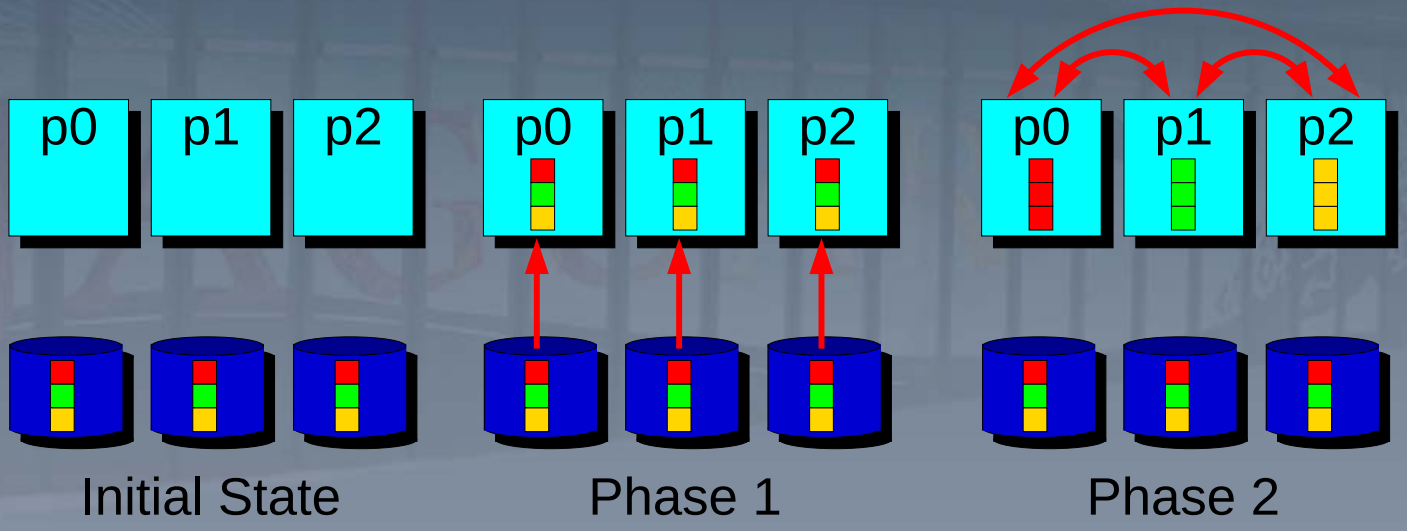
- Using data sieving for writes is more complicated
  - Must read the entire region first
  - Then make our changes
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
  - PFS supporting noncontiguous writes is preferred




Go to Menu

# Two-Phase Collective I/O

- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second ``phase'' moves data to final destinations



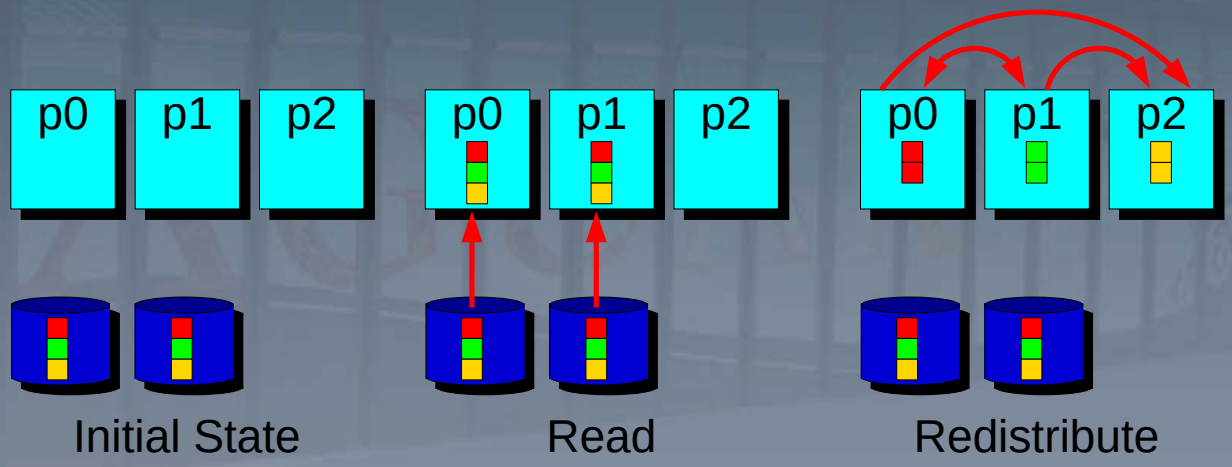

Go to Menu

# Two-Phase Writes

- Similarly to data sieving, we need to perform a read/modify/write for two-phase writes if *combined* data is noncontiguous
- Overhead is substantially lower than independent access to the same regions because there is little or no false sharing
- Note that two-phase is usually applied to file regions, not to actual blocks

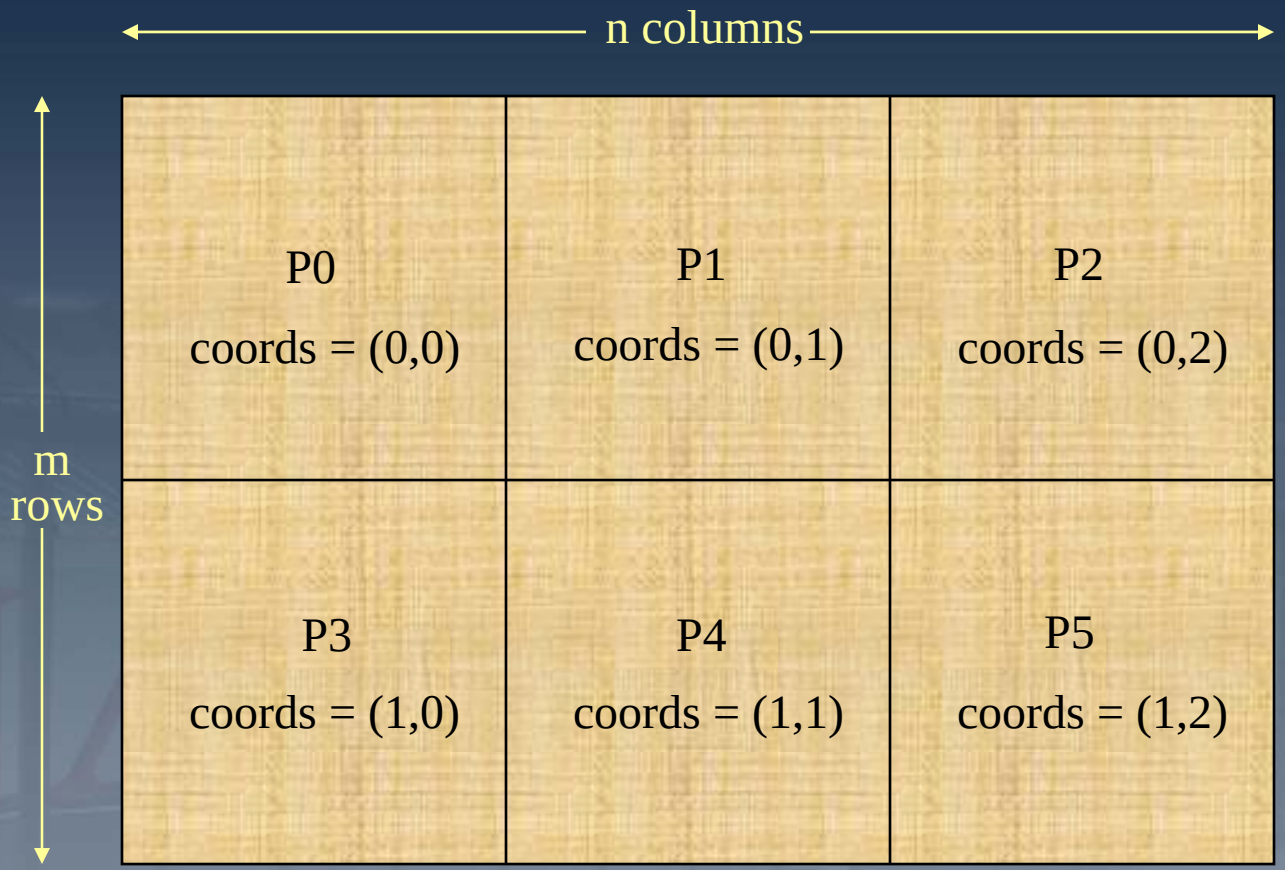
# Aggregation

- Aggregation refers to the more general application of this concept of moving data through intermediate nodes
  - Different #s of nodes performing I/O
  - Could also be applied to independent I/O
- Can also be used for remote I/O, where aggregator processes are on an entirely different system




Go to Menu

# Accessing Arrays Stored in Files




nproc(1) = 2, nproc(2) = 3

# Using the "Distributed Array" (Darray) Datatype

```
int gsizes[2], distribs[2], dargs[2], psizes[2];

gsizes[0] = m;    /* no. of rows in global array */
gsizes[1] = n;    /* no. of columns in global array*/

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; /* no. of processes in vertical dimension
                  of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension
                  of process grid */
```

# Darray (continued)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
                psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
                MPI_FLOAT, &status);

MPI_File_close(&fh);
```

# A Word of Warning about Darray

- The darray datatype assumes a very specific definition of data distribution -- the exact definition as in HPF
- For example, if the array size is not divisible by the number of processes, darray calculates the block size using a *ceiling* division (20 / 6 = 4 )
- darray assumes a row-major ordering of processes in the logical grid, as assumed by cartesian process topologies in MPI-1
- If your application uses a different definition for data distribution or logical grid ordering, you cannot use darray. Use subarray instead.

# Using the Subarray Datatype

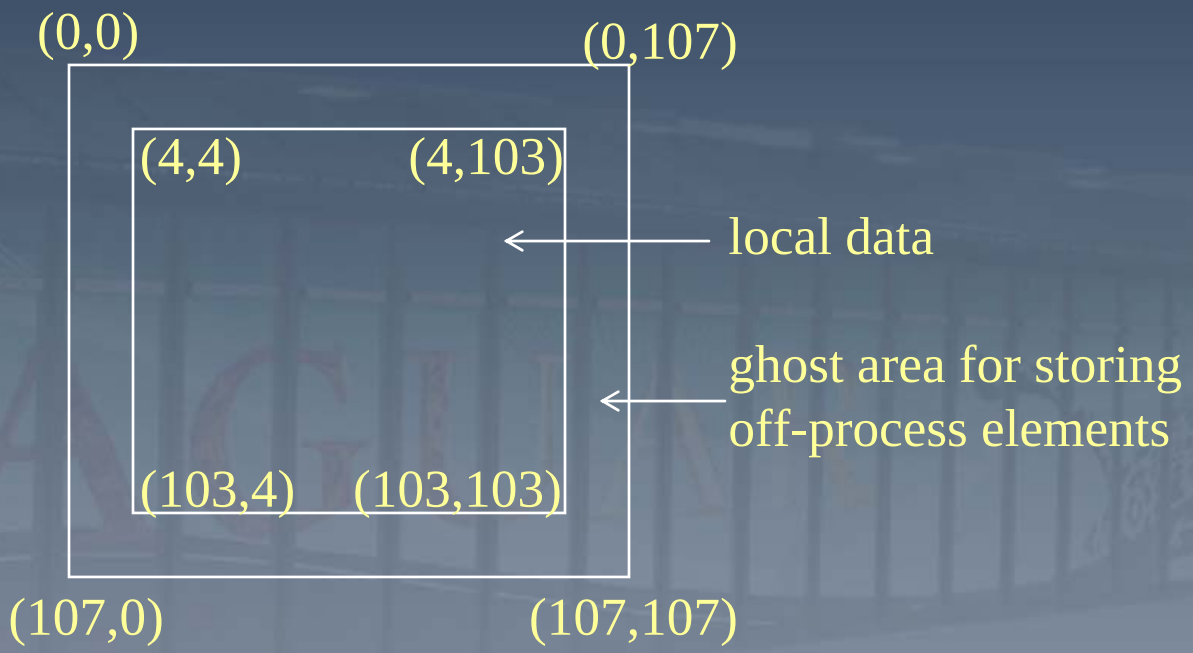
```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

# Subarray Datatype (continued)

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
              MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all(fh, local_array, local_array_size,
               MPI_FLOAT, &status);
```

# Local Array with Ghost Area in Memory

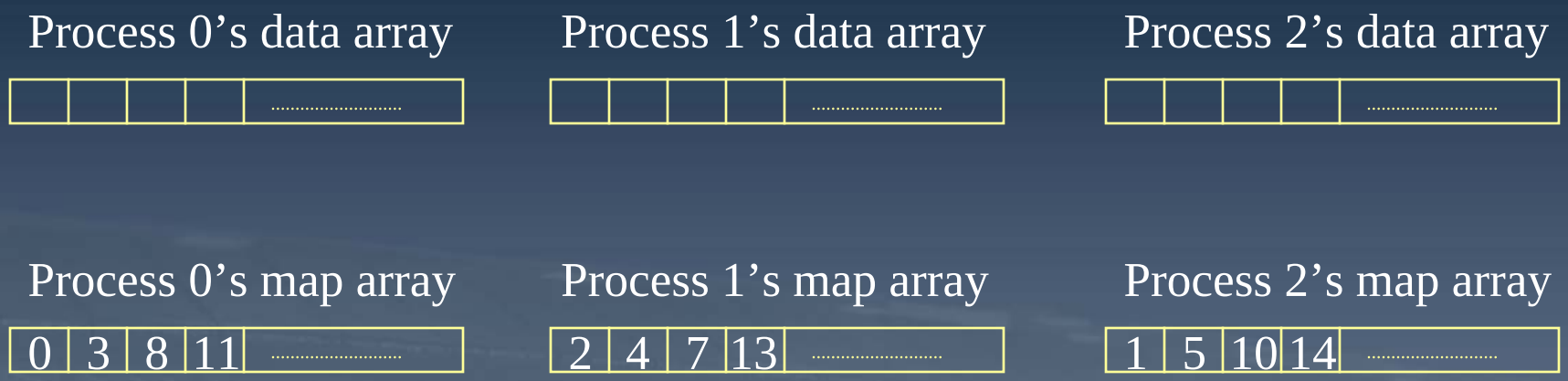
- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to `MPI_File_write_all`



# Local Array with Ghost Area

```
memsizes[0] = lsizes[0] + 8;
   /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8;
   /* no. of columns in allocated array */
start_indices[0] = start_indices[1] = 4;
   /* indices of the first element of the local array
      in the allocated array */

MPI_Type_create_subarray(2, memsizes, lsizes,
         start_indices, MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);

/* create filetype and set file view exactly as in the
   subarray example */

MPI_File_write_all(fh, local_array, 1, memtype, &status);
```

# Accessing Irregularly Distributed Arrays




The map array describes the location of each element of the data array in the common file

# Accessing Irregularly Distributed Arrays (continued)

```
integer (kind=MPI_OFFSET_KIND) disp

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
                   MPI_MODE_CREATE + MPI_MODE_RDWR, &
                   MPI_INFO_NULL, fh, ierr)

call MPI_TYPE_CREATE_INDEXED_BLOCK(bufsize, 1, map, &
                    MPI_DOUBLE_PRECISION, filetype, ierr)
call MPI_TYPE_COMMIT(filetype, ierr)
disp = 0
call MPI_FILE_SET_VIEW(fh, disp, MPI_DOUBLE_PRECISION, &
                  filetype, 'native', MPI_INFO_NULL, ierr)

call MPI_FILE_WRITE_ALL(fh, buf, bufsize, &
                        MPI_DOUBLE_PRECISION, status, ierr)

call MPI_FILE_CLOSE(fh, ierr)
```

Go to Menu

# Nonblocking I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype,
                   &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```

# Split Collective I/O

- A restricted form of nonblocking collective I/O
- Only one active nonblocking collective operation allowed at a time on a file handle
- Therefore, no request object necessary

```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```

# Shared File Pointers

```
#include "mpi.h"
// C++ example
int main(int argc, char *argv[])
{
    int buf[1000];
    MPI::File fh;

    MPI::Init();

    MPI::File fh = MPI::File::Open(MPI::COMM_WORLD,
        "/pfs/datafile", MPI::MODE_RDONLY, MPI::INFO_NULL);
    fh.Write_shared(buf, 1000, MPI_INT);
    fh.Close();

    MPI::Finalize();
    return 0;
}
```

Go to Menu

# Outline: Parallel File System - Lustre



Go to Menu

# Introduction

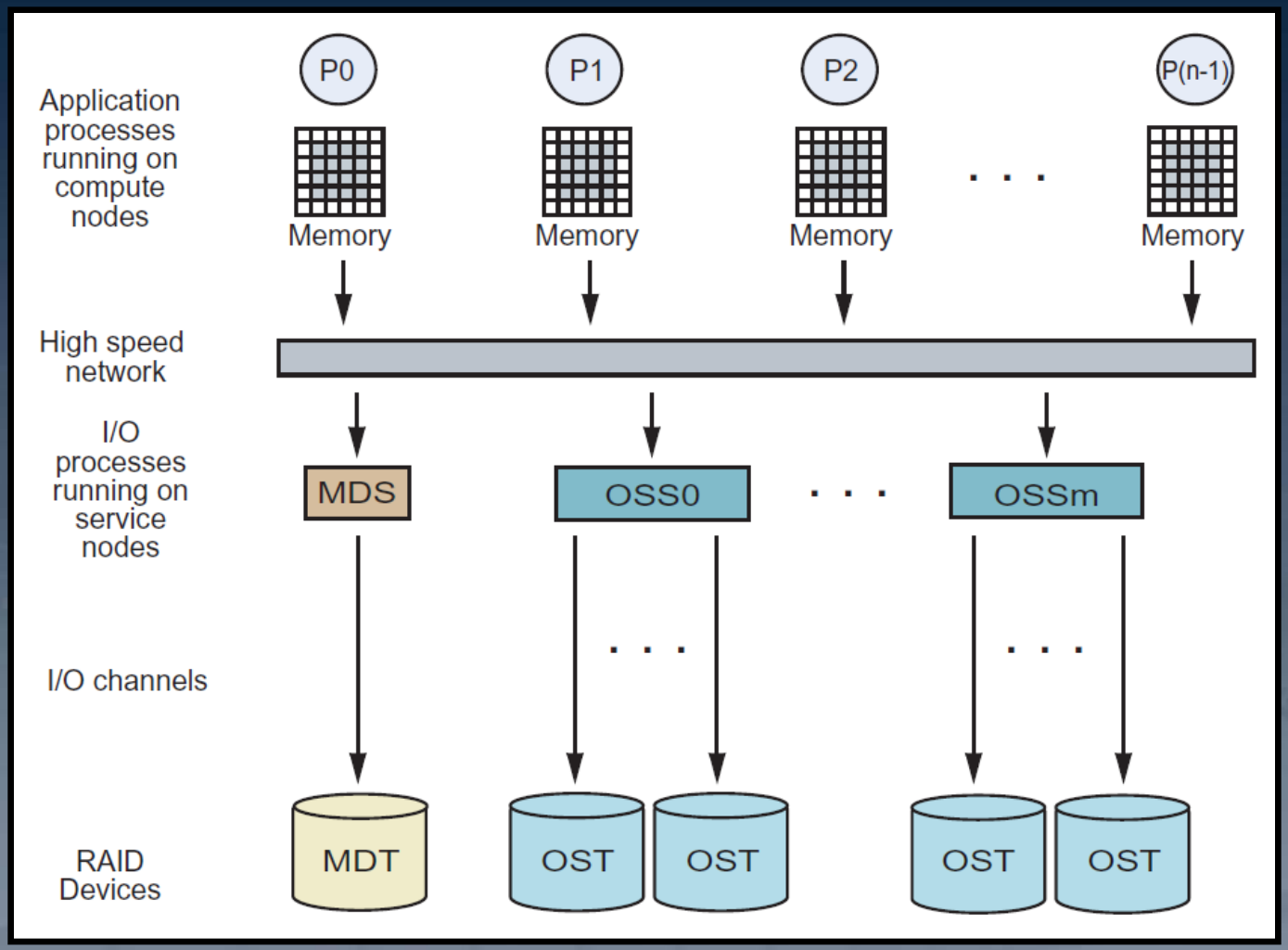
- The parallel file system available on jaguarpf is called Lustre (/tmp/work/$USER), which offers a set of user-level commands to tune and optimize file access operations.
- For many applications a technique called *file striping* will increase I/O performance. File striping will primarily improve performance for codes doing serial I/O from a single node or parallel I/O from multiple nodes writing to a single shared file, such as with MPI-IO, parallel HDF5, or parallel NetCDF.
- The Lustre file system is made up of an underlying set of I/O servers and disks called Object Storage Servers (OSSs) and Object Storage Targets (OSTs) respectively. A file is said to be striped when read and write operations access multiple OST's concurrently. File striping is a way to increase I/O performance since writing or reading from multiple OST's simultaneously increases the available I/O bandwidth.
- Details about the Lustre file system and its configurations are available at http://wiki.lustre.org/.

Go to Menu

# Luster Concepts

- Two types of servers
  - Metadata server (MDS)
    - Holds the directory tree
    - Stores metadata about each file (except for size)
    - Once file is opened, I/O to file does not involve the MDS
  - Object storage server (OSS)
    - Manages OSTs (think single disk/LUN)
    - OSTs hold stripes of the file contents
      - Think RAID0
    - Maintains the locking for the file contents it is responsible for

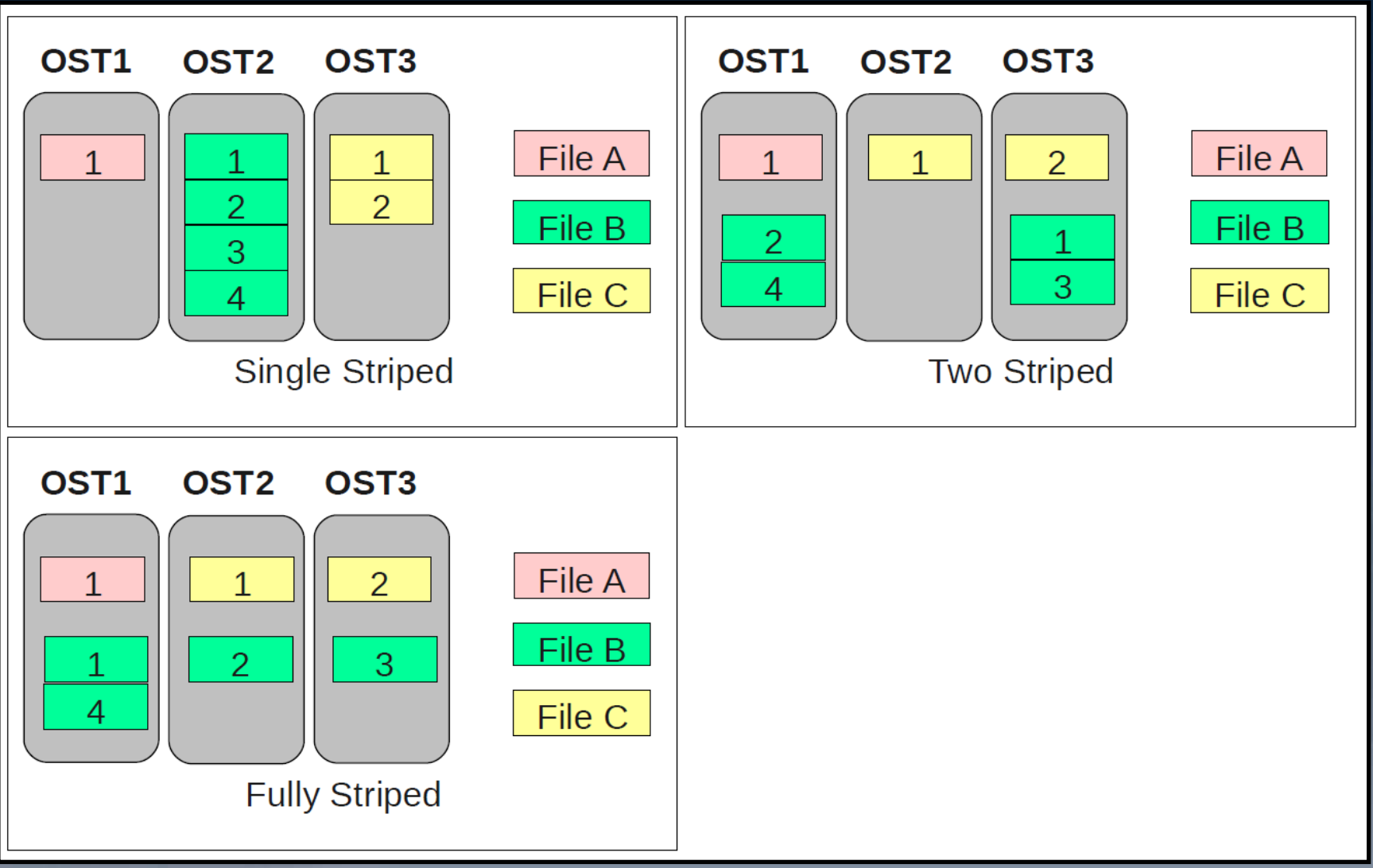
# A Bigger Picture



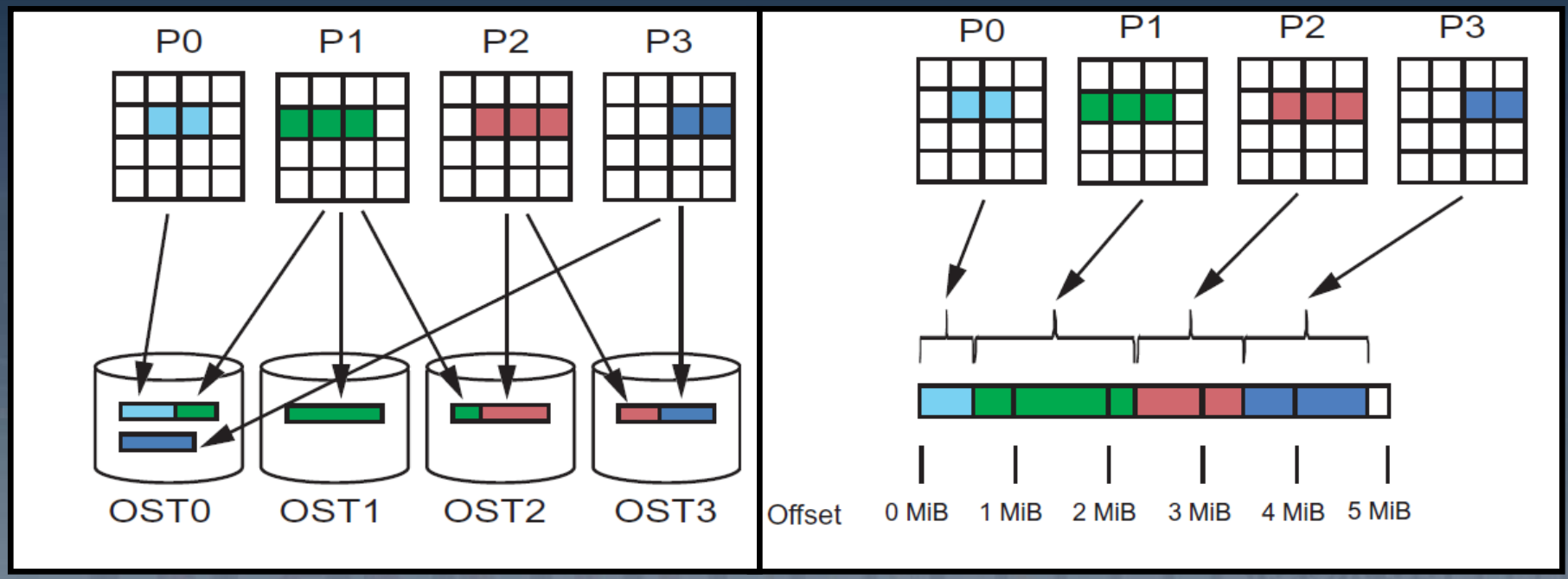

- ← Computational Nodes
  - Jaguarpf: 18,688
- ← Object Storage Server (OSS) Nodes
  - Jaguarpf: 168
  - 100 GB/s
- ← Object Storage Target (OST)
  - Jaguarpf: 672
  - 6.2 TB Disk
  - 4.1 PB

Go to Menu

# Lustre Striping



# File Parallelism



# Default Configuration

- The following command displays the IDs of the 672 file servers (called OSTs) on jaguarpf (as of 9/04/09), as well as the default stripe count, stripe size and stripe offset:

```
> lfs osts
OBDS:
0: widow1-OST0000_UUID ACTIVE
1: widow1-OST0001_UUID ACTIVE
2: widow1-OST0002_UUID ACTIVE
……..……
671: widow1-OST029f_UUID ACTIVE
/lustre/widow1
(Default) stripe_count: 4 stripe_size: 1048576 stripe_offset: 0
```

- The stripe count defines how many file servers a single file can be distributed over; the default stripe count on jaguarpf is 4. The stripe size (default, 1MB=1048576 bytes) is the number of bytes written on one OST before targeting the next (where applicable). The stripe offset is the starting OST ID.

# lfs getstripe

- To find out striping information for files and directories, the following command can be used: lfs getstripe --quiet <dir|file>

  For exemple,

```
> lfs getstripe --quiet file.txt
   539           831832          0xcb158                0
   502           831934          0xcb1be                0
   248           832342          0xcb356                0
   632           830997          0xcae15                0
```

- This example shows IDs for 4 target OSTs on the system.

# Modifications of the Defaults: `setstripe`

- Lustre provides a user command `setstripe` for modifying one or more striping parameters for individual files or directories.
- Syntax:

```
>lfs setstripe filename [stripe-size] [OST-offset] [stripe-count]
```

- For example, the following command would change the default stripe size to 2MB:

```
> lfs setstripe <dir|file> 2m -1 4
```

- Where `dir` is an existing directory, and `file` is a file that does not yet exist. The first parameter (`2m`) represents the stripe size, the second parameter is the stripe offset (`-1` is for round robin assignment starting at `OST 0`), while the third parameter represents the default stripe count.
- It is **HIGHLY** recommended that the default offset value is left unchanged.

# Modifications of the Defaults: `setstripe` (continued)

- When the `setstripe` is invoked on an existing directory, any new files that are created in that directory in the future will inherit the newly defined striping parameters. Existing files in that directory are not affected, however. When `setstripe` is invoked for a new file the file is created with the new striping parameters. Setstripe cannot be invoked for an existing file.
  For example, to limit the number of OSTs to 1 issue the following command:
  ```
  > lfs setstripe <dir|file> 1m -1 1
  ```
  and to use all available OSTs:
  ```
  > lfs setstripe <dir|file> 1m -1 -1
  ```
- Details of the supported Lustre commands are available on the lfs man page.
- Note that the commands relevant to system administrators may not work in user mode.

# General Optimization Tips

- There are different strategies for optimizing I/O performance on Rosa depending on the implementation of file I/O operations in an application and the behavior and sizes of data transfers as well as the file sizes. The table below lists some suggestions for commonly used file I/O implementations in scientific applications.

| File size | I/O pattern | Recommended setting |
|---|---|---|
| < 1GB | Single file per MPI task/core | lfs setstripe <dir\|file> 1m -1 1 |
| < 1GB | Single file (read/written by a single MPI task) | lfs setstripe <dir\|file> 1m -1 1 |
| < 1GB | Single shared file accessed by multiple MPI tasks/core | Default |
| < 100GB | Single file per MPI task/core | Default |
| < 100GB | Single file (read/written by a single MPI task) | Default |
| < 100GB | Single shared file accessed by multiple MPI tasks/core | lfs setstripe <dir\|file> 1m -1 10 |
| >100 GB | Single file per MPI task/core | Potential scaling bottleneck |
| >100 GB | Single file (read/written by a single MPI task) | Potential scaling bottleneck |
| >100 GB | Single shared file accessed by multiple MPI tasks/core | lfs setstripe <dir\|file> 1m -1 40 |

Go to Menu

# Lustre Best Practices for Users

- Use `lfs setstripe` in a safe manner
- Set striping appropriately for your use
- Choose stripe width for your application
- Avoid “excessively” large numbers of files in directories
- Avoid using `ls -l` repeatedly
- More information on website
  - http://www.nccs.gov/user-support/general-support/file-systems/spider

# Lustre Best Practices for Users (continued…)

- Use lfs setstripe in a safe manner
  - Always use the explicit options, not the relative ones
  - Avoid specifying a starting OST index
  - Use `-s` for stripe width (default is 1MB)
    - Can specify in bytes, kilobytes (k), megabytes (m), or gigabytes (g)
  - Use `-c` for stripe count (default is 4)
  - Not specifying an option keeps the current value
- Bad:
  - `lfs setstripe $NAME 1m -1 16`
- Good:
  - `lfs setstripe $NAME -s 1m -c 16`

# Lustre Best Practices for Users (continued…)

- Use `lfs getstripe` to check the striping on a file
- Example: extracting source code

```
# mkdir source
# lfs setstripe source c 1
# cd source
# tar -x –f $TARFILE
```

- Example: fixing incorrect striping

```
# lfs setstripe newfile -c 16
# cp oldfile newfile
# rm oldfile
# mv newfile oldfile
```

Go to Menu

# Lustre Best Practices for Users (continued…)

- Set striping appropriately for your use
  - Default stripe count is 4, but may not match your usage
  - Small files (< 250 MB) should use a single stripe
  - Large files accessed in parallel (single shared-file) should have a stripe count that is a factor of the number of writers (e.g. 20 vs. 21 for 400 writers)
- Cannot use more than 160 stripes currently
  - Maximum number of OSTs currently available

# Lustre Best Practices for Users (continued…)

- For single shared-file, choose per-writer data size as stripe width if possible
  - If each rank will write 256 MB, then use 256 MB as the stripe width
  - Minimizes lock contention
  - NCCS SciComp Liaison can help determine best stripe size
  - May not always be possible to pick a winner

# Lustre Best Practices for Users (continued…)

- Avoid directories with “excessive” numbers of files in them
  - “Excessive” is a fuzzy number
  - Greater 1M - definitely excessive
  - 100k - probably excessive
  - 50k - borderline
  - 10k - OK

# Lustre Best Practices for Users (continued…)

- Avoid doing `ls -l` repeatedly
  - Especially in an excessively large directory!
  - If you are just looking to see if a file exists, use plain `ls`
  - Better yet – look for that file explicitly
  - Avoid options that sort by time stamp or add color to the listing

# Lustre Best Practices for Developers

- Open files read-only when possible
- Read small, shared files once
- Use a directory hierarchy to limit files in a single directory
- Use access(), not stat() to check for existence
- Avoid flock()
- Consider using libLUT or middleware I/O libraries
- Stripe-align I/O if possible

# Lustre Best Practices for Developers (continued…)

- Open files read-only when possible
  - Fortran defaults to READWRITE if no ACTION is given
  - Fortran adds O_CREAT if opening file for writing
- O_CREAT requests an exclusive lock for the file (not contents)
  - Lock ping-pong championships when large job opens the file from all ranks at once

# Lustre Best Practices for Developers (continued…)

- If all ranks need data from a single file, it is better to have one reader and then broadcast the contents than have everyone read the file.

  Fortran example, without error handling and assuming known file size:

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
IF (my_rank .eq. 0) THEN
    OPEN(UNIT=1,FILE=PathName,ACTION='READ')
    READ(1,*) buffer
ENDIF
CALL MPI_BCAST(buffer,SIZE,MPI_CHAR,0,MPI_COMM_WORLD,ierr)
```

# Lustre Best Practices for Developers (continued…)

- Use a directory hierarchy to limit files in a single directory
  - Opening a file currently keeps a lock on the parent directory for one message round-trip
  - Split directory up to avoid contention
  - For two level hierarchy, square root of the total number of files provides best balance

# Lustre Best Practices for Developers (continued…)

- Use access(), not stat() to check for existence
  - Size is not kept on metadata server, so using stat() requires communication with each object storage server that has a portion of the file
  - access() only needs one request

# Lustre Best Practices for Developers (continued…)

- Avoid flock()
  - O(N**2) algorithm for number of lockers on file
  - Ok, if N is small
  - Does not scale to systems the size of Jaguar or JaguarPF

# Lustre Best Practices for Developers (continued…)

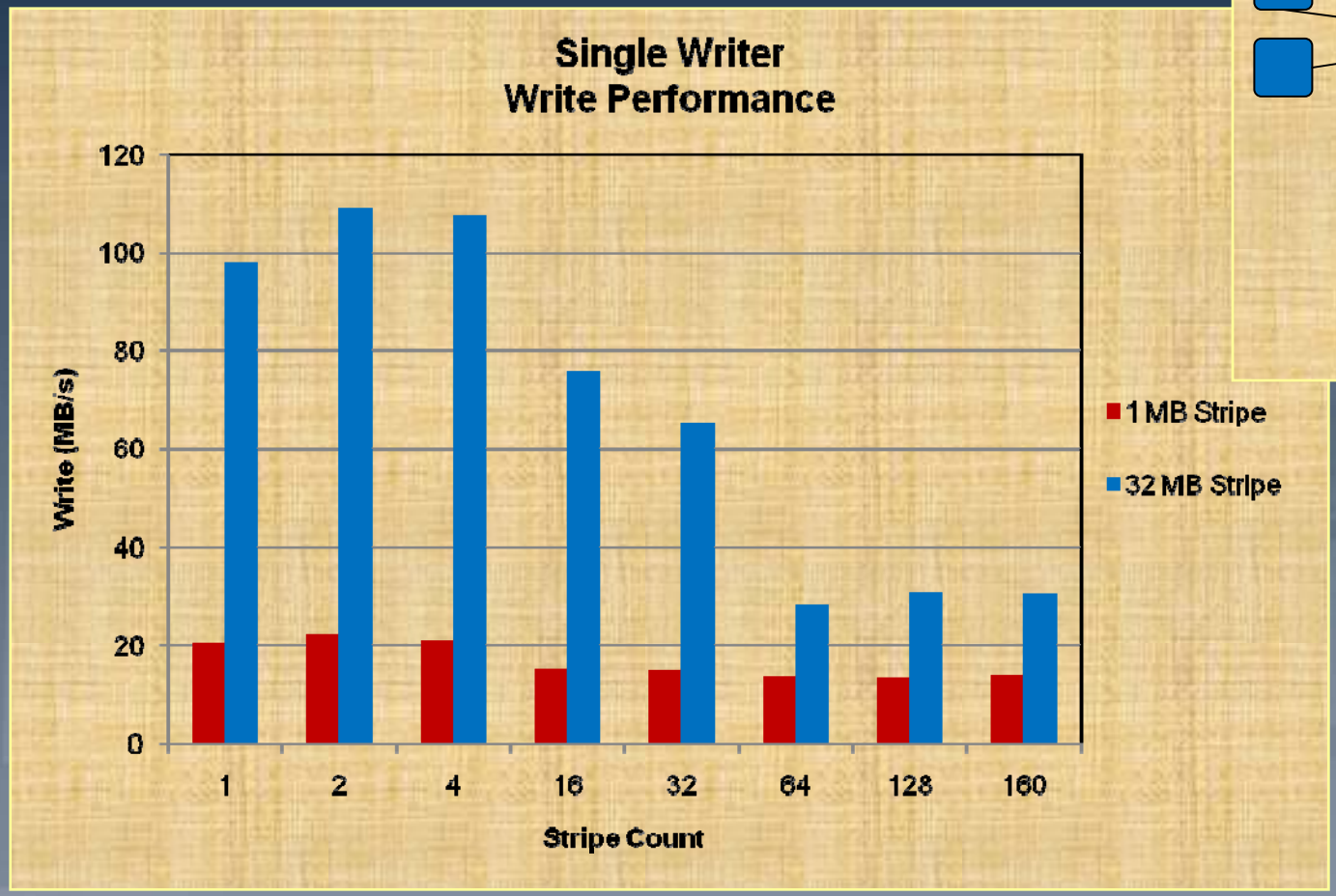
- Consider using libLUT or middleware I/O libraries such as ADIOS
  - Extracting full performance from the file system requires knowledge of the environment
  - Maintaining performance during concurrent access from other users requires constant adaptation
  - Do you really want to write all of this?
    - And maintain it for multiple systems?

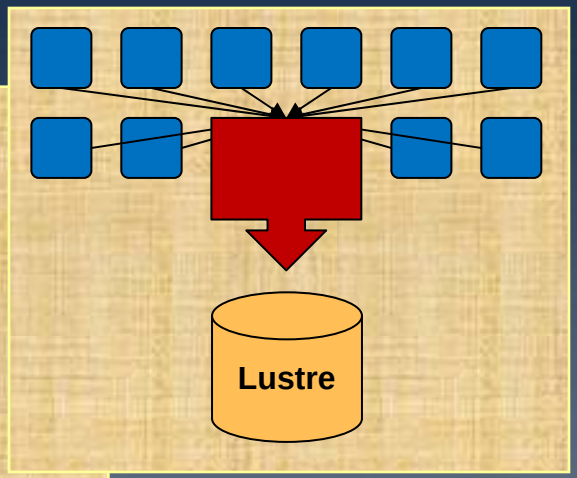
# Lustre Best Practices for Developers (continued…)

- Stripe-align I/O if possible
  - Lustre is a POSIX-compliant file system
  - Overlapping writes are 'last-to-write wins‘
  - This requires locking of the contents
  - Unaligned writes require obtaining locks from multiple servers

# Spokesperson – Serial I/O: importance of data locality

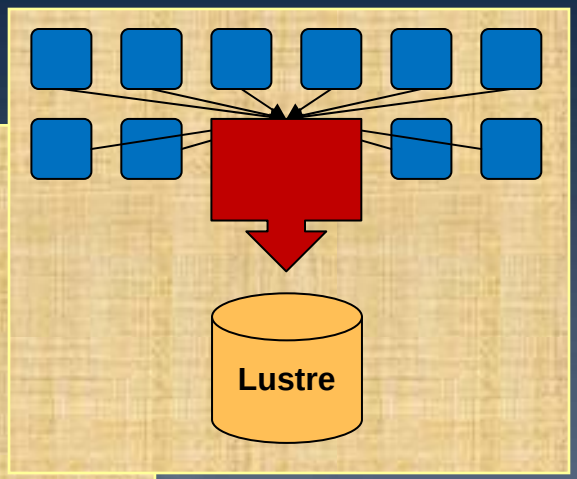
- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size

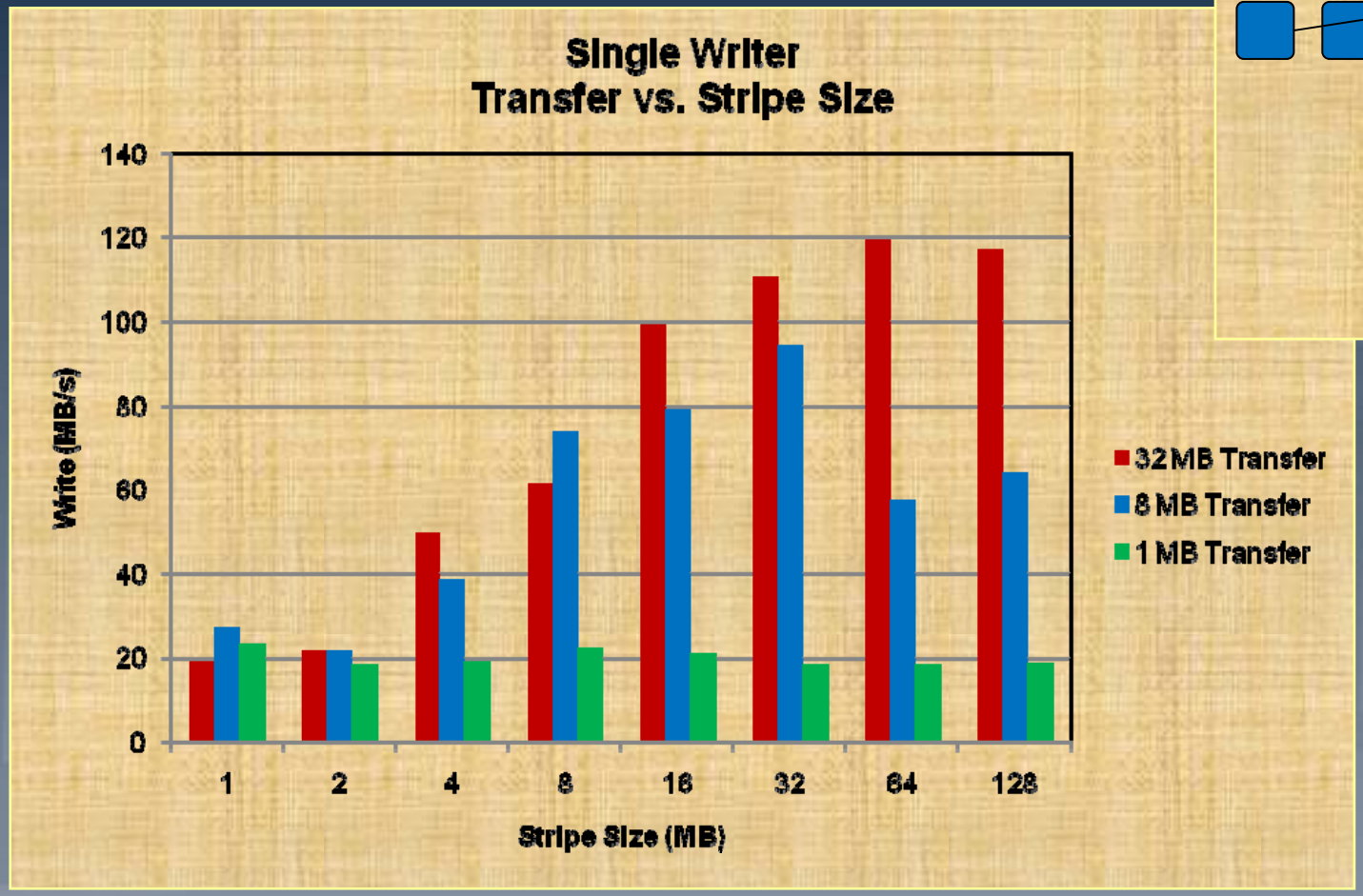


# Spokesperson – Serial I/O: importance of data continuity (cont.)

- Single OST, 256 MB File Size



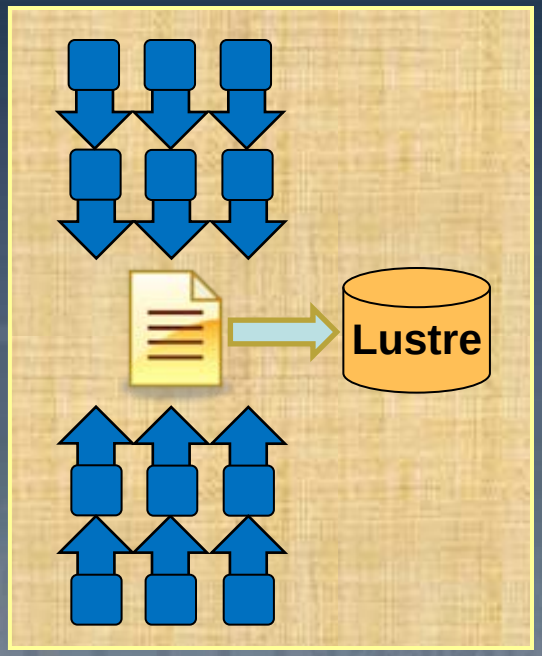
# Serial I/O: Data Locality and Continuity

- Data Locality
  - Performance is decreased when a single process accesses multiple disks.
  - Is limited by the single process which performs I/O.
- Data Continuity
  - Larger read/write operations improve performance.
  - Larger stripe sizes improve performance (places data contiguously on disk).
  - Either may become a limiting factor.

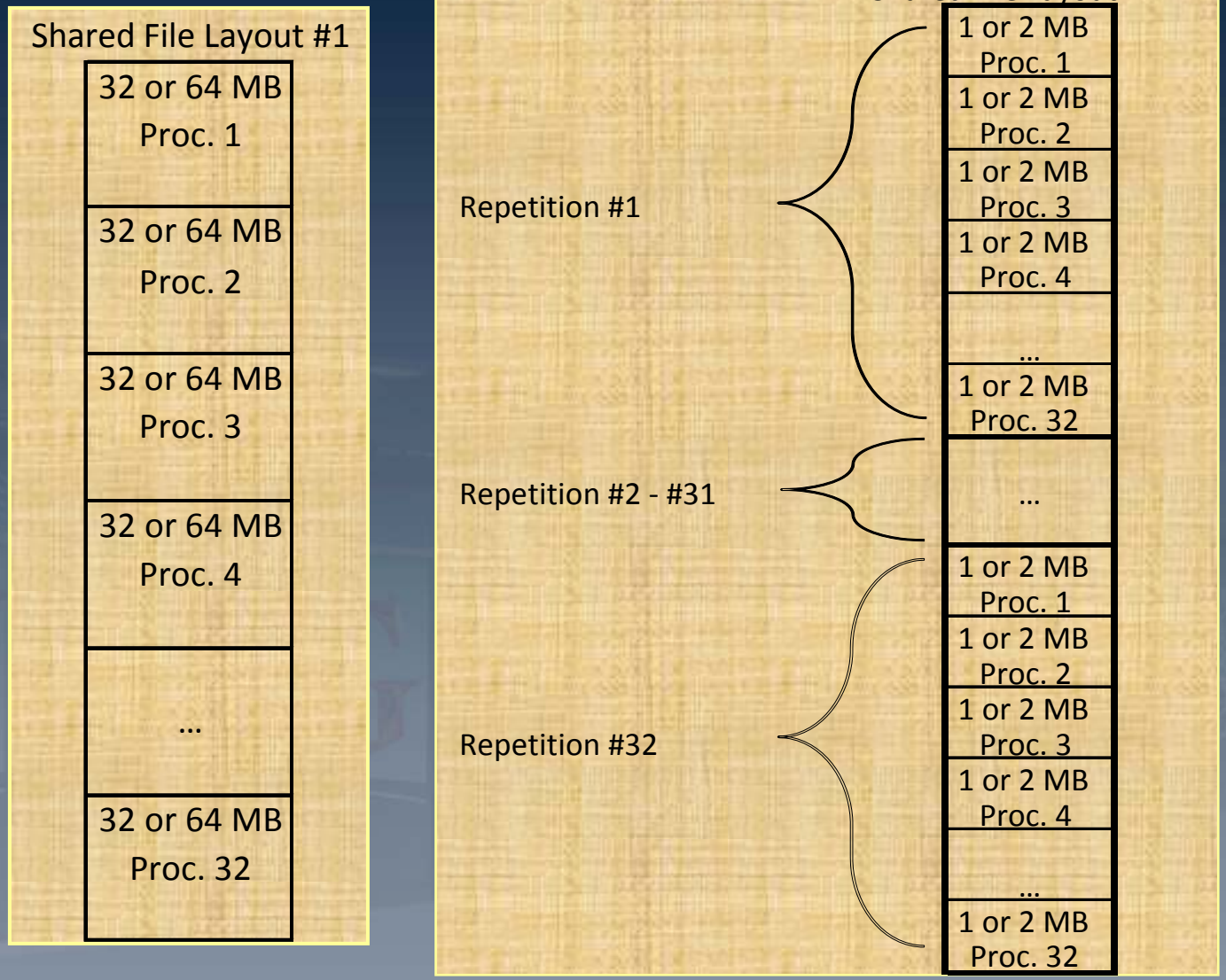
Go to Menu

# Single Shared File

- Important Considerations
  - Data locality
  - Data continuity
- Parallel file Structure

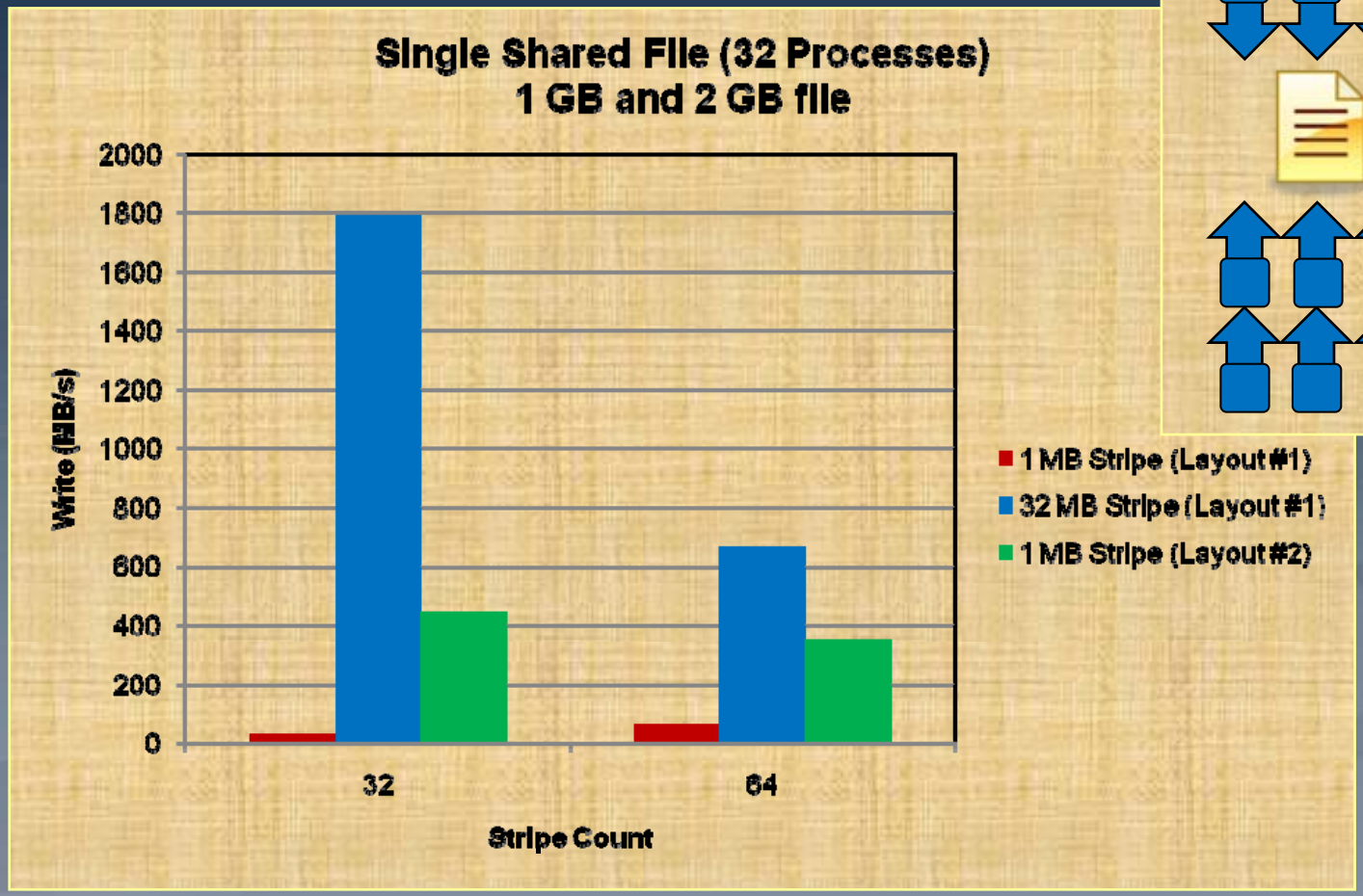


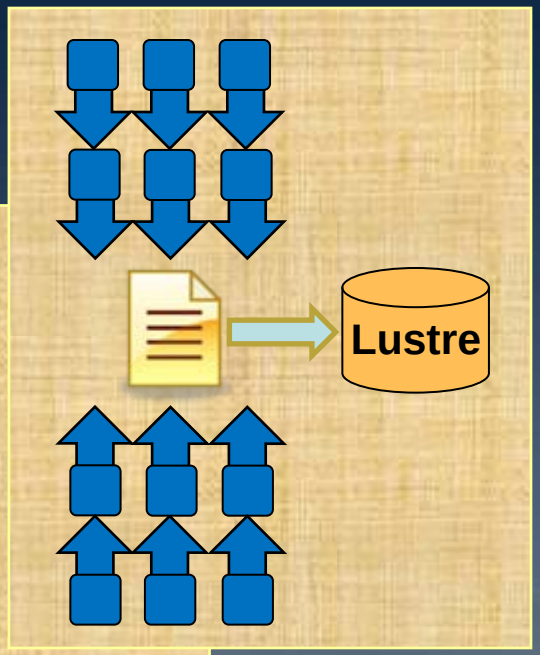
# Single Shared File: Shared File Layouts




Go to Menu

# Single Shared File: Results



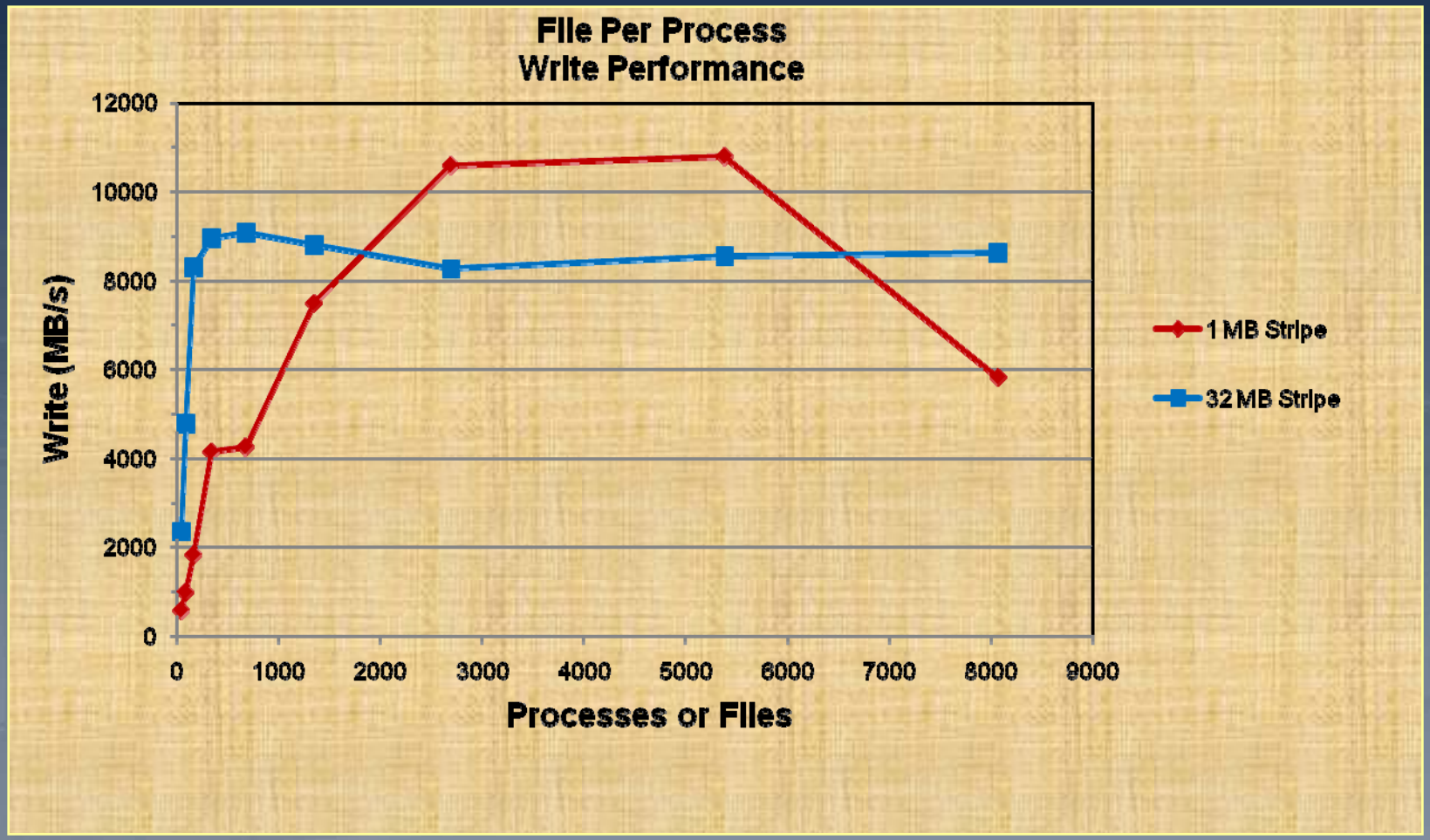

Go to Menu

# Single Shared File: Data Locality and Continuity

- Data Locality
  - Performance is increased when portions of a shared file are localized on a single drive.
- Data Continuity
  - Larger read/write operations improve performance.
  - Larger stripe sizes improve performance (places data contiguously on disk).
  - Either may become a limiting factor.

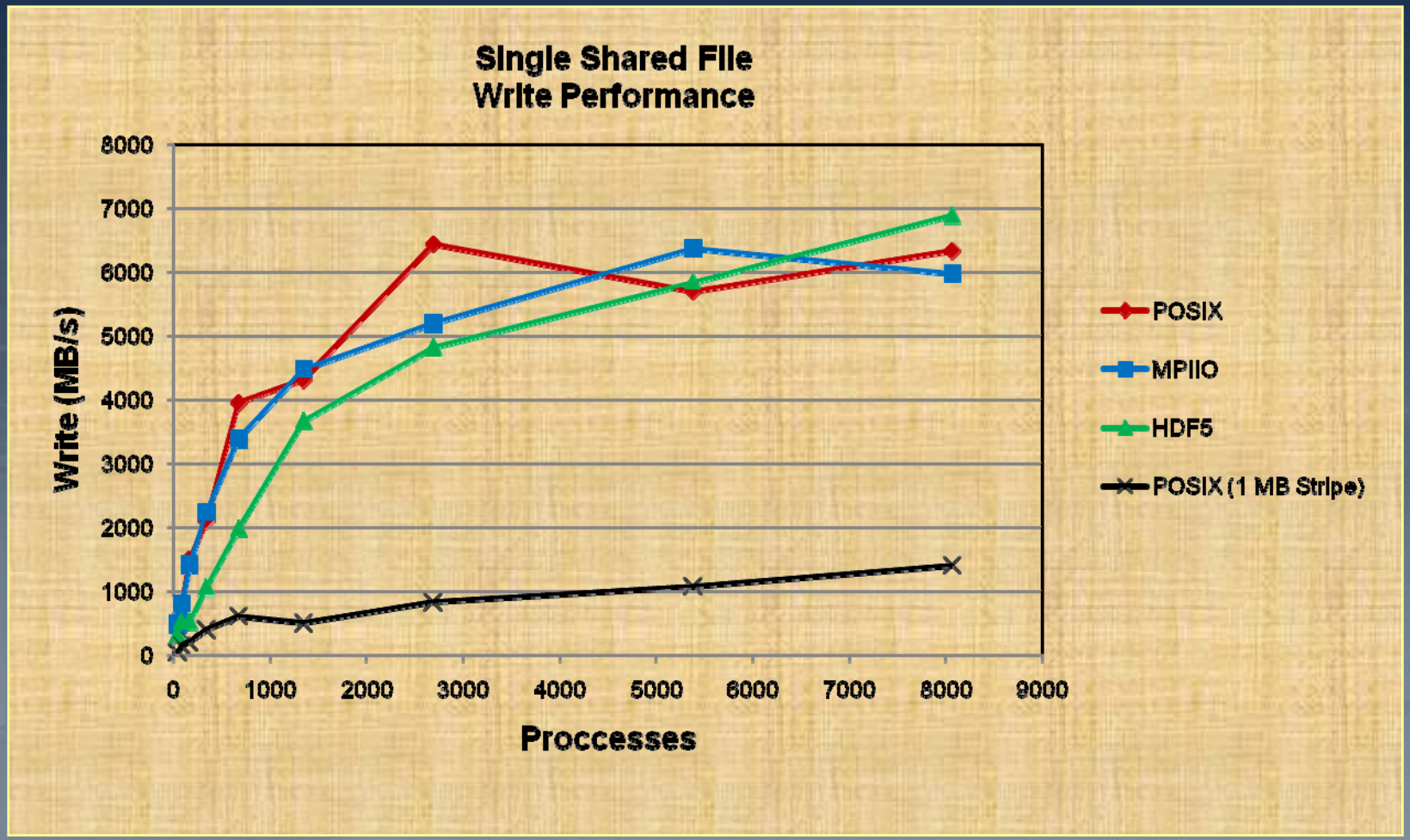
# Scalability: File Per Process

- 128 MB per file and a 32 MB Transfer size



# Scalability: Single Shared File

- 32 MB per process, 32 MB Transfer size and Stripe size



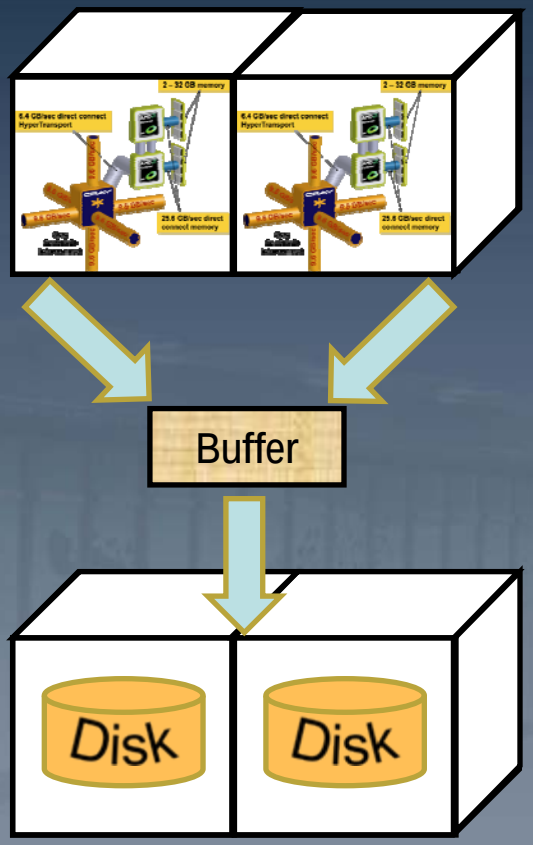
# Scalability: Summary

- Serial I/O
  - Is not scalable. Limited by single process which performs I/O.
- File per Process
  - Limited at large process/file counts by:
    - Metadata Operations
    - Contention on a single drive
- Single Shared File
  - Limited at large process counts by contention on a single drive.
  - File striping limitation of 160 OSTs in Lustre

Go to Menu

# Buffered I/O

- Advantages
  - Aggregates smaller read/write operations into larger operations.
  - Examples: OS Kernel Buffer, MPI-IO Collective Buffering
- Disadvantages
  - Requires additional memory for the buffer.
  - Can tend to serialize I/O.
- Caution
  - Frequent buffer flushes can adversely affect performance.



# Standard Output and Error

- Standard Ouput and Error streams are effectively serial I/O.
- Generally, the MPI launcher will aggregate these requests. (Example: mpirun, mpiexec, aprun, ibrun, etc..)
- Disable debugging messages when running in production mode.
  - "Hello, I'm task 32000!"
  - "Task 64000, made it through loop."

# Binary Files and Endianness

- Writing a big-endian binary file with compiler flag byteswapio

```
File “XXXXXX"
                  Calls    Megabytes    Avg Size
  Open                1
  Write         5918150     23071.28062          4088
  Close               1
  Total         5918152     23071.28062          4088
```

- Writing a little-endian binary

```
File “XXXXXX"
                  Calls    Megabytes    Avg Size
  Open                1
  Write             350     23071.28062      69120000
  Close               1
  Total             352     23071.28062      69120000
```

- Can use more portable file formats such as HDF5, NetCDF, or MPI-IO.

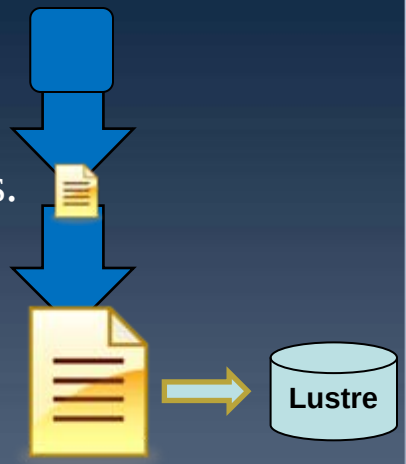
Go to Menu

# Case Study: Parallel I/O

- A particular code both reads and writes a 377 GB file. Runs on 6000 cores.
  - Total I/O volume (reads and writes) is 850 GB.
  - Utilizes parallel HDF5
- Default Stripe settings: count 4, size 1M, index -1.
  - 1800 s run time (~ 30 minutes)
- Stripe settings: count -1, size 1M, index -1.
  - 625 s run time (~ 10 minutes)
- Results
  - 66% decrease in run time.

# Case Study: Buffered I/O

- A post processing application writes a 1GB file.
- This occurs from one writer, but occurs in many small write operations.
  - Takes 1080 s (~ 18 minutes) to complete.
- IOBUF was utilized to intercept these writes with 64 MB buffers.
  - Takes 4.5 s to complete. A 99.6% reduction in time.




```
File "ssef_cn_2008052600f000"
                  Calls         Seconds       Megabytes   Megabytes/sec   Avg Size
   Open               1        0.001119
   Read             217        0.247026        0.105957        0.428931        512
   Write        2083634        1.453222     1017.398927      700.098632        512
   Close              1        0.220755
   Total        2083853        1.922122     1017.504884      529.365466        512
   Sys Read           6        0.655251      384.000000      586.035160   67108864
   Sys Write         17        3.848807     1081.145508      280.904052   66686072
   Buffers used           4 (256 MB)
   Prefetches             6
   Preflushes            15
```

# Fault Tolerance

- Allow application to generate checkpoint files.
  - Should be minimal in size.
  - Should not be written too often.
- Keeping checkpoint files minimal
  - Only incorporate unique information. Allow application to calculate or derive appropriate information.
- Keeping the checkpoint generation low.
  - The goal isn't to keep all information at all times. (checkpointing after every iteration.)
  - Pick a write frequency which allows for a reasonable loss of computation time.

# Outline: Resources for Users

- I/O-Related References
- Getting Started
- Advanced Topics
- More Information

# Resources for Users: I/O-Related References

- PVFS and PVFS2 (open source)
  - www.parl.clemson.edu/pvfs/
  - www.pvfs.org/pvfs2/
- Lustre File System
  - www.lustre.org
- GPFS
  - www.almaden.ibm.com/storagesystems/file_systems/GPFS/
- Lustre File System – White Paper October 2008
  - http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf
- GPFS:  Concepts, Planning, and Installation Guide
  - http://www.publib.boulder.ibm.com/epubs/pdf/a7604132.pbf
- Introduction to HDF5
  - http://www.hdfgroup.org/HDF5/doc/H5.intro.html
- The NetCDF Tutorial
  - http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial.pdf
- John May, Parallel I/O for High Performance Computing, Morgan Kaufmann, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF

Go to Menu

# Resources for Users: Getting Started

- About Jaguar

  http://www.nccs.gov/computing-resources/jaguar/

- Quad Core AMD Opteron Processor Overview

  http://www.nccs.gov/wp-content/uploads/2008/04/amd_craywkshp_apr2008.pdf

- PGI Compilers for XT5

  http://www.nccs.gov/wp-content/uploads/2008/04/compilers.ppt

- NCCS Training & Education – archives of NCCS workshops and seminar series, HPC/parallel computing references

  http://www.nccs.gov/user-support/training-education/

- 2009 Cray XT5 Quad-core Workshop

  http://www.nccs.gov/user-support/training-education/workshops/2008-cray-xt5-quad-core-workshop/

Go to Menu

# Resources for Users: Advanced Topics

- Debugging Applications Using TotalView

  http://www.nccs.gov/user-support/general-support/software/totalview

- Using Cray Performance Tools - CrayPat

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/cray-pat/

- I/O Tips for Cray XT4

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/io-tips/

- NCCS Software

  http://www.nccs.gov/computing-resources/jaguar/software/

# Resources for Users: More Information

- NCCS website

  http://www.nccs.gov/

- Cray Documentation

  http://docs.cray.com/

- Contact us

  help@nccs.gov