HELP US ALL STAY HEALTHY

SIX SIMPLE TIPS

Maintain 1.5 metres distance between yourself and others where possible

Cough and sneeze into your elbow or a tissue (not your hands)

Avoid shaking hands

Put used tissues in the bin

Wash hands with soap and warm water or use an alcohol-based hand sanitiser after you cough or sneeze

Do not touch your face

IF YOU ARE UNWELL AND WORRIED ABOUT COVID-19:

- Call the National Coronavirus Helpline: 1800 020 080
- Call your usual GP for advice
- Call the UWA Medical Centre for advice: 6488 2118

UWA FAQs:
uwa.edu.au/coronavirus

Report COVID-19 hazards and suspected/confirmed cases via RiskWare:
uwa.edu.au/riskware

THE UNIVERSITY OF WESTERN AUSTRALIA

DCS:507757521

THE UNIVERSITY OF WESTERN AUSTRALIA

High-Performance Computing

Lecture 8 MPI Parallel I/O and Process Topologies

CITS5507

Zeyi Wen

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are based on many online sources.

Outline

- **Parallel File I/O**
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ Collective I/O
  - ✓ I/O access pattern
- Process Topologies
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

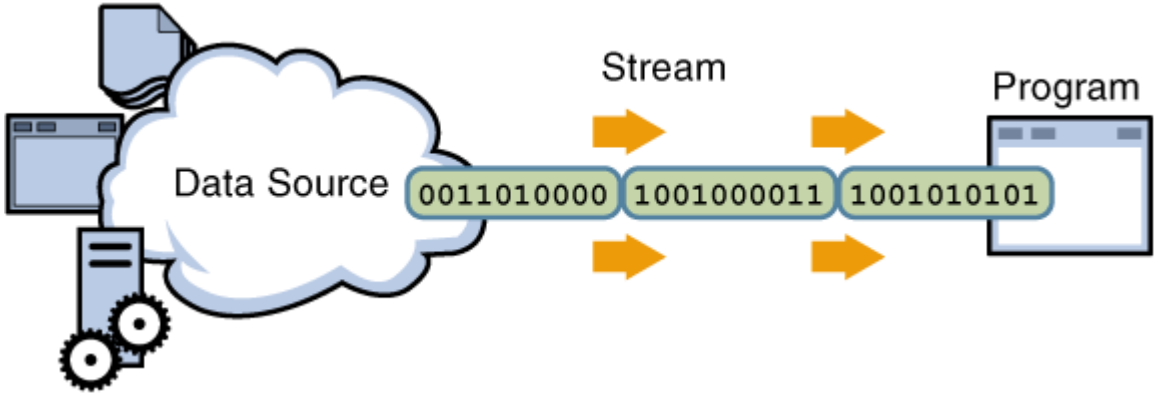
Introduction

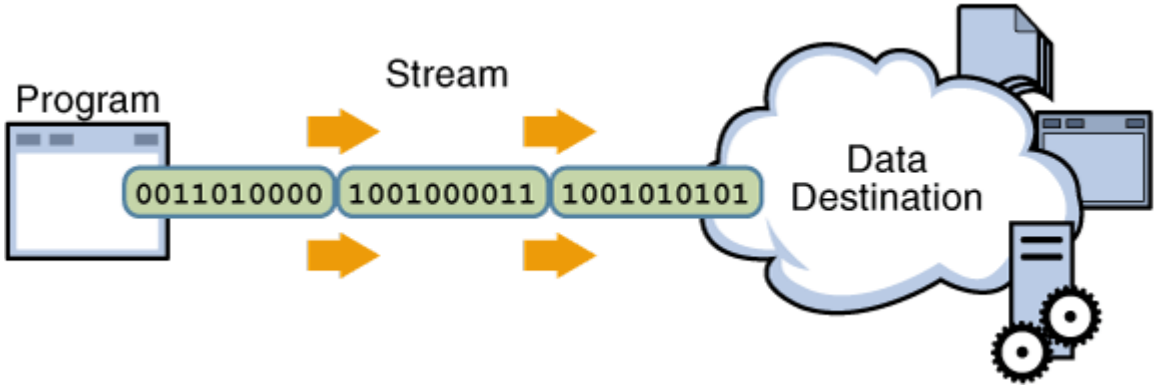
THE UNIVERSITY OF WESTERN AUSTRALIA

- We often use supercomputers to handle large datasets
- Reading in data takes a linear amount of time
- Reading/Writing to files can be parallelised
- Most supercomputers have a bespoke file-system designed for high performance and concurrent access
  - ✓ Your program should use this if possible
- Parallel I/O means concurrent reading/writing to the same file

File I/O

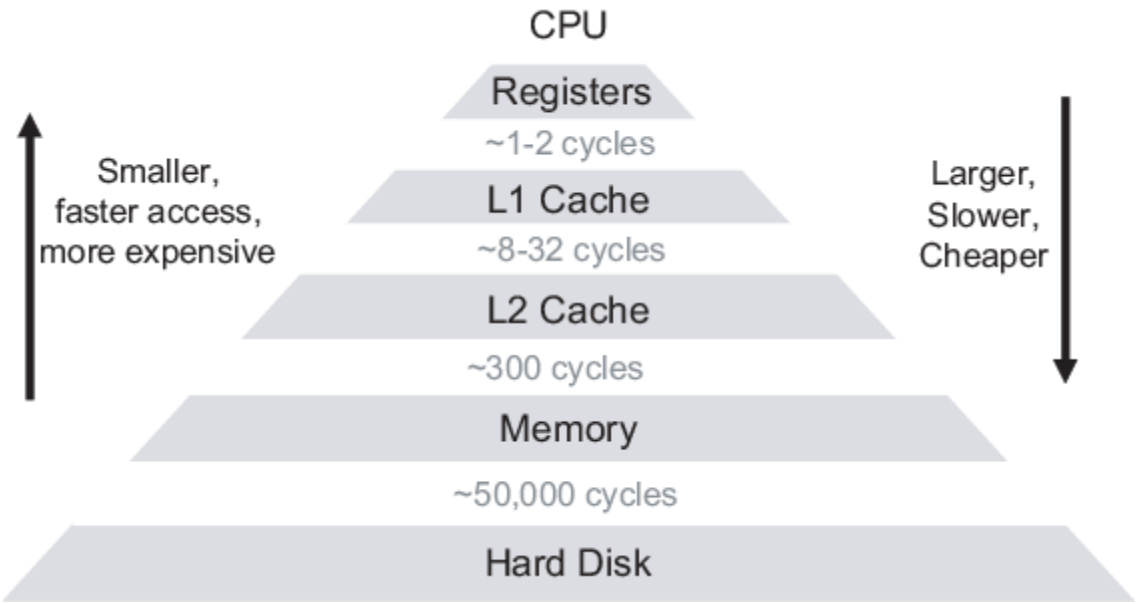
Data sources/destination may be files.



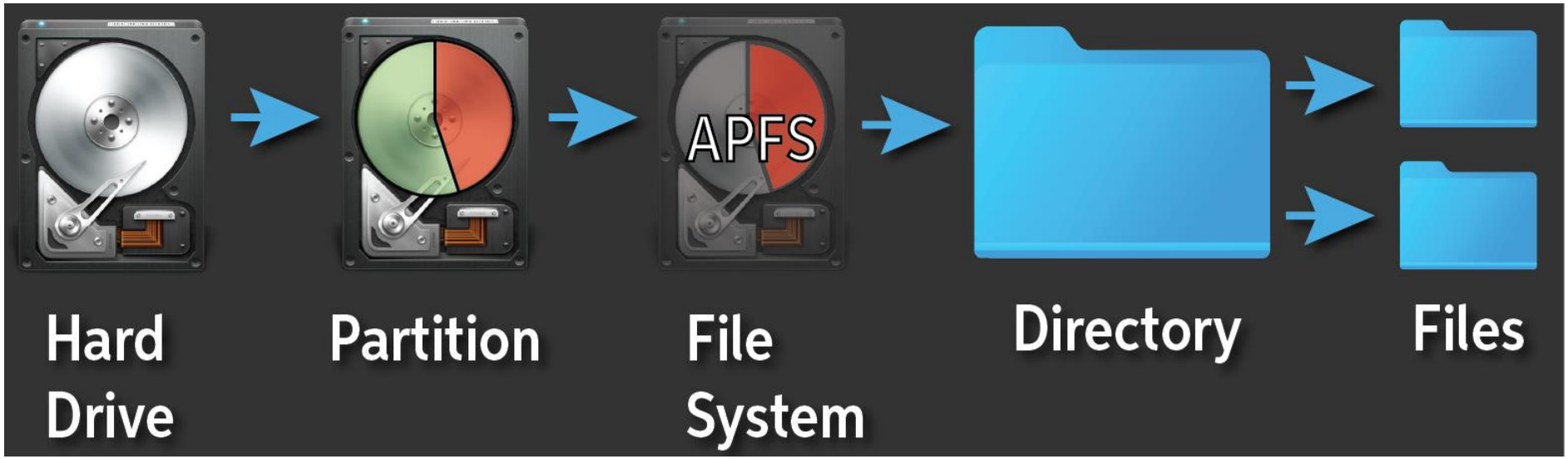


The Memory Hierarchy (Lecture 1)

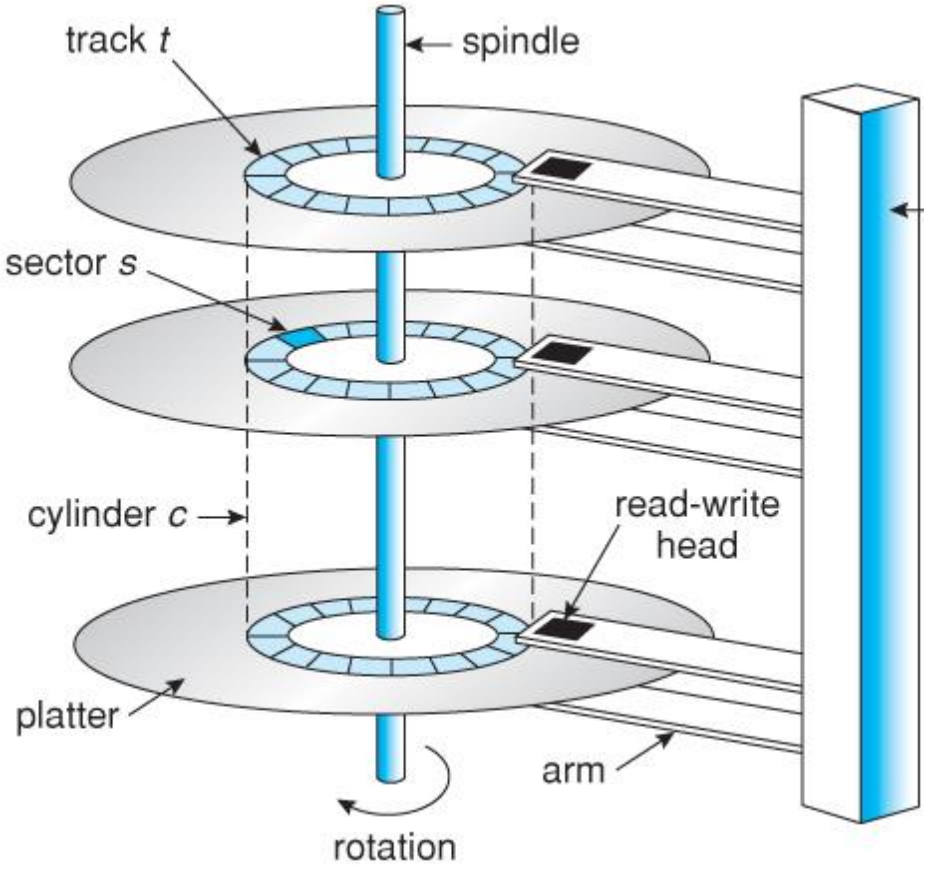
- Why not build computers entirely with fast memories like the cache?
  - ✓ The cost will be high, consumer hardware will not be competitively priced.
  - ✓ The cost is in manufacturing hardware, caches are usually "on-chip" with the processor.



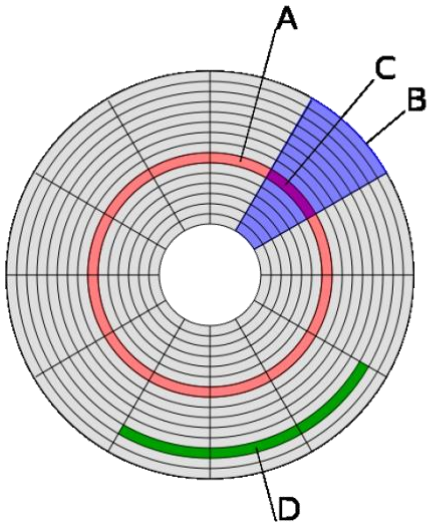
File System Workflow



Hard Drive Overview



A hard dirve



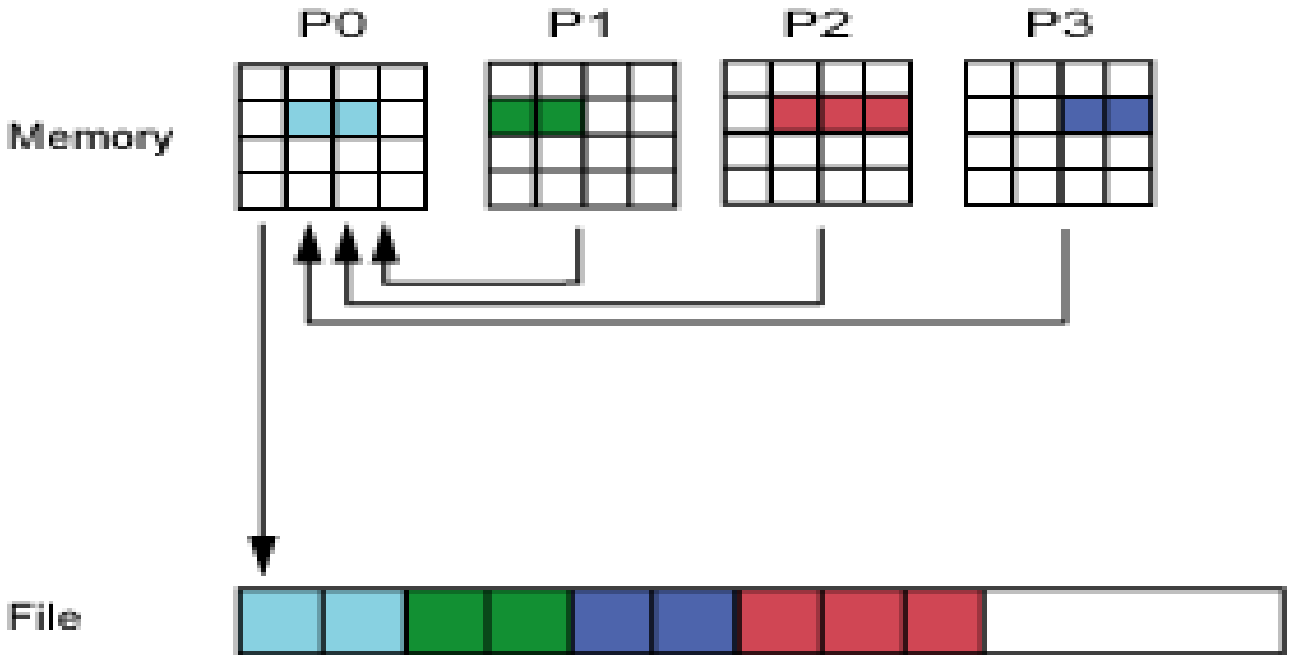
Hard Drive Structure:

A = track
B = sector
C = sector of a track
D = cluster

Single disk

Non-Parallel I/O

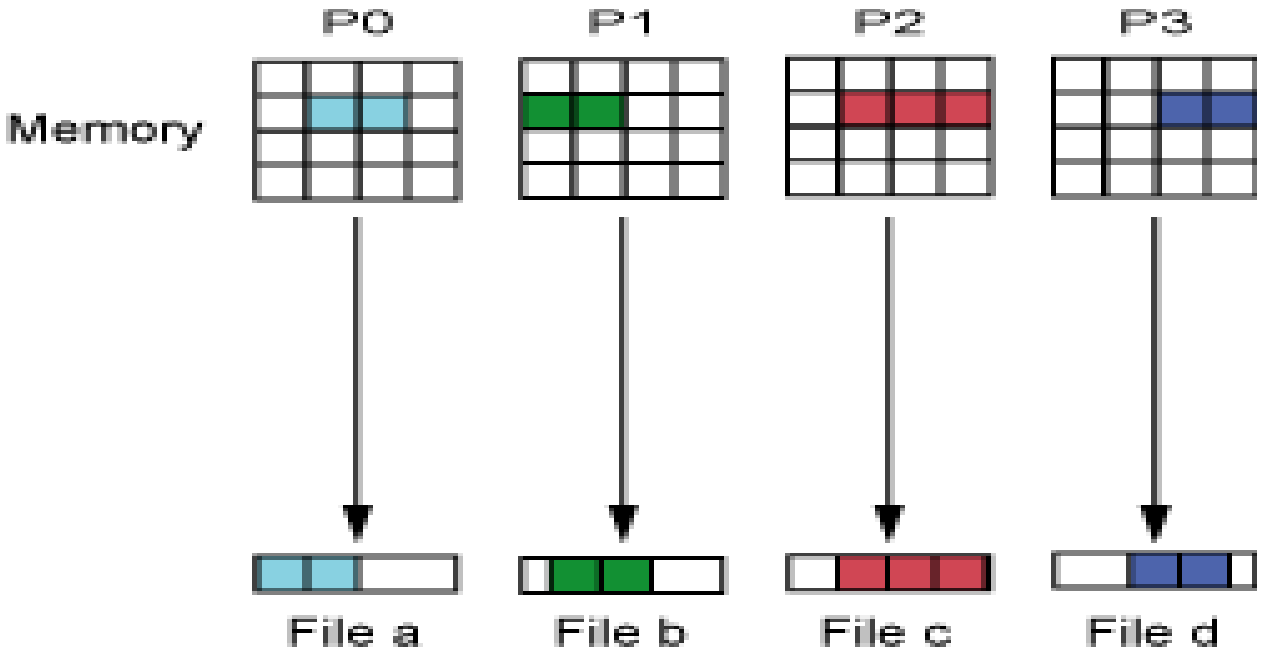
- All processes participating in I/O send write requests to a single process, which writes to the shared file **at once**.
  - ✓ **P1-P3** sends a write request to **P0** which writes the data of **P0-P3** to the shared file at once.
- Cons: Non-parallel; performance **worse** than sequential



Independent Parallel I/O

- Each process writes to a **separate** file

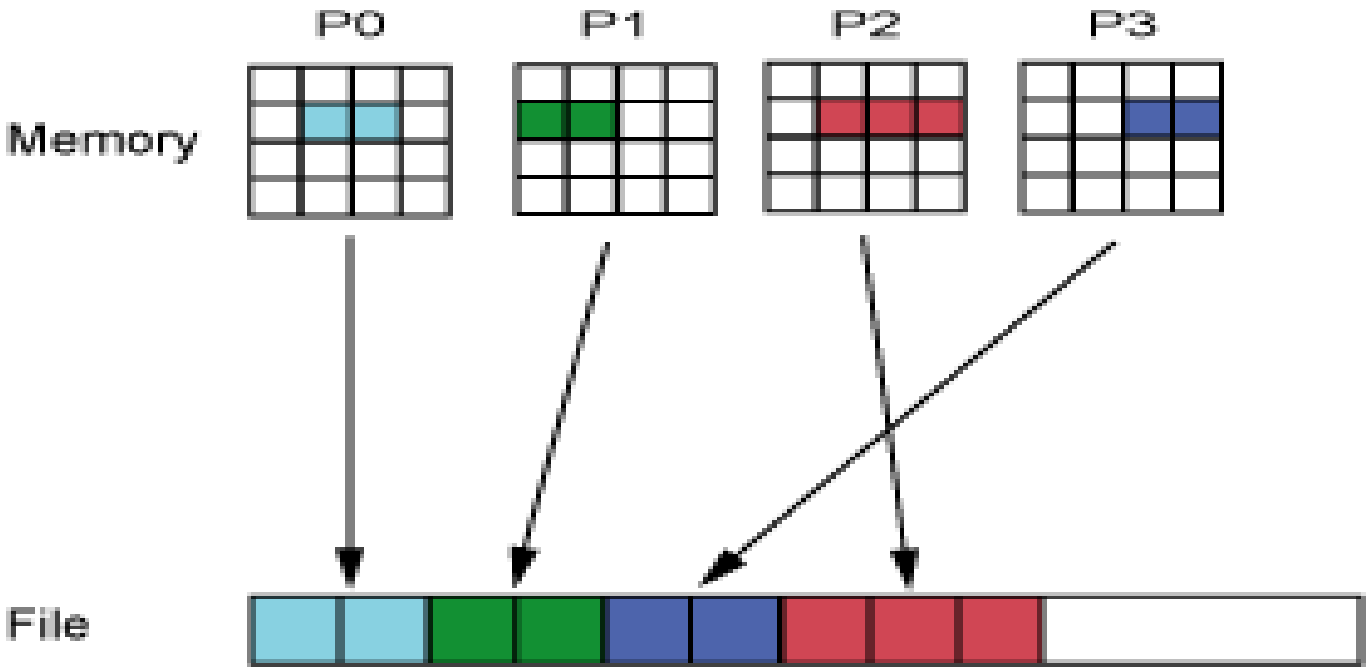
- Pro : Good parallelism
- Con: Lots of small files to manage



Cooperative Parallel I/O

- All processes write a **shared file** at the same time

- Pro: lots of parallelism
- Con: can only be expressed in MPI



Why use MPI for this?

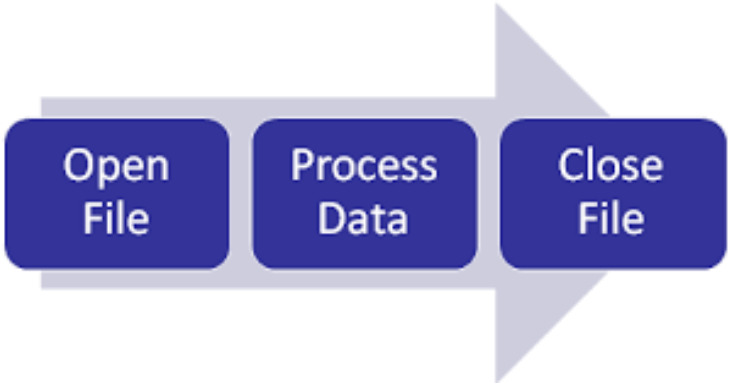
THE UNIVERSITY OF WESTERN AUSTRALIA

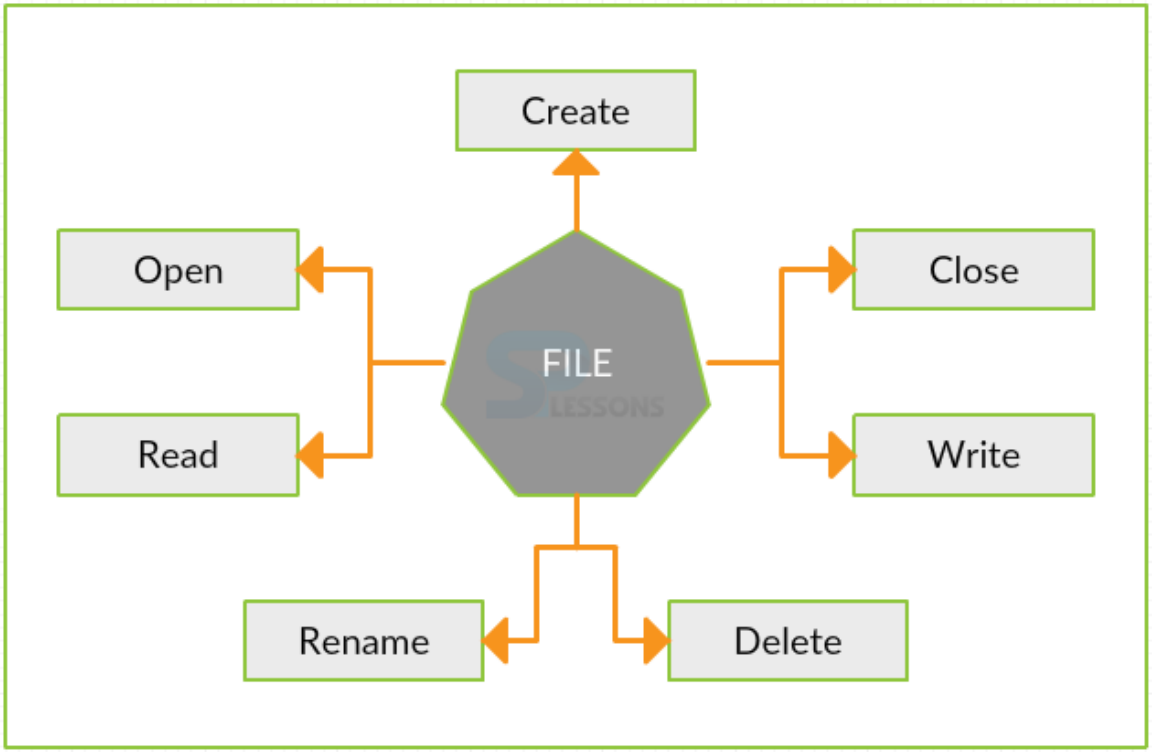
- Reading/Writing is similar to **sending**/**receiving**

- Allows developers to leverage MPI capabilities
  - ✓ Collective communicators
  - ✓ User-defined datatypes to describe file elements
  - ✓ Communicators allow separation from application messages and file messages
  - ✓ Non-blocking operations

- This lecture covers
  - ✓ **Independent I/O**
  - ✓ **Cooperative I/O**

Accessing Files and File Operations





File I/O – Rationale

- The idea is to have each process describe its part of the file
  - ✓ Provides a ‘view’ of the file
  - ✓ Achieves with ‘**offsets**’
  - ✓ Allows the MPI implementation to handle **non-contiguous** access and **shared access**

- Collective I/O combined with non-contiguous access is the key to the highest performance

Two Types of I/O

- MPI I/O supports **two** types of I/O
- Independent
  - ✓ Each process handles its **own** I/O
  - ✓ Supports derived datatypes (different to POSIX)

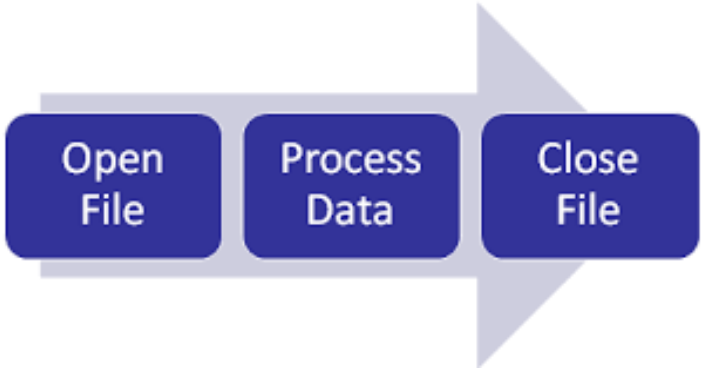
- Collective
  - ✓ I/O calls must be made by **all** processes
  - ✓ Used “shared file, all write”
  - ✓ Optimised by the MPI library

POSIX stands for Portable Operating System Interface, and is an IEEE standard designed to facilitate application portability.

Outline

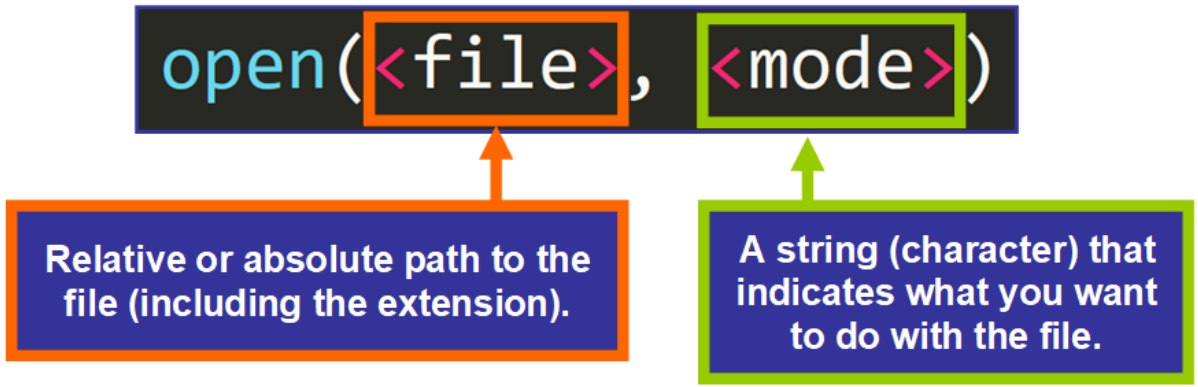
- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ **Independent I/O**
  - ✓ Collective I/O
  - ✓ I/O access pattern
- Process Topologies
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

Independent I/O



- Just like C/C++ I/O you need to
  - ✓ Open the file
  - ✓ Read/Write data from/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**



MPI I/O Example

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include <mpi.h>
#include <stdio.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;
}
```

MPI I/O Example (Cont'd): Comments

THE UNIVERSITY OF WESTERN AUSTRALIA

- File Open is collective over the communicator
  - ✓ Can be used to support **collective I/O**, which is important for performance
  - ✓ Modes similar to Unix file open
  - ✓ **MPI_Info** provides additional hints for performance
- File Write is **independent** (hence choose the rank with **if**)
  - ✓ Many important variations covered in later slides
- File close is collective; similar in style to **MPI_Comm_free**

Independent I/O

- Also like C/C++ you need to provide file flags
  - ✓ **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** (reading and writing)
  - ✓ **MPI_MODE_CREATE** also needed if the file may not exist
  - ✓ Can pass multiple flags with bitwise-or '|'

- When to use Independent I/O
  - ✓ Synchronisation of collective calls
  - ✓ Overhead of collective calls is too high
    - **E.g.** reading a very small proportion of a file

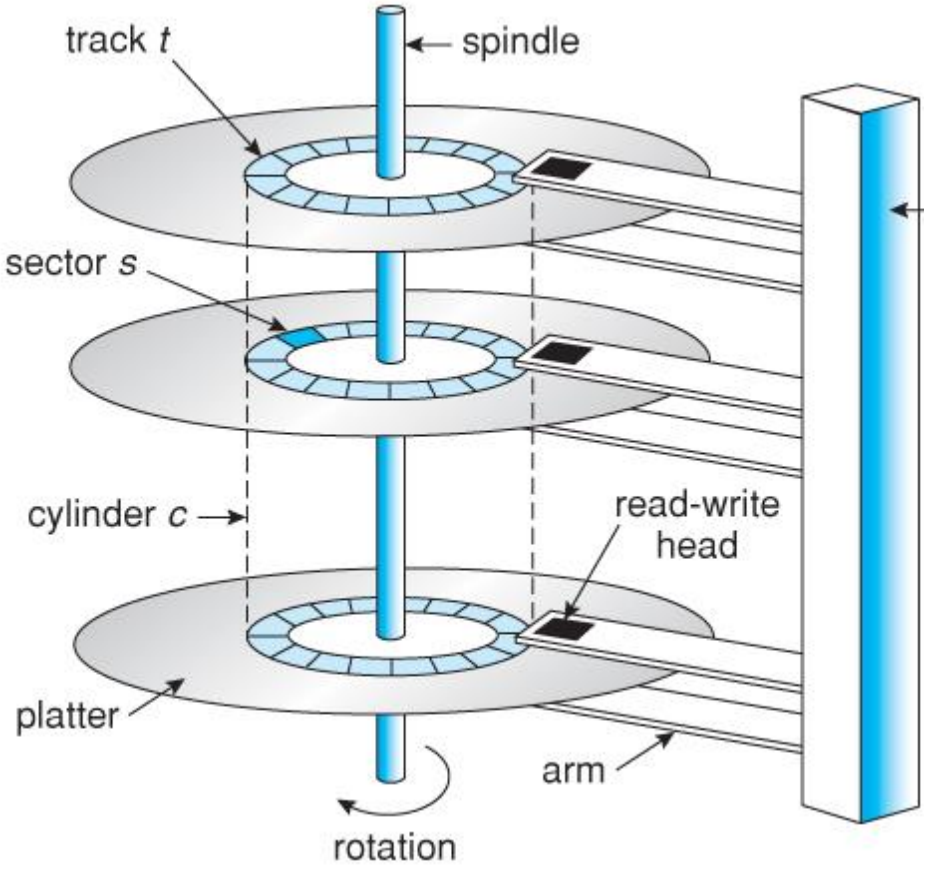
Accessing a Shared File

- MPI_File_open(comm, filename, flags, info, &file_handle)
- MPI_File_close(&file_handle)

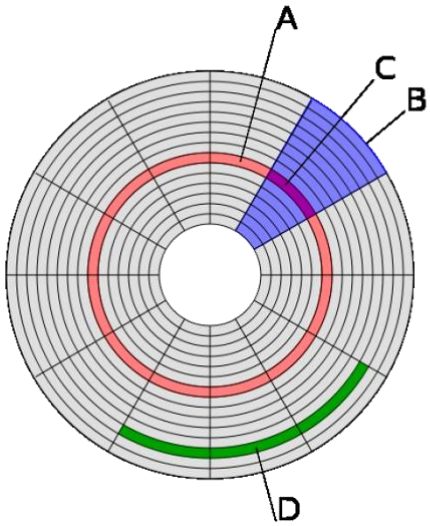
- MPI_File_seek(File_handle, offset, whence)
  - ✓ Updates the file pointer to 'whence' which can be one of
    - MPI_SEEK_SET – Set to the offset value
    - MPI_SEEK_CUR – Set to the current position + offset
    - MPI_SEEK_END – Set to the end of the file + offset

- MPI_File_read(file_handle, buffer, num_read, datatype, status)
- MPI_File_write(file_handle, size, datatype, status)

Hard Drive Overview (MPI_SEEK)



A hard dirve



Hard Drive Structure:

A = track
B = sector
C = sector of a track
D = cluster

Single disk

Accessing a Shared File (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

- MPI_File_read_at(file_handle, offset, buf, num_items, datatype, status)
- MPI_File_write_at(file_handle, offset, buf, count, datatype)

Example: Writing a File

```
#include<stdio.h>
#include "mpi.h"
#define N 16
int main(int argc, char **argv){
  int rank, num_of_process;
  MPI_File fhw;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_of_process);
  int buf[N];
  for (int i=0;i<N;i++)
     buf[i] = i;
  int ave_num_of_ints = N / num_of_process;
  int offset = rank * ave_num_of_ints;
  MPI_File_open(MPI_COMM_WORLD, "dfile",
                MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);
  printf("\nRank: %d, Offset: %d\n", rank, offset);
  MPI_File_write_at(fhw, offset * sizeof(int), buf+offset, ave_num_of_ints,
MPI_INT, &status);
  MPI_File_close(&fhw);
  MPI_Finalize();
  return 0;
}
```

using bytes

Note: try even number of processes (e.g. mpiexec –n 2 my_program)

Example: Reading a File (1)

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include <mpi.h>
#include <stdio.h>
#define N 16
int main(int argc, char **argv){
  int rank, num_of_process;
  MPI_File fh; //Declaring a File Pointer
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_of_process);
  int ave_num_of_ints = N / num_of_process;
  int buf[ave_num_of_ints];
  int bufsize_byte = ave_num_of_ints * sizeof(int);

  MPI_File_open(MPI_COMM_WORLD,"dfile", MPI_MODE_RDONLY, MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize_byte, MPI_SEEK_SET); //File seek
  MPI_File_read(fh, buf, ave_num_of_ints, MPI_INT, &status); //File read
  printf("\nrank: %d, buf[%d]: %d", rank, rank*ave_num_of_ints, buf[0]);
  MPI_File_close(&fh); //Closing a File
  MPI_Finalize();
  return 0;
}
```

Example: Reading a File (2)

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include <mpi.h>
#include <stdio.h>
#define N 16
int main(int argc, char **argv){
  int rank, num_of_process;
  MPI_File fh; //Declaring a File Pointer
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_of_process);
  int ave_num_of_ints = N / num_of_process;
  int buf[ave_num_of_ints];
  int bufsize_byte = ave_num_of_ints * sizeof(int);

  MPI_File_open(MPI_COMM_WORLD,"dfile", MPI_MODE_RDONLY, MPI_INFO_NULL,&fh);
//Combining File Seek & Read in One Step for Thread Safety
  MPI_File_read_at(fh, rank*bufsize_byte, buf, ave_num_of_ints, MPI_INT,
&status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*ave_num_of_ints, buf[0]);
  MPI_File_close(&fh); //Closing a File
  MPI_Finalize();
  return 0;
}
```

Non-contiguous 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 non-contiguous accesses

Non-contiguous Accesses

THE UNIVERSITY OF WESTERN AUSTRALIA

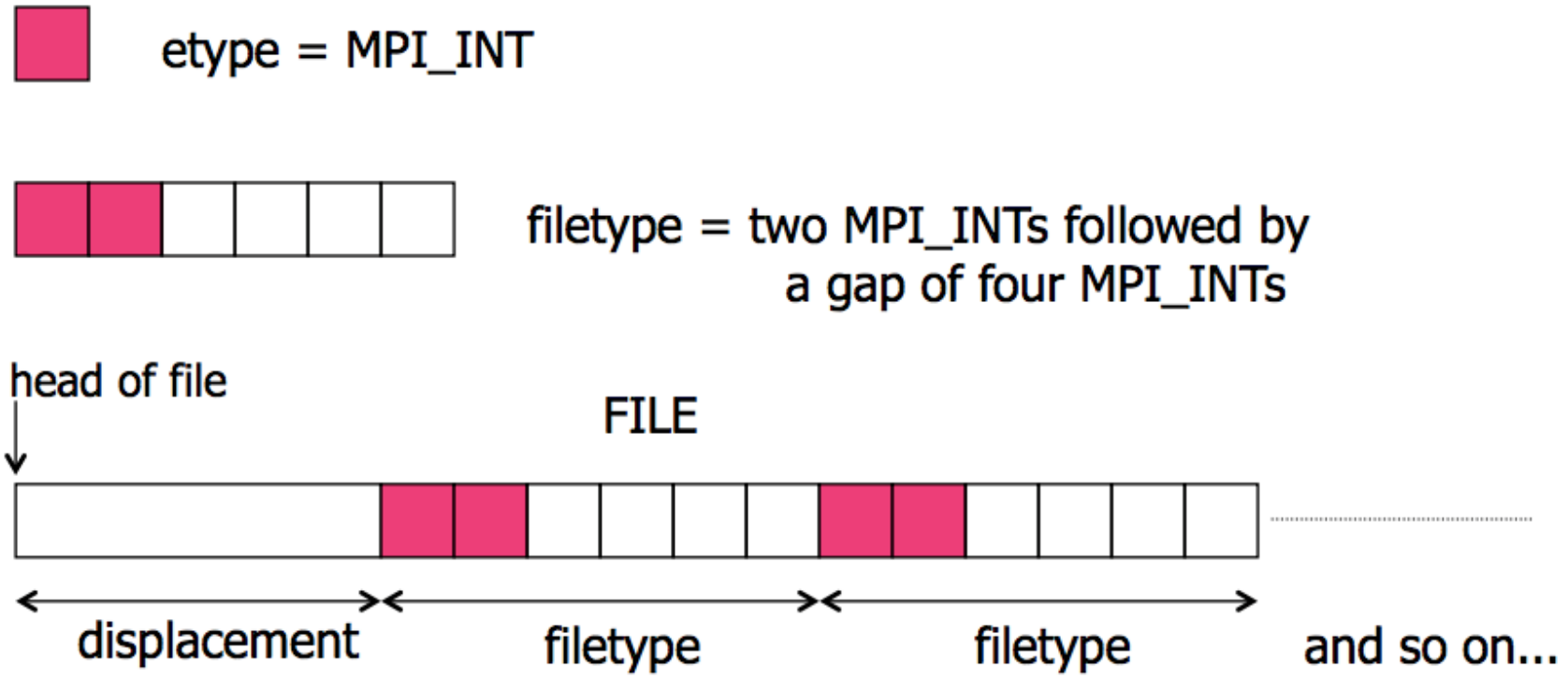
- Common in parallel applications
  - ✓ Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify non-contiguous accesses in memory and file within a single function call by using derived datatypes
- Allows MPI implementations to optimise the access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

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
  - ✓ **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

A Non-contiguous Example



A Non-contiguous Example (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_File fh;
    int buf[1000], rank;
    MPI_Init(0,0);
    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);
    MPI_Finalize();
}
```

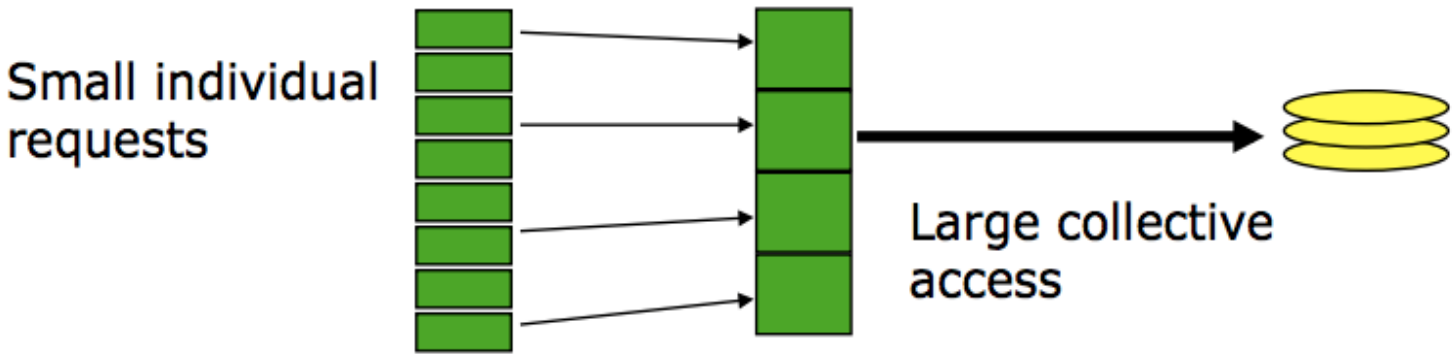
Outline

THE UNIVERSITY OF WESTERN AUSTRALIA

- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ **Collective I/O**
  - ✓ I/O access pattern
- Process Topologies
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

Collective I/O

- Provides massive speedup
- Like communication, all processes (in a communicator) need to call the same function
- Allows implementation to make many optimisations
- Basic idea:
  - ✓ Build large blocks of data, so reads/writes to the I/O system can be large
  - ✓ Merging of requests from different processes
  - ✓ Particularly effective with very non-contiguous but overlapping requests are interleaved



Collective I/O Functions

THE UNIVERSITY OF WESTERN AUSTRALIA

- 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
- MPI_File_read_at_all
- MPI_File_write_at_all
- MPI_File_read_ordered
- MPI_File_write_ordered

Using the Right MPI I/O Function

THE UNIVERSITY OF WESTERN AUSTRALIA

- 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/how I/O functions are used
- We classify the different ways of expressing I/O access patterns in MPI I/O into four levels: **level 0 – level 3**

Outline

- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ Collective I/O
  - ✓ **I/O access pattern**
- Process Topologies
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

Example

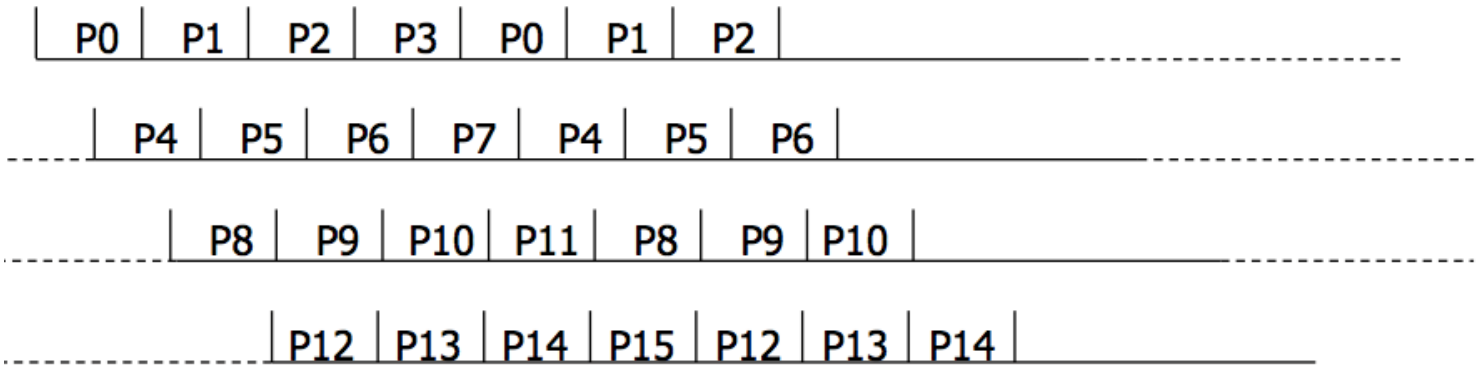
- **Distributed Array Access**

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file



Level-0 Access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
  MPI_File_seek(fh, ...);
  MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-1 Access

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
  MPI_File_seek(fh, ...);
  MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-2 Access

- Each process creates a derived datatype to describe the non-contiguous access pattern, defines a file view, and calls independent I/O functions

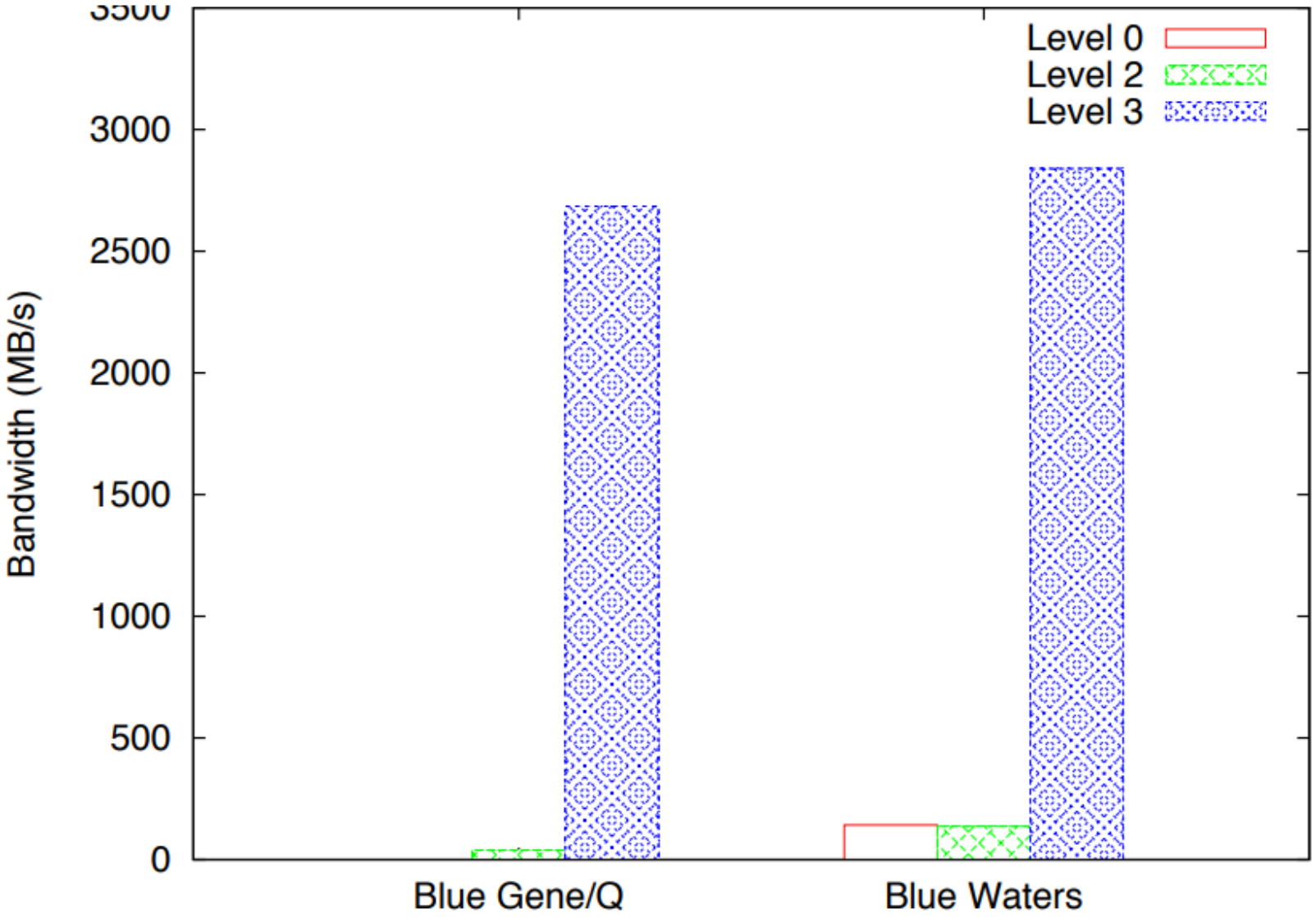
```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-3 Access

- Similar to level 2, except that each process uses collective I/O functions

```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

Performance Comparison



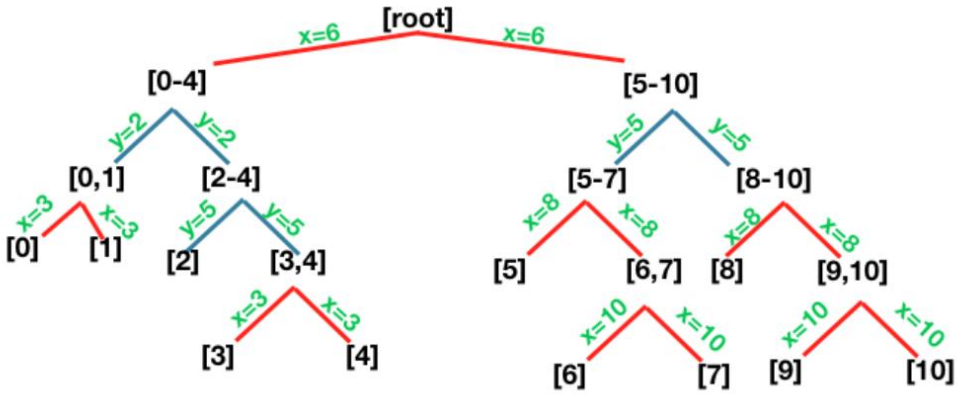
Outline

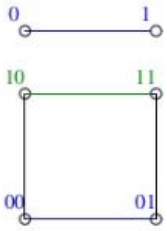
- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ Collective I/O
  - ✓ I/O access pattern
- **Process Topologies**
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

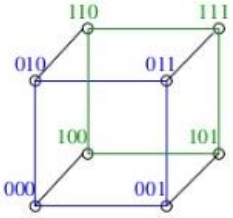
Why we need virtual topologies

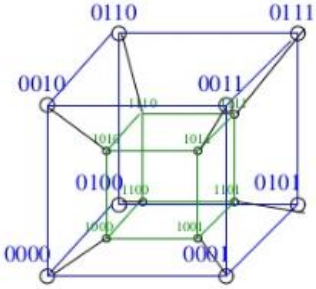
- So far, we deal with MPI processes as a linear range
- The processes needs to be mapped onto the hardware
  - ✓ Which is probably not a line of machines
  - ✓ Many different architectures

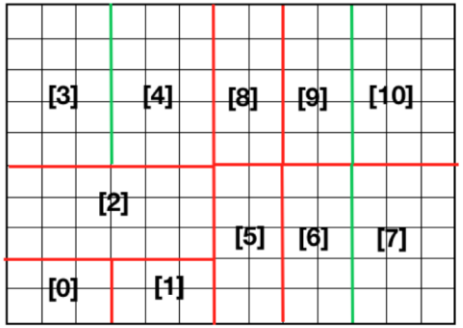
- Most numerical algorithms have some structure to their communication











Why we need virtual topologies (cont'd)

THE UNIVERSITY OF WESTERN AUSTRALIA

- If we don't exploit the structure in an algorithm
  - ✓ Could get 'random' process assignment
  - ✓ Extra communication overhead

- Virtual topologies allow you to specify communication patterns, allowing MPI to make smarter mapping choices no matter what machine you use

Outline

THE UNIVERSITY OF WESTERN AUSTRALIA

- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ Collective I/O
  - ✓ I/O access pattern
- Process Topologies
  - ✓ **Virtual topologies**
  - ✓ Cartesian grid topology
  - ✓ Distributed graph topology
  - ✓ Summary

Virtual and Physical Topologies

THE UNIVERSITY OF WESTERN AUSTRALIA

- A **virtual topology** represents the way that MPI processes communicate.
  - ✓ Communication patterns between processes can be represented as a graph
  - ✓ **Nodes == Processes**
  - ✓ **Edges == Communication**
  - ✓ This definition will work for any possible topology, but many are common, like Grids (2D, 3D, *n*D), Rings and Tori

- A **physical topology** represents that connections between the cores, chips, and nodes in the hardware.

- Issue is mapping of the **virtual topology** onto the **physical topology**

Virtual Topologies

THE UNIVERSITY OF WESTERN AUSTRALIA

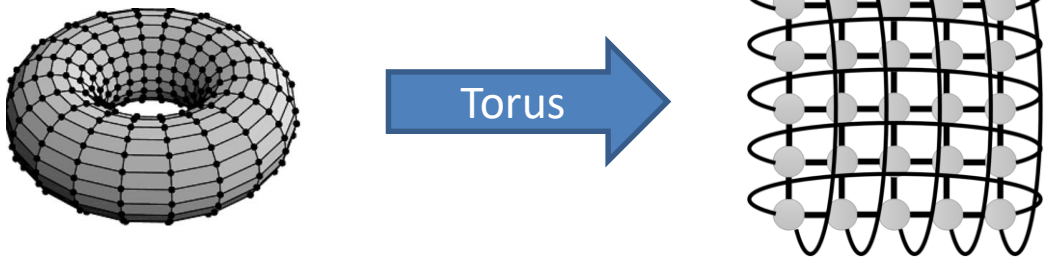
Two types of virtual topologies supported:

- **Cartesian**
  - ✓ *MPI_CART:* This value holds for Cartesian topologies, where processes act as if they are ordered in a multi-dimensional `brick';
- **Graph** (different graph types in MPI)
  - ✓ ~~*MPI_GRAPH*~~: this value describes the graph topology that was defined in MPI; It is unnecessarily burdensome, since each process needs to know the total graph, and should therefore be considered obsolete.
  - ✓ *MPI_DIST_GRAPH*: this value describes the distributed graph topology where each process only describes the edges in the process graph that touch itself.

Cartesian Topologies

- Concept
  - ✓ Any decomposition in natural coordinates
  - ✓ Indexing starts from 0
  - ✓ MPI_COMM_WORLD is a 1D Cartesian topology.

- Can be non-periodic or periodic
  - ✓ Periodic 1D == Ring
  - ✓ Non-Periodic 2D == Rectangle
  - ✓ Periodic 2D (in one dimension)==Cylinder
  - ✓ Full Periodic 2D == Torus



MPI_CART_CREATE

THE UNIVERSITY OF WESTERN AUSTRALIA

- The cartesian topology is specified by giving *MPI_Cart_create* the sizes of the processor grid along each axis, and whether the grid is periodic along that axis.

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```

- comm_old (IN) Input communicator
- ndims (IN) Number of dimensions
- dims (IN) Array of integers specifying number of processes in each dimension
- periods (IN) Logical array of size ndims specifying if each dimension is periodic (true) or not (false)
- reorder (IN) *Whether the ranks can be re-order from the old communicator*
- comm_cart (OUT) Communicator with new Cartesian topology

MPI_CART_CREATE

THE UNIVERSITY OF WESTERN AUSTRALIA

- Returns a handle to a new communicator with the topology specified
- As always, all processes need to call the same function with the same arguments for success
- If the total Cartesian grid is smaller than the number of processes you have, some will receive MPI_COMM_NULL
- This call returns errors if the grid is larger than the number of processes you have

Cartesian Topology: Rank and Coordinate

THE UNIVERSITY OF WESTERN AUSTRALIA

- Communication speed may increase if process addressing reordered by system. Therefore, each point in this new communicator has a coordinate and a rank.

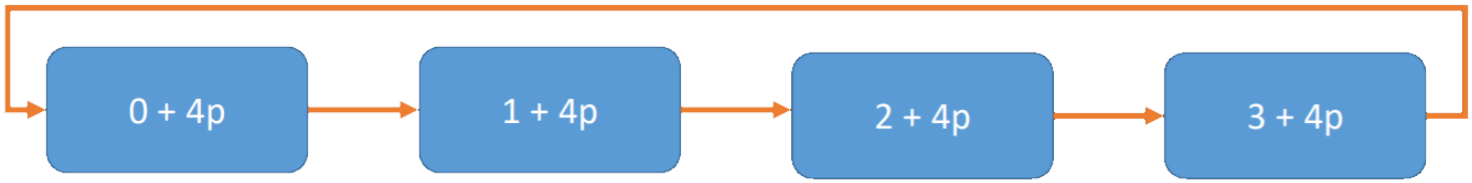
- They can be queried with MPI_Cart_coords and MPI_Cart_rank respectively.

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords);

int MPI_Cart_rank(MPI_Comm comm, init *coords, int *rank);
```

Cartesian Example 1: Ring

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```



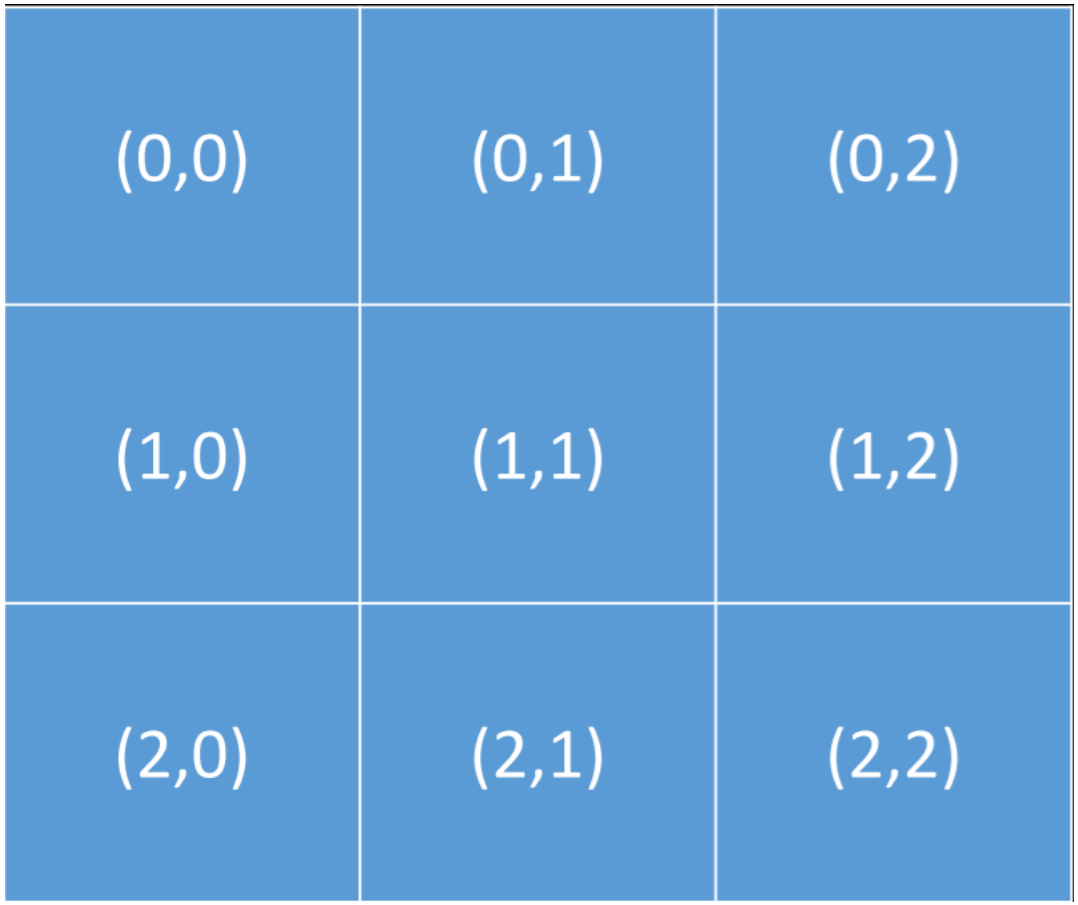
Assume we have access to:

- comm_size – Size of MPI_COMM_WORLD (4)
- rank – Rank of the calling process

```
MPI_Cart_Create(MPI_COMM_WORLD, 1, comm_size, true, true, NEW_COMM);
```

Cartesian Example 2: Square Grid

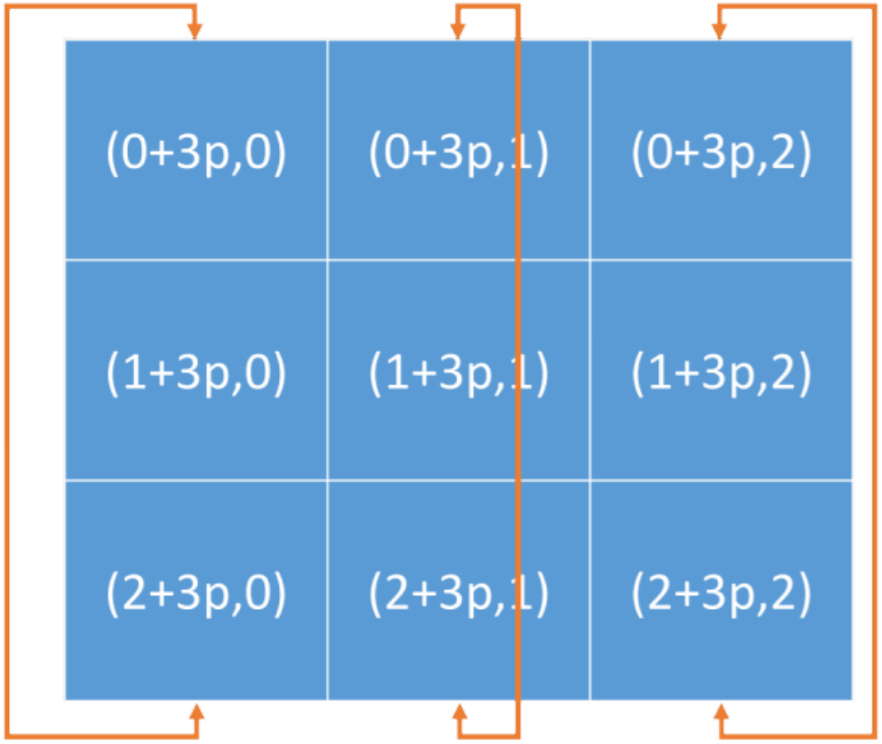
```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```




```
MPI_Cart_Create
(
    MPI_COMM_WORLD,
    2,
    {3,3},
    {False, False},
    True,
    NEW_COMM
);
```

Cartesian Example 3a: Cylinder

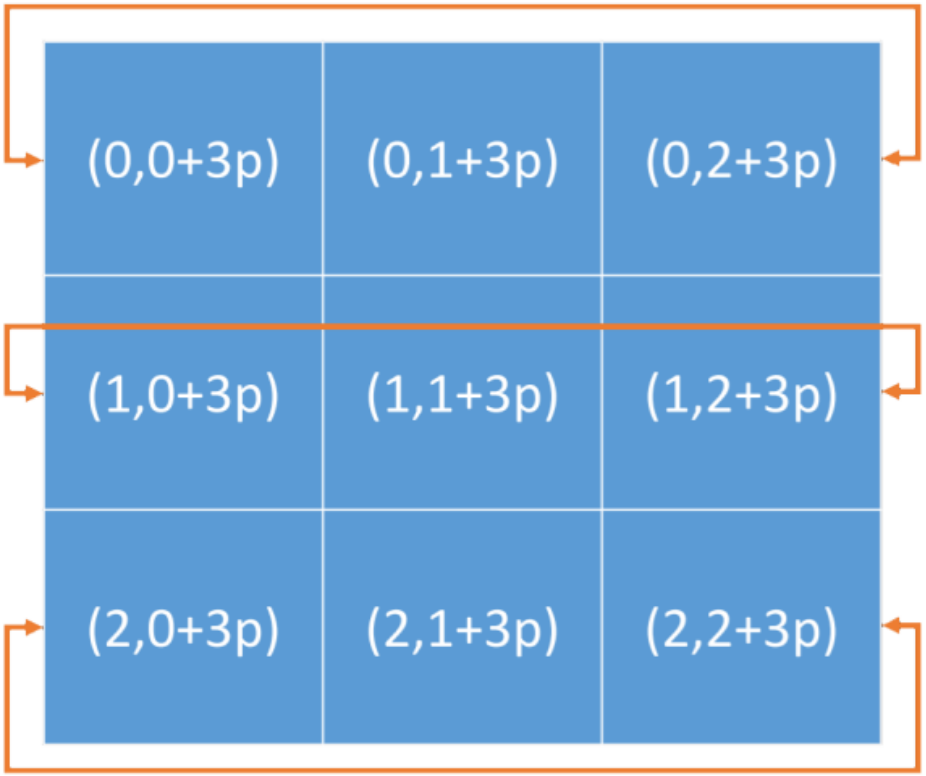
```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```




```
MPI_Cart_Create
(
MPI_COMM_WORLD,
2,
{3,3},
{True, False},
True,
NEW_COMM
);
```

Cartesian Example 3b: Cylinder

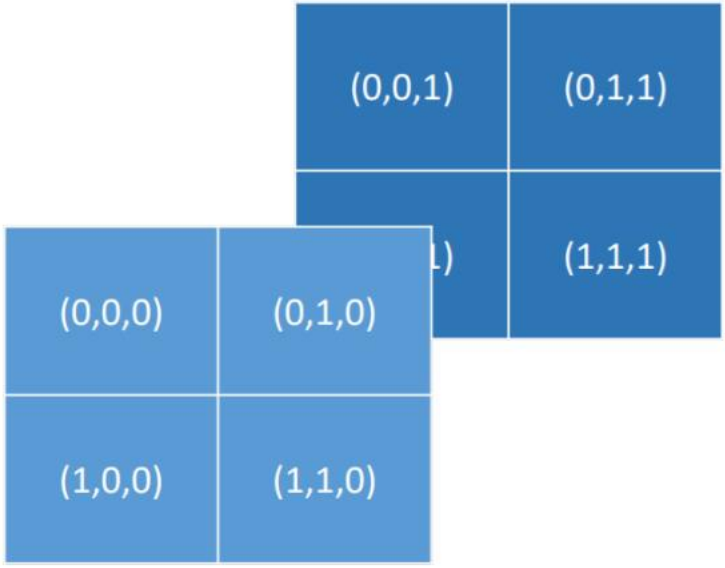
```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```




```
MPI_Cart_Create
(
MPI_COMM_WORLD,
2,
{3,3},
{False, True},
True,
NEW_COMM
);
```

Cartesian Example 4a: 3D Torus

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```




```
MPI_Cart_Create
(
MPI_COMM_WORLD,
3,
{2,2,2},
{True, True, True},
True,
NEW_COMM
);
```

Cartesian Example 4b: 2D Torus

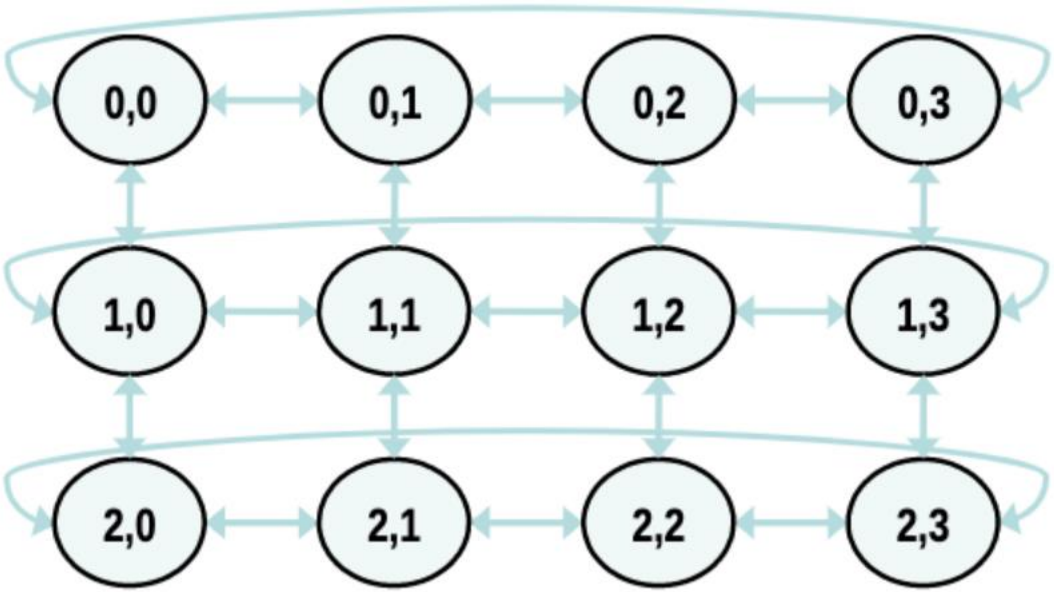
THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include<mpi.h> #include<stdio.h>
/* A two-dimensional torus of 12 processes in a 4x3 grid */
int main(int argc, char *argv[]) {
int rank, size; MPI_Comm comm;
int dim[2], period[2], reorder;
int coord[2], id;

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

dim[0]=4;  dim[1]=3;
period[0]=1; period[1]=0; reorder=1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);
if (rank == 5) {
      MPI_Cart_coords(comm, rank, 2, coord);
      printf("Rank %d coordinates are %d %d\n",
      rank, coord[0], coord[1]);fflush(stdout);
      }
if(rank==0) {
      coord[0]=3;
      coord[1]=1;
      MPI_Cart_rank(comm, coord, &id);
      printf("The processor at position (%d, %d) has rank %d\n",
      coord[0], coord[1], id);fflush(stdout);
      }
MPI_Finalize();
return 0;
}
```

Cartesian Example 4b: 2D Torus (Cont'd)



Cartesian Conveniences

```
MPI_DIMS_CREATE(nnodes, ndims, dims)
```

- Suggests the most balanced division of n-nodes into an n-dimensional grid
- Result is stored in the integer array dims
- If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i;

dims before call	function call	dims on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

Cartesian Conveniences

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int
*source, int *dest)
```

- Returns the shifted source and destination ranks, given a shift direction and amount
- The direction argument is in the range [0,n-1] for an n-dimensional Cartesian mesh

• Comm	(in)	communicator with cartesian structure	(handle)
• direction	(in)	coordinate dimension of shift	(integer)
• displacement	(in)	(> 0: upwards shift, < 0: downwards shift)	(integer)
• rank_source	(out)	rank of source process	(integer)
• rank_dest	(out)	rank of destination process	(integer)

MPI_Cart_shift Example (aside)

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include <stdio.h>
#include <mpi.h>
int main(int argc; char **argv; )
{
     int rank, value, size, false=0;
     int right_nbr, left_nbr;
     MPI_Comm one_dim_comm;
     MPI_Status status;
     MPI_Init( &argc, &argv );
     MPI_Comm_size( MPI_COMM_WORLD, &size );
     MPI_Cart_create( MPI_COMM_WORLD, 1, &size, 1, 1, &one_dim_comm );

     /*   create a one-dimensional grid.
          False means that there is no periodicity
     (the process outside the two ends is identified as MPI_PROC_NULL),
          1 means that it can be reordered,
          and the obtained topological coordinate adjacency relation is:
            MPI_PROC_NULL, 0, 1, ... , size-1 , MPI_PROC_NULL
     */
     MPI_Cart_shift( one_dim_comm, 0, 1, &left_nbr, &right_nbr );
     /*The identity of the left and right processes
     is obtained by shifting over the defined grid*/
     MPI_Comm_rank( one_dim_comm, &rank );
     MPI_Comm_size( one_dim_comm, &size );
```

MPI_Cart_shift Example

```
    do
    {
        if (rank == 0) {
        /*Process 0 is responsible for reading in
         the data and passing it on to the next process*/
        scanf( "%d", &value );
        MPI_Send( &value, 1, MPI_INT, right_nbr, 0, one_dim_comm );
        /*Sends data to the process on the right*/
        }
        else {
        MPI_Recv( &value, 1, MPI_INT, left_nbr, 0, one_dim_comm,
        &status );
        /*The next process receives data from the left process*/
        MPI_Send( &value, 1, MPI_INT, right_nbr, 0, one_dim_comm );
        /*Sends data to the process on the right*/
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);
    /*If the scanned data is non-negative,
    continue reading and passing*/
    MPI_Finalize( );
}
```

Outline

- Parallel File I/O
  - ✓ Introduction
  - ✓ I/O strategy
  - ✓ Independent I/O
  - ✓ Collective I/O
  - ✓ I/O access pattern
- Process Topologies
  - ✓ Virtual topologies
  - ✓ Cartesian grid topology
  - ✓ **Distributed graph topology**
  - ✓ Summary

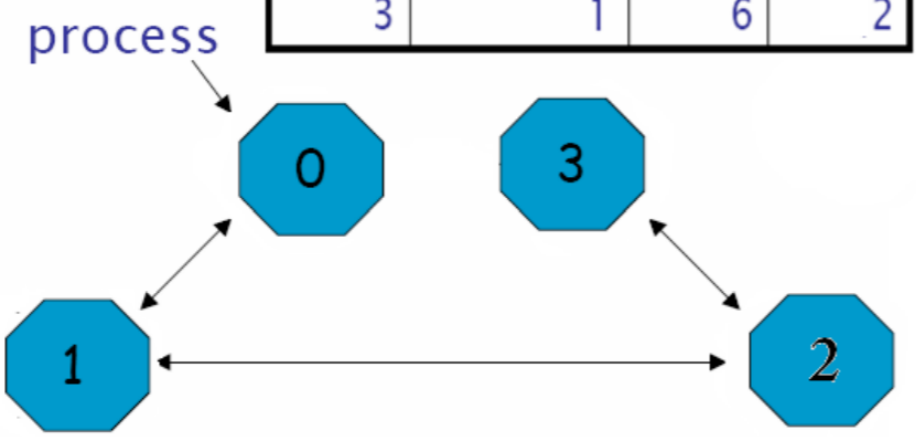
Distributed Graph Topology

THE UNIVERSITY OF WESTERN AUSTRALIA

Elements of Graph Topology

- Nodes: Processors
- Lines: Communicators between nodes
- Arrows: Show origins and destinations of links
- Index: array of integers describing node degrees

Node	Nneighbors	index	edges
0	1	1	1
1	2	3	0,2
2	2	5	1,3
3	1	6	2



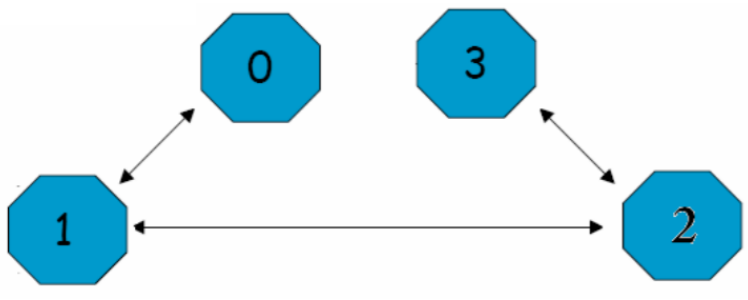
MPI_GRAPH_CREATE

THE UNIVERSITY OF WESTERN AUSTRALIA

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
        int reorder, MPI_Comm *comm_graph)
```

- comm_old [in] input communicator without topology (handle)
- nnodes [in] number of nodes in graph (integer)
- index [in] array of integers describing node degrees
- edges [in] array of integers describing graph edges
- reorder [in] ranking may be reordered (true) or not (false) (logical)
- comm_graph [out] communicator with graph topology added (handle)

MPI_GRAPH_CREATE - Example



Node	Nneighbors	index	edges
0	1	1	1
1	2	3	0,2
2	2	5	1,3
3	1	6	2

```
#include "mpi.h"
MPI_Comm graph_comm;
int nnodes = 4; /* number of nodes */
int index[4] = {1, 3, 5, 6}; /* index definition */
int edges[6] = {1, 0, 2, 1, 3, 2}; /* edges definition */
int reorder = 1; /* allows processes reordered for efficiency */
MPI_Graph_create(MPI_COMM_WORLD, nnodes, index, edges, reorder, graph_comm);
```

**In C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i where i=1, ..., nnodes-1;*

Graph Neighbors

THE UNIVERSITY OF WESTERN AUSTRALIA

- The number of the neighboring processes may be obtained by the following function:

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *mneighbors);
```

- Obtaining **the ranks of the neighboring** vertices is provided by the following function:

```
int MPI_Graph_neighbors(MPI_Comm comm,int rank, int mneighbors, int *neighbors);
```

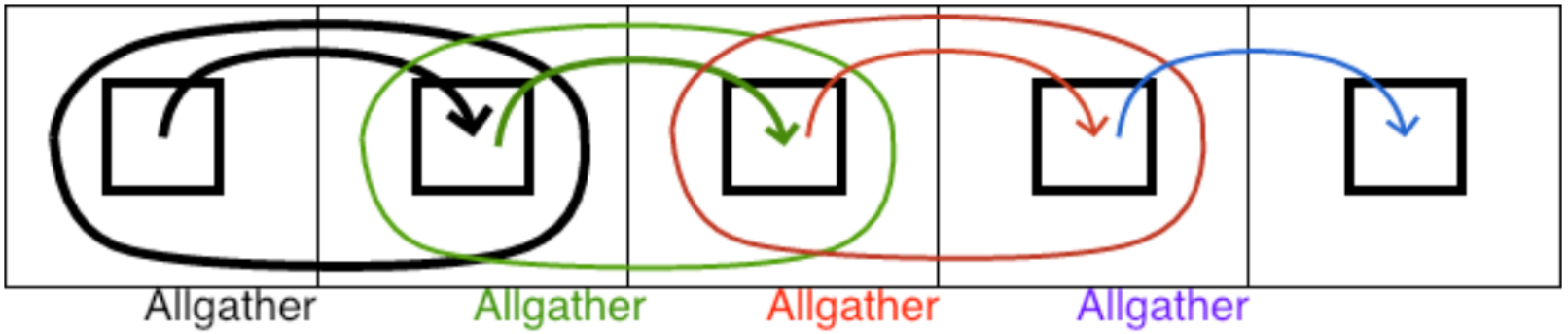
- (where mneighbors is the size of the array neighbors)

Neighbor Collectives

- MPI's notion of neighborhood collectives, offer an elegant way of expressing such communication structures.

- The neighbor collectives have the same argument list as the regular collectives, but they apply to a graph communicator.

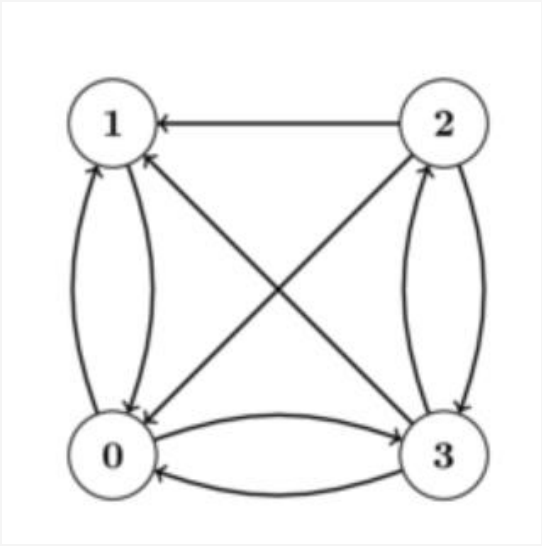
```
int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)
```



Graph Example

```
#include <mpi.h>
#include <stdio.h>
#define nnode 4
int main() {
MPI_Init(NULL, NULL);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int source = rank;
int degree;
int dest[nnode];
int weight[nnode] = {1, 1, 1, 1};
int recv[nnode] = {-1, -1, -1, -1};
int send = rank; // set dest and degree.

if (rank == 0) { dest[0] = 1; dest[1] = 3; degree = 2; }
else if(rank == 1) { dest[0] = 0; degree = 1; }
else if(rank == 2) { dest[0] = 3; dest[1] = 0; dest[2] = 1; degree = 3; }
else if(rank == 3) { dest[0] = 0; dest[1] = 2; dest[2] = 1; degree = 3; }
```



Graph Example (continue)

```
    // create graph.
    MPI_Comm graph;
    MPI_Dist_graph_create(MPI_COMM_WORLD, 1, &source,
    &degree, dest, weight, MPI_INFO_NULL, 1, &graph);

    // send and gather rank to/from neighbors.
    MPI_Neighbor_allgather(&send, 1, MPI_INT, recv, 1, MPI_INT, graph);
    printf("Rank: %i, recv[0] = %i, recv[1] = %i, recv[2] = %i,
    recv[3] = %i\n", rank, recv[0], recv[1], recv[2], recv[3]);
    MPI_Finalize();
    return 0;
}
```

In this function, each process i gathers data items from each process j if an edge (j,i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i,j) exists. The send buffer is sent to each neighboring process and the i-th block in the receive buffer is received from the i-th neighbor.

Summary

- I/O can destroy performance if handled poorly
- MPI I/O offers many ways to access files in parallel
  - ✓ Independent file access
  - ✓ Non-contiguous access to shared files
  - ✓ Collective calls to shared files

- Virtual topologies allows for the user to develop simpler communication calls and hence higher performance
  - ✓ Cartesian grids are flexible and common
  - ✓ Particular useful for numerical routines
  - ✓ Graph topologies

References

- Readings
  - Introduction to Parallel I/O
  - Advanced MPI: Parallel I/O
  - An Article on MPI Process Topologies
  - MPI Topologies
  - MPI_Cart_shift

Copyright Notice

Copyright Notice

Material used in this recording may have been reproduced and communicated to you by or on behalf of **The University of Western Australia** in accordance with section 113P of the *Copyright Act 1968*.

Unless stated otherwise, all teaching and learning materials provided to you by the University are protected under the Copyright Act and is for your personal use only. This material must not be shared or distributed without the permission of the University and the copyright owner/s.