

Advanced Message-Passing Programming

Overview of Parallel IO

ARCHER Training Courses

Sponsors



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Overview

- Lecture will cover
 - Why is IO difficult
 - Why is parallel IO even worse
 - Straightforward solutions in parallel
 - What is parallel IO trying to achieve?
 - Files as arrays
 - MPI-IO and derived data types

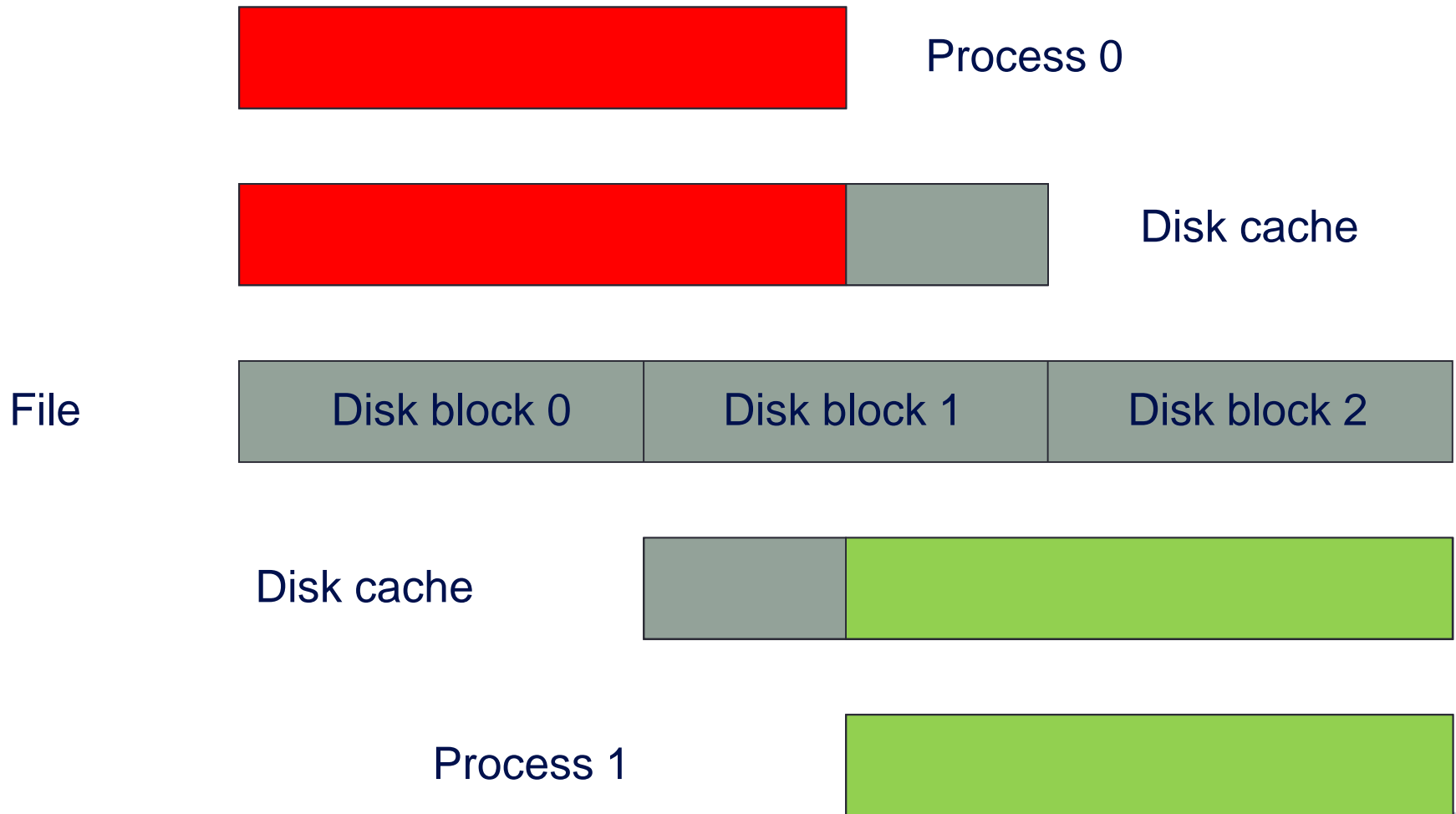
Why is IO hard?

- Breaks out of the nice process/memory model
 - data in memory has to physically appear on an external device
- Files are very restrictive
 - linear access probably implies remapping of program data
 - just a string of bytes with no memory of their meaning
- Many, many system-specific options to IO calls
- Different formats
 - text, binary, big/little endian, Fortran unformatted, ...
- Disk systems are very complicated
 - RAID disks, many layers of caching on disk, in memory, ...
- IO is the HPC equivalent of printing!

Why is Parallel IO Harder?

- Cannot have multiple processes writing a single file
 - Unix generally cannot cope with this
 - data cached in units of disk blocks (eg 4K) and is *not coherent*
 - not even sufficient to have processes writing to distinct parts of file
- Even reading can be difficult
 - 1024 processes opening a file can overload the filesystem (fs)
- Data is distributed across different processes
 - processes do not in general own contiguous chunks of the file
 - cannot easily do linear writes
 - local data may have halos to be stripped off

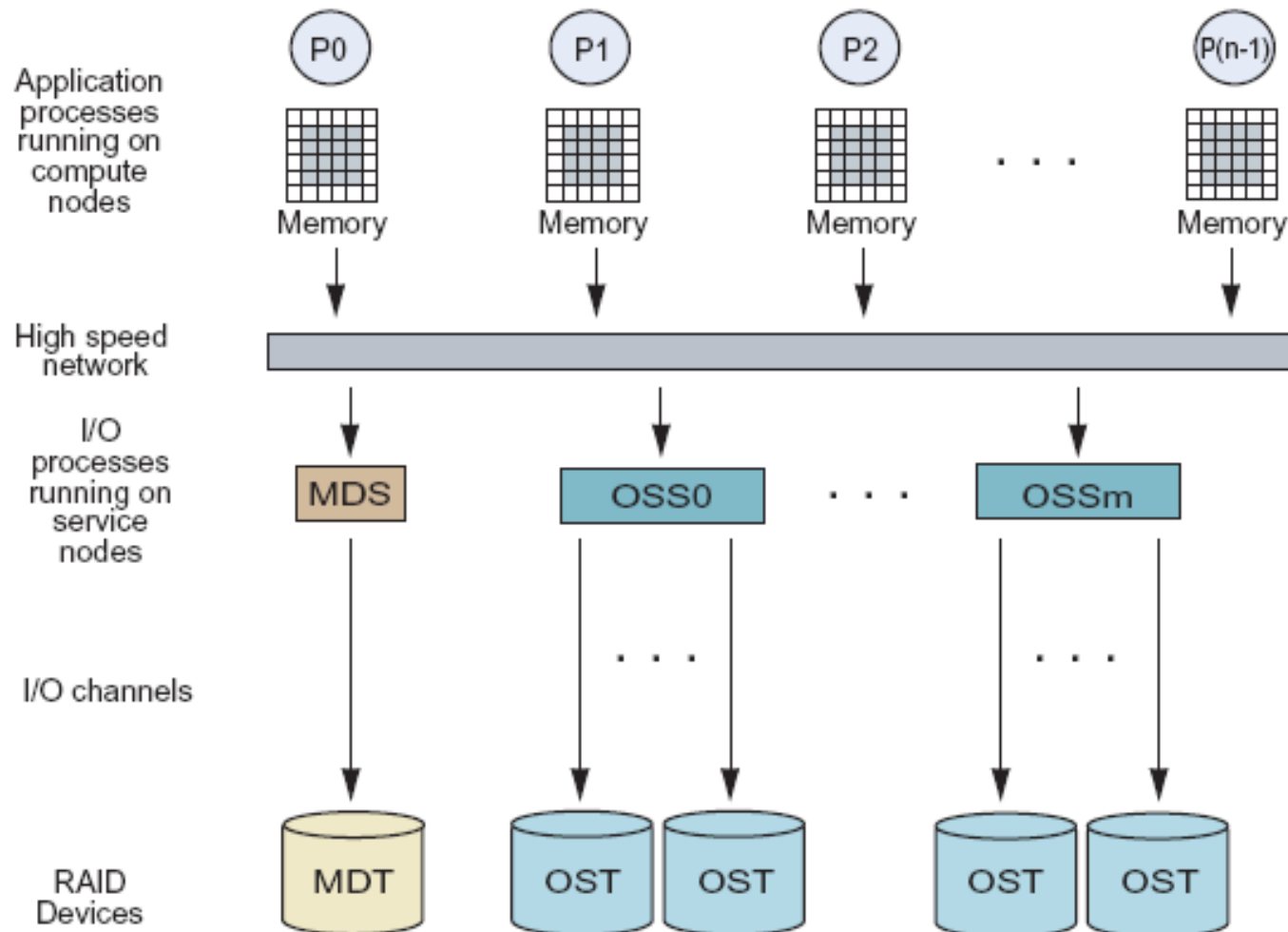
Simultaneous Access to Files



Parallel File Systems

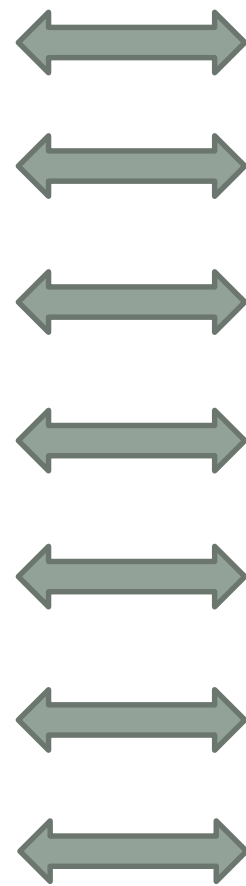
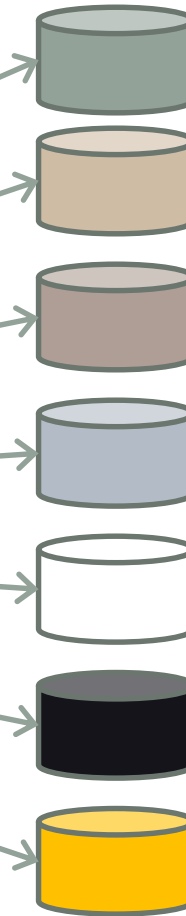
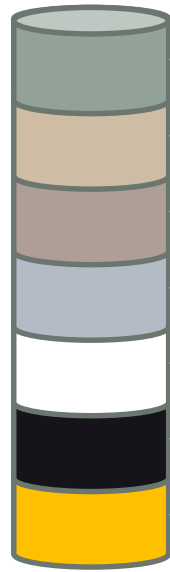
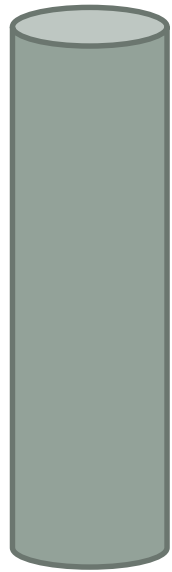
- Parallel computer
 - constructed of many processors
 - each processor not particularly fast
 - performance comes from using many processors at once
 - requires distribution of data and calculation across processors
- Parallel file systems
 - constructed from many standard disk
 - performance comes from reading / writing to many disks
 - requires many *clients* to read / write to different disks at once
 - data from a single file must be *striped* across many disks
- Must appear as a single file system to user
 - typically have a single *MetaData* Server (MDS)
 - can become a bottleneck for performance

Parallel File Systems: Lustre



Lustre data striping

Lustre's performance comes from striping files over multiple OSTs



Single logical user file e.g. `/work/y02/y02/ted`

OS/file-system automatically divides the file into stripes

Stripes read/written to/from their assigned OST

Parallel File Systems

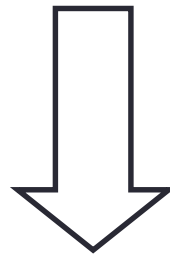
- Allow multiple IO processes to access same file
 - increases bandwidth
- Typically optimised for bandwidth
 - not for latency
 - e.g. reading/writing small amounts of data is very inefficient
- Very difficult for general user to configure and use
 - need some kind of higher level abstraction
 - allow user to focus on data layout across user processes
 - don't want to worry about how file is split across IO servers

4x4 array on 2x2 Process Grid

Parallel Data

2	4	2	4
1	3	1	3
2	4	2	4
1	3	1	3

File



Shared Memory

- Easy to solve in shared memory
 - imagine a shared array called **x**

```
begin serial region
    open the file
    write x to the file
    close the file
end serial region
```

- Simple as every thread can access shared data
 - may not be efficient but it works
- But what about message-passing?

Message Passing: Naive Solutions

- Master IO
 - send all data to/from master and write/read a single file
 - quickly run out of memory on the master
 - or have to write in many small chunks
 - does not benefit from a parallel fs that supports multiple write streams
- Separate files
 - each process writes to a local fs and user copies back to home
 - or each process opens a unique file (dataXXX.dat) on shared fs
- Major problem with separate files is reassembling data
 - file contents dependent on number of CPUs and decomposition
 - pre / post-processing steps needed to change number of processes
 - but at least this approach means that reads and writes are in parallel
- But may overload filesystem for many processes
 - e.g. MDS cannot keep up with requests

2x2 to 1x4 Redistribution

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



11	12	15	16
----	----	----	----

data4.dat

9	10	13	14
---	----	----	----

data3.dat

3	4	7	8
---	---	---	---

data2.dat

1	2	5	6
---	---	---	---

data1.dat



4	8	12	16
---	---	----	----

newdata4.dat

3	7	11	15
---	---	----	----

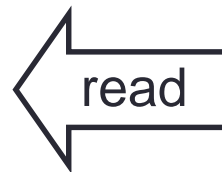
newdata3.dat

2	6	10	14
---	---	----	----

newdata2.dat

1	5	9	13
---	---	---	----

newdata1.dat



What do we Need?

- A way to do parallel IO properly
 - where the IO system deals with all the system specifics
- Want a single file format
 - We already have one: the serial format
- All files should have same format as a serial file
 - entries stored according to position in global array
 - not dependent on which process owns them
 - order should always be 1, 2, 3, 4,, 15, 16

Information on Machine

- What does the IO system need to know about the parallel machine?
 - all the system-specific fs details
 - block sizes, number of IO servers, etc.
- All this detail should be hidden from the user
 - but the user may still wish to pass system-specific options ...

ARCHER's Cray Sonexion Storage

MMU: *Metadata Management Unit*



1

Lustre MetaData Server

- Contains server hardware and storage

SSU: *Scalable Storage Unit*



2

2 x OSSs and 8 x OSTs (Object Storage Targets)

- Contains Storage controller, Lustre server, disk controller and RAID engine
- Each unit is 2 OSSs each with 4 OSTs of 10 (8+2) disks in a RAID6 array

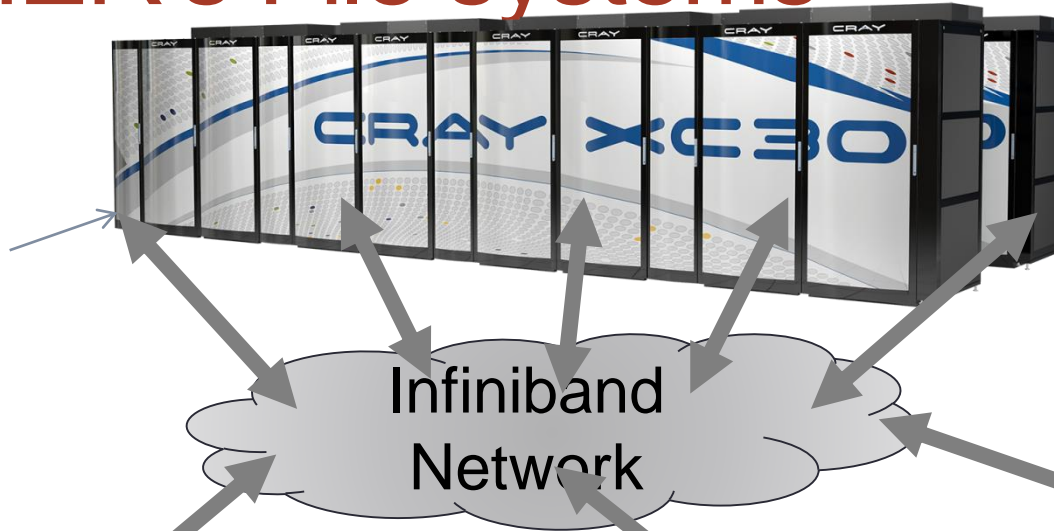


3

Multiple SSUs are combined to form storage racks

ARCHER's File systems

Connected to
the Cray
XC30 via
LNET router
service
nodes.



/fs2

6 SSUs
12 OSSs
48 OSTs
480 HDDs
4TB per
HDD
1.4 PB Total



/fs3

6 SSUs
12 OSSs
48 OSTs
480 HDDs
4TB per
HDD
1.4 PB Total



/fs4

7 SSUs
14 OSSs
56 OSTs
560 HDDs
4TB per
HDD
1.6 PB Total



Information on Data Layout

- What does the IO system need to know about the data?
 - how the local arrays should be stitched together to form the file
- But ...
 - mapping from local data to the global file is only in the mind of the programmer!
 - the program does not know that we imagine the processes to be arranged in a 2D grid
- How do we describe data layout to the IO system
 - without introducing a whole new concept to MPI?
 - cartesian topologies are not sufficient
 - do not distinguish between block and block-cyclic decompositions

Programmer View vs Machine View



4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



Process 4



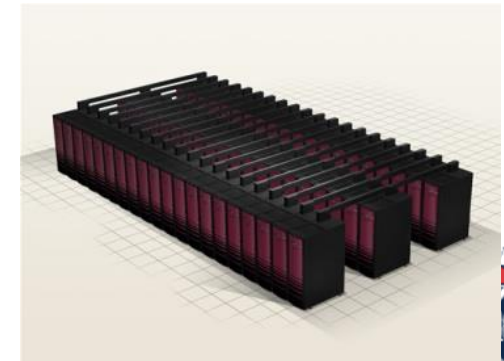
Process 2



Process 3



Process 1



Files vs Arrays

- Think of the file as a large array
 - forget that IO actually goes to disk
 - imagine we are recreating a single large array on a master process
- The IO system must create this array and save to disk
 - without running out of memory
 - never actually creating the entire array
 - ie without doing naive master IO
 - and by doing a small number of large IO operations
 - merge data to write large contiguous sections at a time
 - utilising any parallel features
 - doing multiple simultaneous writes if there are multiple IO nodes
 - managing any coherency issues re file blocks

MPI-IO Approach

- MPI-IO is part of the MPI standard
 - <http://www.mpi-forum.org/docs/docs.html>
- Each process needs to describe what subsection of the global array it holds
 - it is entirely up to the programmer to ensure that these do not overlap for write operations!
- Programmer needs to be able to pass system-specific information
 - pass an `info` object to all calls

Data Sections

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

- Describe 2x2 subsection of 4x4 array
- Using standard MPI derived datatypes
- A number of different ways to do this
 - we will cover three methods in the course

Summary

- Parallel IO is difficult
 - in theory and in practice
- MPI-IO provides a higher-level abstraction
 - user describes global data layout using derived datatypes
 - MPI-IO hides all the system specific fs details ...
 - ... but (hopefully) takes advantage of them for performance
- More flexible formats like NetCDF and HDF5 exist
 - they gain performance by layering on top of MPI-IO
- User requires a good understanding of derived datatypes
 - see next lecture