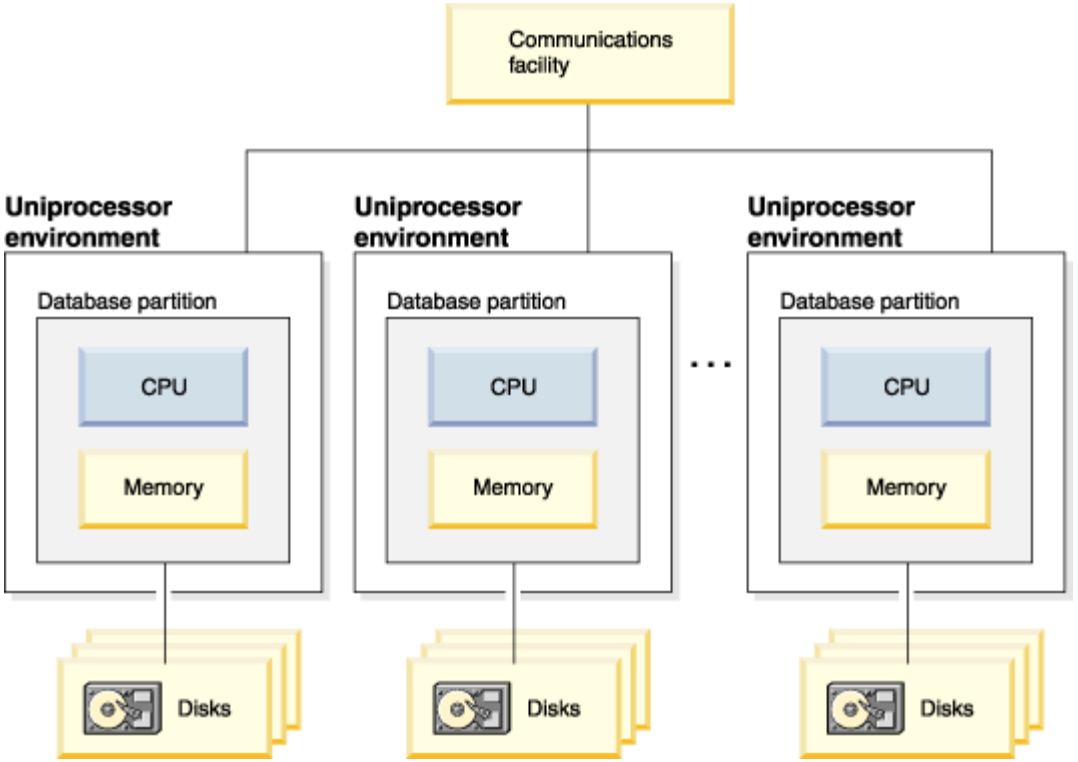
Lecture 3 Message-Passing Programming Using MPI (Part 1)

What is MPI

Message-Passing Interface (MPI)

- Message-Passing is a communication model used on distributed-memory architecture
- MPI is not a programming language (like C, Fortran 77), or even an extension to a language. It is a library that compilers (like cc, f77) uses.
- MPI is a standard that specifies the message-passing libraries supporting parallel programming in C/C++ or Fortran.
- The communication network is opaque to users.
- http://www.mpi-forum.org
    - 1989, first message-passing library called Parallel Virtual Machine (PVM) was written at ORNL.
    - 1993, version 3 of PVM was released.
    - 1994, first version of MPI released by MPI Forum.
    - 1997, MPI-2.0, 2008, MPI-2.1
        - Added I/O and one-sided concepts
    - 2009, MPI-2.2
        - Bug fixes
    - 09/21/2012, MPI-3.0

Message-passing model



This model assumes that the underlying hardware is a collection of processors, each with its own local memory, and an interconnection network supporting message-passing between processors.

MPI Features

- Distributed-memory cluster and multi-processor shared-memory platform support
- Support for virtual process topologies
- Fixed number of available processes during execution ?
- Initial processor allocation and binding to physical processors and interprocessor hardware communication are left to vendor implementation
- Explicit shared-memory operation, I/O functions and task management are not specified in the standard ?
- Designed to provide a **portable** parallel programming interface for:
  - End users
  - Library writers
  - Tool developers

To Learn More about MPI

- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
- http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/

Books

Using MPI, by William Gropp, Ewing Lusk, and Anthony Skjellum

MPI Annotated Reference Manual, by Marc Snir, *et al*

Based on MPI-1 Standards doc. and is almost identical

Designing and Building Parallel Programs, an Foster

Parallel Programming with MPI, Peter Pacheco

High Performance Computing, 2nd Ed., Dowd and Severence

MPI on Linux clusters:

–MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)

–LAM (http://www.lam-mpi.org/)

MPICH

MPICH is a freely available, high-performance and portable implementation of MPI.
(http://wiki.mcs.anl.gov/mpich2/index.php/Frequently_Asked_Questions#Q:_What_are_process_managers.3F)

MPICH2 is an all-new implementation of MPI, designed to support research into high-performance implementations of MPI-1 and MPI-2 functionality. In addition to the features in MPICH, MPICH2 includes support for one-side communication, dynamic processes, intercommunicator collective operations, and expanded MPI-IO functionality. Clusters consisting of both single-processor and SMP nodes and running

- Linux
- FreeBSD
- WinNT
- Solaris

are supported.

Basic Needs in parallel programming

In order to do parallel programming, we need basic functionality:

-- Start Processes

-- Send Messages

-- Receive Messages

-- Synchronize Processes

-- Terminate Processes

MPI Basic Functions

- MPI_Init() – Initiate a MPI computation
- MPI_Finalize() – Terminate a computation
- MPI_Comm_size() – Determine number of processes
- MPI_Comm_rank() – Determine a process's ID number
- MPI_Send() – Send a message
- MPI_Recv() – Receive a message

hello.c

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]) {
    int         my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    printf("Hello from node %d\n", my_rank);
    MPI_Finalize();/* Shut down MPI */
}
```

Compiling and execution on CRC

- module load mpich2/1.4.1-intel /* to load proper libraries and set up environment in CRC */
- mpicc –o hello hello.c
- mpiexec -np 4 ./hello

Execution

- mpiexec -np 4 ./hello

Parallel Programming Environment:

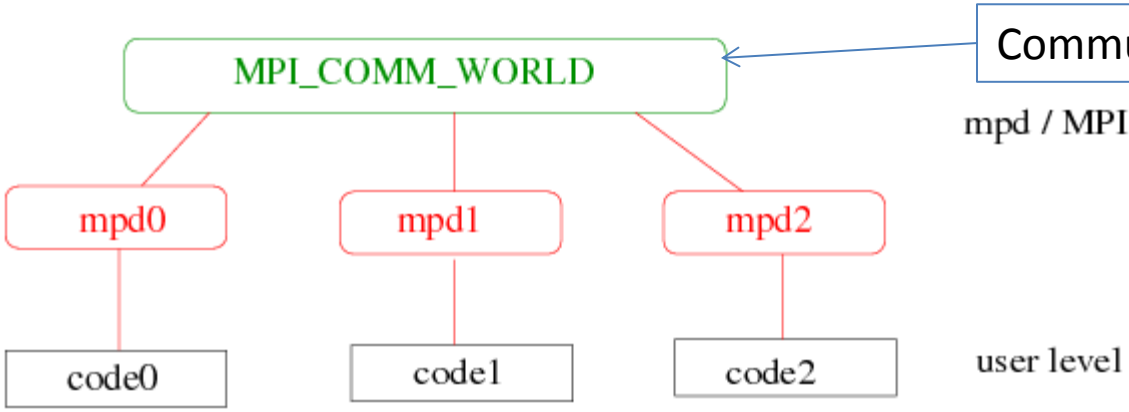
- *Job launcher*: decides what resources a parallel job consisting of multiple processes will run on. ”mpiexec” is used to initialize a parallel job from within a portable batch system (PBS) or other interactive environment. Mpiexec uses the task manager library of PBS to spawn copies of the executable on the nodes in a PBS allocation.
- *Other process manager* ((MPI process manager daemon)mpd (till the 1.2.x release series), hydra (default process manager for MPICH2 (Starting the 1.3.x series) with user interface mpiexec or mpiexec.hydra), smpd): starts and terminates processes and provide them with a number of services
  - http://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework
- *Parallel library*: MPI
- Integration of MPICH and SGE (Sun Grid Engine)
  - http://gridscheduler.sourceforge.net/howto/mpich2-integration/mpich2-integration.html

MPI2 Model

- The MPI2 software model consists in first establishing a virtual machine, the communication ring, within a subset of the physical nodes of the parallel computer and then in running the parallel jobs via the help of communication handles within that virtual computer. Contrary to the MPI standard, the MPI2 standard establishes a distinction between the administrative tasks of establishing and maintaining the communication ring from the administration of the parallel jobs.

- In practice, communication is done via the establishment of computer daemons on each node which are themselves linked within the MPI2 umbrella by point-to-point communication protocol. The users' tasks in any node talk to the local MPI2 daemons, which themselves talk to each other and therefore can establish communication links from any sub-tasks to any other sub-tasks.

- In MPICH2, the multi-purpose daemon (MPD) (for mpd manger) allows the establishment of the communication ring or the virtual machine. Once the communication ring is established specific MPI commands allow the users to load in the sub-tasks, monitor them, signal them and possibly kill them.



See also mpich2-user-guide.

Running MPI Parallel Programs within CRC SGE batch system

- http://wiki.crc.nd.edu/wiki/index.php/Main_Page
- See the other notes.

Program Details

#include “mpi.h”

- Function declarations for all MPI functions

int MPI_Init(int* argc_ptr, char** argv_ptr[])

- Allows the system to do any setup needed to handle further calls to MPI library
- It must be called before any other MPI function
- It requires to pass along the command line arguments.

int MPI_Finalize(void)

- MPI_Finalize() is the companion to MPI_init().
- MPI_Finalize() allows the system to free up resources that have been allocated to MPI.
- It must be the last MPI function call.

How does a process know its position in a set of processes

```
MPI_Comm_rank(MPI_Comm  comm  /* in */,
              int*      result /* out */)
```

- Argument “comm” is called a **communicator**.
- When MPI has been initialized, every active process become a member of a **communicator** called MPI_COMM_WORLD. A **communicator** is an opaque object that provides the environment for message passing among processes. MPI_COMM_WORLD is the default communicator.
- MPI_COMM_WORLD is predefined within MPI and consists of all the processes initiated when we run this program.
- Processes within a communicator are ordered. The **rank** of a process is its position in the overall order.
- In a communicator with *p* processes, each process has a unique rank (ID number) between *0* and *p-1*.

```
MPI_Comm_size(MPI_Comm comm  /* in */,
              int*     size    /* out*/)
```

- It gives total number of processes that have been allocated.

C Language Bindings

- Function arguments are marked as
  - **`in`**: the call uses but does not update the argument
  - **`out`**: the call may update the argument
  - **`inout`**: the call both uses and updates the argument
- All MPI names have an **`MPI_`** prefix
- Defined constants are in all capital letters
- Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase

Summary

1. User issues a directive to the operating system that has the effect of placing a copy of the executable program on each processor.
2. Each processor begins execution of its copy of the executable.
3. Different processes can execute different statements by branching within the program based on their process ranks.

- **Results from execution**

```
[zxu2@newcell ~/ACMS40212]$ mpiexec -np 4 ./hello
Hello from node 2
Hello from node 0
Hello from node 3
Hello from node 1
[zxu2@newcell ~/ACMS40212]$
```

Issues ? :

1. The output might seems out of order. Keep in mind that the code was started on all nodes practically simultaneously. There was no reason to expect one node to finish before another. *It's important for us not to assume that there is any particular order to events unless we do something explicitly.*

2. "how does the output know where to go?" Most IO is file-based and will depend upon a distributed file system.

Recap

- When running with MPI, all processes use the same compiled binary, and hence all processes are running the exact same code.
- Things distinguish the parallel program:

--- Each process uses its *process rank* to determine what part of the algorithm instructions are meant for it.

--- Processes communicate with each other to accomplish the final task.

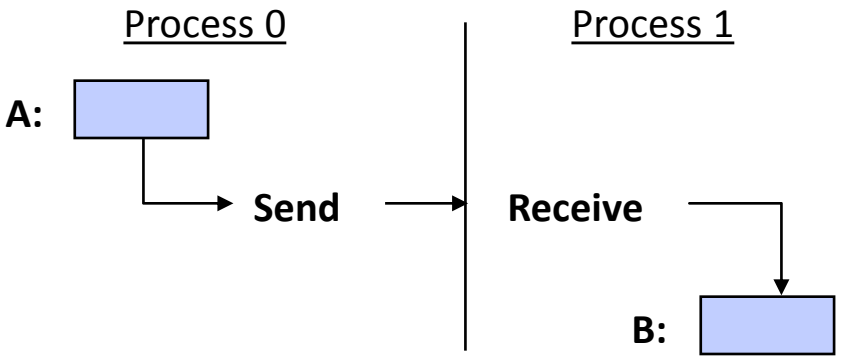
Point-to-Point communications

- Transfer *message* from one process to another process

  -- It involves an explicit "**send**" and "**receive**", which is called "two-sided" communication.

  -- Message: data + (source + destination + communicator + ???)

  -- Almost all of the MPI commands are built around point-to-point operations.



Things need to be considered:

- To whom is data sent?
- Where is the data to be sent?
- What type of data is sent?
- How much of data is sent?
- How does the receiver identify it?
- Where is the received data to be stored?

Message Organization

- Message is divided into data and envelope
- data
  - buffer
  - count
  - datatype
- envelope
  - process identifier (source/destination rank)
  - message tag
  - communicator

Sending and Receiving Routines

- int MPI_Send(void* message /* in */,
  int count /* in */,
  MPI_Datatype datatype /* in */,
  int dest /* in */,
  int tag /* in */,
  MPI_Comm comm /* in */)

- int MPI_Recv(void* message /* out */,
  int count /* in */,
  MPI_Datatype datatype /* in */,
  int source /* in */,
  int tag /* in */,
  MPI_Comm comm /* in */,
  MPI_Status* status /* out */)

Message Bodies

- “void* message”: the starting location in memory where the data is to be found
- “int   count “: number of items to be sent.
- “MPI_Datatype datatype ”: the type of data to be sent.

MPI Datatypes

- MPI defines its own data type that correspond to typical datatypes in C or Fortran
- This allows to code to be portable between systems
- Users are allowed to build their own datatypes in MPI

MPI Datatype	C Datatype
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_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
…	

Message Envelope

What else is needed for A to send a message to B in a communicator?

Example. Process A can send both floats to be printed and floats to be stored. How is process B to distinguish between the two different types?

- We now know where to deliver and where to get message, number of elements in the message and their type, and destination and source IDs.
- Additionally, we also use a message identifier “tag”.

  -- It allows program to label classes of messages (e.g. one for printing data, another for storing data, etc.)

  -- A tag is an int specified by the programmer that the system adds to the message envelope.

  -- **MPI** guarantees that the integers 0 – 32767 can be used as tags.

Blocking vs. Non-Blocking Communication

Blocking: blocking send or receive routines does not return until operation is complete.

- blocking sends ensure that it is safe to overwrite the sent data
- `MPI_Send`(): will not return until the message data and envelope is safely stored away.
  - The message data might be delivered to the matching receive buffer, or copied to some temporary system buffer.
  - After MPI_Send returns, user can safely access or overwrite the send buffer.
- blocking receives make sure that the data has arrived and is ready for use
- `MPI_Recv`(): returns only after the receive buffer has the received message
  - After it returns, the data is here and ready for use.

Non-blocking: Non-blocking send or receive routines returns immediately, with no information about completion.

-- User should test for success or failure of communication.

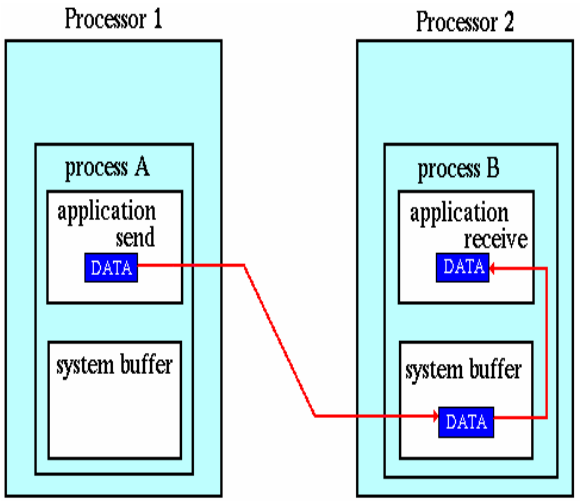
-- In between, the process is free to handle other tasks.

-- It is less likely to form deadlocking code

-- It is used with MPI_Wait() or MPI_Test()

Buffering

- Send and matching receive operations usually are not synchronized in reality because of the work loads. MPI implementation must decide what happens when send/recv are not sync.
- Why buffering:
  - Send occurs 5 seconds before receive is ready; where is the message when receive is being posted?
  - Multiple sends arrive at the same receiving task which can receive one send at a time – what happens to the messages that are backing up?
- MPI implementation (not the MPI standard) typically uses a system buffer to hold data in transit.
  - System buffer:
    - Invisible to users and managed by MPI library
    - Finite resource that can be easily exhausted
    - May exist on sending or receiving side, or both
    - May improve performance.
  - Users can allocate memory for MPI message buffering.



Path of a message buffered at the receiving process

Type of Communication	MPI Function
blocking send (Standard mode)	MPI_Send System decides whether the outgoing message will be buffered or not Usually, small messages → buffering mode; large messages, no buffering, synchronous mode.
non-blocking send	MPI_Isend
blocking receive	MPI_Recv
non-blocking receive	MPI_Irecv

C Code

Goal: Process 1 sends a number 77 to process 0.

```
/** send_recv.c **/
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{

      int my_rank, numbertoreceive, numbertosend=77;
      MPI_Status status;

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

       if (my_rank==0){
            MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Number received is: %d\n", numbertoreceive);
       }
       else if(my_rank == 1)
            MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

       MPI_Finalize();
}
```

```
MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD)
```

&numbertosend	a pointer to whatever we wish to send. In this case it is simply an integer. It could be anything from a character string to a column of an array or a structure. It is even possible to pack several different data types in one message.
1	the number of items we wish to send. If we were sending a vector of 10 int's, we would point to the first one in the above parameter and set this to the size of the array.
MPI_INT	the type of object we are sending. Possible values are: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LING, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED
0	Destination of the message (the rank of the receiving process). In this case process 0.
10	Message tag. All messages have a tag attached to them that can be useful for sorting messages. We just picked 10 at random.
MPI_COMM_WORLD	We don't really care about any subsets of PEs here. So, we just chose this "default".

MPI_Recv(&numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &status)

&numbertoreceive	A pointer to the variable that will receive the item. In our case it is simply an integer that has has some undefined value until now.
1	Number of items to receive. Just 1 here.
MPI_INT	Datatype. Must be an int to match with what we send.
MPI_ANY_SOURCE	The node to receive from. We could use 1 here since the message is coming from there, but the "wild card" – MPI_ANY_SOURCE allows to receive a message from anywhere.
MPI_ANY_TAG	We could use a value of 10 here to filter out any other messages (there aren't any) but, “wild card” MPI_ANY_TAG allows to receive any tag.
MPI_COMM_WORLD	Just using default set of all Processes.
&status	A structure that receive the status data which includes the source and tag of the message.

- MPI_ANY_SOURCE: there is no wildcard for specifying destination.
- MPI_ANY_TAG: this wildcard can not be used by sender. Namely, process 1 must use a tag and process 0 can receive with either an identical tag or MPI_ANY_TAG
- Status of receive: MPI_Status type. It returns information on the data that was actually received. MPI_Status structure contains at least three members:

---- status.MPI_SOURCE

----status.MPI_TAG

----status.MPI_ERROR

- MPI_Send() and MPI_Recv() have integer return values. These return values are error codes.

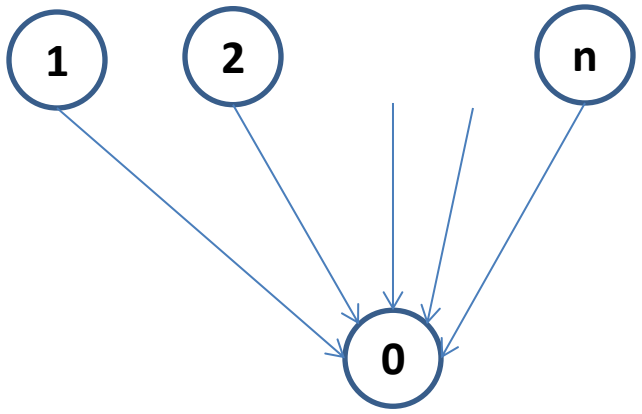
- To get size of the message received, we call

```
int MPI_Get_count(
                MPI_Status*       status  /* in */,
                MPI_Datatype     datatype /* in */,
                int*                    count_ptr  /* out */)
```

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

int     main(int argc, char** argv)
{   /**   send_recv_count.c  **/
      int my_rank, numbertoreceive[10], numbertosend[3]={73, 2, -16};
      int  recv_count, i;
      MPI_Status status;

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

      if (my_rank==0){
            MPI_Recv( numbertoreceive, 3, MPI_INT, MPI_ANY_SOURCE,  MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
            printf("status.MPI_SOURCE = %d\n", status.MPI_SOURCE);
            printf("status.MPI_TAG = %d\n", status.MPI_TAG);
            printf("status.MPI_ERROR = %d\n", status.MPI_ERROR);

            MPI_Get_count(&status, MPI_INT, &recv_count);
            printf("Receive %d data\n",  recv_count);
            for(i = 0; i < recv_count; i++)
               printf("recv[%d] = %d\n", i, numbertoreceive[i]);
      }
      else MPI_Send( numbertosend, 3, MPI_INT, 0, 10, MPI_COMM_WORLD);

      MPI_Finalize();
      return 0;
}
```

Parallel Summation Example

```
#include<iostream.h>
#include<mpi.h>
/**** chapter2c11P.cpp, add numbers from 1 to 1000 *****/
int main(int argc, char ** argv){
 int mynode, totalnodes;
 int sum,startval,endval,accum;
 MPI_Status status;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
 MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

 sum = 0;
 startval = 1000*mynode/totalnodes+1;
 endval =   1000*(mynode+1)/totalnodes;

 for(int i=startval;i<=endval;i=i+1)
   sum = sum + i;

 if(mynode!=0)
  MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
 else
  for(int j=1;j<totalnodes;j=j+1){
   MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
   sum = sum + accum;
  }

 if(mynode == 0)
  cout << "The sum from 1 to 1000 is: " << sum << endl;

 MPI_Finalize();
}
```



More to Think About

- Suppose process 1 calls MPI_Send, but process 0 fails to call MPI_Recv to receive from process 1. What happens to the program?

- **Blocking send/receive restrictions**

  -- *source, tag,* and *comm* must match those of a pending message for the message to be received.

  -- Wildcards can only be used for *source* and *tag,* but not communicator.

  -- An error will be returned if the message buffer exceeds that allowed for by the receive.

  -- User must make sure that the send/receive datatypes agree. If they do not, the results are not defined.

Message Buffering

- Definition of “completion” for MPI_Recv() is trivial – the data can now be used.
- Definition of “completion” for MPI_Send() is trickier. Completion implies that the data has been stored away such that the program is free to overwrite the send “message” buffer.

  -- **Non-local:** the data can be sent directly to the receive buffer.

  -- **Local (buffering)**: the data can be stored in a local buffer (system provided or user provided), in which case the send could return before the receive is initiated.

Write Safe Code

- A safe MPI program should not rely on system buffering for success.
- Any system will eventually run out of buffer space as message sizes are increased.
- User should design proper send/receive orders to avoid **deadlock**

Safe Code

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

/* process 0 send a number to and receive a number from process 1.
    process 1 receive a number from and send a number to process 0
*/
int     main(int argc, char** argv)
{          /*** sample_safe1.c   ****/
      int my_rank, numbertoreceive, numbertosend = -16;
      MPI_Status status;

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

      if (my_rank==0){
            MPI_Send( &numbertosend, 1, MPI_INT, 1,    10, MPI_COMM_WORLD);
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 1,  20, MPI_COMM_WORLD, &status);
      }
      else  if(my_rank == 1)
      {
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0,  10, MPI_COMM_WORLD, &status);
            MPI_Send( &numbertosend,     1, MPI_INT, 0, 20, MPI_COMM_WORLD);
      }
      MPI_Finalize();
      return 0;
}
```

Deadlock Code

```
#include <stdio.h>
#include "mpi.h“
/* process 0 receive a number from and send a number from process 1.
    process 1 receive a number from and send a number to process 0
*/
int     main(int argc, char** argv)
{        /**** sample_deadlock.c   *****/
      int my_rank, numbertoreceive, numbertosend = -16;
      MPI_Status status;

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

      if (my_rank==0){
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 1,  20, MPI_COMM_WORLD, &status);
            MPI_Send( &numbertosend, 1, MPI_INT, 1,    10, MPI_COMM_WORLD);
       }
      else  if(my_rank == 1)
      {
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0,  10, MPI_COMM_WORLD, &status);
            MPI_Send( &numbertosend,     1, MPI_INT, 0, 20, MPI_COMM_WORLD);
      }
      MPI_Finalize();
      return 0;
}
```

Buffering dependent Code

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

/* process 0 receive a number from and send a number from process 1.
    process 1 receive a number from and send a number to process 0
*/
int     main(int argc, char** argv)
{
      int my_rank, numbertoreceive, numbertosend = -16;
      MPI_Status status;

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

      if (my_rank==0){
            MPI_Send( &numbertosend, 1, MPI_INT, 1,    10, MPI_COMM_WORLD);
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 1,  20, MPI_COMM_WORLD, &status);
      }
      else  if(my_rank == 1)
      {
            MPI_Send( &numbertosend,     1, MPI_INT, 0, 20, MPI_COMM_WORLD);
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0,  10, MPI_COMM_WORLD, &status);
      }
      MPI_Finalize();
      return 0;
}
```

Success of this code is dependent on buffering. One of the send must buffer and return. Otherwise, deadlock occurs.