高性能计算程序设计基础

任课教师：黄聃（Huang，Dan）

Parallel I/O Techniques and Performance Optimization

Some of slides by Lonnie Crosby, NICS Scientific Computing Group

Some of slides by UIUC

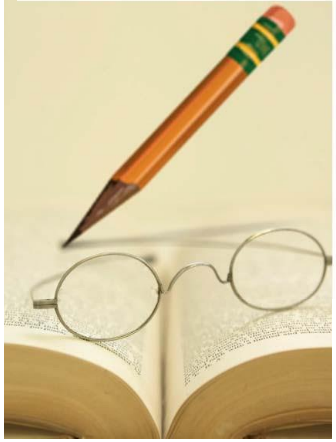
Outline

- Introduction to I/O
- Path from Application to File System
  - Data and Performance
  - I/O Patterns
  - Lustre File System
  - I/O Performance Results
- MPI-IO
  - General File I/O
  - Derived MPI DataTypes
  - Collective I/O
- Common I/O Considerations

Application Examples

Factors which affect I/O

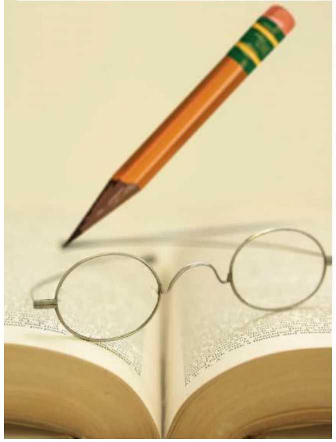
- I/O is simply data migration.
  - Memory ⟷ Persistent devices, e.g. SSD, Disk
- I/O is a very expensive operation.
  - Interactions with data in memory and on persistent devices.
- How is I/O performed?
  - I/O Pattern
    - Number of processes and files.
    - Characteristics of file access.



- Where is I/O performed?
  - Characteristics of the computational system.
  - Characteristics of the file system.

I/O Performance

- There is no "One Size Fits All" solution to the I/O problem.
- Many I/O patterns work well for some range of parameters.
- Bottlenecks in performance can occur in many locations. (Application and/or File system)
- Going to extremes with an I/O pattern will typically lead to problems.
- Increase performance by decreasing number of I/O operations (latency) and increasing size (bandwidth).

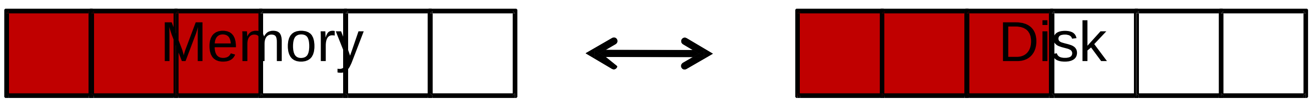


Outline

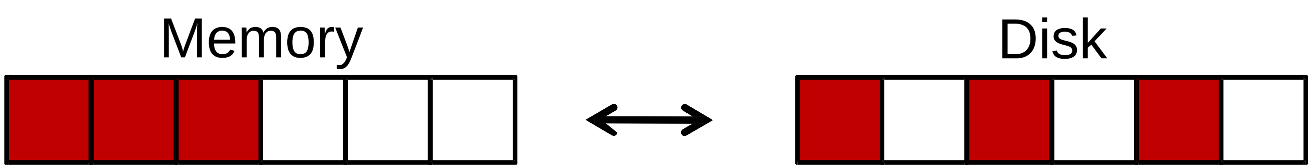
- Path from Application to File System
  - Data and Performance
  - I/O Patterns
  - Lustre File System
  - I/O Performance Results

Data and Performance

- The best performance comes from situations when the data is accessed contiguously in memory and on disk.

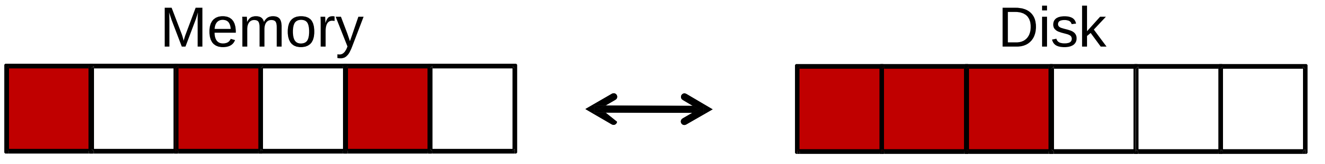


- Commonly, data access is contiguous in memory but noncontiguous on disk. For example, to reconstruct a global data structure via parallel I/O.

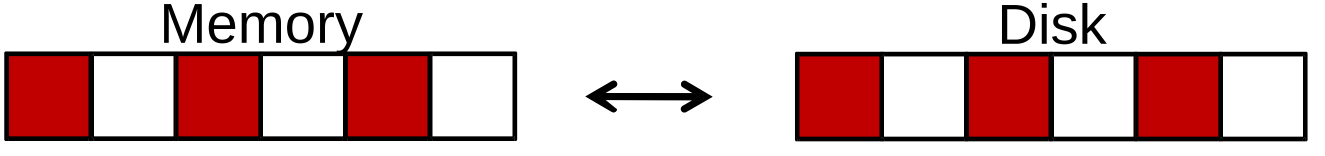


Data and Performance

- Sometimes, data access may be contiguous on disk but noncontiguous in memory. For example, writing out the interior of a domain without ghost cells.

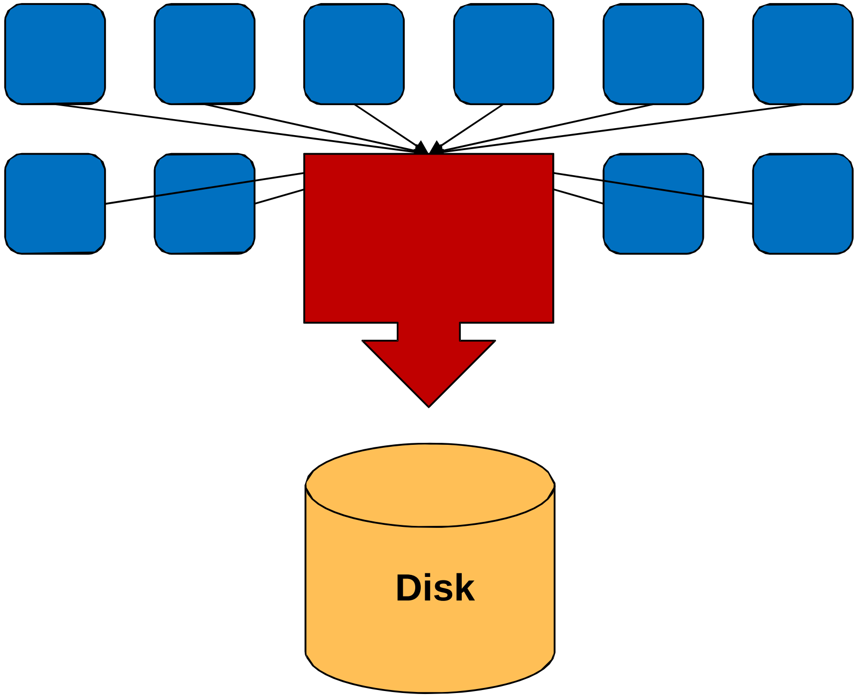


- A large impact on I/O performance would be observed if data access was noncontiguous both in memory and on disk.



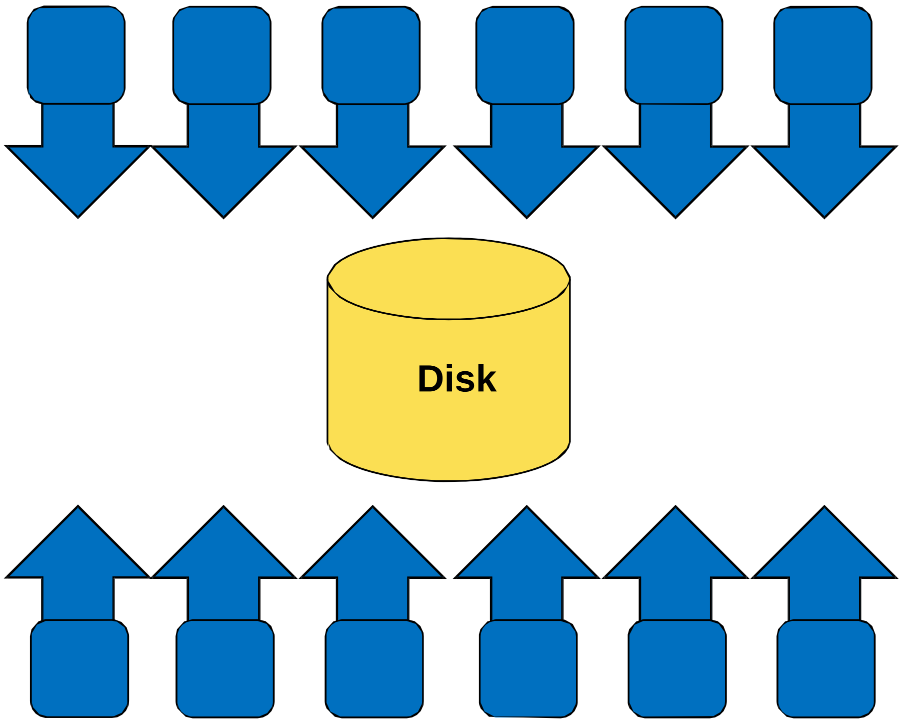
Serial I/O:

- Serial I/O
  - One process performs I/O.
    - Data Aggregation or Duplication (I/O size large)
    - Performance limited by single I/O process.
  - Pattern does not scale.
    - I/O rate does not increase with process count.



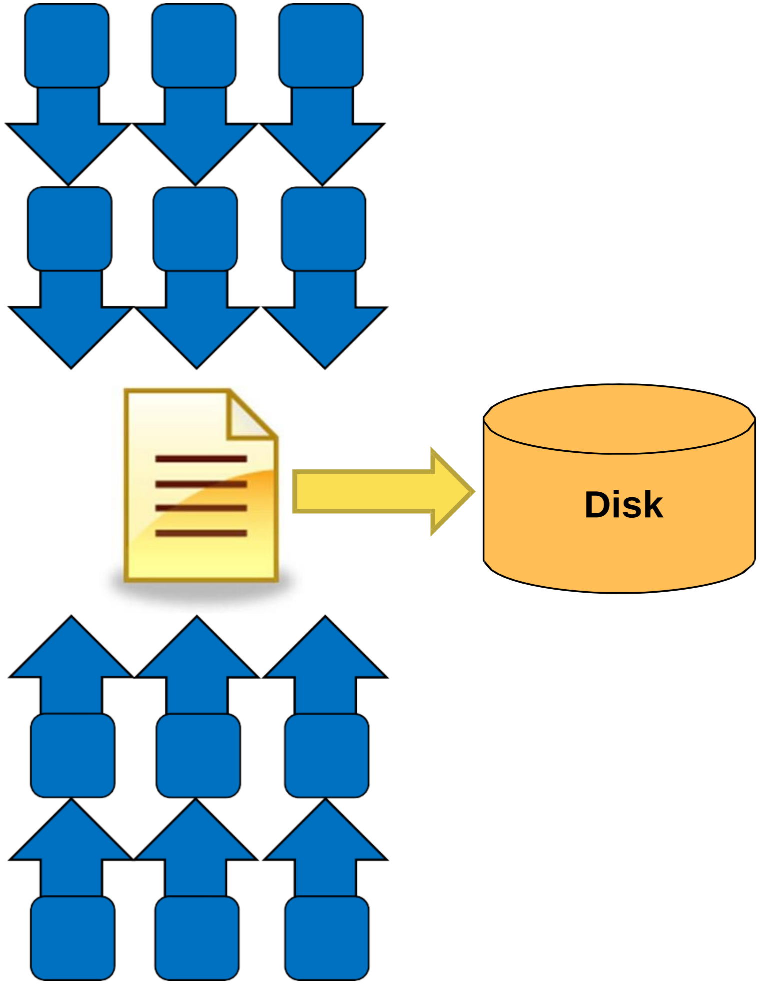
Parallel I/O: File-per-Process

- File per process
  - All processes perform I/O to individual files.
    - I/O rate increases with number of files.
    - Limited by file system.
    - I/O size small
  - Pattern does not scale at large process counts.
    - Hardware constraints such as number of disks, arrays, etc. limits the I/O rate increase with number of processes.



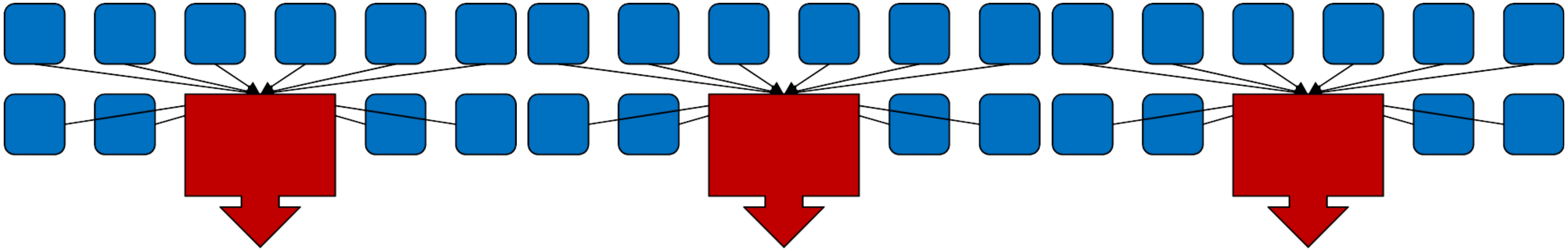
Parallel I/O: Shared File

- Shared File
  - Each process performs I/O to a single file which is shared.
    - I/O rate increases with number of processes
    - Limited by filesystem
    - I/O size small
  - Performance
    - Data layout within the shared file is very important.



Pattern Combinations

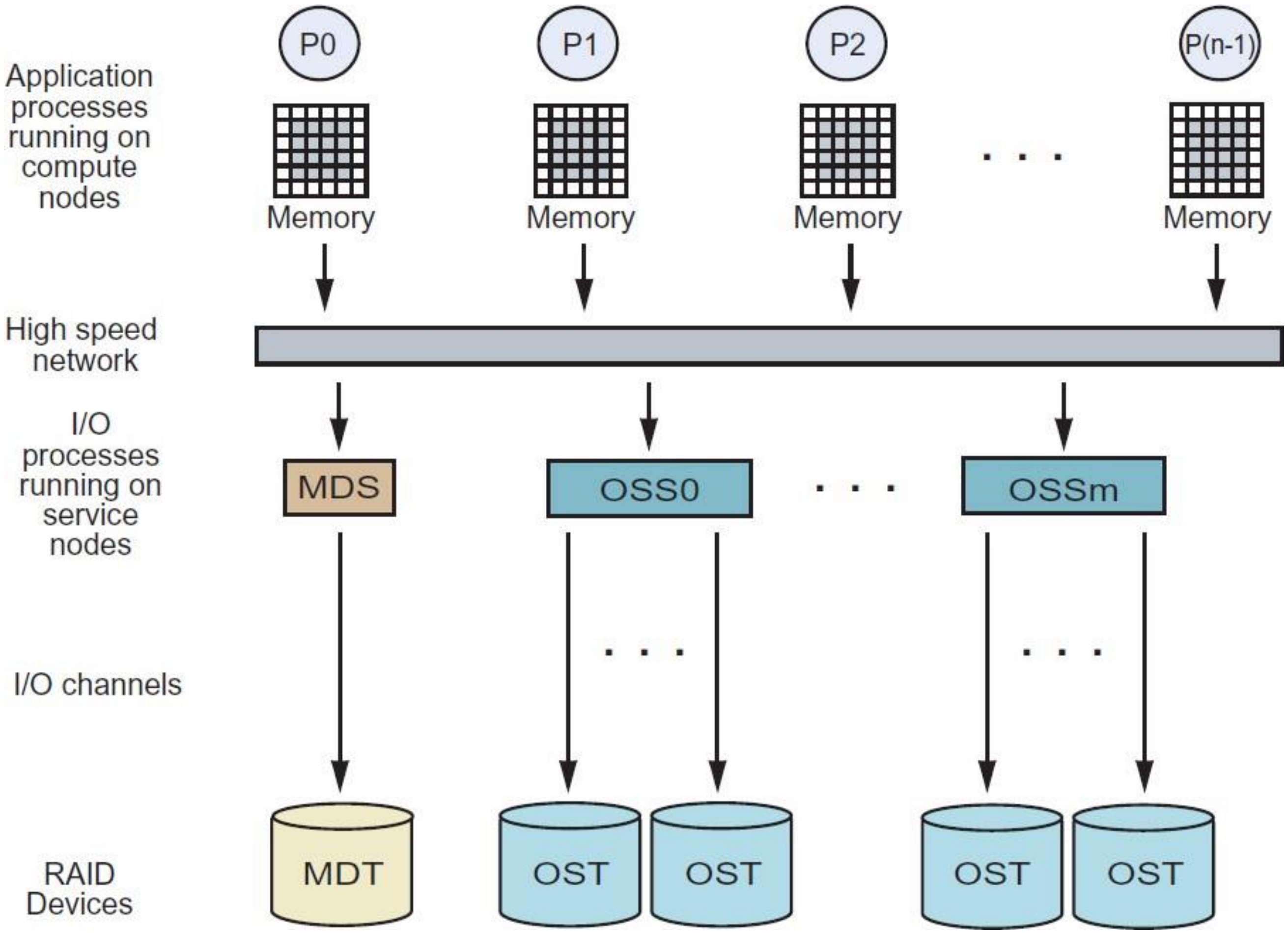
- Subset of processes which perform I/O.
  - Aggregation of a group of processes data.
    - Serializes I/O in group. Increases I/O size.
  - I/O process may access independent or shared files.
    - Limits the number of files accessed.
  - Group of processes perform parallel I/O to a shared file.
    - Increases the number of shared files to increase file system usage.
    - Decreases number of processes which access a shared file to decrease file system contention.



Performance Mitigation Strategies

- File-per-process I/O
  - Restrict the number of processes/files written simultaneously. Limits file system limitation.
  - Buffer output to increase the I/O operation size.
- Shared file I/O
  - Restrict the number of processes accessing file simultaneously. Limit file system limitation.
  - Aggregate data to a subset of processes to increase the I/O operation size.
  - Decrease the number of I/O operations by writing/reading strided data.

A Bigger Picture: Lustre File System



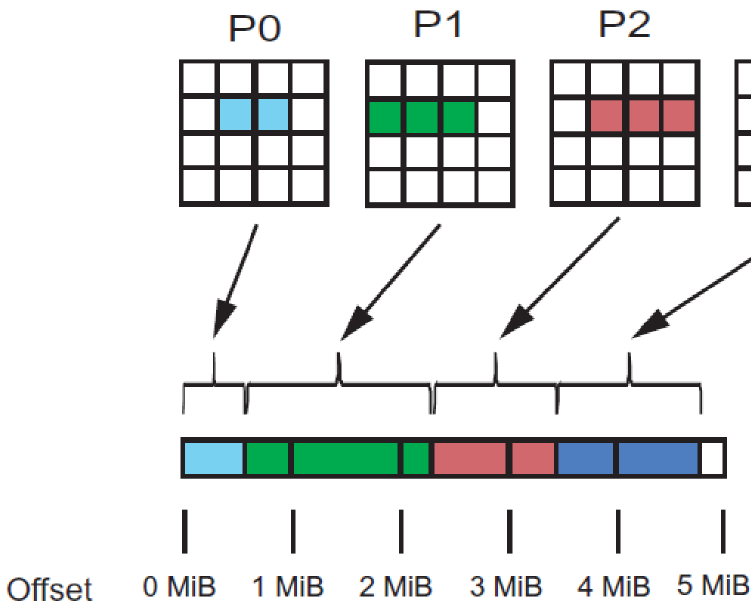
Striping on the Lustre File system

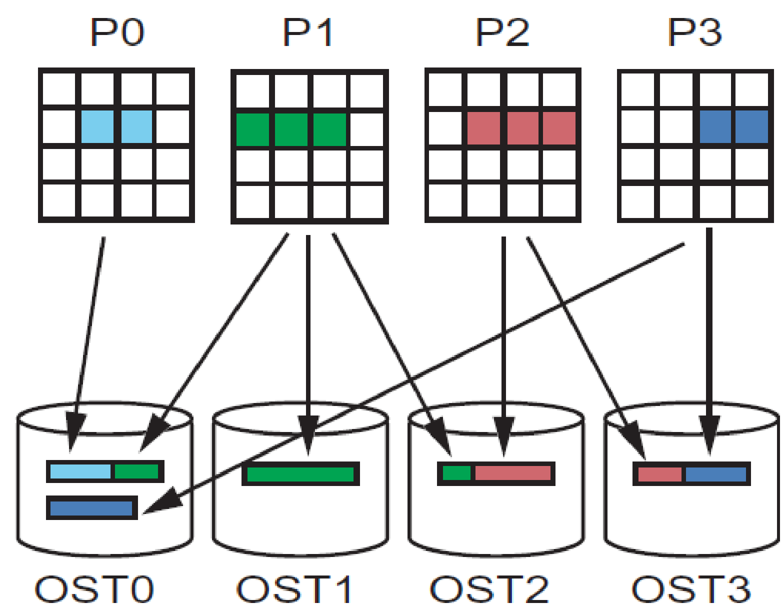
- lfs setstripe and getstripe command
  - Syntax: lfs setsripe –c <stripe_count> -s <stripe_size> -i <stripe_index> <file or directory>
  - <stripe_count>
    - 0 (Use default)
    - -1 (Use all available OSTs, max = 160)
  - <stripe_size>
    - 0 (Use default)
    - In bytes, although can be specified with k, m or g (in KB, MB and GB respectively)
  - <stripe_index>
    - -1 (allow MDS to choose starting OST, recommended)
  - <file or directory>
    - Cannot change the striping characteristics of existing files
    - Striping characteristics of directories can be changed at any time.

LFS setstripe and getstripe commands

- Example:
  - % lfs setstripe -c 5 -s 2M -i 27 test_file_stripe
  - % lfs getstripe test_file_stripe
  - test_file_stripe
  - lmm_stripe_count: 5
  - lmm_stripe_size: 2097152
  - lmm_stripe_offset: 27
  - obdidx objid objid group
  - 27 29421438 0x1c0ef7e 0
  - 97 29211011 0x1bdb983 0
  - 87 29386728 0x1c067e8 0
  - 90 28982042 0x1ba3b1a 0
  - 13 29013598 0x1bab65e 0

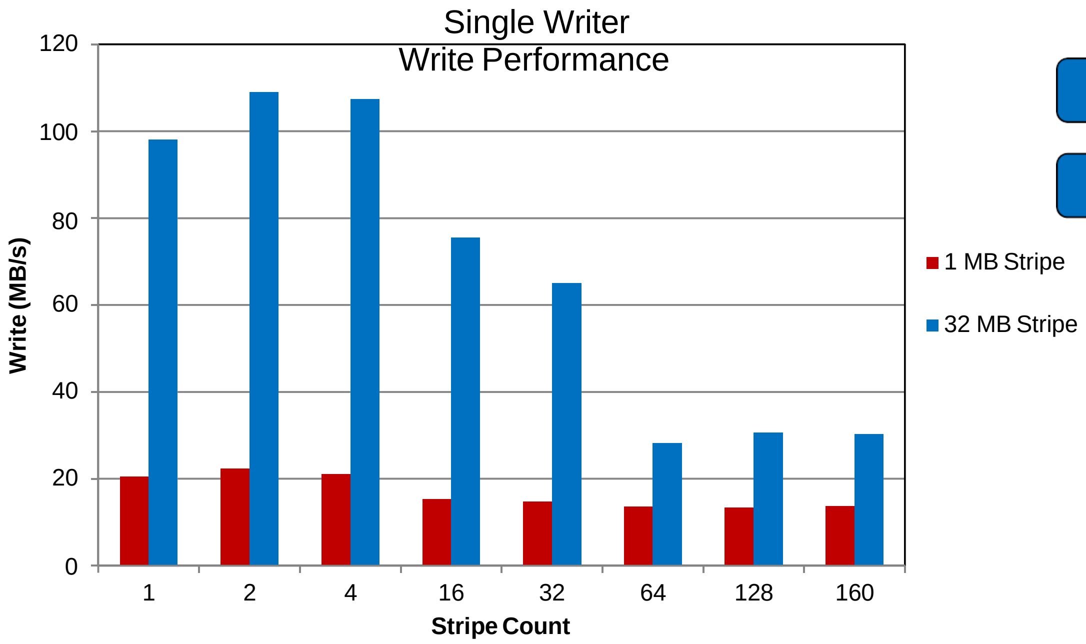
File Striping: Physical and Logical Views

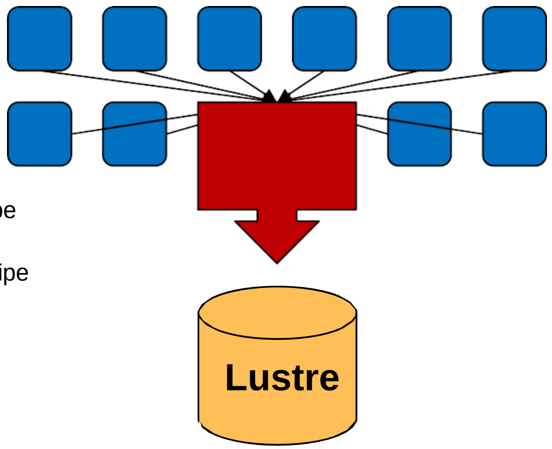




Single writer performance and Lustre

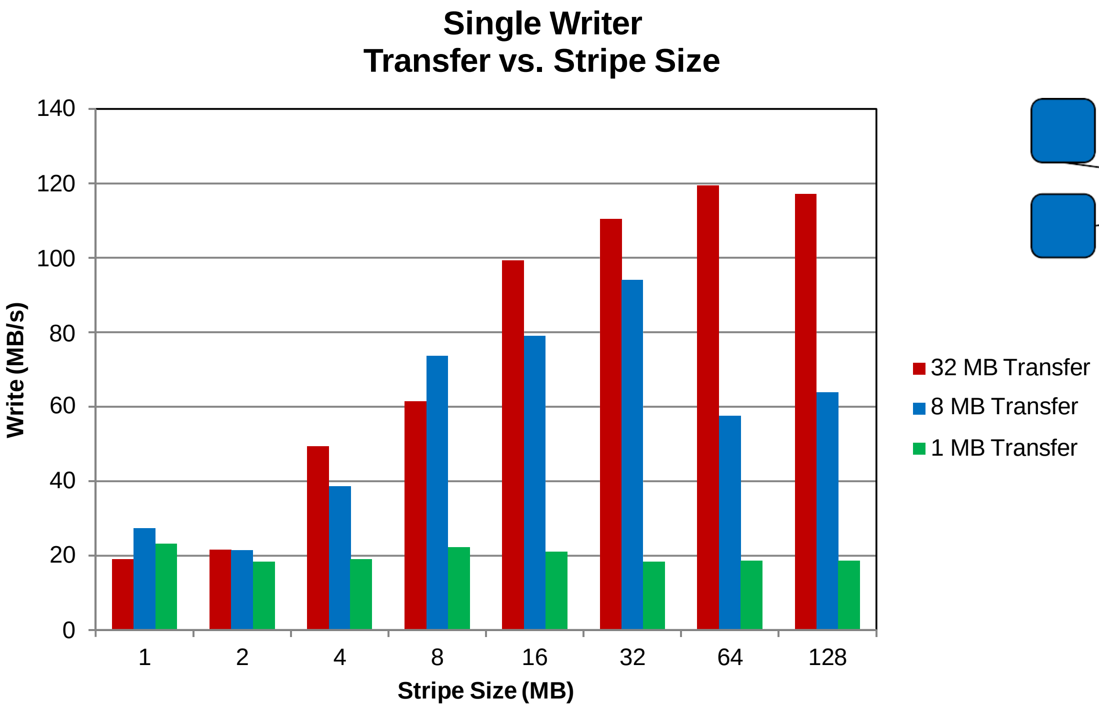
- File size 32 MB – 5 GB (32 MB per OST)
- 32 MB I/O write (transfer) size (large)
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance

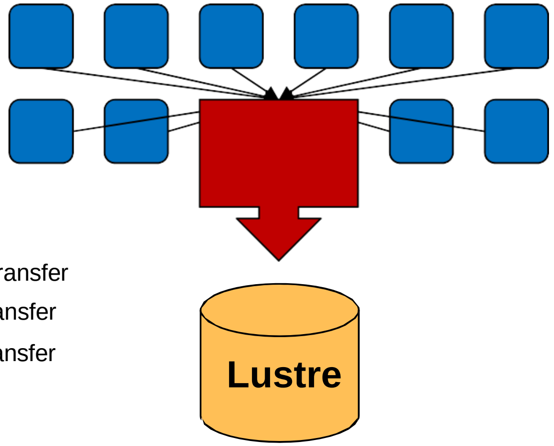




Stripe size and I/O Operation size

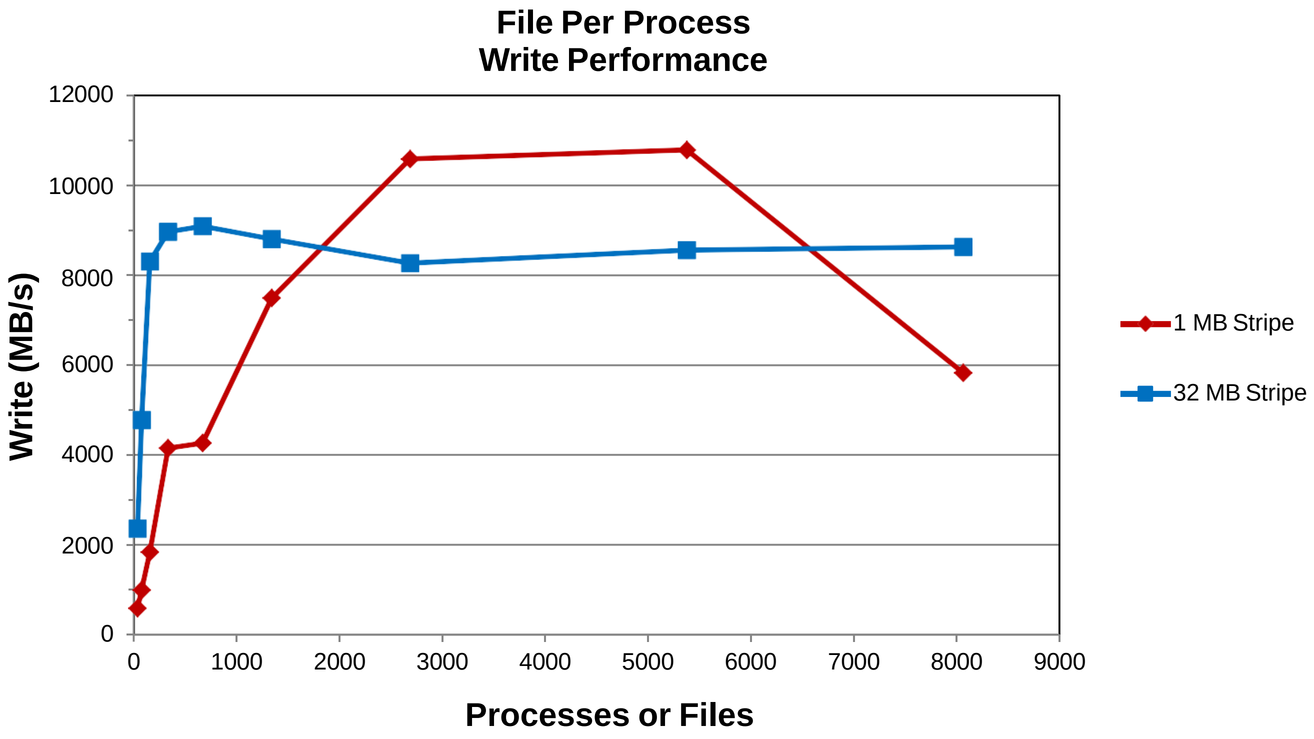
- Single OST, 256 MB File Size
  - Performance can be limited by the process (transfer size) or file system (stripe size)



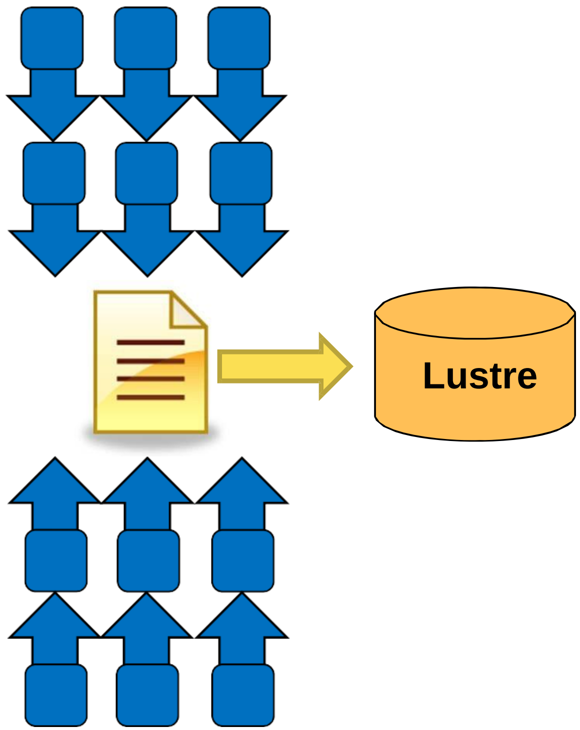


Scalability: File Per Process

- 128 MB per file and a 32 MB Transfer size



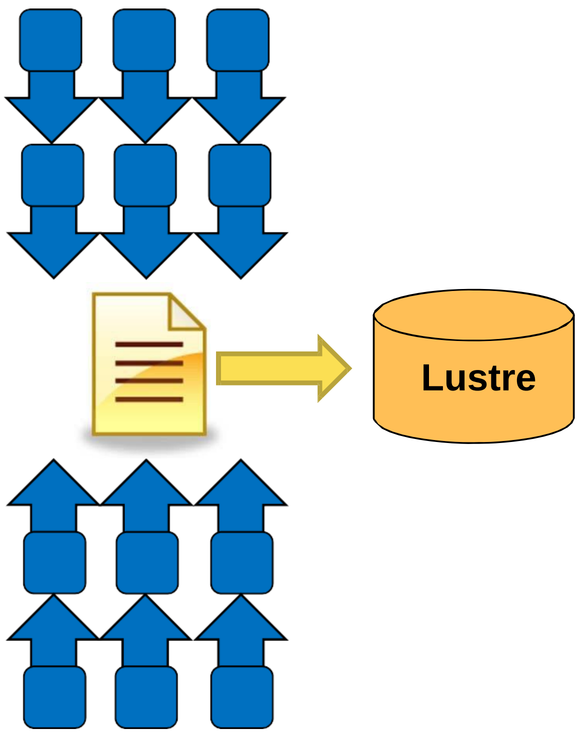
Single Shared Files and Lustre Stripes

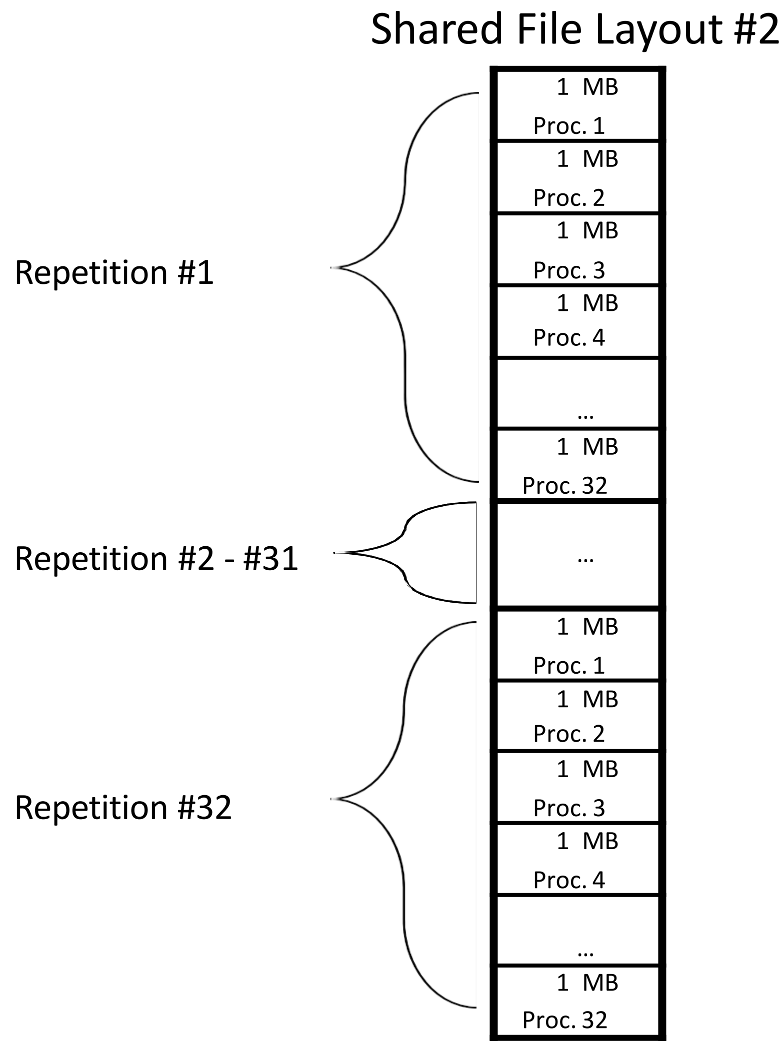


Shared File Layout #1

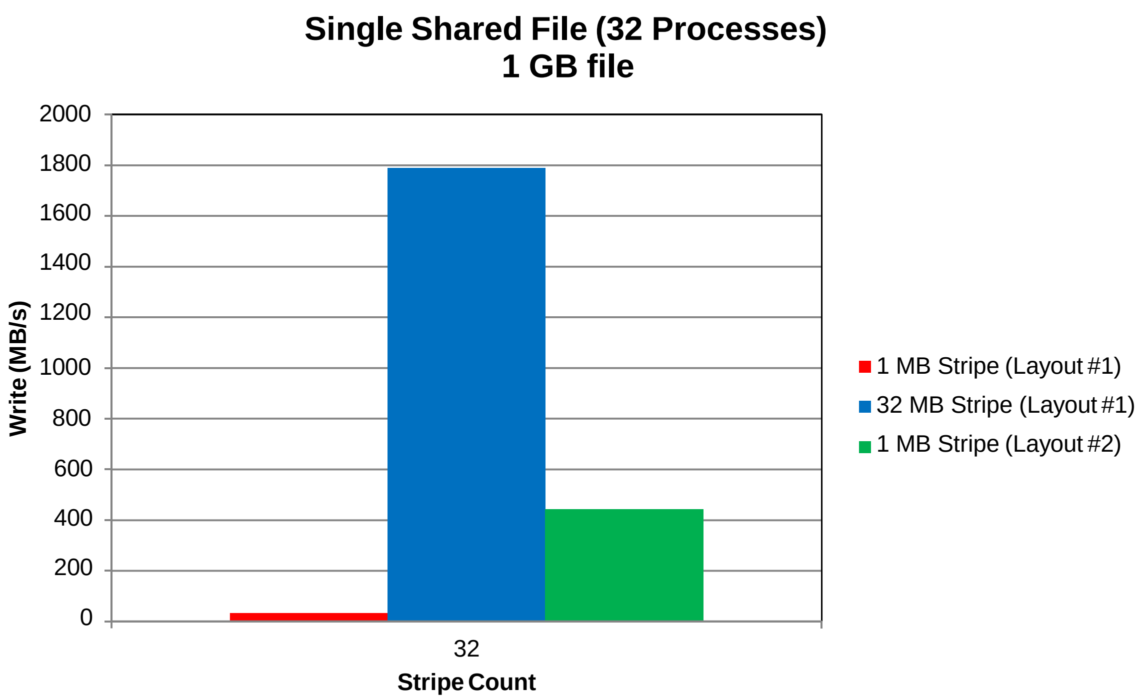
32 MB Proc. 1
32 MB Proc. 2
32 MB Proc. 3
32 MB Proc. 4
...
32 MB Proc. 32

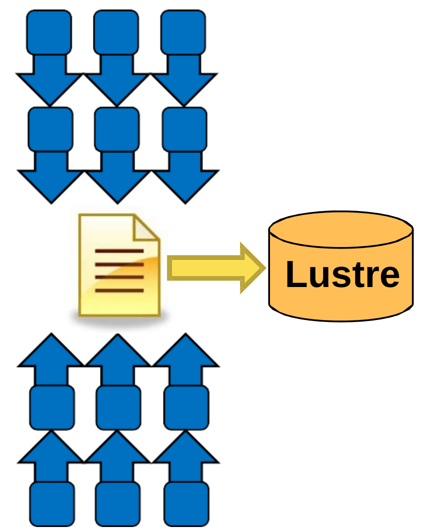
Single Shared Files and Lustre Stripes





File Layout and Lustre Stripe Pattern





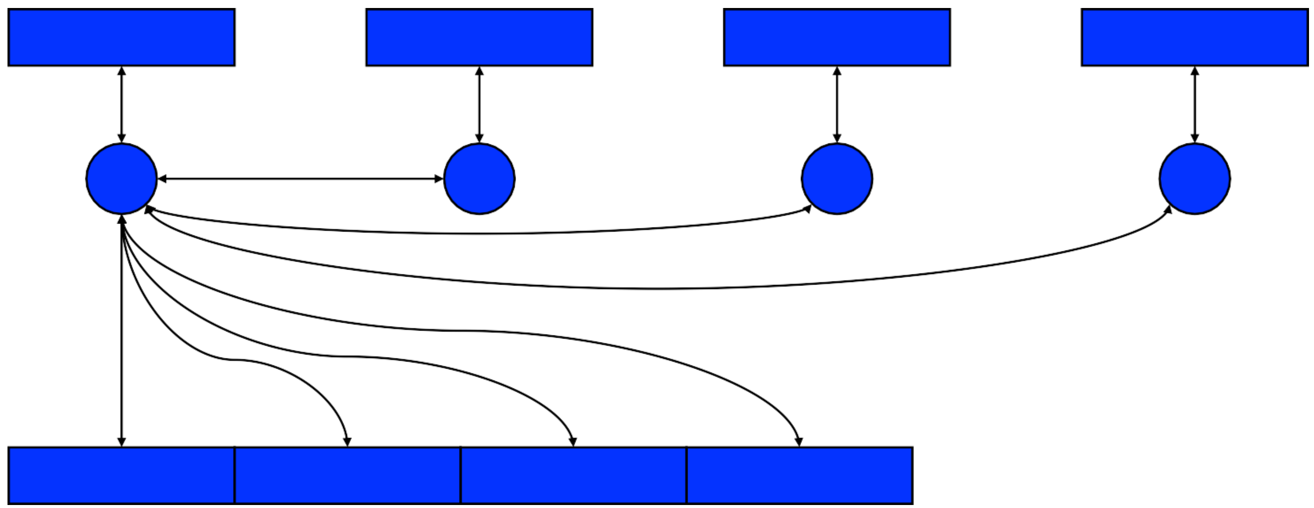
Summary

- Lustre
  - Minimize contention for file system resources.
  - A process should not access more than one or two OSTs.
  - Decrease the number of I/O operations (latency).
  - Increase the size of I/O operations (bandwidth).
- Performance
  - Performance is limited for single process I/O.
  - Parallel I/O utilizing a file-per-process or a single shared file is limited at large scales by the file system.
  - Potential solution is to utilize multiple shared files or a subset of processes which perform I/O.

Parallel I/O in MPI

- Why do I/O in MPI?
  - ♦ Why not just POSIX?
    - Parallel performance
    - Single file (instead of one file / process)
- MPI has replacement functions for POSIX I/O
  - ♦ Provides migration path
- Multiple styles of I/O can all be expressed in MPI
  - ♦ Including some that cannot be expressed without MPI

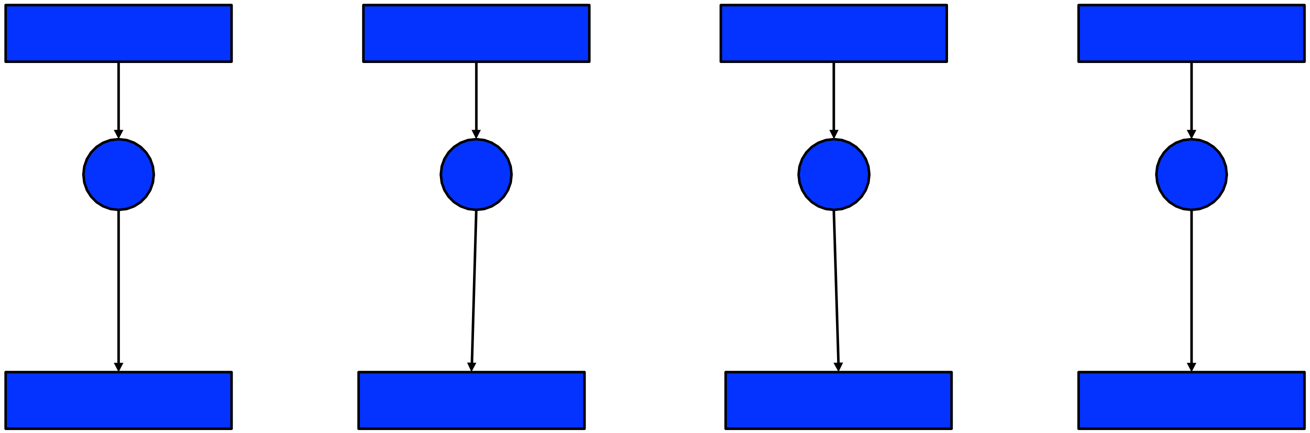
Non-Parallel I/O



- Non-parallel
- Performance worse than sequential
- Legacy from before application was parallelized
- Either MPI or not

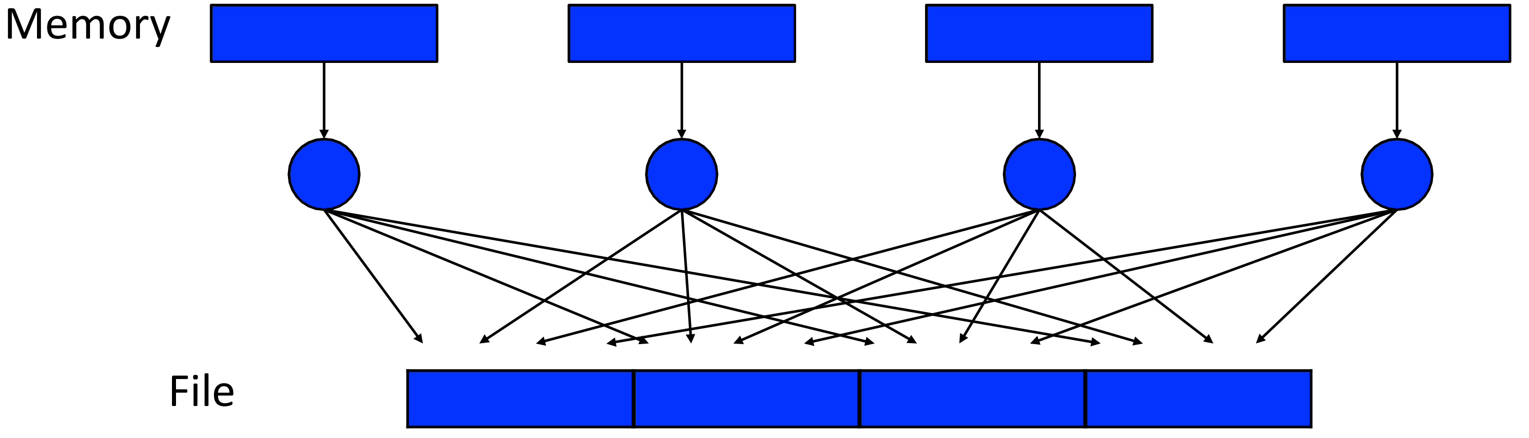
Independent Parallel I/O

- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage
- Legacy from before MPI
- MPI or not

Cooperative Parallel I/O



- Parallelism
- Can only be expressed in MPI
- Natural once you get used to it

Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations

What does Parallel I/O Mean?

- At the program level:
  - Concurrent reads or writes from multiple processes to a common file
- At the system level:
  - A parallel file system and hardware that support such concurrent access

The Basics: An Example

- Just like POSIX I/O, you need to
  - Open the file
  - Read or Write data to the file
  - Close the file
- In MPI, these steps are almost the same:
  - Open the file: MPI_File_open
  - Write to the file: MPI_File_write
  - Close the file: MPI_File_close

A Complete Example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_File fh;
    int buf[1000], rank;
    MPI_Init(0,0);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File_open(MPI_COMM_WORLD, "test.out",
                  MPI_MODE_CREATE|MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    if (rank == 0)
       MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

Comments on Example

- File Open is collective over the communicator
  - Will be used to support collective I/O, which we will see is important for performance
  - Modes similar to Unix open
  - MPI_Info provides additional hints for performance
- File Write is independent (hence the test on rank)
  - Many important variations covered in later slides
- File close is collective; similar in style to MPI_Comm_free

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

Ways to Access a Shared File

- **MPI_File_seek**
- **MPI_File_read**
- **MPI_File_write**
- **MPI_File_read_at**
- **MPI_File_write_at**
- **MPI_File_read_shared**
- **MPI_File_write_shared**

like Unix I/O

combine seek and I/O for thread safety

use shared file pointer

Using Explicit Offsets

```
#include “mpi.h”
MPI_Status   status;
MPI_File     fh;
MPI_Offset   offset;

MPI_File_open(MPI_COMM_WORLD, “/pfs/datafile”,
                    MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;
MPI_File_read_at(fh, offset, buf, nints,  MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
printf(“process %d read %d ints\n”, rank, count);

MPI_File_close(&fh);
```

Why Use Independent I/O?

- Sometimes the synchronization of collective calls is not natural
- Sometimes the overhead of collective calls outweighs their benefits
  - ♦ Example: very small I/O during header reads

Noncontiguous I/O in File

- Each process describes the part of the file for which it is responsible
  - ♦ This is the "file view"
  - ♦ Described in MPI with an offset (useful for headers) and an MPI_Datatype
- Only the part of the file described by the file view is visible to the process; reads and writes access these locations
- This provides an efficient way to perform *noncontiguous accesses*

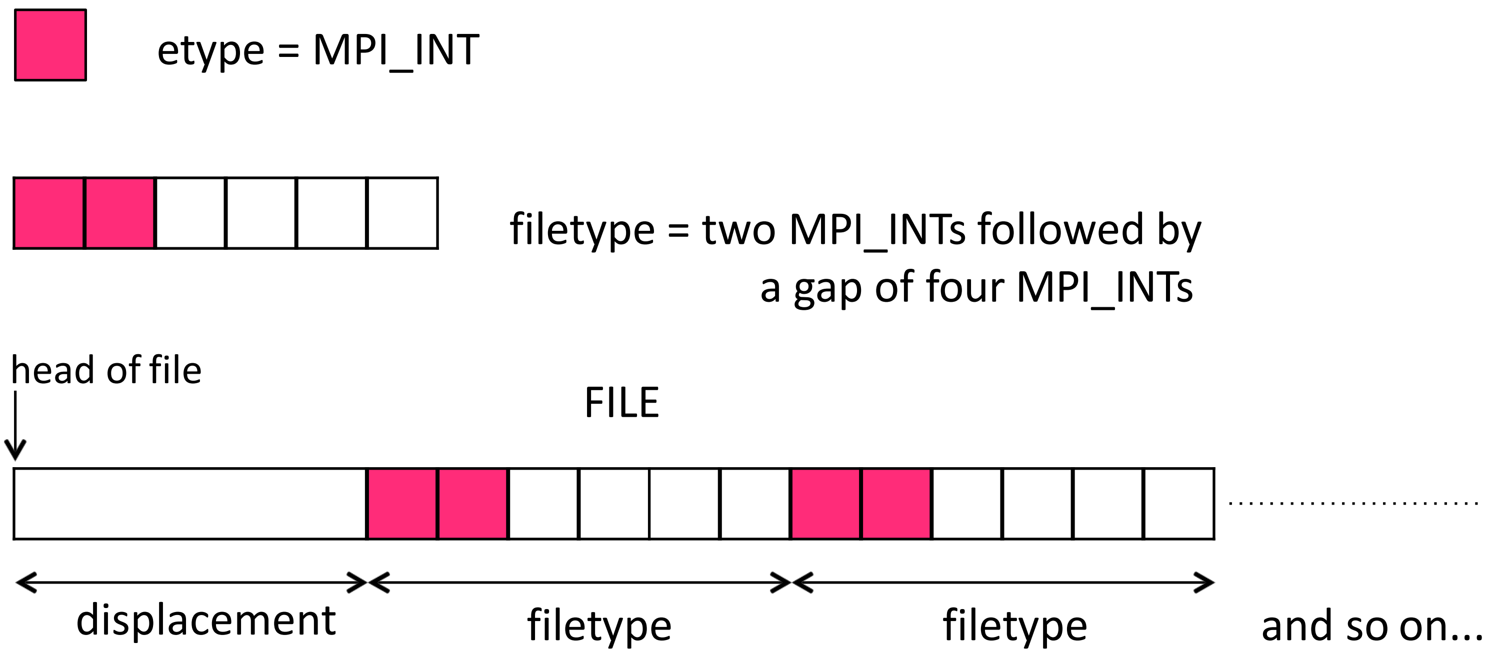
Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory **and** file within a single function call by using derived datatypes
  - POSIX only supports non-contiguous in file
- Allows implementation to optimize the access
- Collective I/O combined with noncontiguous accesses yields the highest performance

File Views

- MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)
  - 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
    - e.g., to skip a file header
  - *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
  - datarep may be ‘NATIVE’ for machine dependent binary.

A SimpleNoncontiguous File View Example



Noncontiguous File View Code

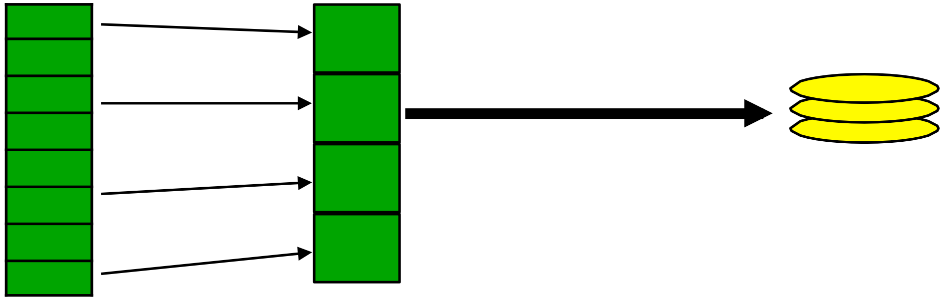
```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

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 and MPI

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - Framework for I/O optimizations at the MPI-IO layer
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
  - Requests from different processes may be merged together
  - Particularly effective when the accesses of different processes are noncontiguous and interleaved
  - Small individual requests
  - Large collective



Collective I/O Functions

- **MPI_File_write_at_all**, etc.
  - **_all** indicates that all processes in the group specified by the communicator passed to **MPI_File_open** will call this function
  - **_at** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate “seek” call
- Each process specifies only its own access information — the argument list is the same as for the non-collective functions

The Other Collective I/O Calls

- **MPI_File_seek**
- **MPI_File_read_all**
- **MPI_File_write_all**

like Unix I/O

- **MPI_File_read_at_all**
- **MPI_File_write_at_all**

combine seek and I/O for thread safety

- **MPI_File_read_ordered**
- **MPI_File_write_ordered**

use shared file pointer

Collective non-contiguous MPI-IO examples

```
#define “mpi.h”
#define FILESIZE 1048576
#define INTS_PER_BLK 16

int main(int argc, char **argv){
  int *buf, rank, nprocs, nints, bufsize;
  MPI_File fh;
  MPI_Datatype filetype;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  bufsize = FILESIZE/nprocs;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);

  MPI_File_open(MPI_COMM_WORLD, “filename”, MPI_MODE_RD_ONLY, MPI_INFO_NULL, &fh);
  MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK*nprocs, MPI_INT, &filetype);
  MPI_Type_commit(&filetype);
  MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT, filetype, “native”, MPI_INFO_NULL);
  MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);

  MPI_Type_free(&filetype);
  free(buf)
  MPI_Finalize();
  return(0);
}
```

Using the Right MPI-IO Function

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- We classify the different ways of expressing I/ O access patterns in MPI-IO into four levels: level 0 – level 3
- We demonstrate how the user’s choice of level affects performance

Further Information

- Lustre Operations Manual
  - http://dlc.sun.com/pdf/821-0035-11/821-0035-11.pdf
- GPFS: Concepts, Planning, and Installation Guide
  - http://publib.boulder.ibm.com/epubs/pdf/a7604133.pdf
- HDF5 User Guide
  - http://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UG_r183. pdf
- The NetCDF Tutorial
  - http://www.unidata.ucar.edu/software/netcdf/docs/netcdf- tutorial.pdf

Further Information MPI-IO

– Tzu-Hsien Wu, Jerry Chi-Yuan Chou, Shyng Hao, Bin Dong, Scott Klasky, Kesheng Wu: Optimizing the query performance of block index through data analysis and I/O modeling. 12:1-12:10

– Sarp Oral, Sudharshan S. Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross G. Miller, Dustin Leverman, Scott Atchley, Verónica G. Vergara Larrea: End-to-end I/O portfolio for the summit supercomputing ecosystem. 63:1-63:14

– Tirthak Patel, Suren Byna, Glenn K. Lockwood, Devesh Tiwari: Revisiting I/O behavior in large-scale storage systems: the expected and the unexpected. 65:1-65:13