**Message Parsing Interface**

The Message Passing Interface (MPI) is a library specification that allows HPC to pass information between its various nodes and clusters. HPC uses OpenMPI, an open-source, portable implementation of the MPI standard. OpenMPI contains a complete implementation of version 1.2 of the MPI standard and also MPI-2

**Compilers**

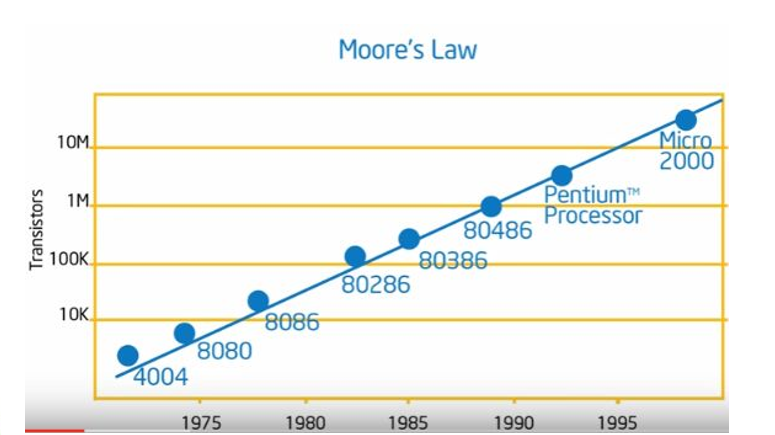
OpenMPI supports several different compiler sets, but the user must specify which they wish to use for each job.OpenMPI supports C, C++, F77, and F90. However not all compiler sets support all languages. Portland Group’s CDK and Intel support all four languages. KCC supports only C++. Absoft supports F77 and F90. GNU supports C, C++, and F77.

**Moore's Law:**

Moore's Law refers to Moore's perception that the number of transistors on a microchip doubles every two years, though the cost of computers is halved. Moore's Law states that we can expect the speed and capability of our computers to increase every couple of years, and we will pay less for them. Another tenet of Moore's Law asserts that this growth is exponential.

**Understanding Moore's Law:**

In 1965, Gordon E. Moore—co-founder of Intel —postulated that the number of transistors that can be packed into a given unit of space will double about every two years. Today, however, the doubling of installed transistors on silicon chips occurs closer to every 18 months instead of every two years.

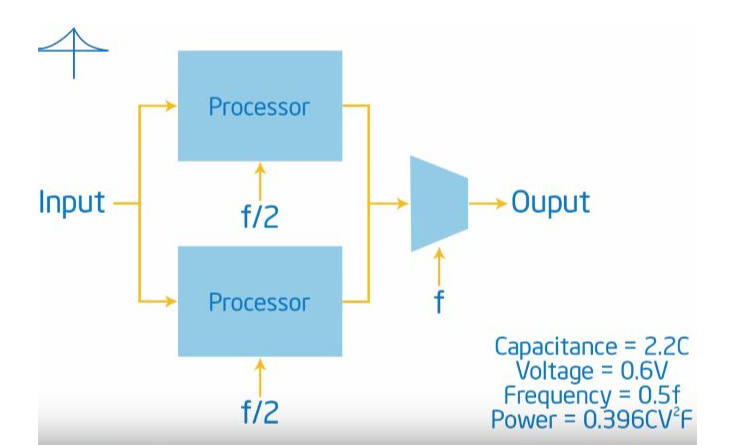
**Background:** Gordon Moore did not call his observation "Moore's Law," nor did he set out to create a "law." Moore made that statement based on noticing emerging trends in chip manufacturing at Intel. Eventually, Moore's insight became a prediction, which in turn became the golden rule known as Moore's Law. 

* Moore's Law states that the number of transistors on a microchip doubles about every two years, though the cost of computers is halved.
* In 1965, Gordon E. Moore, the co-founder of Intel, made this observation that became Moore's Law.
* Another tenet of Moore's Law says that the growth of microprocessors is exponential.
* Moore's Law states that the number of transistors on a microchip doubles about every two years, though the cost of computers is halved.
* In 1965, Gordon E. Moore, the co-founder of Intel, made this observation that became Moore's Law.
* Another statement of Moore's Law says that the growth of microprocessors is exponential.

From Prediction to Truism:

In the decades that followed Gordon Moore's original observation, Moore's Law guided the semiconductor industry in long-term planning and setting targets for research and development (R&D). Moore's Law has been a driving force of technological and social change, productivity, and economic growth that are hallmarks of the late-twentieth and early twenty-first centuries.

Parallel Architecture:

MPI follows the SPMD style, i.e., it splits the workload into different tasks that are executed on multiple processors. Originally, MPI was designed for distributed memory architectures, which were popular at that time. Fig. 9.1 illustrates the characteristics of these traditional systems, with several CPUs connected to a network and one memory module per CPU. A parallel MPI program consists of several processes with associated local memory. In the traditional point of view each process is associated with one core. Communication among processes is carried out through the interconnection network by using send and receive routines.

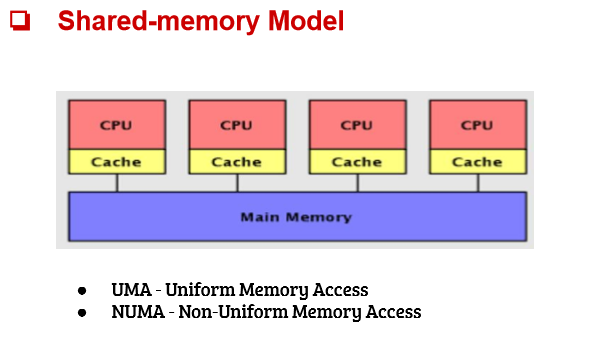
As architectural trends changed, the majority of current clusters contain shared-memory nodes that are interconnected through a network forming a hybrid distributed-memory/shared-memory system, as illustrated in Fig. 9.2. Modern clusters could even include manycore accelerators attached to the nodes. Nowadays MPI implementations are able to spawn several processes on the same machine. However, in order to improve performance, many parallel applications use the aforementioned hybrid approach: one MPI process per node that calls multithreaded [3,10] or CUDA [1,13] functions to fully exploit the compute capabilities of the existing CPUs and accelerators cards within each node.

# Shared Memory Model:

# In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.

* Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
* This is perhaps the simplest parallel programming model.
* An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.
* An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality:
  + Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  + Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

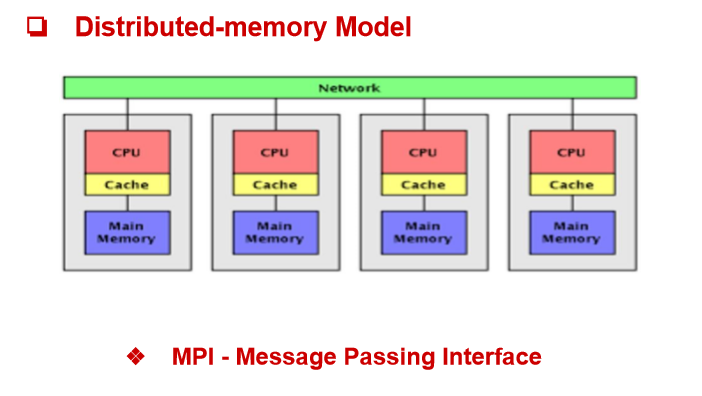
## [**222.PNG**](https://hpc.llnl.gov/files/222png)



Distributed memory:

In computer science, distributed memory refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a shared memory multiprocessor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

In a distributed memory system there is typically a processor, a memory, and some form of interconnection that allows programs on each processor to interact with each other. The interconnect can be organised with point to point links or separate hardware can provide a switching network. The network topology is a key factor in determining how the multiprocessor machine scales. The links between nodes can be implemented using some standard network protocol (for example Ethernet), using bespoke network links (used in for example the Transputer), or using dual-ported memories.

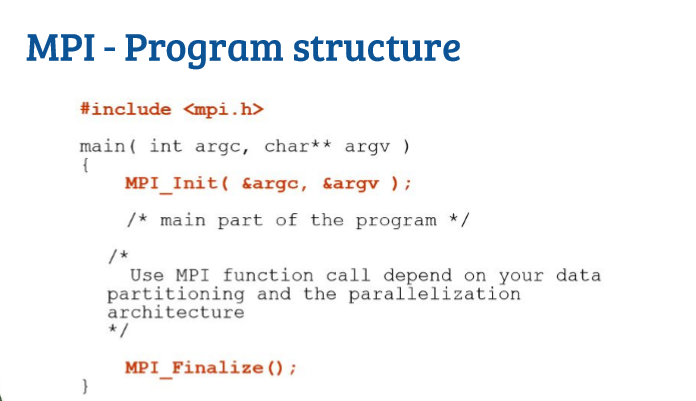


MPI - Message Passing Interface

● The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum

● In MPI a Message is passed from one process to another process

● MPI is based on Routines.



Collective communication and synchronization points:

One of the things to remember about collective communication is that it implies a synchronization point among processes. This means that all processes must reach a point in their code before they can all begin executing again.

Before going into detail about collective communication routines, let’s examine synchronization in more detail. As it turns out, MPI has a special function that is dedicated to synchronizing processes:

MPI\_Barrier(MPI\_Comm communicator)

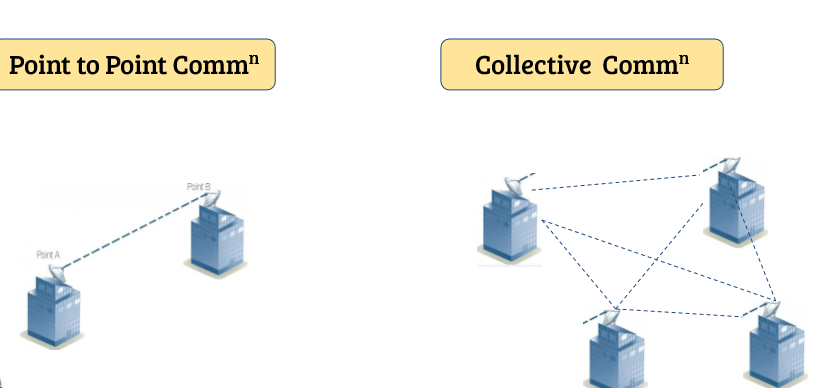
The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function. Here’s an illustration. Imagine the horizontal axis represents execution of the program and the circles represent different processes.

Process zero first calls MPI\_Barrier at the first time snapshot (T 1). While process zero is hung up at the barrier, process one and three eventually make it (T 2). When process two finally makes it to the barrier (T 3), all of the processes then begin execution again (T 4).

MPI\_Barrier can be useful for many things. One of the primary uses of MPI\_Barrier is to synchronize a program so that portions of the parallel code can be timed accurately.

Want to know how MPI\_Barrier is implemented? Sure you do :-) Do you remember the ring program from the sending and receiving tutorial? To refresh your memory, we wrote a program that passed a token around all processes in a ring-like fashion. This type of program is one of the simplest methods to implement a barrier since a token can’t be passed around completely until all processes work together.

One final note about synchronization - Always remember that every collective call you make is synchronized. In other words, if you can’t successfully complete an MPI\_Barrier, then you also can’t successfully complete any collective call. If you try to call MPI\_Barrier or other collective routines without ensuring all processes in the communicator will also call it, your program will idle.



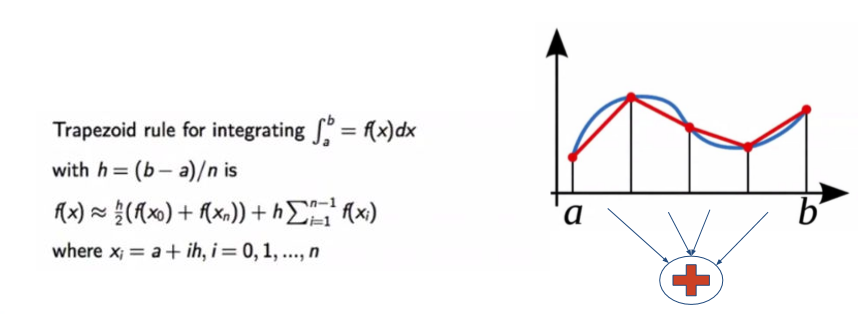
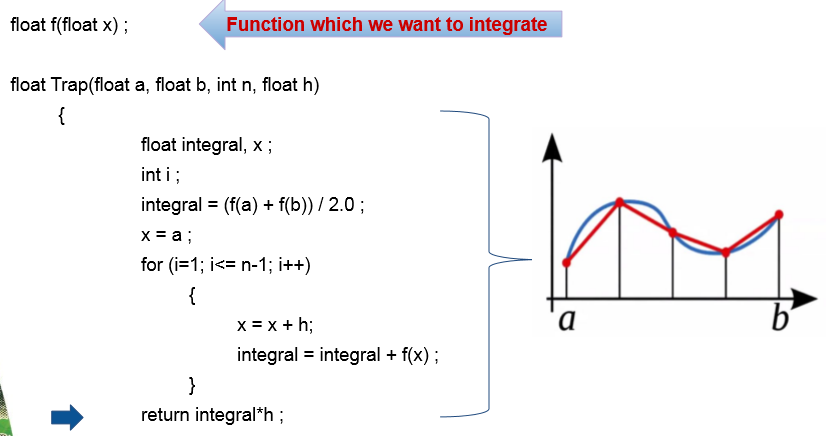
Trapezoidal Rule:

The Trapezoidal Rule for approximating b∫af(x)dx is given by

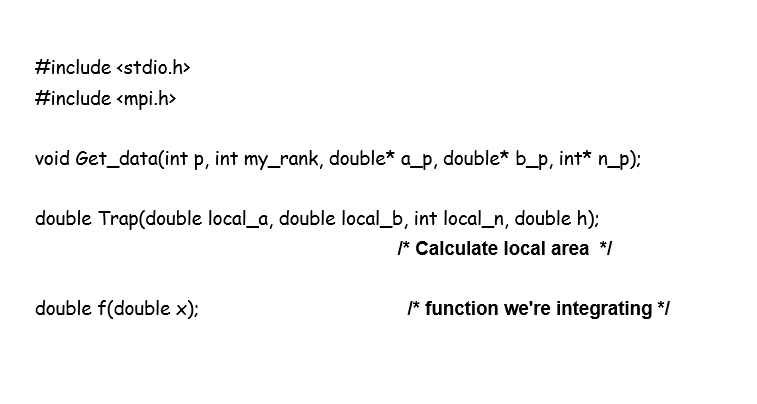
b∫af(x)dx≈Tn=Δx2[f(x0)+2f(x1)+2f(x2)+⋯+2f(xn−1)+f(xn)],

where Δx=b−an and xi=a+iΔx.

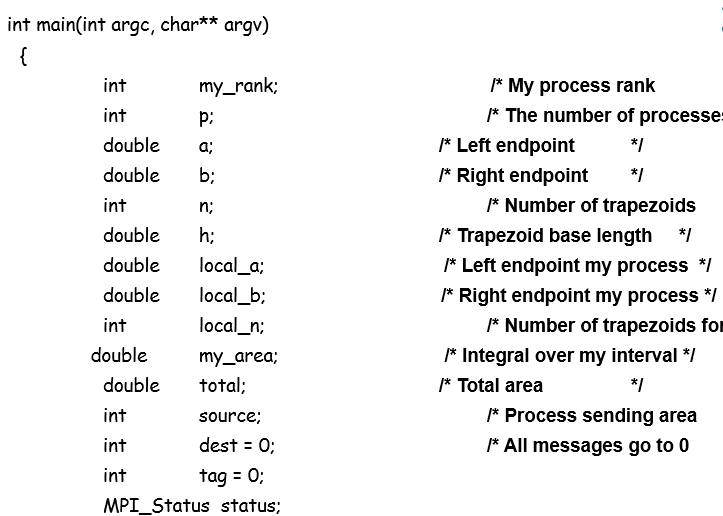
As n→∞, the right-hand side of the expression approaches the definite integral b∫af(x)dx.

The program that follows:  


Trapezoidal approach:



The integrating factors for the trapezoidal estimation:

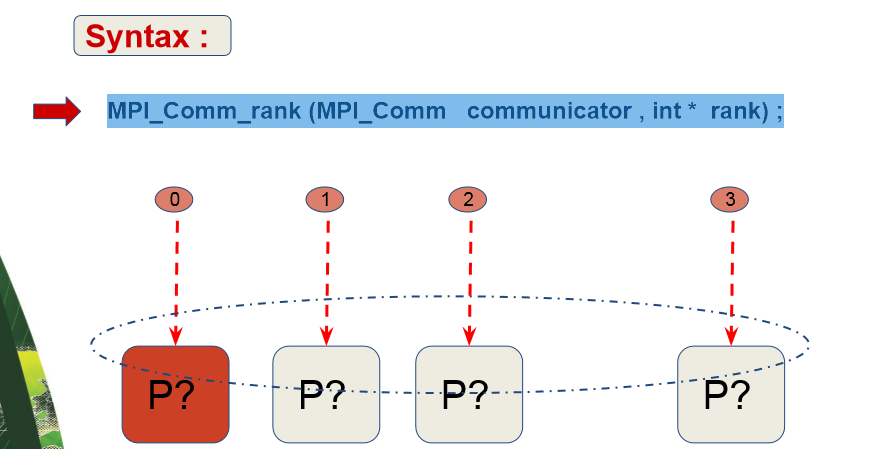


Syntax:

1. MPI\_Comm\_rank(.....)

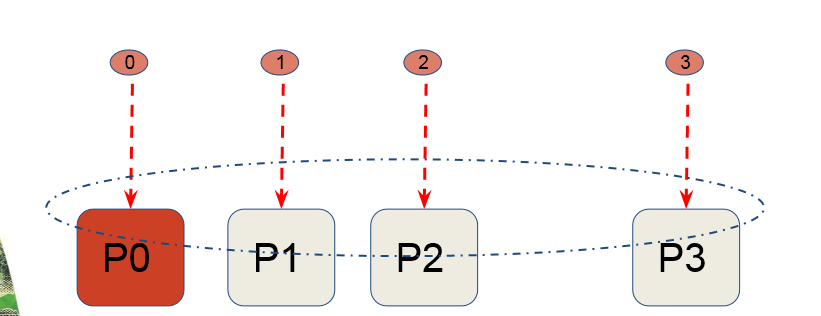
MPI\_Comm\_rank (MPI\_Comm communicator , int \* rank) ;

1. MPI\_Comm\_rank (MPI\_Comm communicator , int \* rank) ;



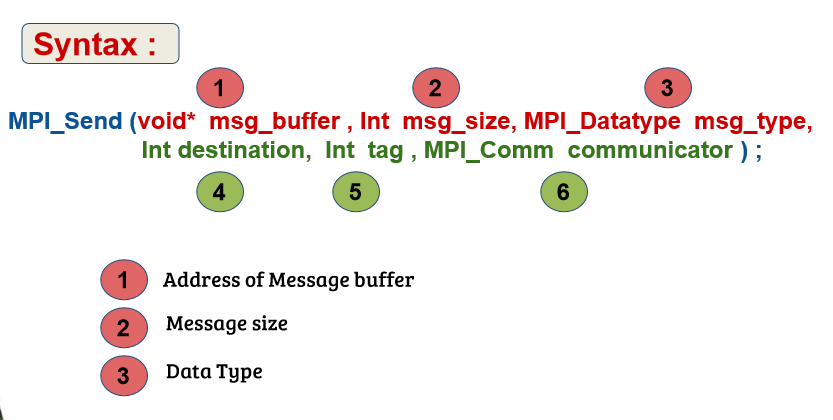
MPI\_Comm\_size(.....):

MPI\_Comm\_size (MPI\_Comm communicator , int \* size) ;



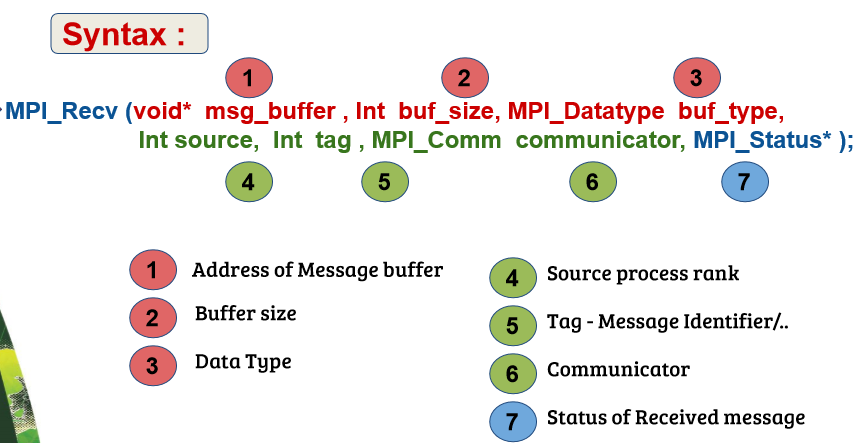
1. MPI\_Send(.....)

MPI\_Send (void\* msg\_buffer , Int msg\_size, MPI\_Datatype msg\_type, Int destination, Int tag , MPI\_Comm communicator ) ;



1. MPI\_Recv(.....)

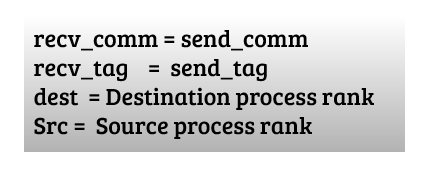
MPI\_Recv (void\* msg\_buffer , Int buf\_size, MPI\_Datatype buf\_type, Int source, Int tag , MPI\_Comm communicator, MPI\_Status\* );



Successful transmission of Message:

MPI\_Send (void\* msg\_buffer , Int msg\_size, MPI\_Datatype msg\_type, Int destination, Int tag , MPI\_Comm communicator ) ;

MPI\_Recv (void\* msg\_buffer , Int buf\_size, MPI\_Datatype buf\_type, Int source, Int tag , MPI\_Comm communicator, MPI\_Status\* );



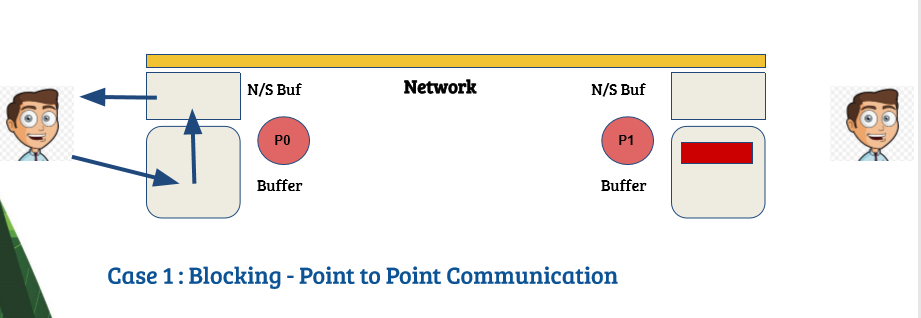
Types of communication for different cases:

Point-to-Point Communication

The most elementary form of message-passing communication involves two nodes, one passing a message to the other. Although there are several ways that this might happen in hardware, logically the communication is point-to-point: one node calls a send routine and the other calls a receive.

A message sent from a sender contains two parts: data (message content) and the message envelope. The data part of the message consists of a sequence of successive items of the type indicated by the variable datatype. MPI supports all the basic C datatypes and allows a more elaborate application to construct new datatypes at runtime (discussed in an advanced topic tutorial). The basic MPI datatypes for C are MPI\_INT, MPI\_FLOAT, MPI\_DOUBLE, MPI\_COMPLEX, MPI\_CHAR. The message envelope contains information such as the source (sender), destination (receiver), tag and communicator.

As with most existing message-passing systems today, MPI provides blocking send and receive as well as nonblocking send and receive. We will introduce and discuss both blocking and nonblocking communication in this tutorial.



Blocking Send and Receive

Blocking Send Operation:

Below is the syntax of the blocking send operation:

MPI\_Send(void\* buf, int count, MPI\_Datatype datatype,

int dest, int tag, MPI\_Comm comm);

where buf is the address of the send buffer, count is the number of items in the send buffer, datatype (MPI\_INT, MPI\_FLOAT, MPI\_CHAR, etc.) describes these items' datatype, dest is the rank of the destination processor, tag is the message type identifier and comm is the communicator to be used (see MPI advanced topics for more information on communicators).

The blocking send call does not return until the message has been safely stored away so that the sender can freely reuse the send buffer.

Blocking Receive Operation:

The MPI blocking receive call has the form:

MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype,

int source, int tag, MPI\_Comm comm,

MPI\_Status \*status);

where all the arguments have the same meaning as for the blocking send except for source and status. Source is the rank of the node sent the message and status is an MPI-defined integer array of size MPI\_STATUS\_SIZE. The information carried by status can be used in other MPI routines.

The blocking receive does not return until the message has been stored in the receive buffer.

Order:

Messages are non-overtaking: if a sender sends two messages in succession to the same destination and both match the same receive, then this operation cannot receive the second message while the first is still pending. If a receiver posts two receives in succession and both match the same message, then this message cannot satisfy the second receive operation, as long as the first one is still pending. This requirement facilitates matching sends to receives. It guarantees that message-passing code is deterministic if processes are single-threaded and the wildcard MPI\_ANY\_SOURCE is not used in receives.

Progress:

If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independent of other action in the system. The send operation will complete unless the receive is satisfied and completed by another message. The receive operation will complete unless the message sent is consumed by another matching receive that was posted at the same destination process.

Avoid a Deadlock:

It is possible to get into a deadlock situation if one uses blocking send and receive. Here is a fragment of code to illustrate the deadlock situation:

MPI\_Comm\_rank(comm,&rank);

if (rank == 0) {

MPI\_Recv(recvbuf,count,MPI\_REAL,1,tag,comm,&status);

MPI\_Send(sendbuf,count,MPI\_REAL,1,tag,comm);

}

elseif (rank == 1) {

MPI\_Recv(recvbuf,count,MPI\_REAL,0,tag,comm,&status);

MPI\_Send(sendbuf,count,MPI\_REAL,0,tag,comm);

}

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second process is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. To avoid deadlock, one can use one of the following two examples:

MPI\_Comm\_rank(comm,&rank);

if (rank == 0) {

MPI\_Send(sendbuf,count,MPI\_REAL,1,tag,comm,);

MPI\_Recv(recvbuf,count,MPI\_REAL,1,tag,comm,&status);

}

elseif(rank == 1) {

MPI\_Recv(recvbuf,count,MPI\_REAL,0,tag,comm,&status);

MPI\_Send(sendbuf,count,MPI\_REAL,0,tag,comm);

}

or

MPI\_Comm\_rank(comm,&rank);

if (rank == 0) {

MPI\_Recv(recvbuf,count,MPI\_REAL,1,tag,comm,&status);

MPI\_Send(sendbuf,count,MPI\_REAL,1,tag,comm);

}

elseif(rank == 1) {

MPI\_Send(sendbuf,count,MPI\_REAL,0,tag,comm);

MPI\_Recv(recvbuf,count,MPI\_REAL,0,tag,comm,&status);

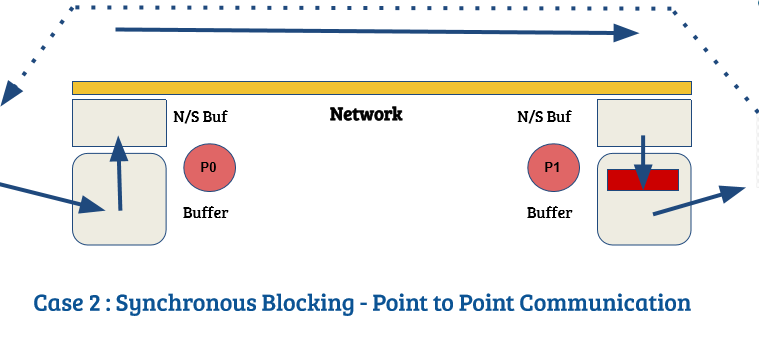
}

Synchronous Blocking - Point to Point Communication: A nonblocking send call initiates the send operation, but does not complete it. The send start call will return before the message is copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data have been copied out of the send buffer. Here is the syntax of the nonblocking send operation:

MPI\_Isend(void\* buf, int count, MPI\_Datatype datatype,

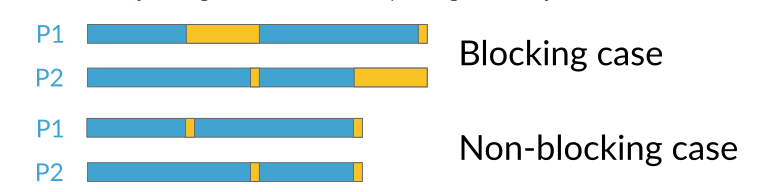
int dest, int tag, MPI\_Comm comm,

MPI\_Request \*request);



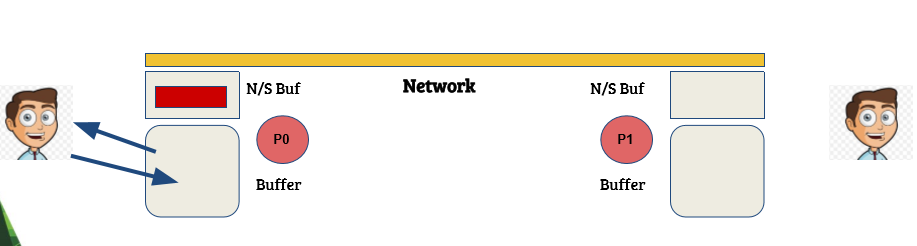
Non-blocking communications:

For the moment, we have only seen blocking point-to-point communication. That means that when a process sends or receive information, it has to wait (Animation from Cornell virtual workshop) for the transmission to end to get back to what it was doing.

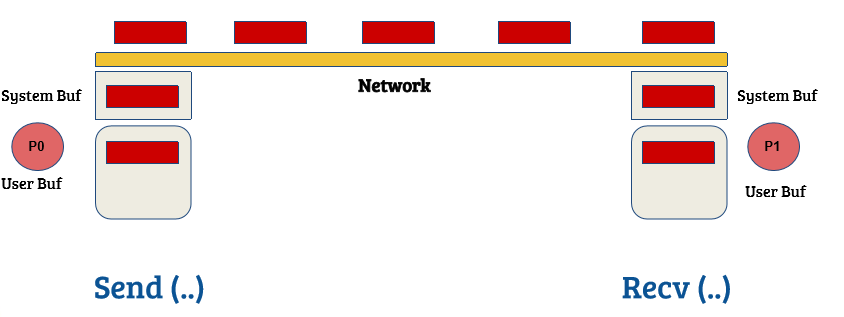


In this case, process 0 has some information to send to process 1. But both are working on very different things and, as such, take different time to finish their computations. Process 0 is ready to send its data first, but since process 1 has not finished its own computations, process 0 has to wait for process 1 to be ready before getting back to its own work. Process 1 finishes treating the data really quickly and now waits for process 0 to finish for getting new data. This way of sending messages is possible in MPI and called non-blocking communications.

What is happening in MPI is a bit different. Non-blocking communications always require to be initialised and completed. What that means is that now, we will call a send and a receive commands to initialise the communication. Then, instead of waiting to complete the send (or the receive), the process will continue working, and will check once in a while to see if the communication is completed. This might be a bit obscure so let's work an example together. First in pseudo-code and then in C++. Let's imagine that process 0 has first to work for 3 seconds, then for 6. At the same time, process 1 has to work for 5 seconds, then for 3. They must synchronise some time in the middle, and at the end.

****

**Network buffer mechanism:**

****

**Non - Blocking point to point communication :**

**● MPI\_Isend (&buf,count,datatype,dest,tag,comm,&request)**

**● MPI\_Irecv (&buf,count,datatype,source,tag,comm,&request)**

**● MPI\_Issend (&buf,count,datatype,dest,tag,comm,&request) ○ Synchronous non-blocking send.**

**● Check for Asynchronous Transfer :**

**○ MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \* status) ■**

**Flag: ● if flag == 0, the send/receive operation is not yet complete**

**● if flag != 0, the send/receive operation is complete and the variable status contains information about the messag**

**■ status: contains information about the message (use the information only if flag != 0**

**Collective Communication Routines:**

**MPI\_Bcast ( void\* data , Int count , MPI\_Datatype datatype , Int source\_process , MPI\_Comm comm ) ;**

**Eg : MPI\_Bcast(a, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) ;**

**MPI\_Scatter ( void\* send\_buffer , Int send\_count , MPI\_Datatype send\_datatype , void\* recv\_buffer , Int recv\_count , MPI\_Datatype recv\_datatype , Int source\_process , MPI\_Comm comm ) ;**

**Eg : MPI\_Scatter ( a, local\_n, MPI\_DOUBLE, local\_a, local\_n, MPI\_DOUBLE, 0, comm) ;**