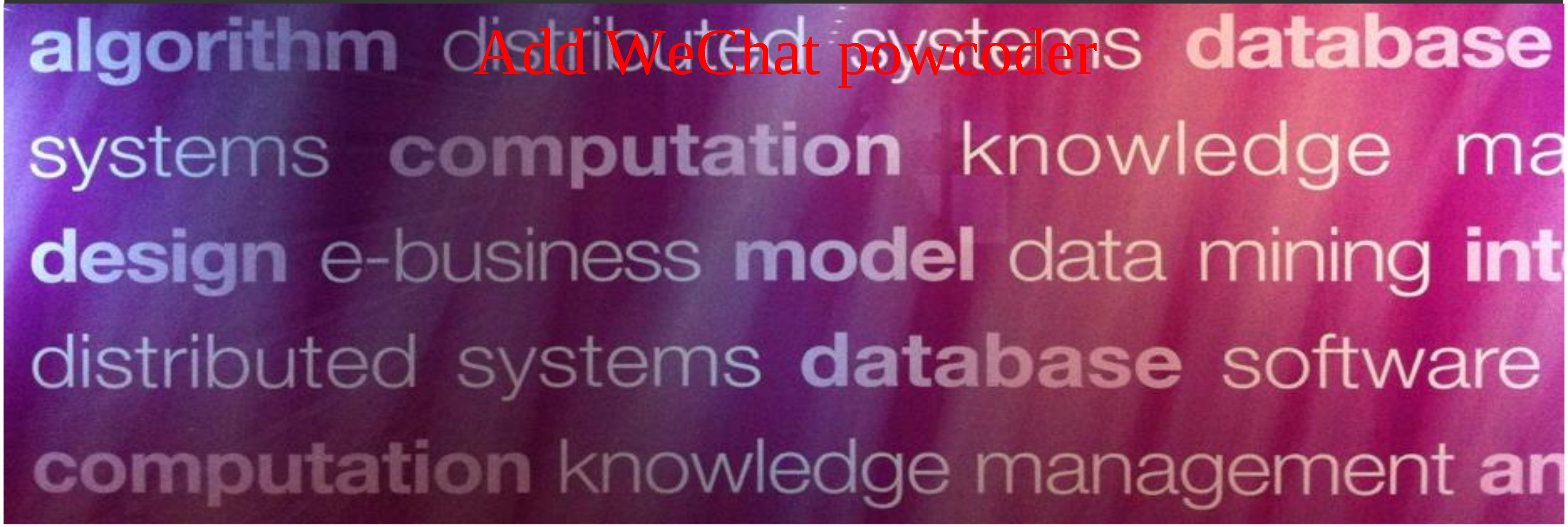
MONASH University

Information Technology

FIT3143 - LECTURE WEEK 5

PARALLEL COMPUTING IN DISTRIBUTED MEMORY – MESSAGE PASSING LIBRARY



Overview

1. Message Passing Interface (MPI)
2. MPI Routines

Learning outcome(s) related to this topic

- Explain the fundamental principles of parallel computing architectures and algorithms (LO1)
- Design and develop parallel algorithms for various parallel computing architectures (LO3)

What is MPI

- **M P I** = **M**essage **P**assing **I**nterface
- MPI is a ***specification*** for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be
  - practical
  - portable
  - efficient
  - flexible

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all major platforms and many specialised HPC systems. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.

Programming Model

- MPI lends itself to virtually any distributed memory parallel programming model. In addition, MPI is commonly used to implement (behind the scenes) some shared memory models, such as Data Parallel, on distributed memory architectures.

- Hardware platforms:
  - Distributed Memory: Originally, MPI was targeted for distributed memory systems.
  - Shared Memory: As shared memory systems became more popular, particularly SMP / NUMA architectures, MPI implementations for these platforms appeared.
  - Hybrid: MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.

Programming Model

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

- The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time. (MPI-2 addresses this issue).

Getting Started

- **MPI is native to ANSI C**
  - **C++ and Java bindings are available**
    - **MPI C++ classes – www.mcs.anl.gov**
    - **mpiJava API – www.hpjava.org**

- **MPI versions**
- **MPI - C**

  - **Header File:**
    - Required for all programs/routines which make MPI library calls.


```
#include "mpi.h"
Or
#include <mpi.h>
```

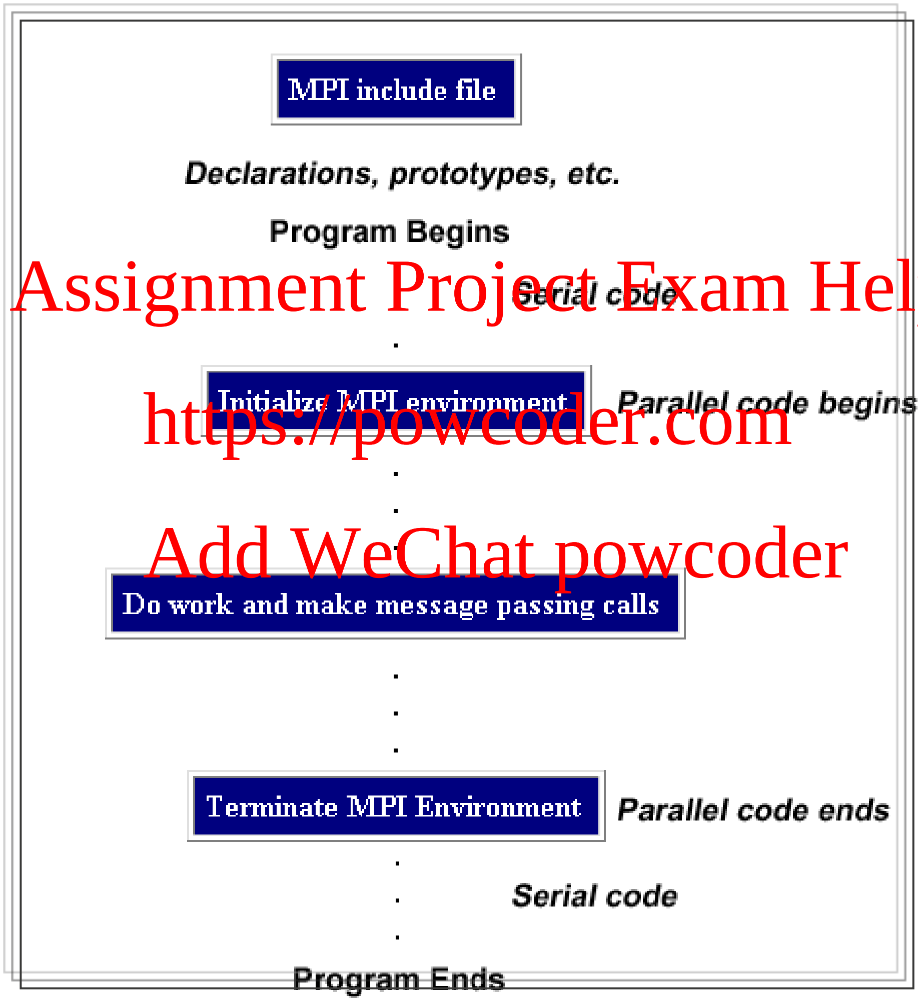
  - **Format of MPI Calls:**

Format: rc = MPI_Xxxxx(parameter, ...)

Example: rc = MPI_Bsend(&buf,count,type,dest,tag,comm)

Error code: rc value is set to MPI_SUCCESS if successful

General MPI Program Structure



MPI’s “Hello World”

1. **Create Source Code File: hello.c**

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

int main(int argc, char *argv[])
{    int numprocs, rank, namelen;
     char processor_name[MPI_MAX_PROCESSOR_NAME];
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Get_processor_name(processor_name, &namelen);

     printf("Process %d on %s out of %d\n", rank, processor_name,
     numprocs);

     MPI_Finalize();
}
```


2. **Compile: `mpicc hello.c –o hello-mp`**
3. **Execute: `mpirun -np 2 hello-mp`**

Communicators and Groups

- MPI uses objects called *communicators* and *groups* to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.

- Communicators and groups will be covered in more detail later. For now, simply use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.

Communicators and Groups

- Rank:
  - Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
  - Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that etc).

Environment Management Routines

- **MPI_Init**
  - Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

    MPI_Init (&argc, &argv)

- **MPI_Comm_size**
  - Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.

    MPI_Comm_size (comm, &size)

Environment Management Routines

- **MPI_Comm_rank**
  - Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID.

MPI_Comm_rank (comm, &rank)

- **MPI_Abort**
  - Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

MPI_Abort (comm, errorcode)

Environment Management Routines

- **MPI_Get_processor_name**
  - Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

    MPI_Get_processor_name (&name, &resultlength)

- **MPI_Initialized**
  - Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

    MPI_Initialized (&flag)

Environment Management Routines

- **MPI_Wtime**
  - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
    MPI_Wtime ()

- **MPI_Wtick**

  - Returns the resolution in seconds (double precision) of MPI_Wtime.
    MPI_Wtick ()

- **MPI_Finalize**
  - Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.
    MPI_Finalize ()

Environment Management Routines Example

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

int main(int argc, char *argv[])
{
   int numtasks, rank, rc;
   rc = MPI_Init(&argc,&argv);
   if (rc != MPI_SUCCESS)
      {  printf ("Error starting MPI program. Terminating.\n");
MPI_Abort(MPI_COMM_WORLD, rc);
      }
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

   /******* do some work *******/

   MPI_Finalize();
}
```

Point to Point (P2P) Communication

General Concepts

- **Types of Point-to-Point Operations:**
  - MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

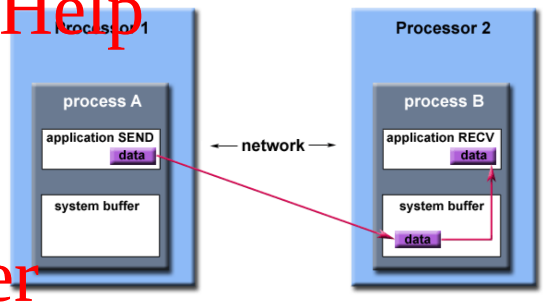
  - There are different types of send and receive routines used for different purposes. For example:

    - Synchronous send
    - Blocking send / blocking receive

    - Non-blocking send / non-blocking receive
    - Buffered send
    - Combined send/receive
    - "Ready" send
  - Any type of send routine can be paired with any type of receive routine.
  - MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

P2P general concepts

- **Buffering:**
  - In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
  - Consider the following two cases:
    - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
    - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
  - The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a **system buffer** area is reserved to hold data in transit.



Path of a message buffered at the receiving process

P2P general concepts

- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.
  - User managed address space (i.e. your program variables) is called the **application buffer**. MPI also provides for a user managed send buffer.

P2P general concepts

- **Blocking vs. Non-blocking:**
- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

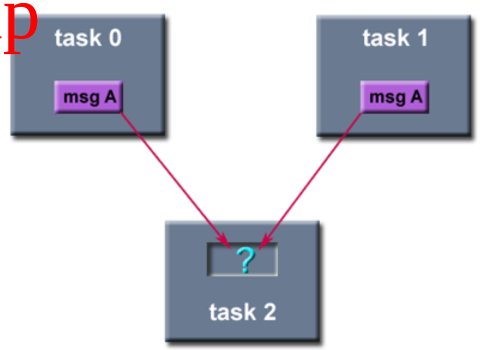
- Blocking:
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is a handshake occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

P2P general concepts

- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

P2P general concepts

- **Order and Fairness:**
- Order:
  - MPI guarantees that messages will not overtake each other.
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
  - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
  - Order rules do not apply if there are multiple threads participating in the communication operations.
- Fairness:
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



Point to Point Communication Routines and Arguments

- MPI Message Passing Routine Arguments

Blocking sends	**MPI_Send(buffer, count, type, dest, tag, comm)**
Non-blocking sends	**MPI_Isend(buffer, count, type, dest, tag, comm, request)**
Blocking receive	**MPI_Recv(buffer, count, type, source, tag, comm, status)**
Non-blocking receive	**MPI_Irecv(buffer, count, type, source, tag, comm, request)**

- **Buffer**
  - Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: **&var1**

- **Data Count**
  - Indicates the number of data elements of a particular type to be sent.

- **Data Type**
  - For reasons of portability, MPI predefines its elementary data types. The table in next slide lists those required by the standard.

Data Types

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI Message Passing Routine Arguments

- **Destination**
  - An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

- **Source**

  - An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

- **Tag**
  - Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag.
  - The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

MPI Message Passing Routine Arguments

- **Communicator**
  - Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.

- **Status**

  - For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG). Additionally, the actual number of bytes received are obtainable from Status via the MPI_Get_count routine.

- **Request**
  - Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request.

Blocking Message Passing Routines

- **MPI_Send**

  – Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.

  MPI_Send (&buf,count,datatype,dest,tag,comm)

- **MPI_Recv**

  – Receive a message and block until the requested data is available in the application buffer in the receiving task.

  MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

Blocking Message Passing Routines

- **MPI_Ssend**
  - Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

  MPI_Ssend (&buf,count,datatype,dest,tag,comm)

- **MPI_Bsend**
  - Buffered blocking send: permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Insulates against the problems associated with insufficient system buffer space. Routine returns after the data has been copied from application buffer space to the allocated send buffer. Must be used with the MPI_Buffer_attach routine.

  MPI_Bsend (&buf,count,datatype,dest,tag,comm)

Blocking Message Passing Routines

- **MPI_Buffer_attach**

  – Used by programmer to allocate/deallocate message buffer space to be used by the MPI_Bsend routine. The size argument is specified in actual data bytes - not a count of data elements. Only one buffer can be attached to a process at a time. Note that the IBM implementation uses MPI_BSEND_OVERHEAD bytes of the allocated buffer for overhead.

  MPI_Buffer_attach (&buffer,size)
  MPI_Buffer_detach (&buffer,size)

- **MPI_Rsend**

  – Blocking ready send. Should only be used if the programmer is certain that the matching receive has already been posted.

  MPI_Rsend (&buf,count,datatype,dest,tag,comm)

Blocking Message Passing Routines

- **MPI_Sendrecv**

  – Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

**MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag,
&recvbuf,recvcount,recvtype,source,recvtag, comm,&status)**

Blocking Message Passing Routines

- **MPI_Wait**
  MPI_Waitany
  MPI_Waitall
  MPI_Waitsome

  – MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

MPI_Wait (&request,&status)
MPI_Waitany (count,&array_of_requests,&index,&status)
MPI_Waitall (count,&array_of_requests,&array_of_statuses)
MPI_Waitsome (incount,&array_of_requests,&outcount,
&array_of_offsets, &array_of_statuses)

Blocking Message Passing Routines

- **MPI_Probe**

  – Performs a blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

  MPI_Probe (source,tag,comm,&status)

Example: Blocking Message Passing Routines

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

int main(int argc,char *argv[])
{ int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
{ dest = 1; source = 1;
rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
else if (rank == 1)
{ dest = 0; source = 0;
rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank, count,
                                       Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Non-Blocking Message Passing Routines

- **MPI_Isend**

  – Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

  MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)

- **MPI_Irecv**

  – Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

  MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)

Non-Blocking Message Passing Routines

- MPI_Issend

  – Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

  MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)

- MPI_Ibsend

  – Non-blocking buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.

  MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request)

- MPI_Irsend

  – Non-blocking ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Should only be used if the programmer is certain that the matching receive has already been posted.

  MPI_Irsend (&buf,count,datatype,dest,tag,comm,&request)

Non-Blocking Message Passing Routines

- **MPI_Test**
  MPI_Testany
  MPI_Testall
  MPI_Testsome

  – MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

MPI_Test (&request,&flag,&status)
MPI_Testany (count,&array_of_requests,&index,&flag,&status)
MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
MPI_Testsome (incount,&array_of_requests,&outcount, &array_of_indices, &array_of_statuses)

Non-Blocking Message Passing Routines

- **MPI_Iprobe**

  – Performs a non-blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. The integer "flag" parameter is returned logical true (1) if a message has arrived, and logical false (0) if not. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

  MPI_Iprobe (source,tag,comm,&flag,&status)

Example: Non-Blocking Message Passing Routines

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

int main(int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks); MPI_Comm_rank(MPI_COMM_WORLD,
   &rank);
prev = rank-1;
next = rank+1;
if (rank == 0) prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
 { /* do some work */ }
MPI_Waitall(4, reqs, stats);
printf("I am proc %d buf[0]=%d buf[1]=%d\n",rank,buf[0],buf[1]);
MPI_Finalize();
}
```

Collective Communication

- **All or None:**
- Collective communication must involve **all** processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD.
- It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations.
- **Types of Collective Operations:**
- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Collective Communication

Programming Considerations and Restrictions:

- Collective operations are blocking.
- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators (discussed in the Group and Communicator Management Routines section).
- Can only be used with MPI predefined datatypes - not with MPI Derived Data Types.

Collective Communication Routines

- **MPI_Barrier**
  - Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.

    MPI_Barrier (comm)

- **MPI_Bcast**
  - Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

    MPI_Bcast (&buffer,count,datatype,root,comm)

Collective Communication Routines

- **MPI_Scatter**
  - Distributes distinct messages from a single source task to each task in the group.

    MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
    ..... recvcnt,recvtype,root,comm)

- **MPI_Gather**

  - Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

    MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,
    ...... recvcount,recvtype,root,comm)

Collective Communication Routines

- MPI_Allgather
  - Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

    MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
    ...... recvcount,recvtype,comm)

- MPI_Reduce
  - Applies a reduction operation on all tasks in the group and places the result in one task.

    MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

Collective Communication Routines

- **MPI_Allreduce**

  – Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.

  MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)

- **MPI_Reduce_scatter**

  – First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation.

  MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype, op,comm)

Collective Communication Routines

- **MPI_Alltoall**

  – Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

  MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)

- **MPI_Scan**

  – Performs a scan operation with respect to a reduction operation across a task group.

  MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)

Examples: Collective Communications

```
#include "mpi.h"
#include <stdio.h>
 #define SIZE 4
int main(argc,argv)
int argc; char *argv[];
{ int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = { {1.0, 2.0, 3.0, 4.0}, {5.0, 6.0, 7.0, 8.0}, {9.0,
   10.0, 11.0, 12.0}, {13.0, 14.0, 15.0, 16.0} };
float recvbuf[SIZE];
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks == SIZE)
{ source = 1; sendcount = SIZE; recvcount = SIZE;
MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
   MPI_FLOAT,source,MPI_COMM_WORLD);
printf("rank= %d Results: %f %f %f %f\n",rank,recvbuf[0],
   recvbuf[1],recvbuf[2],recvbuf[3]); }
else printf("Must specify %d processors. Terminating.\n",SIZE);
MPI_Finalize();
}
```

Derived data type

- MPI predefines its primitive data types:

- MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.

- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.

- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

C Data Types

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG
MPI_UNSIGNED
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED

Derived Data Type Routines

- **MPI_Type_contiguous**
  - The simplest constructor. Produces a new data type by making count copies of an existing data type.

MPI_Type_contiguous (count,oldtype,&newtype)

- **MPI_Type_vector**

  - Similar to contiguous, but allows for regular gaps (stride) in the displacements. MPI_Type_hvector is identical to MPI_Type_vector except that stride is specified in bytes.

MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)

- **MPI_Type_indexed**
  - An array of displacements of the input data type is provided as the map for the new data type. MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are specified in bytes.

MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)

Derived Data Type Routines

- **MPI_Type_struct**

  – The new data type is formed according to completely defined map of the component data types.

  MPI_Type_struct (count,blocklens[],offsets[],old_types,&newtype)

- **MPI_Type_extent**

  – Returns the size in bytes of the specified data type. Useful for the MPI subroutines that require specification of offsets in bytes.

  MPI_Type_extent (datatype,&extent)

Derived Data Type Routines

- **MPI_Type_commit**

  – Commits new datatype to the system. Required for all user constructed (derived) datatypes.

  MPI_Type_commit (&datatype)

- **MPI_Type_free**

  – Deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.

  MPI_Type_free (&datatype)

Examples: Contiguous Derived Data Type

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(argc,argv) int argc; char *argv[];
{ int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
   11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype rowtype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD,
   &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
if (numtasks == SIZE) { if (rank == 0) { for (i=0; i<numtasks; i++)
   MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD); }
MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
   rank,b[0],b[1],b[2],b[3]); } else printf("Must specify %d processors.
   Terminating.\n",SIZE); MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

Examples: Vector Derived Data Type

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4 int main(argc,argv) int argc; char *argv[];
{ int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
   11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype columntype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD,
   &numtasks);
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
   MPI_Type_commit(&columntype);
if (numtasks == SIZE) { if (rank == 0) { for (i=0; i<numtasks; i++)
MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD); }
MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
   rank,b[0],b[1],b[2],b[3]); } else printf("Must specify %d processors.
   Terminating.\n",SIZE); MPI_Type_free(&columntype);
MPI_Finalize();
}
```

Examples: Indexed Derived Data Type

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6
int main(argc,argv) int argc; char *argv[];
{ int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0,
   12.0, 13.0, 14.0, 15.0, 16.0};
float b[NELEMENTS]; MPI_Status stat;
MPI_Datatype indextype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD,
   &numtasks); blocklengths[0] = 4; blocklengths[1] = 2; displacements[0]
   = 5; displacements[1] = 12; MPI_Type_indexed(2, blocklengths,
   displacements, MPI_FLOAT, &indextype); MPI_Type_commit(&indextype); if
   (rank == 0) { for (i=0; i<numtasks; i++) MPI_Send(a, 1, indextype, i,
   tag,
MPI_COMM_WORLD); }
MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
   rank,b[0],b[1],b[2],b[3],b[4],b[5]); MPI_Type_free(&indextype);
MPI_Finalize();
}
```

Group and Communicator Management Routines

- **Groups vs. Communicators:**

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator that comprises all tasks is MPI_COMM_WORLD.

- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Group and Communicator Management Routines

- **Primary Purposes of Group and Communicator Objects:**
  - Allow you to organize tasks, based upon function, into task groups.
  - Enable Collective Communications operations across a subset of related tasks.
  - Provide basis for implementing user defined virtual topologies
  - Provide for safe communications

Group and Communicator Management Routines

Programming Considerations and Restrictions:

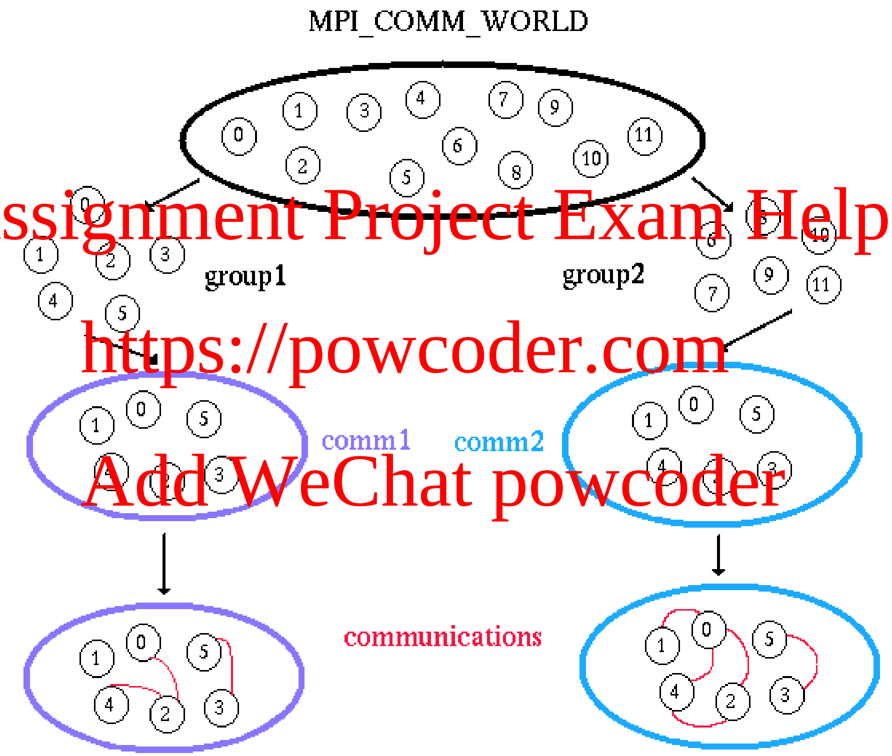
- Groups/communicators are dynamic - they can be created and destroyed during program execution.

- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.

- MPI provides over 40 routines related to groups, communicators, and virtual topologies.

- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

Group and Communicator Management Routines



MPI routines for communicator and groups

MPI Routines

- MPI includes routines for accessing information on groups or communicators, for creating new groups or communicators from existing ones, and for deleting groups or communicators. A list follows.

- Communicator creation routines are collective. They require all processes in the input communicator to participate, and may require communication amongst processes. All other group and communicator routines are local. As will be discussed later, it often makes sense to have all members of an input group call a group creation routine, if a communicator will later be created for that group.

MPI Group routines

Group Accessors

MPI_Group_size	returns number of processes in group
MPI_Group_rank	returns rank of calling process in group
MPI_Group_translate_ranks	translates ranks of processes in one group to those in another group
MPI_Group_compare	compares group members and group order

Group Constructors

MPI_Comm_group	returns the group associated with a communicator
MPI_Group_union	creates a group by combining two groups
MPI_Group_intersection	creates a group from the intersection of two groups
MPI_Group_difference	creates a group from the difference between two groups
MPI_Group_incl	creates a group from listed members of an existing group
MPI_Group_excl	creates a group excluding listed members of an existing group
MPI_Group_range_incl	creates a group according to first rank, *stride*, last rank
MPI_Group_range_excl	creates a group by deleting according to first rank, stride, last rank

MPI Group routines

Group Destructors

MPI_Group_free	marks a group for deallocation

Communicator Accessors

MPI_Comm_size	returns number of processes in communicator's group
MPI_Comm_rank	returns rank of calling process in communicator's group
MPI_Comm_compare	compares two communicatorsCommunicator Constructors
MPI_Comm_dup	duplicates a communicator
MPI_Comm_create	creates a new communicator for a group
MPI_Comm_split	splits a communicator into multiple, non-overlapping communicators

Communicator Destructors

MPI_Comm_free	marks a communicator for deallocation

Virtual Topologies

- **What Are These?**
- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.

Virtual topology

- **Why Use Them?**
- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Virtual topology

- **Example:**
- A simplified mapping of processes into a Cartesian virtual topology appears as follows:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

MPI Implementations

Platform	Implementations	Comments
IBM AIX	IBM MPI library	Thread-safe
Intel Linux	Quadrics MPI	Clusters with a switch. Not thread-safe
	MPICH	Clusters without a switch. On-node communications only. Not thread safe.
Opteron Linux	MVAPICH	Clusters with a switch. Not thread-safe
	MPICH	Clusters without a switch. On-node communications only. Not thread safe.
Mac OS X	Open MPI	Widely available

Summary

- What is the purpose of MPI?
  - MPI: Message Passing Interface
  - MPI is a ***specification*** for the developers and users of message passing libraries
- Reasons for using MPI?
  - **Standardization** - MPI is the only message passing library which can be considered a standard.
  - **Portability** - There is no need to modify your source code when you port your application to a different platform
  - **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
  - **Functionality** - Over 115 routines are defined in MPI-1 alone.
  - **Availability** - A variety of implementations are available, both vendor and public domain.
- MPI Communicator & Groups?
  - MPI uses objects called ***communicators*** and ***groups*** to define which collection of processes may communicate with each other
- Blocking MPI message routines.
  - **MPI_Send(), MPI_Recv(), MPI_Ssend(), MPI_Bsend()**
- Non-Blocking MPI message routines?
  - **MPI_Isend(), MPI_Irecv()**
- Collective communication routines?
  - **MPI_Scatter(), MPI_Gather(), MPI_Reduce(), MPI_Allreduce()**
- What are MPI derived data types?
  - Predefines its primitive data types:
  - MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types.
- What is a MPI virtual topology?
  - In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.