Universitat Autònoma de Barcelona

Departamento de Arquitectura de Computadores y Sistemas Operativos (CAOS/DACSO)

Escuela de Ingeniería
Universidad Autónoma de Barcelona

Tutorial

Parallel Programming with MPI

1.- Objective

A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, programming languages, compilers, libraries, communications systems, and parallel I/O. Due to the difficulties in automatic parallelization today, people have to choose a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform.

Parallel models may be implemented as libraries invoked from traditional sequential languages. The popular model is the Message Passing Interface Standard (MPI). It 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.

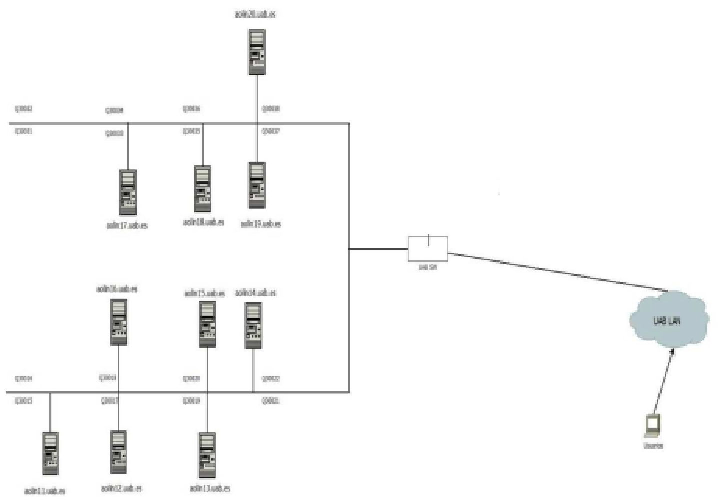
The objective of this tutorial is to introduce you into parallel programming. We will see, step by step, how to develop a parallel program according to the MPI standard and how to run it on many processors simultaneously. Developing a parallel application must take into account the execution on many processors. The objective of this approach is the reduction of application execution time.

The tutorial begins with an introduction, background, and basic information for getting started with MPI. It contains a set of programs starting with the simplest one and then extending its functionality introducing more MPI routines that are most useful for new MPI programmers, including MPI Environment Management, Point-to-Point Communications, and Collective Communications routines.

2.- Laboratory

You will develop the programs presented in tutorial at the laboratory (Laboratory of Computer Architecture Q5-0004) and then compile, link and execute them. The laboratory contains 10 machines (AOLIN11 … AOLIN21) connected via network. The operating system of each machine is 64-bit Linux. This environment supports MPI standard through openmpi (several versions installed, including openmpi 3.0).

The laboratory contains 10 machines, connected through a 100 Mbits network, with 4 cores each (Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz), 6 GB of main memory (1067 MHz), 500 GB of hard disk, a NVIDIA GeForce GT 430 (96 cores), and 64-bit operating system. These machines will work during the day as a non-dedicated cluster and during the night as a dedicated cluster. Consequently, during the day each student can do functional testing of his programs, but he cannot gather information for doing performance analysis, which should be done during the night. The distribution of the computers at the laboratory is as follows:



2.- Getting Started

Using a tool/environment

For listing all available software use: *module avail*
For loading a particular software use: *module load soft_name/version*
For unloading a particular software use: *module unload soft_name/version*

Hardware Information

We will start by using *likwid* (https://code.google.com/p/likwid/wiki/Introduction) to get the hardware information of our working environment.
Ex.
Use ***likwid-topology***. What information are you getting?

3.- MPI Applications

Header File:
Header file is required for all programs/routines which make MPI library calls.

```
#include "mpi.h"
```

Format of MPI Calls:

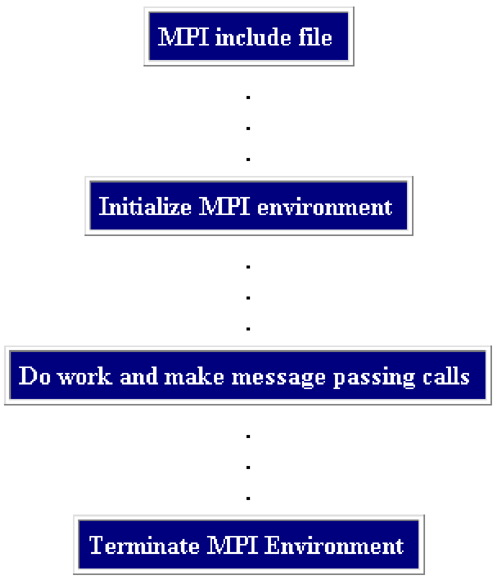
Format: **`rc = MPI_Xxxxx (parameter, ...)`**

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

Return code: Each call returns a value equal to `MPI_SUCCESS` on successful call or error code otherwise.

Note: C names are case sensitive;

General MPI Program Structure:


MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Communicators and Groups:

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
- Use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.

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 "process ID". Ranks are contiguous and begin at zero.

Environment Management Routines

MPI environment management routines are used for initializing and terminating the MPI environment, querying the environment and identity, etc.

MPI function	**Description**
`MPI_Init(&argc,&argv)`	Initializes the MPI execution environment. This function must be called only once in every MPI program before any other MPI functions.
`MPI_Comm_size(comm,&size)`	Determines the number of processes in the group associated with a communicator.
`MPI_Comm_rank(comm,&rank)`	Determines the rank of the calling process within the communicator.
`MPI_Abort(comm,errorcode)`	Terminates all MPI processes associated with the communicator.
`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.
`MPI_Initialized(&flag)`	Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false (0).
`MPI_Wtime()`	Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
`MPI_Finalize()`	Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program

4.- Example: Environment Management Routines

Now, let's implement a first simple MPI program. We can use a command environment. You have to open a new terminal window. Each person may create a directory called **`mpi`** inside the $HOME directory (/home/master/masterX) and store there the programs we will implement during this tutorial. For example:

```
mkdir mpi
```

To write the program code you can use a graphical editor or **`vim`** editor that is available at /usr/bin/vim location. To edit this file simply write:

```
vim mpi_basic.c
```

Enter to your mpi directory and create the file `mpi_basic.c` inside:

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

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

      rc = MPI_Init (&argc, &argv);
      if (rc != MPI_SUCCESS)
      {
            printf ("Error starting MPI program. Terminating.\n");
            MPI_Abort (MPI_COMM_WORLD, rc);
            return -1;
      }

      MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
      MPI_Comm_rank (MPI_COMM_WORLD, &rank);
      printf ("Number of tasks= %d, My rank= %d\n", numtasks, rank);

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

      MPI_Finalize();
      return 0;
}
```

Code compilation

Once you have the program implemented, compile it and link:

```
        mpicc –g –o mpi_basic mpi_basic.c
```

`mpicc` is inside the /soft/openmpi/bin directory

Application execution

You need to start your application with the **`mpirun`** command that is inside the directory:

```
        mpirun
```

Enter into directory where you generated the executable program and put the following command:

```
        mpirun [option list] [executable] [args]
```

For example:

```
        mpirun -np 2 mpi_basic
```

would run a 2 process MPI application called "mpi_basic". You should obtain the following output:

```
        masterp@aolin11:~/mpi/example1$ mpirun -np 2 mpi_basic
        Number of tasks= 2, My rank= 0
        Number of tasks= 2, My rank= 1
```

*Note: the command must have a space between -np 2 and all mpirun options must appear before the executable name. Use the **`mpirun -h`** command for more information.*

OpenMPI uses the shared memory MPI implementation, and permits you to run with more processes than CPUs on a given node. However, if you use more MPI processes than there are CPUs, there will be a reduction in the parallelism achieved as processes will compete with each other for CPUs. They will also compete with any other users on the node.

5.- SLURM batch system

Machines in the lab (aolin[11-21]) work as an heterogeneous cluster for executing batch jobs, during certain time periods (from 21:00 to 9:00 hours and weekends). For this reason, several execution queues (partitions) have been configured using SLURM (Simple Linux Utility for Resource Management) as a job manager.

The available partitions in our system are:

aolin.q: consists of aolin11...aolin20

Each node in this partition offers 1 socket, 4 cores/socket and 2 threads/core, i.e., **8 execution resources/node**.
The maximum allowed execution time for a job is 8h in this partition.

cuda.q: consists of aolin[11-15], aolin7, aolin19, and aolin21, i.e., all the machines with 2 GPUs.

The resources offered depend on the used nodes:

- aolin11, 13, 15, 17 and 19 offer Gres=gpu:GeForceGTX480:1
- aolin12 and 14 offer Gres=gpu:Quadro2000:1
- aolin21 offers Gres=gpu:GeForceGTX680:1,gpu:GeForceGTX1080:1

test.q: only in aolin21

This node offers up to 8 threads and 2 GPUs and this partition is in place for allowing users to test the submission scripts. For this reason, it is always available, but the maximum job execution time is limited to 10 minutes.

5.1.- Basic commands for managing jobs

Command **sbatch** submits a job (script) to the indicated partition and returns the job id.

```
druiz@aolin21:$ sbatch submit_job_script.sub

Submitted batch job 320
```

Command **squeue** shows the partition and our jobs state. For example, state *qw* indicates *queue wait.*

```
druiz@aolin21:$  squeue –u druiz

JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
  320    test.q     test    druiz PD       0:00      1 aolin21
```

The job id is useful for example for eliminating a job using the **scancel** command:

```
druiz@aolin21:$  scancel 320
```

5.2.- Use cases

Normally, submission scripts are defined to send jobs to SLURM partitions. In this section, we show several basic examples, but you can find all the information about scheduling with SLURM at:

http://slurm.schedmd.com/

The first example shows a script that executes command *hostname* on 20 cores of 5 nodes.

The script would usually be in the working directory. Its initial lines would be comments for a regular shell (ex. cshell or bshell), but actually they define parameters for the job manager.

The description of these configuration parameters is:

Line	**Explanation**
1	Shell that will interpret the current script
2	Job name
3	File name for storing the job output
4	Number of required nodes
5	Number of required cores
6	Partition (queue) our job will be inserted in.

Submit_job_script.sub

```
 1  #!/bin/bash
 2  #SBATCH --job-name=test
 3  #SBATCH --output=result.txt
 4  #SBATCH -N 5 # number of nodes
 5  #SBATCH -n 20 # number of total cores
 6  #SBATCH --partition=aolin.q
 7
 8  srun hostname

10
```

The script may include any command that can be executed locally, but it could be necessary to load the appropriated modules if the command uses functions of a library.

By default, if the parameter –output is not indicated, a file <job-name-job-id.out> is generated.

If the number of required cores (parameter –n) is smaller than the number of cores available in the required nodes (parameter –N), SLURM will allocate N/n cores in each node (if N is not divisible by n, SLURM will allocate 1 more core in n – (N/n)*N nodes).

Next, we show how to submit different kinds of jobs: sequential or parallel.

<u>Sequential jobs</u>

These jobs only require one Core/CPU for being executed and SLURM can assign them to any free resource.

In this case, the command line could like the following:

```
$ sbatch –partition aolin.q ./serial.sub
```

where serial.sub would be a script like this:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --job-name=HOSTNAME-SERIAL

echo "Running on: `/bin/hostname`"
```

This job will be executed in a free CPU selected by the scheduler.

Parallel MPI jobs

A MPI application is a program composed of multiple processes running in different cores, where the processes interchange information using the MPI communication library.

By default, SLURM uses a scheduling policy called “backfill”, that consists in executing lower priority jobs without affecting higher priority ones. This means that SLURM tries to “fill” unused resources in nodes that have not been reserved “exclusively”.

In general, the SLURM default allocation method across nodes is block allocation (allocate all available execution resources in a node before using another node). The default allocation method within a node is cyclic allocation (allocate available CPUs in a round-robin fashion across the sockets within a node).

In addition, nodes can be reserved exclusively to the application or can be shared with other applications. This means that if an application uses for example 2/4 cores of a node, the other 2 cores can be left unused (the node has been assigned exclusively to the application) or can be used by another application. By default, resources are not assigned exclusively.

Example 1: non-exclusive.

Command line (the same for all the examples):

```
sbatch ./parallel.sub
```

Submit script:

```
#!/bin/bash
#
#SBATCH --job-name=N10n10
#SBATCH --output=N10n10.txt
#
#SBATCH -N 10 # number of nodes
#SBATCH -n 10 # number of cores
#SBATCH --partition=aolin.q

source /soft/modules-3.2.10/Modules/3.2.10/init/bash

module load openmpi/1.10.2

mpirun /home/caos/druiz/samples-SLURM/OpenMPI/mpihello-1.10.2
```

In this example, we have asked for 10 execution resources (total) in 10 different nodes, i.e., 1 execution resource/node. As far as the job is non-exclusive (default) the 7 execution resources remaining in each node can be assigned to other applications.

Example 2: exclusive

```
#!/bin/bash
#
#SBATCH --job-name=N10n10
#SBATCH --output=N10n10.txt
#
#SBATCH -N 10 # number of nodes
#SBATCH -n 10 # number of cores
#SBATCH --partition=aolin.q
#SBATCH --exclusive

source /soft/modules-3.2.10/Modules/3.2.10/init/bash

module load openmpi/1.10.2

mpirun /home/caos/druiz/samples-SLURM/OpenMPI/mpihello-1.10.2
```

As in the previous case, we have asked for 10 execution resources (total) in 10 different nodes, i.e., 1 execution resource/node. However, in this case the 10 nodes will be exclusively reserved for the application (leaving 7 execution resources free in each node)[1].

<u>Parallel GPU jobs</u>

Applications to be executed in a GPU must be submitted to the cuda.q partition (queue) and the environment variable CUDA_VISIBLE_DEVICES should be assigned in the script.

[1] This doesn't mean that the resources are not going to be used, they are available for the application, which can use them for example for running different threads (using OpenMP).

It must be indicated that we want to reserve a GPU resource, using the option Gres=gpu:name_GPU:number (for example, for aolin11 it would be something like Gres=gpu:GeForceGTX480:1). If we do not want to force a particular type of GPU, then we can use Gres=gpu:1.

Consequently, an example of a GPU cuda job could be:

```
#!/bin/bash
#
#SBATCH --job-name=GPU
#SBATCH -N 1 # number of nodes
#SBATCH -n 4 # number of cores
#SBATCH --partition=cuda.q
#$BATCH --gres=gpu:GeForceGTX680:1


export CUDA_VISIBLE_DEVICES=0
/usr/local/cuda-7.5/samples/0_Simple/clock/clock

exit 0
```

In this example, we are submitting the *clock* application to a to a “GeForceGTX680” device, independently of the node that has it (the particular node will be selected by SLURM). In addition, we are reserving 4 execution resources in that node.

It is also possible to ask for GPU (or more) without specifying the type, doing for example:

> #SBATCH --gres=gpu:2 (we are asking for 2 GPUs independently of the node that has them and the GPU type)

6.- Point to point communication

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

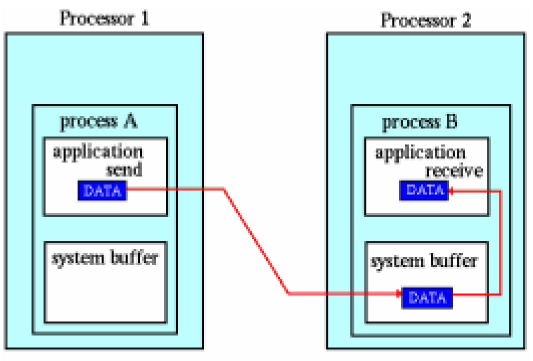
Note: Any type of send routine can be paired with any type of receive routine.

Buffers

There are two types of buffers when sending/receiving messages:

- **System buffer**
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust

  - Able to exist on the sending side, the receiving side, or both
- User managed address space (i.e. your program variables) is called the **application buffer**.



Path of a message buffered at the receiving process

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 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.
- 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.

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

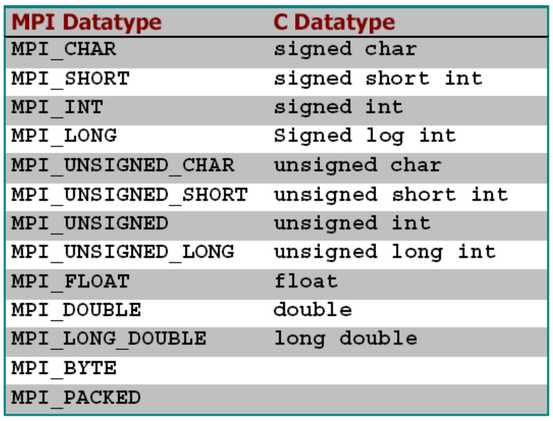
Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`

Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

where:

Parameter	**Description**
`buffer`	starting *address* of the data to be sent
`count`	number of elements to be sent
`datatype`	MPI datatype of each element
`dest`	rank of destination process
`tag`	message marker (set by user)
`comm`	MPI communicator of processors involved
`status`	actual source and tag of the message. In C, it is a pointer to a predefined structure MPI_Status (ex. `stat.MPI_SOURCE stat.MPI_TAG`). The actual number of bytes received is obtained from Status via `MPI_Get_count()`
`request`	a unique handle number used in a WAIT type routine to determine completion of the non-blocking operation

MPI data type and their corresponding C types:



MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blocking functions	
`MPI_Send`	Basic blocking send operation.
`MPI_Recv`	Receive a message and block until the requested data is available in the application buffer in the receiving task.
`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_Bsend`	Buffered blocking send: permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Routine returns after the data has been copied from application buffer space to the allocated send buffer. Must be used with the MPI_Buffer_attach routine.
`MPI_Rsend`	Blocking ready send. Should only be used if the programmer is certain that the matching receive has already been posted.

`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 (&req,&stat)`
	`MPI_Waitany (count,&array_of_reqs,&index,&stat)`
	`MPI_Waitall (count,&array_of_reqs,&array_of_stats)`
	`MPI_Waitsome (incount,&array_of_reqs,&outcount,`
	`       &array_of_offsets, &array_of_stats)`

Non-Blocking functions	
`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.
`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.
`MPI_Issend`	Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.
`MPI_Ibsend`	Non-blocking buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.
`MPI_Irsend`	Non-blocking ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

`MPI_Test`
`MPI_Testany`
`MPI_Testall`
`MPI_Testsome`
	MPI_Test checks the status of a specified non-blocking send or receive operation.
	`MPI_Test (&req,&flag,&stat)`
	`MPI_Testany (count,&array_of_reqs,&idx,&flag,&stat)`
	`MPI_Testall`
	`(count,&array_of_reqs,&flag,&array_of_stats)`
	`MPI_Testsome (incount,&array_of_reqs,&outcount,`
	`       &array_of_offsets, &array_of_stats)`

```
#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);
      printf("Process %d of %d is alive\n", rank, numtasks);

      if (numtasks != 2)
      {
         printf ("This program only works on 2 processors\n");
         return -1;
      }

      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("Process %d: Received %d char(s) from task %d with tag %d
             \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

      MPI_Finalize();
      return 0;
}
```

7.- Example: Point to point communication

Blocking communication

Let’s implement a program where process 0 pings process 1 and awaits return ping. Use blocking MPI_SEND and MPI_RECV. Write a program to do the following:

- Process 0 should send a single character (char) to the process 1 and should wait for the response character
- Process 1 should receive the character and send the response as a single character (char) to the process 0
- Use blocking MPI send and receive to pass the character between processes

Non-Blocking communication

Let’s implement a program which provides neighbour exchange in ring topology. The objective is to pass a message to the nearest neighbours one time. Use non-blocking MPI_SEND and MPI_RECV. Write a program to do the following:

- Each process should pass a single integer (it can be the rank of the process) to its neighbours (previous and next process)
- If the process rank is 0, this process should send a message to the last process
- If the process is the last process, it should send a message to the first process
- Use non-blocking MPI send and receive to pass the integer around a ring

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

int main(int argc, char **argv)
{
      int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
      MPI_Request reqs[4];
      MPI_Status stats[4];

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

      prev = rank-1;
      next = rank+1;
      if (rank == 0)  prev = numtasks - 1;
      if (rank == (numtasks - 1))  next = 0;

      MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD,
                &reqs[0]);
      MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD,
                &reqs[1]);

      MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
      MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      // do some work

      MPI_Waitall(4, reqs, stats);

      printf("Process %d: received %d value from task %d with tag %d \n",
             rank, buf[0], stats[0].MPI_SOURCE, stats[0].MPI_TAG);

      printf("Process %d: received %d value from task %d with tag %d \n",
             rank, buf[1], stats[1].MPI_SOURCE, stats[1].MPI_TAG);

      MPI_Finalize();
      return 0;
}
```

Write, compile and link both programs, then analyze the results and differences.

8.- Collective communication

Collective communication must involve **all** processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD. It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations.

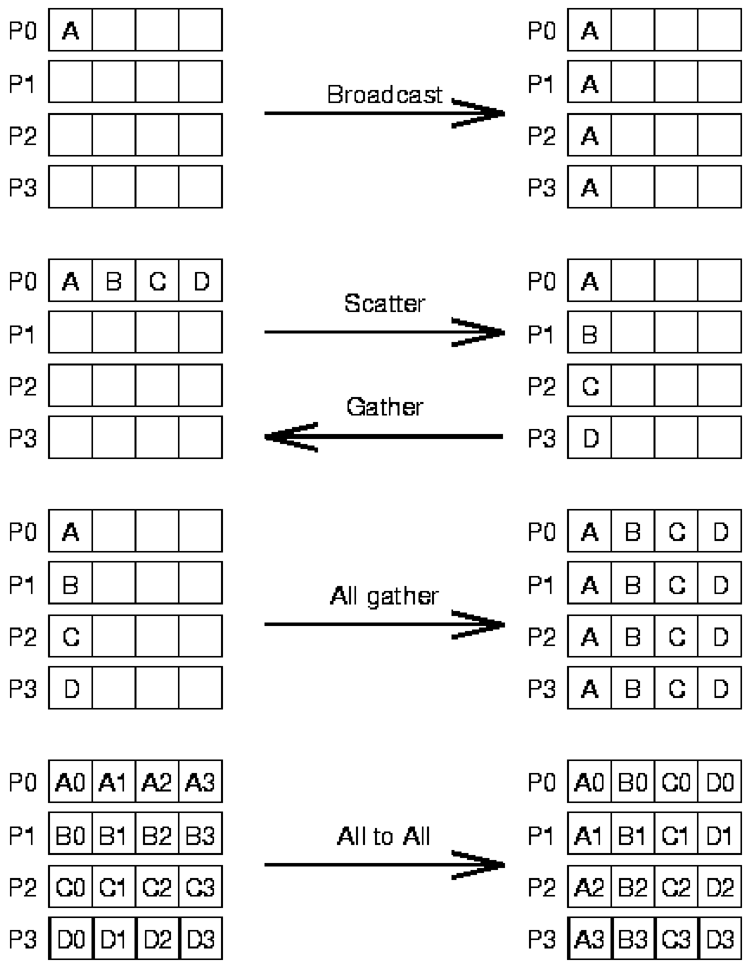
Types of Collective Operations:

- Synchronization - processes wait until all members of the group have reached the synchronization point.
- Data Movement - broadcast, scatter/gather, all to all.
- Collective Computation (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Programming Considerations and Restrictions:

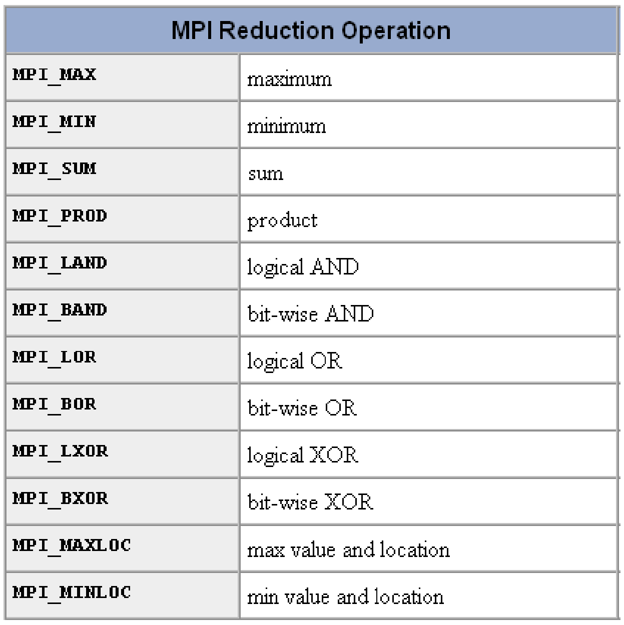
- Collective operations are blocking.
- Collective communication routines do not take message tag arguments.

Here we present the resume of the collective communications:



`MPI_Barrier(comm)`	Creates barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.
`MPI_Bcast(&buffer, count, datatype, root, comm)`	Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.
`MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`	Distributes distinct messages from a single source task to each task in the group.
`MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)`	Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.
`MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)`	Applies a reduction operation on all tasks in the group and places the result in one task with rank “root”.
`MPI_Alltoall(&sendbuf, sendcount, sendtype, &recvbuf, recvcnt, recvtype, comm)`	Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

The predefined MPI reduction operations appear below:



MPI Reduction Operation	
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

9.- Example: collective communication

Distribution of vectors

Now we will implement a program which contains a matrix of N vectors and distributes each vector to distinct processes. The objective is to perform a scatter operation on the vectors of a matrix. Use MPI scatter collective communication. Write a program to do the following:

- The program defines a matrix of size N x N.
- The program must be executed with N processes.
- The process 0 scatters the matrix to the rest of the process.

```
#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 = 0;
        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();
      return 0;
}
```

Analyze the program output. See the sample:

```
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
```

Operations on a vector

Now we will implement a program which contains a vector and distributes its elements between distinct processes. Then each process performs local calculations. All the processes find the global sum of the calculated vectors and finally the root process gathers the results.

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

main(int argc, char **argv)
{
int rank, size, myn, i, N;
double *vector, *myvec, mysum, total;

MPI_Init(&argc, &argv );
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
/* In the root process read the vector length, initialize
the vector and determine the sub-vector sizes */
if (rank == 0)
{
  printf("Enter the vector length : ");
  scanf("%d", &N);
  vector = (double *)malloc(sizeof(double) * N);
  for (i = 0; i < N; i++)
    vector[i] = 1.0;
  myn = N / size;
}
/* Broadcast the local vector size */
MPI_Bcast(&myn, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* allocate the local vectors in each process */
myvec = (double *)malloc(sizeof(double)*myn);

/* Scatter the vector to all the processes */
MPI_Scatter(vector,myn,MPI_DOUBLE,myvec,myn,MPI_DOUBLE,0,MPI_COMM_WORLD);

/* Find the sum of all the elements of the local vector */
mysum = 0;
for (i = 0; i < myn; ++i)
  mysum += myvec[i];
printf("[%d] my sum is: %f\n", rank, mysum);

/* Find the global sum of the vectors */
MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

/* Multiply the local part of the vector by the global sum */
for (i = 0; i < myn; i++) myvec[i] *= total;

/* Gather the local vector in the root process */
MPI_Gather(myvec,myn,MPI_DOUBLE,vector,myn,MPI_DOUBLE,0,MPI_COMM_WORLD);
if (rank == 0)
  for (i = 0; i < N; i++)
      printf("[%d] %f\n", rank, vector[i]);
MPI_Finalize();
return 0;
}
```

Bibliography

1. Training tutorial: Message Passing Interface (MPI) - LLNL
http://www.llnl.gov/tutorials/mpi/

2. OpenMPI
http://www.open-mpi.org