Message Passing Interface (MPI)

Umang Brahmakshatriya

ICSI 520

University At Albany

What is MPI?

- The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users

- The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs.

What is MPI?

- 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.

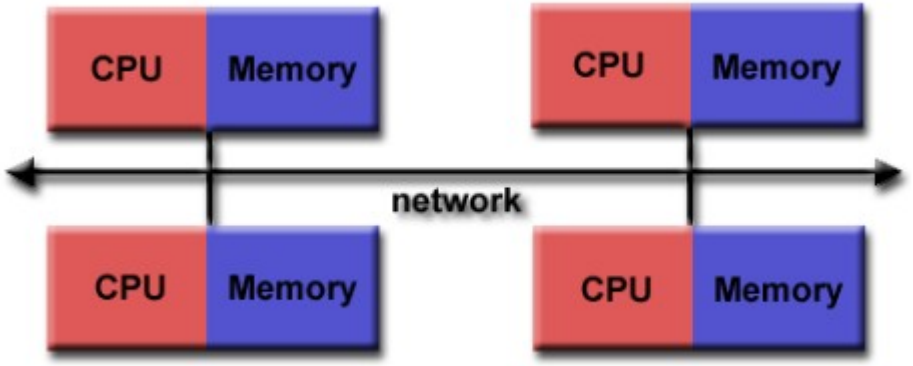
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

What is MPI?

- 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

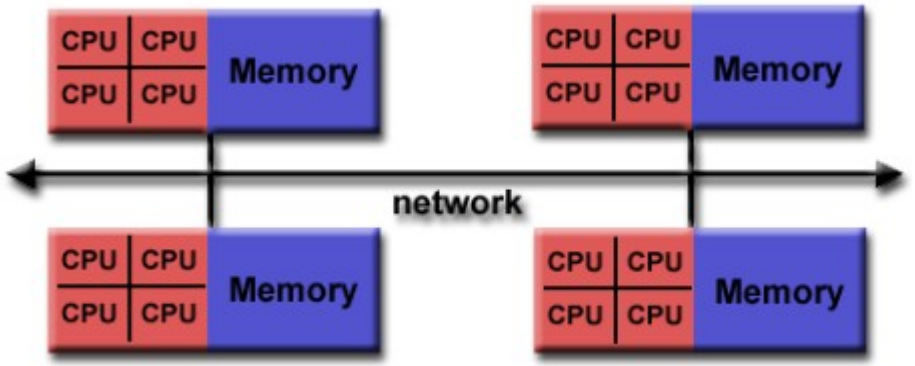
Programming model

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s)



Programming model

- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



Programming model

- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Why use MPI?

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is little or 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. Any implementation is free to develop optimized algorithms.

Why use MPI?

- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- **Availability** - A variety of implementations are available, both vendor and public domain.

Building MPI programs

- MPI compiler wrapper scripts are used to compile MPI programs - these should all be in your default $PATH unless you have changed it. These scripts mimic the familiar MPICH scripts in their functionality, meaning, they automatically include the appropriate MPI include files and link to the necessary MPI libraries and pass switches to the underlying compiler.
- If mpicc is not on your path on Ualbany HPC environment do this:
  - module load openmpi-x86_64
  - **OR if the above doesn't work** module load mpi/openmpi-x86_64

Building MPI programs

Language	Script Name	Underlying Compiler
C	mpicc	gcc
	mpigcc	gcc
	mpiicc	icc
	mpipgcc	pgcc
C++	mpiCC	g++
	mpig++	g++
	mpiicpc	icpc
	mpipgCC	pgCC

Running MPI jobs

- An MPI job can be simply run using mpirun after compiling
- A very simplistic command line for mpirun is as follows:
  - mpirun [-np X] [--hostfile <filename>] <program>
  - This will run X copies of <*program*> in your current run-time environment (if running under a supported resource manager, Open MPI's *mpirun* will usually automatically use the corresponding resource manager process starter, as opposed to, for example, *rsh* or *ssh*, which require the use of a hostfile, or will default to running all X copies on the localhost), scheduling (by default) in a round-robin fashion by CPU slot.
  - See https://www.open-mpi.org/doc/v1.8/man1/mpirun.1.php for more details.
  - Specifically, mpirun is a symbolic link to a common back-end launcher command named orterun (Open MPI's run-time environment interaction layer is named the Open Run-Time Environment, or ORTE -- hence orterun).

Running MPI jobs

- **-c, -n, --n, -np <#>**
  - Run this many copies of the program on the given nodes.
  - This option indicates that the specified file is an executable program and not an application context.
  - If no value is provided for the number of copies to execute (i.e., neither the "-np" nor its synonyms are provided on the command line), Open MPI will automatically execute a copy of the program on each process slot.

Running MPI jobs

- hostfile, --hostfile <hostfile>
  - Provide a hostfile to use.
  - The hostfile may contain the ip addresses or hostnames of the processors to use to execute copies of the program

- For more see
  - https://www.open-mpi.org/doc/v1.8/man1/mpirun.1.php

Running MPI jobs

- Ualbany uses SLURM http://slurm.schedmd.com/ to run MPI jobs
- MPI executables are launched using the SLURM srun or sbatch command with the appropriate options.
  - For example, to launch an 8-process MPI job split across two different nodes in the pdebug pool:
    - srun -n8 -ppdebug a.out
    - sbatch -n8 -p batch --wrap="mpirun /network/rit/home/ub532371/mpi/mpitest"
- More information here (we will discuss this more)
  - https://www.rit.albany.edu/wiki/Main_Page
- For this class to run mpi jobs we will use:
  - salloc and mpirun OR
  - sbatch

Running MPI jobs

- salloc
- salloc allows you to run a multi-step job interactively

```
-bash-4.1$ salloc -n 10
salloc: Granted job allocation 154704
bash-4.1$
```

- Can now run the job issuing command mpirun

```
-bash-4.1$ salloc -n 10
salloc: Granted job allocation 154704
bash-4.1$
bash-4.1$
bash-4.1$
bash-4.1$ mpirun -nolocal /network/rit/home/ub532371/mpi/mpitest
```

Running MPI jobs

- salloc (contd..)
- Once done, you can exit or run more jobs

```
bash-4.1$ mpirun -nolocal /network/rit/home/ub532371/mpi/mpitest
Number of tasks= 10 My rank= 2 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 9 Running on cc1-02.rit.albany.edu
Number of tasks= 10 My rank= 3 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 8 Running on cc1-02.rit.albany.edu
Number of tasks= 10 My rank= 1 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 4 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 5 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 6 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 0 Running on cc1-01.rit.albany.edu
Number of tasks= 10 My rank= 7 Running on cc1-01.rit.albany.edu
bash-4.1$ exit
exit
salloc: Relinquishing job allocation 154704
salloc: Job allocation 154704 has been revoked.
-bash-4.1$
```

Running MPI jobs

- sbatch

```
-bash-4.1$
-bash-4.1$ sbatch -n8 -p batch --wrap="mpirun /network/rit/home/ub532371/mpi/mpi
test"
Submitted batch job 154708
-bash-4.1$
```

- The output is piped into (written into the a file in the same directory)
  - slurm-<jobNumber>.out
  - In our case it is slurm-154708.out
  - Do a cat slurm-154708.out and see the results

```
-bash-4.1$ cat slurm-154708.out
Number of tasks= 8 My rank= 4 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 5 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 1 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 2 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 3 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 6 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 7 Running on cc1-01.rit.albany.edu
Number of tasks= 8 My rank= 0 Running on cc1-01.rit.albany.edu
-bash-4.1$
```

File system

- Do you need a common file system on all the nodes?
  - No
  - But have this would make your life easier
- Let us talk about the outputs and I/O
  - Files being created in an MPI program

MPI on the path

- If Open MPI was installed with a prefix of /opt/openmpi, then the following should be in your PATH and LD_LIBRARY_PATH
  - PATH: /opt/openmpi/bin
  - LD_LIBRARY_PATH: /opt/openmpi/lib

- If not on path / cannot set the path use prefix:

  mpirun --prefix /opt/openmpi -np 8 a.out

In summary to run the mpi job

- SPMD (single process multiple data) job
  - mpirun -np 4 theParallelApp
  - Can use a host file (if not using SLURM)

    cat my_hostfile

    host01.example.com

    host02.example.com

    mpirun --hostfile my_hostfile -np 4 theParallelApp
  - MPMD (multiple process multiple data) job
    - mpirun -np 2 a.out : -np 2 b.out
    - This will launch a single parallel application, but the first two processes will be instances of the a.out executable, and the second two processes will be instances of the b.out executable.
    - In MPI terms, this will be a single MPI_COMM_WORLD, but the a.out processes will be ranks 0 and 1 in MPI_COMM_WORLD, while the b.out processes will be ranks 2 and 3 in MPI_COMM_WORLD

In summary to run the mpi job (contd..)

- mpirun can also accept a parallel application specified in a file instead of on the command line. For example
  - mpirun --app myFile
- Where myFile contains:

  # parallel job, one per line. The first sub-application is the 2

  # a.out processes:

  -np 2 a.out

  # The second sub-application is the 2 b.out processes:

  -np 2 b.out

Can you run non-mpi jobs with mpirun

- YES
- Open MPI's mpirun is actually a synonym for the underlying launcher named orterun (i.e., the Open Run-Time Environment layer in Open MPI, or ORTE).
- So you can use mpirun to launch any application

mpirun -np 2 --host a,b uptime

This will launch a copy of the unix command uptime on the hosts a and b.

Hostfile

- The --hostfile option to mpirun takes a filename that lists hosts on which to launch MPI processes.
- Hostfiles are simple text files with hosts specified, one per line.
- Each host can also specify a default a maximum number of slots to be used on that host (i.e., the number of available processors on that host).
- Comments are also supported, and blank lines are ignored.
- Example next slide

Hostfile – example file contents

```
# This is an example hostfile.  Comments begin with #
#
# The following node is a single processor machine:
foo.example.com

# The following node is a dual-processor machine:
bar.example.com slots=2

# The following node is a quad-processor machine, and we absolutely
# want to disallow over-subscribing it:
wow.example.com slots=4 max-slots=4
```

Hostfile - working

- Lets us assume that a SLURM job contains hosts node01 through node04. If you run:

cat my_hosts

node03

mpirun -np 1 --hostfile my_hosts hostname

- Where will this run?
  - This will run a single copy of hostname on the host node03.

Hostfile - working

- However, similar to the previous slide
  - Lets us assume that a SLURM job contains hosts node01 through node04 again. If you run:

```
cat my_hosts
  node17
mpirun -np 1 --hostfile my_hosts hostname
```

- Where will this run?
  - This is an error (because node17 is not provisioned for you by SLURM); mpirun will abort.
- Tip: use squeue command to get the hosts allocated by SLURM

Controlling how processes are run

- If you are not oversubscribing your nodes (i.e., trying to run more processes than you have told Open MPI are available on that node), scheduling is pretty simple and occurs either on a by-slot or by-node round robin schedule.
- --byslot flag
- --bynode flag

```
cat my-hosts
 node0 slots=2 max_slots=20
 node1 slots=2 max_slots=20
mpirun --hostfile my-hosts -np 8 --byslot hello | sort
 Hello World I am rank 0 of 8 running on node0
 Hello World I am rank 1 of 8 running on node0
 Hello World I am rank 2 of 8 running on node1
 Hello World I am rank 3 of 8 running on node1
 Hello World I am rank 4 of 8 running on node0
 Hello World I am rank 5 of 8 running on node0
 Hello World I am rank 6 of 8 running on node1
 Hello World I am rank 7 of 8 running on node1
```

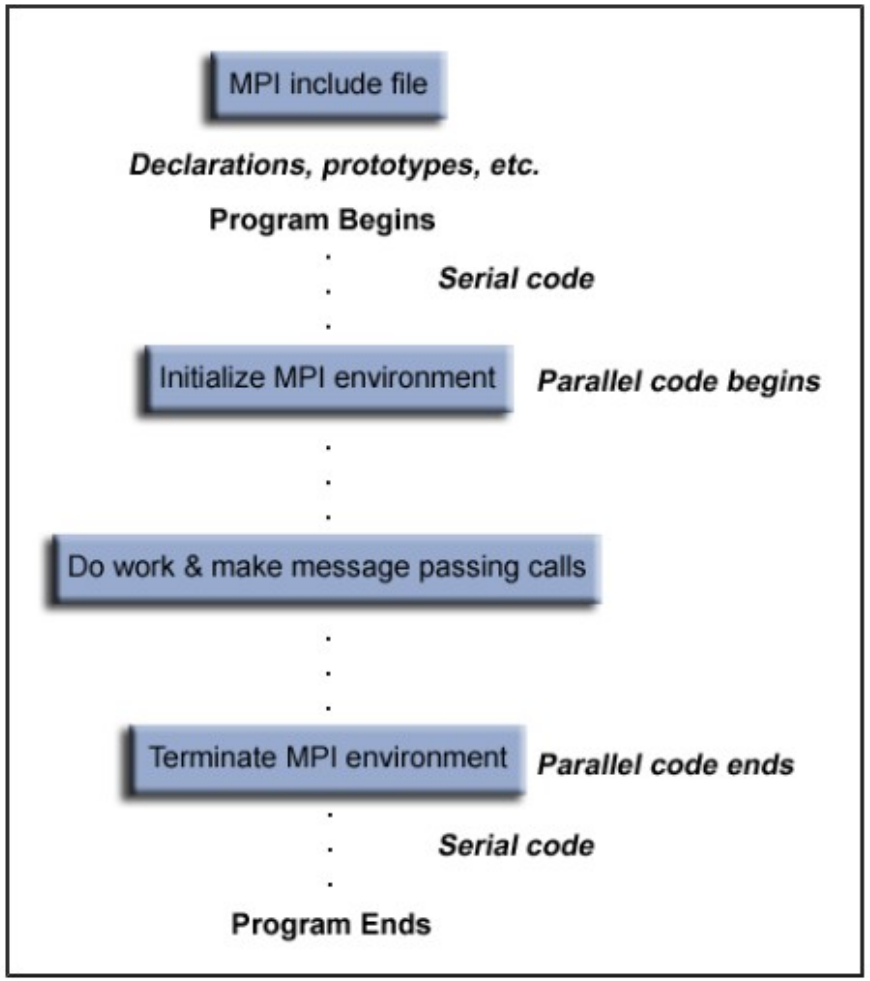
```
cat my-hosts
 node0 slots=2 max_slots=20
 node1 slots=2 max_slots=20
mpirun --hostname my-hosts -np 8 --bynode hello | sort
 Hello World I am rank 0 of 8 running on node0
 Hello World I am rank 1 of 8 running on node1
 Hello World I am rank 2 of 8 running on node0
 Hello World I am rank 3 of 8 running on node1
 Hello World I am rank 4 of 8 running on node0
 Hello World I am rank 5 of 8 running on node1
 Hello World I am rank 6 of 8 running on node0
 Hello World I am rank 7 of 8 running on node1
```

Degraded and aggressive mode

- Degraded: When Open MPI thinks that it is in an oversubscribed mode (i.e., more processes are running than there are processors available), MPI processes will automatically run in degraded mode and frequently yield the processor to its peers, thereby allowing all processes to make progress

- Aggressive: When Open MPI thinks that it is in an exactly- or under-subscribed mode (i.e., the number of running processes is equal to or less than the number of available processors), MPI processes will automatically run in aggressive mode, meaning that they will never voluntarily give up the processor to other processes. This means that Open MPI will spin in tight loops attempting to make message passing progress.

MPI program structure



Header file

- Required for all programs that make MPI library calls.

C include file
`#include "mpi.h"`

MPI calls

- C names are case sensitive
- Programs must not declare variables or functions with names beginning with the prefix MPI_ or PMPI_ (profiling interface).

C Binding	
Format:	`rc = MPI_Xxxxx(parameter, ... )`
Example:	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Error code:	Returned as "rc". MPI_SUCCESS if successful

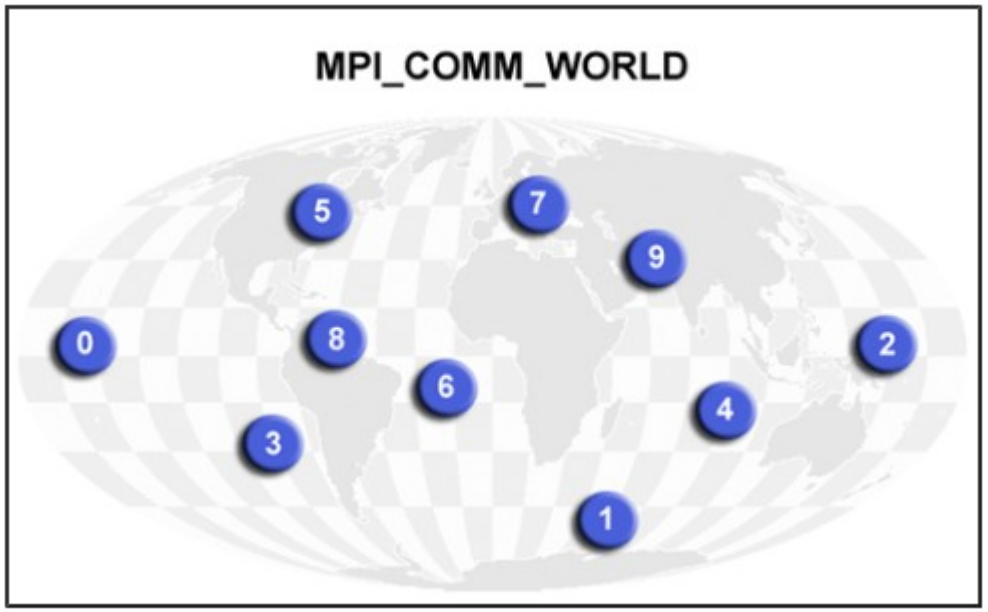
MPI communicators and groups

- 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.

MPI communicators and groups

- 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.



MPI 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.
- This is 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).

MPI error handling

- Most MPI routines include a return/error code parameter
- However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error.
- This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).
- Also, the types of errors displayed to the user are implementation dependent.

MPI environment management routines

- This group of routines is used for interrogating and setting the MPI execution environment, and covers an assortment of purposes, such as initializing and terminating the MPI environment, querying a rank's identity, querying the MPI library's version, etc.
- We will go over the most of the commonly used ones

MPI environment management routines

- MPI_Init(&argc,&argv);
- 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, 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 environment management routines

- MPI_Comm_size(comm,&size);
- Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD.
- If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

MPI environment management routines

- MPI_Comm_rank(comm,&rank);
- Returns the rank of the calling MPI process within the specified communicator.
- Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID.
- If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

MPI environment management routines

- MPI_Abort(comm,errorcode);
- Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

MPI environment management routines

- MPI_Get_processor_name (&name,&resultlength)
- 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 environment management routines

- MPI_Get_version (&version,&subversion)
- Returns the version and subversion of the MPI standard that's implemented by the library.

MPI environment management routines

- MPI_Initialized (&flag)
- 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 environment management routines

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

MPI environment management routines

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

MPI environment management routines

- 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.

Lab 7

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

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

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);
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);

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

MPI_Finalize();
}
```

Lab 7

- Login to Uabany cluster environment
- Setup your mpi environment module load openmpi-x86_64
- Create a file called mpitest.c and write the C code in it
- Compile the mpi program into an executable called mpitest mpicc mpitest.c -o mpitest
- Run the program in the following two ways (use 8 nodes):

1. Using mpirun after doing salloc –n 8
    - mpirun /network/rit/home/ub532371/mpi/mpitest
2. Using sbatch
    - sbatch –n8 -p batch --wrap="mpirun /network/rit/home/ub532371/mpi/mpitest“

Point to Point Communication Routines

- The value of PI can be calculated in a number of ways. Consider the following method of approximating PI
  - Inscribe a circle in a square
  - Randomly generate points in the square
  - Determine the number of points in the square that are also in the circle
  - Let r be the number of points in the circle divided by the number of points in the square
  - PI ~ 4 r
  - Note that the more points generated, the better the approximation

Point to Point Communication Routines

- Serial pseudo code for this procedure (PI):

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

Point to Point Communication Routines

- Leads to an "embarassingly parallel" solution:
  - Break the loop iterations into chunks that can be executed by different tasks simultaneously.
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
  - Master task receives results from other tasks **using send/receive point-to-point operations**.

Point to Point Communication Routines

- Pseudo code solution: **red** highlights changes for parallelism.

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```

Key Concept: Divide work between available tasks which communicate data via point-to-point message passing calls.

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

Types of Point to Point Operations

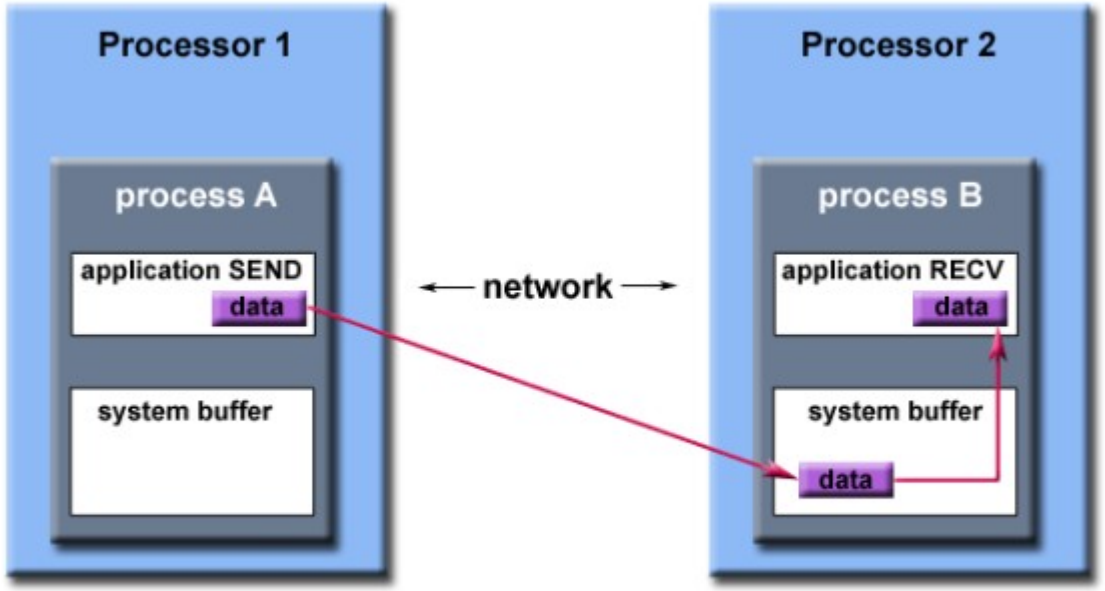
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

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"?

Buffering

- 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. For example:



Path of a message buffered at the receiving process

Buffering

- 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.

Blocking vs Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

Blocking vs Non-blocking

- 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 handshaking 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.

Blocking vs Non-blocking

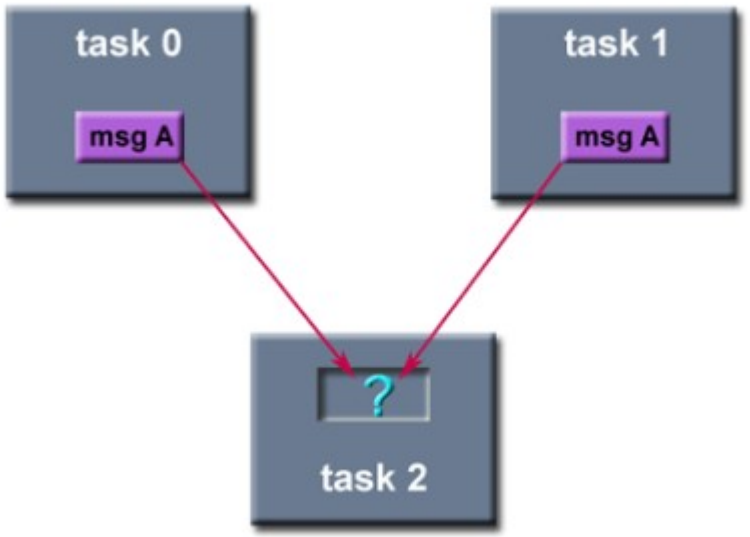
- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "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.

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.

Order and Fairness

- 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.



MPI Message Passing Routine Arguments

- MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

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.

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 is obtainable from Status via the MPI_Get_count routine.

MPI Message Passing Routine Arguments

- 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.
  - This argument is a pointer to a predefined structure MPI_Request.

MPI Message Passing Routine Arguments

- MPI Data Types

C Data Types	
MPI_CHAR	signed char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	int8_t int16_t int32_t int64_t
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI 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.
  - **MPI_Send (&buf,count,datatype,dest,tag,comm)**

MPI Blocking message passing routines

- 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)**

MPI 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 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)**

MPI Blocking message passing routines

- MPI_Probe
  - Performs a blocking test for a message. (Test if the message is ready)
  - 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.
  - **MPI_Probe (source,tag,comm,&status)**
- MPI_Get_count
  - **MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)**
  - status - return status of receive operation (Status)
  - datatype - datatype of each receive buffer element (handle)
  - count - number of received elements (integer)

Lab 8

- Task 0 pings task 1 and awaits return ping
- See how send and receive routines are implemented and called
- Observations?
  - What happens if you use more than 2 processors for this mpi job?
  - What do you see in your output and why?

```
#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();
}
```

Lab 9 – probe and ta

- Observe how probe works
- Observe how tags work

- Submit binary and code

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SENDRECV_NUMS_TAG 0

int main(int argc, char** argv) {
  MPI_Init(NULL, NULL);

  int comm_size;
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  if (comm_size != 2) {
    fprintf(stderr, "Must use two processes for this example\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int numbersToSend;
  if (rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
    /* Pick a random amount of integers to send to process one */
    srand(time(NULL));
    numbersToSend = (rand() / (float)RAND_MAX) * MAX_NUMBERS;
    /* Send the amount of integers to process one */
    /* See the use of tag */
    MPI_Send(numbers, numbersToSend, MPI_INT, 1, SENDRECV_NUMS_TAG, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", numbersToSend);
  } else if (rank == 1) {
    MPI_Status status;
    /* Probe for an incoming message from process zero */
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
    /* When probe returns, the status object has the size and other
       attributes of the incoming message. Get the size of the message. */
    MPI_Get_count(&status, MPI_INT, &numbersToSend);
    /* Allocate a buffer just big enough to hold the incoming numbers */
    int* number_buf = (int*)malloc(sizeof(int) * numbersToSend);
    /* Now receive the message with the allocated buffer */
    MPI_Recv(number_buf, numbersToSend, MPI_INT, 0, SENDRECV_NUMS_TAG, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
           numbersToSend);
    free(number_buf);
  }
  MPI_Finalize();
}
```

MPI 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 indicate that the non-blocking send has completed.
  - **MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)**

MPI Non-Blocking message passing routines

- 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 application buffer.
  - A communication request handle is returned for handling the pending message status.
  - The program must use calls to MPI_Wait 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)**

MPI 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)

MPI 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_offsets, &array_of_statuses)

  - **This is non-blocking compared to MPI_Wait**

MPI 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.
  - **MPI_Iprobe (source,tag,comm,&flag,&status)**

  - **MPI_Probe will block until the message from specified source has been received**

Lab 10

- Observations?
  - What are all the tasks doing?
- Add code after MPI_Waitall to show who sends messages to whom
  - Print the messages received from source at the destination

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

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);


MPI_Finalize();
}
```

Lab 11

- Create a new mpi C program called lab11Mpi.c
- Initialize the MPI environment
- After the master task has printed the number of tasks, but before MPI_Finalize
  - Have each task determine a unique partner task to send/receive with. One easy way to do this:

    if (taskid < numtasks/2) then partner = numtasks/2 + taskid
    else if (taskid >= numtasks/2) then partner = taskid - numtasks/2

  - Each task sends its partner a single integer message: its taskid
  - Each task receives from its partner a single integer message: the partner's taskid
  - For confirmation, after the send/receive, each task prints something like "Task ## is partner with ##" where ## is the taskid of the task and its partner.
  - Compile, run, see if you get proper output
  - First use blocking calls then use non-blocking calls
    - Submit two different source files and two binaries – one for blocking and one for non-blocking

Collective Communication Routines

- Collective communication routines must involve **all** processes within the scope of a communicator
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations

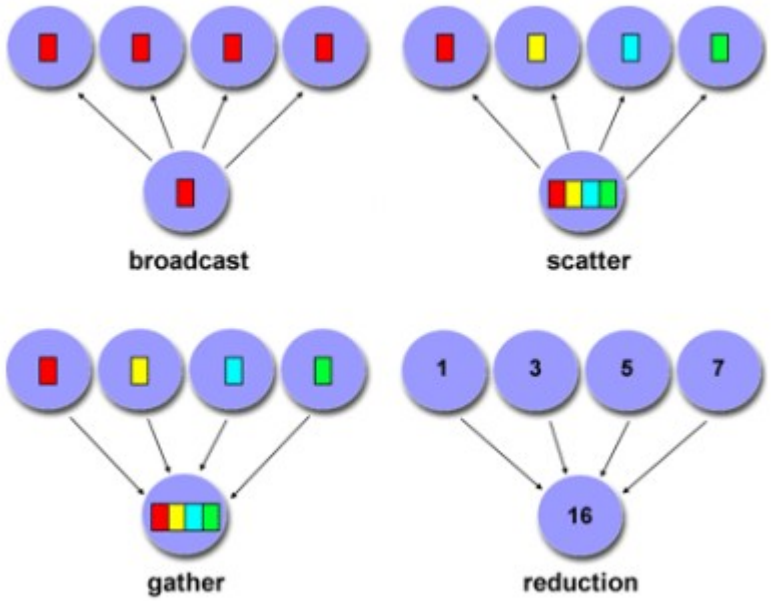
Collective Communication Routines

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 Routines

Types of Collective Operations



Collective Communication Routines

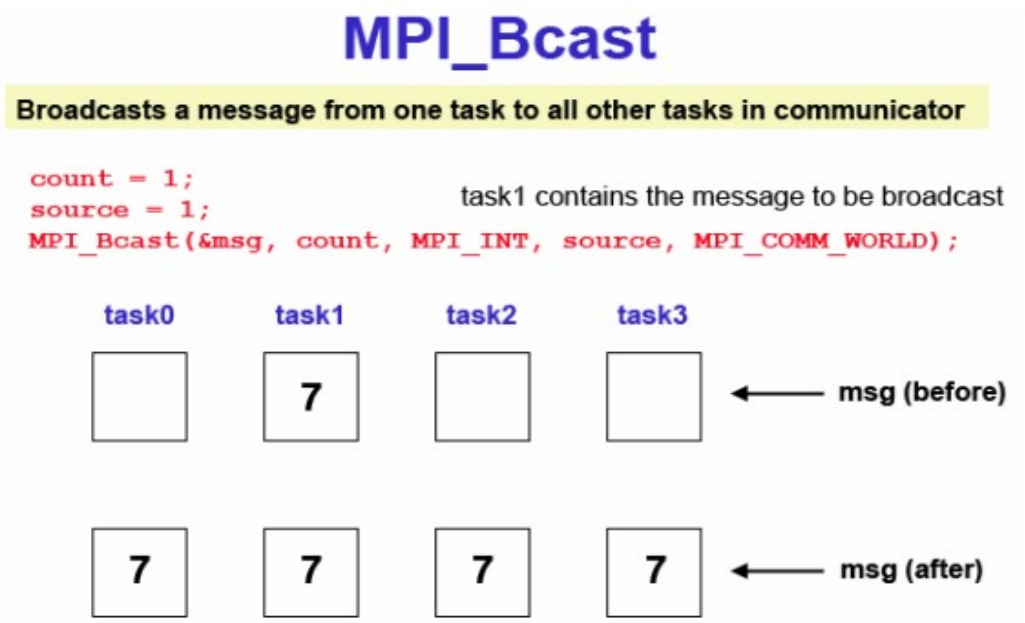
MPI_Barrier MPI_Barrier (comm)

- Synchronization operation.
- 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.
- Then all tasks are free to proceed.

Collective Communication Routines

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

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

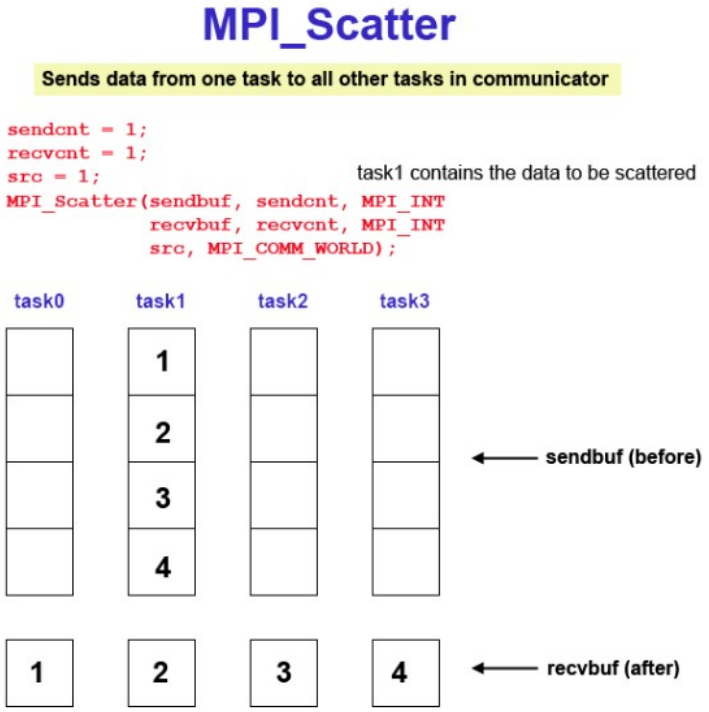


Collective Communication Routines

MPI_Scatter

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

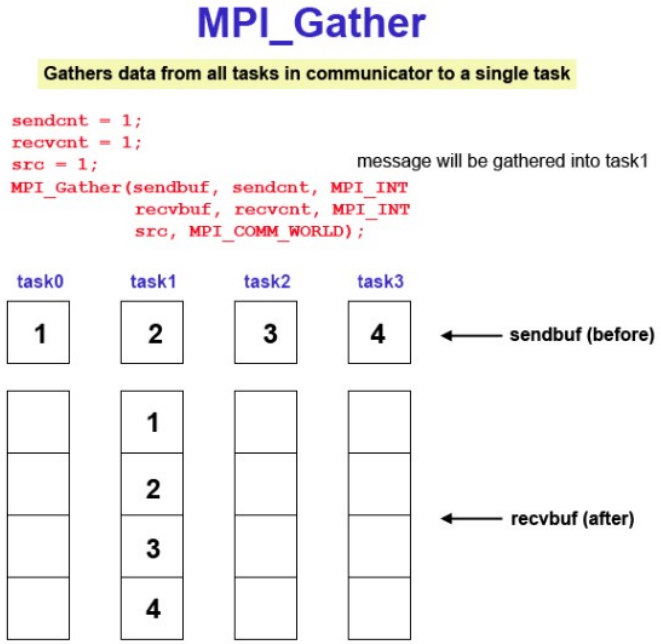
- Data movement operation.
- Distributes distinct messages from a single source task to each task in the group.



Collective Communication Routines

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

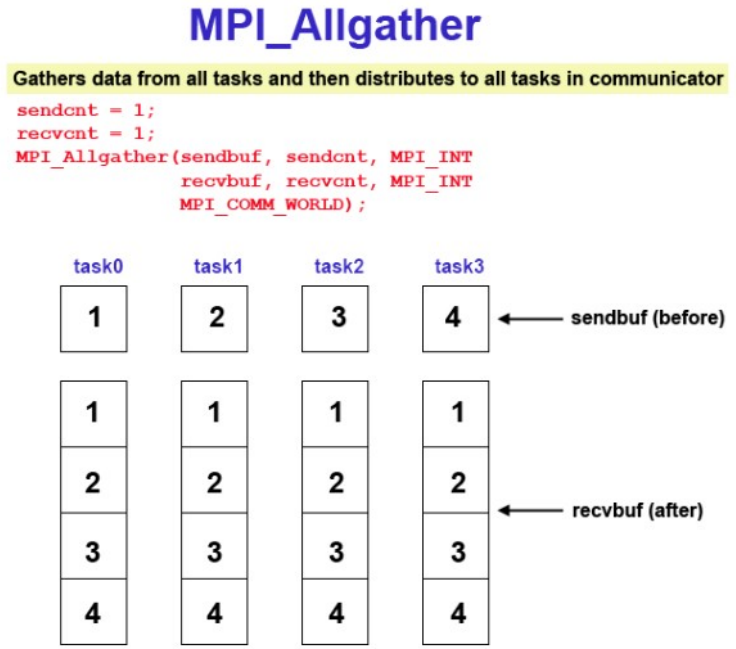
- Data movement operation.
- Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.



Collective Communication Routines

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

- Data movement operation.
- 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.

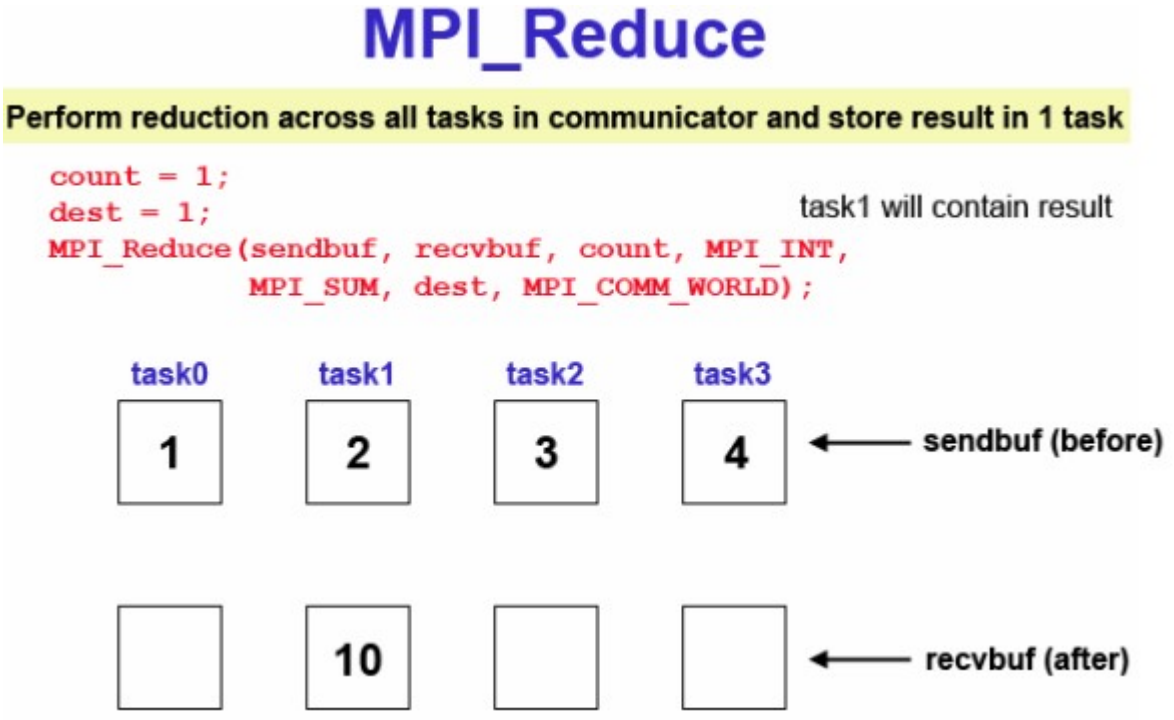


Collective Communication Routines

MPI_Reduce

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

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



Collective Communication Routines

MPI_Reduce: The predefined MPI reduction operations appear below.

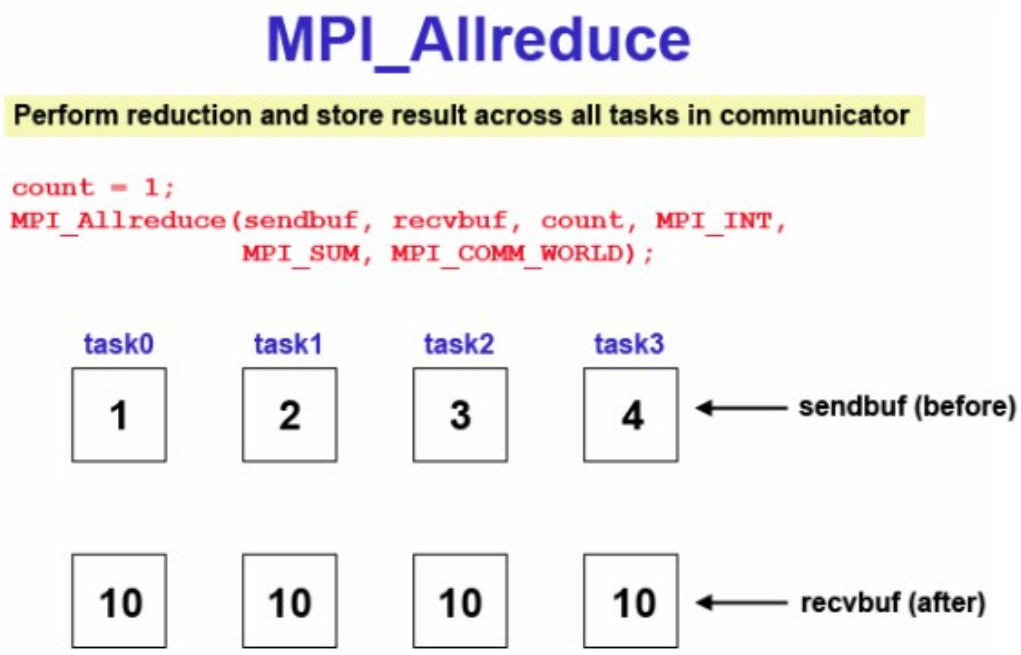
MPI Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

Collective Communication Routines

MPI_Allreduce

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

- Collective computation operation + data movement.
- 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.



Collective Communication Routines

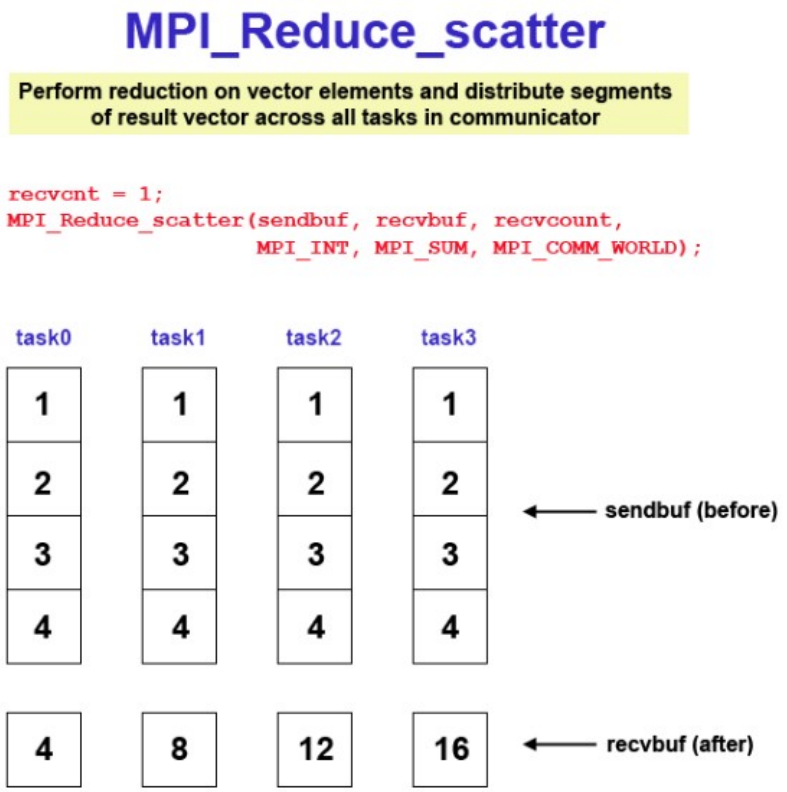
MPI_Reduce_scatter

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

- Collective computation operation + data movement.
- 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.

Collective Communication Routines

MPI_Reduce_scatter



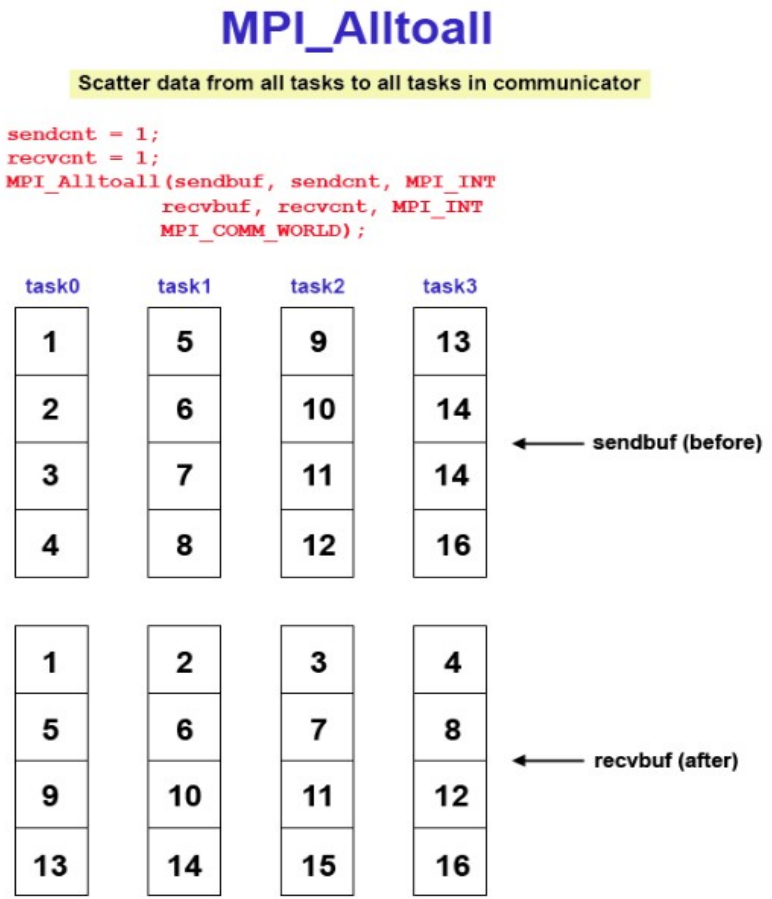
Collective Communication Routines

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

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

Collective Communication Routines

MPI_Alltoall

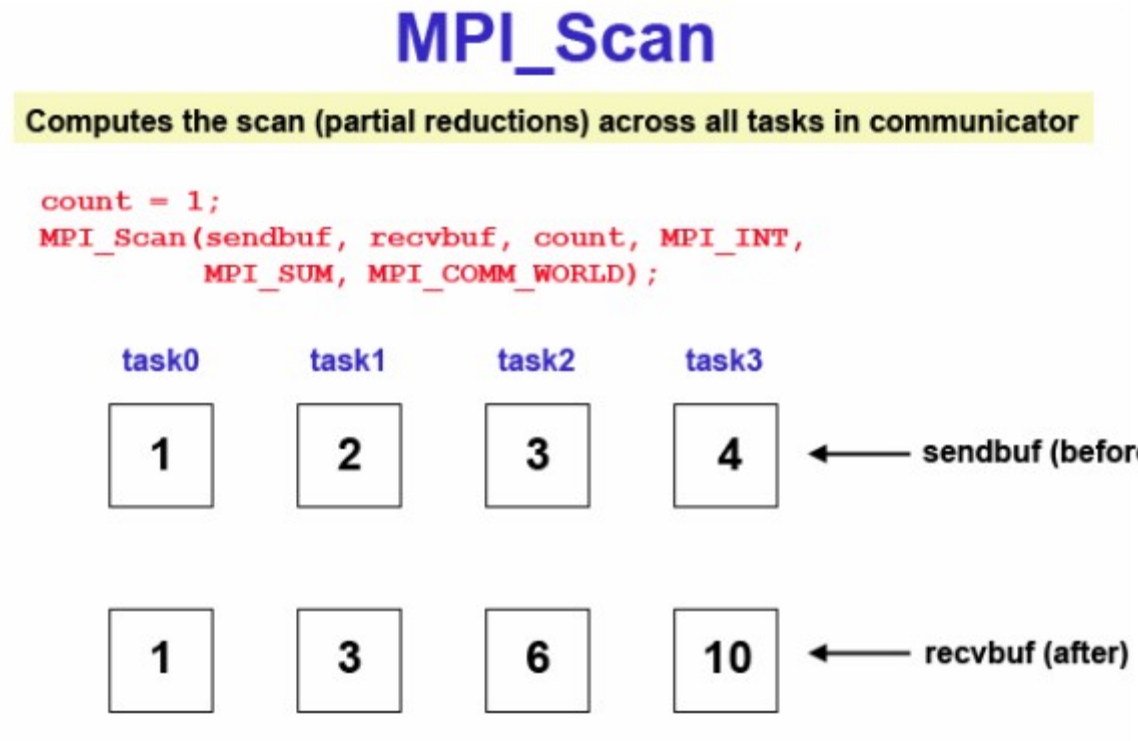



```
rank    send buf                          recv buf
----    --------                          --------
 0      a,b,c          MPI_Alltoall       a,A,#
 1      A,B,C        ---------------->    b,B,@
 2      #,@,%                             c,C,%

(a more elaborate case with two elements per process)

rank    send buf                          recv buf
----    --------                          --------
 0      a,b,c,d,e,f    MPI_Alltoall       a,b,A,B,#,@
 1      A,B,C,D,E,F  ---------------->    c,d,C,D,%,$
 2      #,@,%,$,&,*                       e,f,E,F,&,*
```

Collective Communication Routines

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

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



Sending a struct

- See example on blackboard
- Remember a few things
  - Easiest way to use MPI_BYTE (if you are not dynamically allocating the elements of the struct)
    - Need to know how big the struct is
  - If you are dynamically allocation the elements of the struct
    - Send the struct first
    - Send the dynamically allocated elements next
    - Always remember you have to allocate the same amount of memory on the receiving side so you need to know how much you are sending and how much you are receiving

Lab 12

- Perform a scatter operation on the rows of an array
- Run this program with 4 processors
- OR change the SIZE definition to whatever #processors you wish to run with and recompile

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(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();
}
```

Lab 13

- Perform a simple broadcast
- Send an integer to all processors
- Observations?

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


int main(int argc, char *argv[])  {
  int numtasks, rank;

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

  if(rank==0) //master node
  {
      int broadcastMsg;
      broadcastMsg = 5000;
      MPI_Bcast(&broadcastMsg, 1, MPI_INT, rank, MPI_COMM_WORLD);
      printf("\nProcessor %d : sent message as broadcast (MPI_INT): %d\n",rank, broadcastMsg);
  }

  if(rank!=0)
  {
      int receiveBroadcastedMsg = 0;
     MPI_Bcast(&receiveBroadcastedMsg, 1, MPI_INT, 0, MPI_COMM_WORLD);     // notice source = 0
     printf("\nProcessor %d : received message: %d\n",rank, receiveBroadcastedMsg);
  }

  MPI_Finalize();
}
```

Lab 14

- Perform a simple gather
- What is the program doing?
- Observations?

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


int main(int argc, char *argv[])  {
  int numtasks, rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  int sendBuf[] = {-1,-2};

  sendBuf[0] = rank + 100;
  sendBuf[1] = rank + 10;

  // 2 times the procs because each sends 2 ints
  int receiveBufSize = numtasks*2;

  int receiveBuf[receiveBufSize];
  int i=0;
  //initialize the receiveBuf
  for(i=0;i<receiveBufSize;i++)
  {
      receiveBuf[i] = 0;
  }
  MPI_Gather(sendBuf, 2, MPI_INT, // sending 2 ints
             receiveBuf, 2, MPI_INT, //gathering 2 ints from each
             0, MPI_COMM_WORLD); // master processor will do the gather
  if(rank==0) //master can now print the result
  {

    for(i=0;i<receiveBufSize;i++){
        printf("\nReceived buf[%d] = %d\n", i,receiveBuf[i]);
    }
  }

  MPI_Finalize();
}
```

Lab 15

- Perform an all gather
- What changed?
- Observations?

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


int main(int argc, char *argv[])  {
  int numtasks, rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  int sendBuf[] = {-1,-2};

  sendBuf[0] = rank + 100;
  sendBuf[1] = rank + 10;

  // 2 times the procs because each sends 2 ints
  int receiveBufSize = numtasks*2;

  int receiveBuf[receiveBufSize];
  int i=0;
  //initialize the receiveBuf
  for(i=0;i<receiveBufSize;i++)
  {
      receiveBuf[i] = 0;
  }
  MPI_Allgather(sendBuf, 2, MPI_INT, // sending 2 ints
                receiveBuf, 2, MPI_INT, //gathering 2 ints from each
                 MPI_COMM_WORLD); //all will do the gather
  if(rank==1) //now any one can print
  {
    printf("\nAt processor %d\n",rank);
    for(i=0;i<receiveBufSize;i++){
        printf("\nReceived buf[%d] = %d\n", i,receiveBuf[i]);
    }
  }

  MPI_Finalize();
}
```

Lab 16

- Perform a reduce
- Reduce a distributed array containing at least 2 elements
- Observations?

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


int main(int argc, char *argv[])  {
  int numtasks, rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  int sendBuf[] = {-1,-2};

  sendBuf[0] = rank + 100;
  sendBuf[1] = rank + 10;

  // 2 times the procs because each sends 2 ints
  int receiveBufSize = 2;

  int receiveBuf[receiveBufSize];
  int i=0;
  //initialize the receiveBuf
  for(i=0;i<receiveBufSize;i++)
  {
      receiveBuf[i] = 0;
  }
  MPI_Reduce(sendBuf, receiveBuf, receiveBufSize, MPI_INT, // sending (receiveBufSize) ints
             MPI_SUM, 0, MPI_COMM_WORLD); //master will contain the reduced result
  if(rank==0) //master can print
  {
    printf("\nAt processor %d\n",rank);
    for(i=0;i<receiveBufSize;i++){
       printf("\nReceived buf[%d] = %d\n", i,receiveBuf[i]);
    }
  }

  MPI_Finalize();
}
```

Lab 17

- Perform an all reduce
- Reduce a distributed array containing at least 2 elements
- Observations?
- Differences from previous?

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


int main(int argc, char *argv[])  {
  int numtasks, rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  int sendBuf[] = {-1,-2};

  sendBuf[0] = rank + 100;
  sendBuf[1] = rank + 10;

  // 2 times the procs because each sends 2 ints
  int receiveBufSize = 2;

  int receiveBuf[receiveBufSize];
  int i=0;
  //initialize the receiveBuf
  for(i=0;i<receiveBufSize;i++)
  {
      receiveBuf[i] = 0;
  }
  MPI_Allreduce(sendBuf, receiveBuf, receiveBufSize, MPI_INT, // sending (receiveBufSize) ints
                MPI_SUM, MPI_COMM_WORLD); //all processors will contain the reduced result
  if(rank==2) //any one can print - let the proc # 2 print
  {
    printf("\nAt processor %d\n",rank);
    for(i=0;i<receiveBufSize;i++){
        printf("\nReceived buf[%d] = %d\n", i,receiveBuf[i]);
    }
  }

  MPI_Finalize();
}
```

Lab 18

- Perform a reduce scatter
- Observations?
- Notice the size of send buffer?

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

int main(int argc, char *argv[]) {
    int rank, size, i, n;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
   // we have to have an array of size = #procs
   //because if the size is smaller, one of the procs will receive
   //garbage values
    int sendbuf[size];
    int recvbuf;

    for (i=0; i<size; i++)
        sendbuf[i] = rank + i;
/*
    printf("Proc %d: ", rank);
    for (i=0; i<size; i++) printf("%d ", sendbuf[i]);
    printf("\n");
*/
    //this tells the size to receive
    //because we have a 1D array we initialize all to be 1
    int recvcounts[size];
    for (i=0; i<size; i++)
        recvcounts[i] = 1;

    MPI_Reduce_scatter(sendbuf, &recvbuf, recvcounts, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    printf("Proc %d: %d\n", rank, recvbuf);

    MPI_Finalize();

    return 0;
}
```

Lab 19

- Perform an all to all operation
- Observations?
- Can you use this to do a distributed matrix transpose?

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

int main(int argc, char *argv[]) {
    int rank, size, i, n;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
   // we have to have an array of size = #procs
   //because if the size is smaller, one of the procs will receive
   //garbage values
    int sendbuf[size];
    int recvbuf[size];
    int whoPrints = 1;

    for (i=0; i<size; i++)
        sendbuf[i] = 1+ i + 4*rank;

    if(rank == whoPrints){
        printf("\nAt proc %d ",rank);
        for(i=0;i<size;i++)
            printf("%d ",sendbuf[i]);
    }

    MPI_Alltoall(sendbuf, 1, MPI_INT,
                 recvbuf, 1, MPI_INT,
                 MPI_COMM_WORLD);


    if(rank == whoPrints)
    {
        printf("\nAt proc %d: ",rank);
        for(i=0;i<size;i++)
            printf("%d ",recvbuf[i]);
    }
    MPI_Finalize();

    return 0;
}
```

Lab 20

- Perform an scan operation
- Observations?

- Change this to calculate factorials

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <mpi.h>

int main (int argc, char *argv [])
{
  int output;
  int input;
  int rank;

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

  input = rank;
  MPI_Scan (&input, &output, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  printf ("Process %d: Factorial %d\n", rank, output);
  MPI_Finalize ();


}
```