Assignment No.: 03

Title: An MPI program for calculating a quantity called coverage from data files

Aim: Write a MPI program for calculating a quantity called coverage from data files

Objectives:

- To understand clustering
- To implement the Message Passing Interface

Theory:

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.

- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:

    - Practical
    - Portable
    - Efficient
    - Flexible

Reasons for using MPI:

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

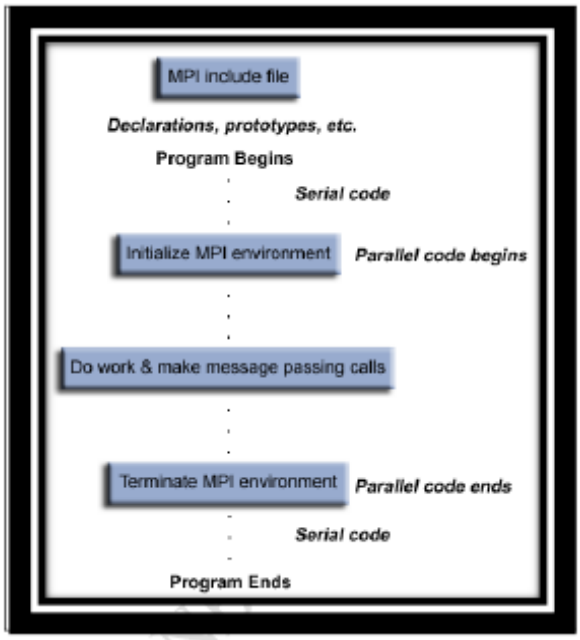
- **Portability**- There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities -**Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

- **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.

General MPI program structure:



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. Most of the commonly used ones are described below.

1) **MPI_Init :**

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

MPI_Init (&argc,&argv)
MPI_INIT (ierr)

2) **MPI_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_Comm_size(comm,&size)
MPI_COMM_SIZE (comm, size, ierr)

3) **MPI_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 numbers 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_Comm_rank(comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)

4) MPI_Abort :

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

MPI_Abort(comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)

5) MPI_Get_processor_name :

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

MPI_Get_processor_name(&name,&resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)

6) MPI_Get_version:

Returns the version and subversion of the MPI standard that's implemented by the library.

MPI_Get_version(&version,&subversion)
MPI_GET_VERSION (version,subversion,ierr)

7) MPI_Initialized :

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

MPI_Initialized(&flag)
MPI_INITIALIZED (flag, ierr)

8) MPI_Wtime :

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

9) **MPI_Finalize :**

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

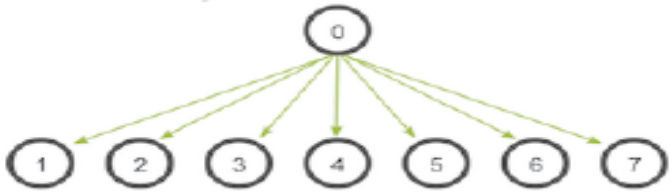
MPI_Finalize()

MPI_FINALIZE (ierr)

10) MPI_Bcast :

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

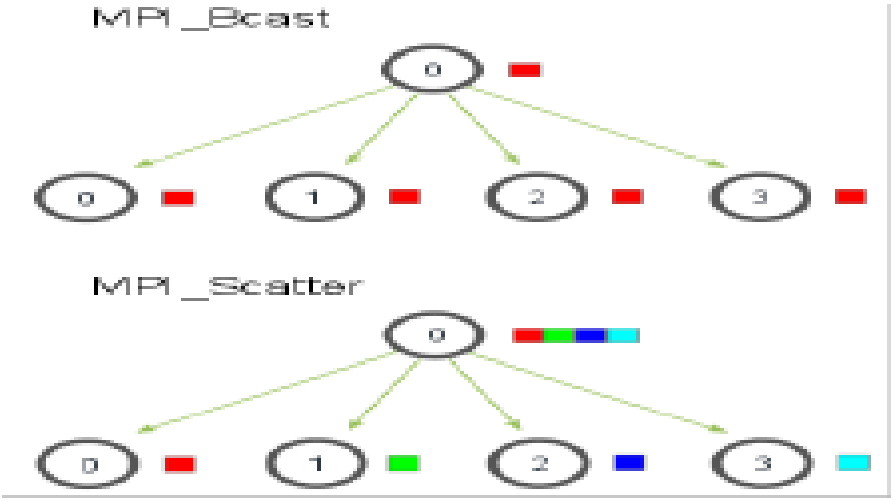
The communication pattern of a broadcast looks like this:



In this example, process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.

11) MPI_Scatter:

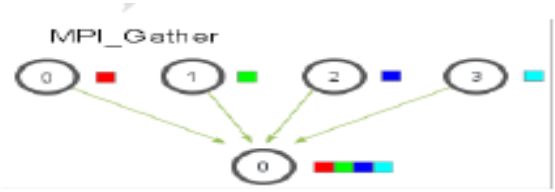
MPI_Scatter is a collective routine that is very similar to MPI_Bcast. MPI_Scatter involves a designated root process sending data to all processes in a communicator. The primary difference between MPI_Bcast and MPI_Scatteris small but important. MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.



In the illustration, MPI_Bcast takes a single data element at the root process (the red box) and copies it to all other processes. MPI_Scatter takes an array of elements and distributes the elements in the order of process rank. The first element (in red) goes to process zero, the second element (in green) goes to process one, and so on. Although the root process (process zero) contains the entire array of data, MPI_Scatter will copy the appropriate element into the receiving buffer of the process.

12) MPI_Gather:

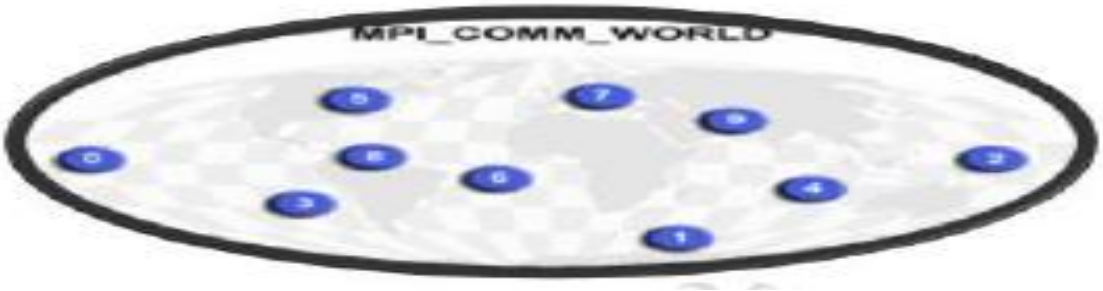
MPI_Gather is he inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching. Below is a simple illustration of this algorithm.



Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers them to the root process. The elements are ordered by the rank of the process from which they were received.

RANK:

- Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
- Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).
- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- Communicators and groups will be covered in more detail later. For now, simply use MPI_COMM_WORLD
- whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.



Steps for creating cluster of computers:

1. At least Two Computers with a Linux Distribution installed in it (I'll use Ubuntu 14.04 here). Make sure that your system has GCC installed in it.

2. A network connection between them. If you have just two computers, you can connect them using an Ethernet wire. Make sure that IP addresses are assigned to them. If you don‟t have a router to assign IP, you can statically assign them IP addresses.

3. Rest of the document will assume that we are having two computers having host names node0 and node1. Let node0 be the master node.

4. The following steps are to be done for every node:

5. Add the nodes to the /etc/hosts file. Open this file using your favorite‟s text editor and add your node's IP address followed by its host name. Give one node information per line. For example,

   node0 10.1.1.1
   node1 10.1.1.2

6. Create a new user in both the nodes. Let us call this new user as mpiuser. You can create a new user through GUI by going to System->Administration->Users and Groups and click "Add User". Create a new user called mpiuser and give it a password. Give administrative privileges to that user. Make sure that you create the same user on all nodes. Although same password on all the nodes is not necessary, it is recommended that you do so because it'll eliminate the need to remember passwords for every node.

7. Now download and install ssh-server in every node. Execute the command sudo apt-get install open ssh-server in every machine

8. Now logout from your session and log in as mpiuser.

9. Open terminal and type the following ssh-keygen -t dsa. This command will generate a new ssh key. On executing this command, it'll ask for a paraphrase. Leave it blank as we want to create a passwordlessssh (Assuming that you've a trusted LAN with no security issues)

10. A folder called .ssh will be created in your home directory. It‟s a hidden folder. This folder will contain a file id_dsa.pub that contains your public key. Now copy this key to another file called authorized_keys in the same directory. Execute the command in the terminal cd /home/mpiuser/.ssh; cat id_dsa.pub >>authorized_keys;.

11. Now download MPICH from the following website (MPICH1). Please download the MPICH 1.xx version from the website. Do not download MPICH 2 version. I was unable to get MPICH 2 to work in the cluster.

12. Untar the archive and navigate into the directory in the terminal. Execute the following commands:

    mkdir/home/mpiuser/mpich1
    ./configure--prefix=/home/mpiuser/mpich1
    make
    make install

13. Open the file .bashrc in your home directory. If file does not exist, create one. Copy the following code into thatfile

    exportPATH=/home/mpiuser/mpich1/bin:$PATH
    exportPATH
    LD_LIBRARY_PATH="/home/mpiuser/mpich1/lib:$LD_LIBRARY_PATH"
    export LD_LIBRARY_PATH

14. Now we'll define the path to MPICH for SSH. Run the following command: sudo echo /home/mpiuser/mpich1/bin >> /etc/environment

15. Now logout and login back into the user mpiuser.

16. In the folder mpich1, within the sub-directory share or util/machines/ a file called machines. LINUX will be found. Open that file and add the hostnames of all nodes except the home node ie. If you're editing the machines. LINUX file of node0, then that file will contain host names of all nodes except node0. By default MPICH executes a copy of the program in the home node. LINUX file for the machine node0 is as follows
    node1: 2
    The number after: indicates number of cores available in each of the nodes.

Mathematical Model:

Let M be the system.

M = {I, P, O, S, F}

Let I be the input.
I = {k}
k = any number.

Let P be the process.
P= {n,q,r}
n=squaring the number.
q=calculating the run time
r= plotting the graph

Let O be the Output.
O = {c}
c =square of the number and run time.

Let S be the case of Success.
S = {l}
l = Satisfied all conditions.

Let F be the case of Failure.
F = {f}

f = Satisfied result not generated.

Conclusion:

Hence we studied MPI program for calculating a quantity called coverage from data files

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define v 1 /* verbose flag, output if 1, no output if 0 */
int main ( int argc, char *argv[] )
{
        int myid,j,*data,tosum[25],sums[5],sums2[5];
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        if(myid==0) /* manager allocates and initializes the data */
        {

                data = (int*)calloc(100,sizeof(int));
                for (j=0; j<100; j++) data[j] = j+1;
                if(v>0)
                {
                        printf("The data to sum : ");
                        for (j=0; j<100; j++)
                        printf(" %d",data[j]);
                        printf("\n");
                }
        }
        MPI_Scatter(data,25,MPI_INT,tosum,25,MPI_INT,0,MPI_COMM_WORLD);
        if(v>0) /* after the scatter, every node has 25 numbers to sum*/
        {
                printf("Node %d has numbers to sum :",myid);
                for(j=0; j<25; j++) printf(" %d", tosum[j]);
                printf("\n");
        }
        sums[myid] = 0;
        for(j=0; j<25; j++) sums[myid] += tosum[j];
        if(v>0) printf("Node %d computes the sum %d\n",myid,sums[
        myid]);

        MPI_Gather(&sums[myid],1,MPI_INT,&sums2[myid],1,MPI_INT,0,MPI_COMM_WORLD);

        if(myid==0) /* after the gather, sums contains the four sums*/
        {
                printf("The four sums : ");
                printf("%d",sums2[0]);
                for(j=1; j<4; j++) printf(" + %d", sums2[j]);
                for(j=1; j<4; j++) sums2[0] += sums2[j];
                printf(" = %d, which should be 5050.\n",sums2[0]);
        }

        MPI_Finalize();
        return 0;
}
```

machinefile

```
ub0:20

ub1:20
```

Code

```
mpiu@DB21:/mirror/mpiu/CL4 prog/A-3$ mpicc A3_MPI.c



mpiu@DB21:/mirror/mpiu/CL4 prog/A-3$ mpiexec -n 3 -f machinefile ./a.out
```

```
The data to sum :  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99 100

Node 0 has numbers to sum :Node 1 has numbers to sum : 26 27 28 29 30 31 32 33 34 35 36 37
38 39Node 2 has numbers to sum : 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75

Node 2 computes the sum 1575

 40 41 42 43 44 45 46 47 48 49 50

Node 1 computes the sum 950

 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Node 0 computes the sum 325

The four sums : 325 + 950 + 1575 + 32598 = 35448, which should be 5050.

mpiu@DB21:/mirror/mpiu/CL4 prog/A-3$
```