Learning Objectives

- Learn about Cartesian topology.
- Learn about how to submit multi jobs on cluster
Dealing with Communicators

- Many MPI operations deal with all the processes in a communicator.
- MPI_COMM_WORLD by default contains every task in your MPI job.
- Other communicators can be defined for more complex operations; for different parts of the task, to add topology, to segregate different kinds of messaging.
- MPI allows you to create subsets of communicators.
Why Communicators?

- Isolate communication to a small number of processors
- Useful for creating libraries
- Different processors can work on different parts of the problem
- Useful for communicating with "nearest neighbors"
MPI Communication Features

- Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

- Communicators (see below) provide a “caching” mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. E.g., virtual topologies.

- Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving).
  - Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations.
Communicators and Groups

- Many MPI users are only familiar with the communicator MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines
Communicators and Groups (cont)

- An intracommunicator is used for communication within a single group.
- An intercommunicator is used for communication between 2 disjoint groups.
Communicators and Groups (cont)
Communicators and Groups (cont)

- Refer to previous slide
  - There are 4 distinct groups.
  - These are associated with intracommunicators (MPI_COMM_WORLD, comm1, and comm2, and comm3).
  - $P_3$ is a member of 2 groups and may have different ranks in each group (say 3 & 4).
  - If $P_2$ wants to send a message to $P_1$, it must use MPI_COMM_WORLD (intracommunicator) or comm5 (intercommunicator).
  - If $P_2$ wants to send a message to $P_3$, it can use MPI_COMM_WORLD or comm1 (rank number will be different).
  - $P_0$ can broadcast a message to all processes associated with comm2 by using intercommunicator comm4.
**MPI_Comm_create**

- MPI_Comm_create creates a new communicator newcomm with group members defined by a group data structure.

**Name**

**MPI_Comm_create** - Creates a new communicator.

**Syntax**

**C Syntax**

```
#include <mpi.h>
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

**Fortran Syntax**

```
INCLUDE 'mpi.f'
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
     INTEGER    COMM, GROUP, NEWCOMM, IERROR
```

**C++ Syntax**

```
#include <mpi.h>

MPI::Intercomm MPI::Intercomm::Create(const Group& group) const
MPI::Intracomm MPI::Intracomm::Create(const Group& group) const
```

How do you define a group?
MPI_Comm_group

- Given a communicator, MPI_Comm_group returns in group associated with the input communicator.

**Name**

MPI_Comm_group - Returns the group associated with a communicator.

**Syntax**

**C Syntax**

```c
#include <mpi.h>
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

**Fortran Syntax**

```fortran
#include 'mpi.h'
MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR
```

**C++ Syntax**

```cpp
#include <mpi.h>
Group Comm::Get_group() const
```
MPI_Group_incl

- Produces a group by reordering an existing group and taking only listed members.

**Syntax**

**C Syntax**

```c
#include <mpi.h>
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup)
```

**Fortran Syntax**

```fortran
INCLUDE ’mpif.h’
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
   INTEGER   GROUP, N, RANKS(*), NEWGROUP, IERROR
```

**C++ Syntax**

```cpp
#include <mpi.h>
Group Group::Incl(int n, const int ranks[]) const
```
**MPI_Comm_split**

- Creates new communicators based on colors and keys.

**Syntax**

**C Syntax**

```c
#include <mpi.h>
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

**Fortran Syntax**

```fortran
INCLUDE 'mpi.f'
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
  INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

**C++ Syntax**

```cpp
#include <mpi.h>
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

```c
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```
Lab: MPI_Comm_split

```c
int main(int argc, char *argv[])
{
    // MPI variables
    int world_rank, world_size;
    int new_rank, new_size;

    MPI::Init(argc, argv);
    world_rank = MPI::COMM_WORLD.Get_rank();
    world_size = MPI::COMM_WORLD.Get_size();

    // Color for split
    int color = world_rank%3;
    cout << "world_rank = " << world_rank << ", color = " << color << endl;

    // Split the communicator
    MPI::Intercomm new_comm;
    new_comm = MPI::COMM_WORLD.Split(color, world_rank);

    // Determine this process's rank
    new_rank = new_comm.Get_rank();

    // Determine the number of available processes
    new_size = new_comm.Get_size();

    // Print output
    cout << "World Rank/Size: " << world_rank << "/" << world_size << ", New Rank/Size: " << new_rank << "/" << new_size << endl;

    MPI::Finalize();
    return 0;
}
```
int main(int argc, char *argv[]) {
    // MPI variables
    int world_rank, world_size;
    int new_rank, new_size;
    int N, members_1[8], members_2[8];

    world_rank = MPI::COMM_WORLD.Get_rank();
    world_size = MPI::COMM_WORLD.Get_size();

    // MPI World Group
    MPI::Group world_group;
    world_group = MPI::COMM_WORLD.Get_group();

    // Processes of MPI_COMM_WORLD are divided
    N = world_size/2;

    // members
    for (int i=0; i<N; i++) {
        members_1[i] = i;
        members_2[i] = N+i;
    }

    // MPI Groups
    MPI::Group new_group;

    // Divide tasks into two groups
    if (world_rank < N) {
        new_group = world_group.Incl(N, members_1);
    } else {
        new_group = world_group.Incl(N, members_2);
    }

    // Create new communicator
    MPI::Intracomm new_comm;
    new_comm = MPI::COMM_WORLD.Create(new_group);

    new_rank = new_comm.Get_rank();
    new_size = new_comm.Get_size();

    cout << "World Rank/Size: " << world_rank << "/" << world_size << ", new_comm Rank/Size: "
         << new_rank << "/" << new_size << endl;
    MPI::Finalize();
    return 0;
}
Communicators and Groups

- Many MPI users are only familiar with the communicator MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines
An intracommunicator is used for communication within a single group

An intercommunicator is used for communication between 2 disjoint groups
A topology is a mechanism for associating different addressing schemes with processes.

A topology can provide a convenient naming mechanism for processes.

A topology can allow MPI to optimize communications.

Simplifies writing of code.

There are virtual process topology and topology of the underlying hardware.

The virtual topology can be exploited by the system in assigning of processes to processors.
Introduction

- Any process topology can be represented by graphs.
- MPI provides defaults for ring, mesh, torus and other common structures

![Graph Table]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>4</td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>8</td>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>

![Matrix Data]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
How to Use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides “mapping functions”
- Mapping functions compute processor ranks, based on the topology naming scheme

2D mesh  
2D torus  
3D mesh
Topology Types

- Cartesian topologies
  - Each process is connected to its neighbors in a virtual grid
  - Boundaries can be cyclic
  - Processes can be identified by cartesian coordinates

- Graph topologies
  - Any process topology can be represented by graphs
  - General graphs
  - Will not be covered here
Cartesian Topology

- **Cartesian topologies** assume the presentation of a set of processes as a rectangular grid and the use of Cartesian coordinate system for pointing to the processes,
- Cartesian structures of arbitrary dimensions
- Can be periodic along any number of dimensions
- Popular Cartesian structures – linear array, ring, rectangular mesh, cylinder, torus (hypercubes)
## Cartesian Topology

- Process coordinates begin with 0
- Row-major numbering
- Example: 12 processes arranged on a 3 x 4 grid

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>
Virtual Topology MPI Routines

- Some of the MPI topology routines are
  - MPI_CART_CREATE
  - MPI_CART_COORDS
  - MPI_CART_RANK
  - MPI_CART_SUB
  - MPI_CARTDIM_GET
  - MPI_CART_GET
  - MPI_CART_SHIFT
**MPI_Cart_create**

- Used to create Cartesian coordinate structures, of arbitrary dimensions, on the processes. The new communicator receives no cached information.

### C Syntax

```c
#include <mpi.h>
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                     const int periods[], int reorder, MPI_Comm *comm_cart)
```

### Fortran Syntax

```fortran
INCLUDE 'mpif.h'
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
                 COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```

### C++ Syntax

```cpp
#include <mpi.h>
Cartcomm Intracommp.Create_cart(int[] ndims, int[] dims[],
                                 const bool periods[], bool reorder) const
```
MPI_Cart_create

C++ Syntax:

```cpp
MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const;
```

- `ndims` – number of dimensions of Cartesian grid
- `dims` – number of processes along each dimension
- `periods` – whether a dimension is cyclic (true) or not
- `reorder` – ranking of initial processes may be reordered (true) or not (rank preserved)
C++ Syntax Example

// Variables required by Create_cart
int ndims, dims[2];
bool periods[2], reorder;
ndims = 2;              // 2D matrix/grid
dims[0] = 3;            // rows
dims[1] = 4;            // columns
periods[0] = true;      // row periodic (each column forms a ring)
periods[1] = false;     // columns non-periodic
reorder = true;         // allows processes reordered for efficiency

// MPI new Cartcomm by Create_cart
MPI::Cartcomm new_comm = MPI::COMM_WORLD.Create_cart(ndims, dims, periods, reorder);
MPI_Cart_coords & MPI_Cart_rank

- Rank-to-coordinates translator
- Coordinates-to-rank translator

C Syntax

```c
#include <mpi.h>
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

C++ Syntax

```c++
#include <mpi.h>
void Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const
```

Fortran Syntax

```fortran
INCLUDE 'mpi.f'
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
   INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

C++ Syntax

```c++
#include <mpi.h>
int Cartcomm::Get_cart_rank(const int coords[]) const
```
MPI_Cart_coords & MPI_Cart_rank

C++ Syntax:

```cpp
void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const;
```

- rank – rank of a process within group of comm
- maxdims – length of vector coords in the calling program
- coords – array containing the Cartesian coordinates of specified process

```cpp
int MPI::Cartcomm::Get_cart_rank(const int coords[]) const;
```

- coords – array containing the Cartesian coordinates of the process
int main(int argc, char *argv[]) {

    // MPI variables
    int world_rank, world_size;
    int new_rank, new_size;

    // MPI Init and get rank
    MPI::Init(argc, argv);  // Initialize MPI
    world_rank = MPI::COMM_WORLD.Get_rank(); // Determine this process's rank
    world_size = MPI::COMM_WORLD.Get_size(); // Determine the number of available processes

    // Variables required by Create_cart
    int ndims, dims[2];
    bool periods[2], reorder;
    ndims = 2;  // 2D matrix/grid
    dims[0] = 3;  // rows
    dims[1] = 4;  // columns
    periods[0] = true; // row periodic (each column forms a ring)
    periods[1] = false; // columns non-periodic
    reorder = true; // allows processes reordered for efficiency

    // MPI new Cartcomm by Create_cart
    MPI::Cartcomm grid_comm = MPI::COMM_WORLD.Create_cart(ndims, dims, periods, reorder);

    // Determine this process's rank
    int grid_rank;
    grid_rank = grid_comm.Get_rank();

    // Translates task rank in a communicator into cartesian task coordinates.
    int grid_coords[2];
    grid_comm.Get_coords(grid_rank, ndims, grid_coords);

    // Translate the coordinates to ranks
    int cart_rank = grid_comm.Get_cart_rank(grid_coords);

    // Print
    cout << "Process:" << world_rank
         << " > grid_rank=" << grid_rank
         << ", coords=(" << grid_coords[0] << "," << grid_coords[1] << ")" << " 
         << " > cart_rank=" << cart_rank << endl;

    // MPI finalize and exit
    return 0;
}
MPI_Cart_sub

- Used to partition a communicator group into subgroups when MPI_CART_CREATE has been used to create a Cartesian topology.

C++ Syntax:

```cpp
MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const;
```

- `remain_dims` – The ith entry of remain_dims specifies whether the ith dimension is kept in the subgrid (true) or is dropped (false) (logical vector).
Sending and receiving in Cartesian topology

- There is no MPI_Cart_send or MPI_Cart_recv which would allow you to send a message to process (1,0) in your Cartesian topology, for example.

- You must use standard communication functions.

- There is a convenient way to obtain the rank of the desired destination/source process from your Cartesian coordinate grid.

- Usually one needs to determine which are the adjacent processes in the grid and obtain their ranks in order to communicate.
MPI_Cart_shift

- Returns the shifted source and destination ranks, given a shift direction and amount.

**C Syntax**

```c
#include <mpi.h>
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest)
```

**Fortran Syntax**

```fortran
INCLUDE 'mpif.h'
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
               RANK_DEST, IERROR)
    INTEGER     COMM, DIRECTION, DISP, RANK_SOURCE
    INTEGER     RANK_DEST, IERROR
```

**C++ Syntax**

```cpp
#include <mpi.h>
void Cartcomm::Shift(int direction, int disp, int& rank_source,
                     int& rank_dest) const
```
MPI_Cart_shift

C++ Syntax:

```cpp
void MPI::Cartcomm::Shift(int direction, int disp, int &rank_source, int &rank_dest) const;
```

- **direction** – the cartesian grid index, has range (0, 1, \ldots, ndim-1). For a 2D grid, the 2 choices for direction are 0 and 1.
- **disp**: Amount and sense of shift (<0: downward shifts, >0: upwards shifts, or 0)
- **source** – rank of process to receive data from
- **dest** – rank of process to send data to
Test below

// MPI Cart shift
int src, dst;
grid_comm.Shift(0,1,src,dst);
cout << "Process:" << world_rank << " > src: " << src << ", dst:" << dst << endl;

// Test again
//grid_comm.Shift(1,1,src,dst);
//cout << "Process:" << world_rank << " > src: " << src << ", dst:" << dst << endl;
MPI Primer

- https://computing.llnl.gov/tutorials/mpi/
Running multiple jobs

- http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/
#!/bin/bash

#SBATCH --export=ALL
#SBATCH -n 8
#SBATCH --partition=defq,med_mem,hi_mem

for i in sim/*; do
    mkdir $i/$SLURM_JOB_NAME
    ./integ.py ecoli.fa viral/*basename $i* > $i/$SLURM_JOB_NAME/g.fa
    wgsim -N 1000000 -1 150 -2 150 $i/$SLURM_JOB_NAME/g.fa $i/$SLURM_JOB_NAME/l.fq $i/$SLURM_JOB_NAME/r.fq
    bwa mem -a -T 0 -t 8 index/all_viral.fa $i/$SLURM_JOB_NAME/l.fq $i/$SLURM_JOB_NAME/r.fq \
        | samtools view -b | samtools sort -@8 - > $i/$SLURM_JOB_NAME/aln.bam
    samtools index $i/$SLURM_JOB_NAME/aln.bam
    ./m1.py $i/$SLURM_JOB_NAME/aln.bam > $i/$SLURM_JOB_NAME/stats
    rm -rf $i/$SLURM_JOB_NAME/*.*
done