****

**Name of the Student: Choudhari Arti Rajendra Roll no: 08**

**CLASS:- B.E.[I.T] Division: A Course:- 2015**

**Assignment No.2**

**COMPUTER LABORATORY-IX**

**Marks: /10**

**Date of Performance: Sign with Date:**

## ASSIGNMENT NO.2

**ProblemStatement:**

To develop any distributed application using Message Passing Interface (MPI).

**Objective:**

1) The course aims to provide an understanding of the principles on which the distributed systems are based; their architecture, algorithms and how they meet the demands of Distributed applications.

2) The course covers the building blocks for a study related to the design and the implementation of distributed systems and applications.

**Outcomes:**

1) Demonstrate knowledge of the core concepts and techniques in distributed systems.

2) Learn how to apply principles of state-of-the-Art Distributed systems in practical application.

3) Design, build and test application programs on distributed systems.

**PEOs:2; POs: a,b,c,d,f,g,i, l, m ; PSOs: 1,2,3 and COs satisfied: 1, 2, 3.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Assignment No.** | **Assignment Title** | **Assignment Statement** | **Scenarios** | **Software Required** |
| 2 | MPI | To develop any distributed application using Message Passing Interface (MPI). | Three clients want to communicate with the server. The server decides to whom should I communicate. The communication be sequentially or based on server’s decision. | 1) mpj.jar file for java MPI :mpjv0\_44.jar |

## Tools /Environment:

Java Programming Environment, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express (mpj.jar)

## RelatedTheory:

**Message passing** is a popularly renowned mechanism to implement parallelism in applications; it is also called MPI. The MPI interface for Java has a technique for identifying the user and helping in lower startup overhead. It also helps in collective communication and could be executed on both **shared memory and distributed systems**. MPJ is a familiar Java API for MPI implementation. mpiJava is the near flexible Java binding for MPJstandards.

Currently developers can produce more efficient and effective parallel applications using messagepassing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. The parallel computing world is mainly concerned with

`symmetric' communication, occurring in groups of interacting peers. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI).

## Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

#include <mpi.h>

The mpi.hfile contains the definitions and declarations necessary for compiling an MPI program.

**MPI\_Init**initializes the execution environment for MPI. It is a “share nothing” modality in which the outcome of any one of the concurrent processes can in no way be influenced by the intermediate results of any of the other processes. Command has to be called before any other MPI call is made, and it is an error to call it more than a single time within the program. **MPI\_Finalize**cleans up all the extraneous mess that was first put into place by MPI\_Init.

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases

are also communicating with each other. To do this, an object called the “communicator” is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. “**MPI\_COMM\_WORLD**” communicator contains all the concurrent processes making up an MPI program.

The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

intMPI\_Comm\_size(MPI\_Commcomm, int \_size).

The function "MPI\_Comm\_size” required to return the number of processes; int size. MPI\_Comm\_size(MPI\_COMM\_WORLD,&size); This will put the total number of processes in the MPI\_COMM\_WORLD communicator in the variable size of the process data context. Every process within the communicator has a unique ID referred to as its “rank”. MPI system automatically and arbitrarily assigns a unique positive integer value, starting with 0, to all the processes within the communicator. The MPI command to determine the process rank is:

intMPI\_Comm\_rank (MPI\_Commcomm, int \_rank).

The sendfunction is used by the source process to define the data and establish the connection of the message. The send construct has the following syntax:

intMPI\_Send (void \_message, int count, MPI\_Datatypedatatype, intdest, int tag, MPI\_Commcomm)

The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies the number of data elements of which the message is composed. The third operand indicates the data type of the elements that make up the message.

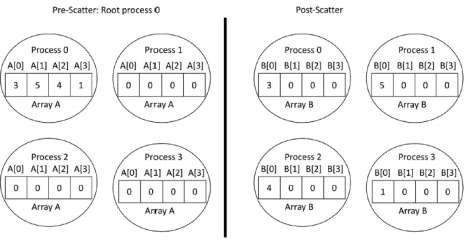
The receive command (MPI\_Recv) describes both the data to be transferred and the connection to be established. The MPI\_Recv construct is structured as follows:

intMPI\_Recv (void \_message, int count, MPI\_Datatypedatatype, int source, int tag, MPI\_Commcomm, MPI\_Status \_status)

The source field designates the rank of the process sending the message.

**Communication Collectives:** Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.

**The scatter operation:** The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separateprocess.



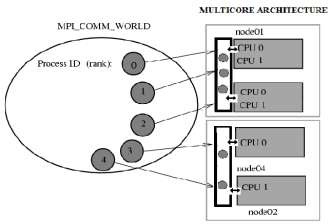
**MPJ Express** is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds.** The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.

MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).**

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We except that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platforms

## Designing thesolution:

While designing the solution, we have considered the multi-core architecture as per shown in the diagram below. The communicator has processes as per input by the user. MPI program will execute the sequence as per the supplied processes and the number of processor cores available for the execution.



## Implementing thesolution:

1. For implementing the MPI program in multi-core environment, we need to install MPJ expresslibrary.
   1. Download MPJ Express (mpj.jar) and unpack it.
   2. Set MPJ\_HOME and PATH environmentvariables:
   3. exportMPJ\_HOME=/path/to/mpj/
   4. exportPATH=$MPJ\_HOME/bin:$PATH
2. Write Hello World parallel Java program and save it as HelloWorld.java(Asign2.java).
3. Compile a simple Hello World (Asign)parallel Java program
4. Running MPJ Express in the Multi-coreConfiguration.

**Conclusion:**

There has been a large amount of interest in parallel programming using Java. mpj is an MPI binding with Java along with the support for multicore architecture so that user can develop the code on its own laptop or desktop. This is an effort to develop and run parallel programs **according to MPI standard.**

**Code:-**

import mpi.MPI;

public class ScatterGather {

public static void main(String args[]){

//Initialize MPI execution environment

MPI.Init(args);

//Get the id of the process

int rank = MPI.COMM\_WORLD.Rank();

//total number of processes is stored in size

int size = MPI.COMM\_WORLD.Size();

int root=0;

//array which will be filled with data by root process

int sendbuf[]=null;

sendbuf= new int[size];

//creates data to be scattered

if(rank==root){

sendbuf[0] = 10;

sendbuf[1] = 20;

sendbuf[2] = 30;

sendbuf[3] = 40;

//print current process number

System.out.print("Processor "+rank+" has data: ");

for(int i = 0; i< size; i++){

System.out.print(sendbuf[i]+" ");

}

System.out.println();

}

//collect data in recvbuf

int recvbuf[] = new int[1];

//following are the args of Scatter method

//send, offset, chunk\_count, chunk\_data\_type, recv, offset, chunk\_count, chunk\_data\_type, root\_process\_id

MPI.COMM\_WORLD.Scatter(sendbuf, 0, 1, MPI.INT, recvbuf, 0, 1, MPI.INT, root);

System.out.println("Processor "+rank+" has data: "+recvbuf[0]);

System.out.println("Processor "+rank+" is doubling the data");

recvbuf[0]=recvbuf[0]\*2;

//following are the args of Gather method

//Object sendbuf, int sendoffset, int sendcount, Datatype sendtype,

//Object recvbuf, int recvoffset, int recvcount, Datatype recvtype,

//int root)

MPI.COMM\_WORLD.Gather(recvbuf, 0, 1, MPI.INT, sendbuf, 0, 1, MPI.INT, root);

//display the gathered result

if(rank==root){

System.out.println("Process 0 has data: ");

for(int i=0;i<4;i++){

System.out.print(sendbuf[i]+ " ");

}

}

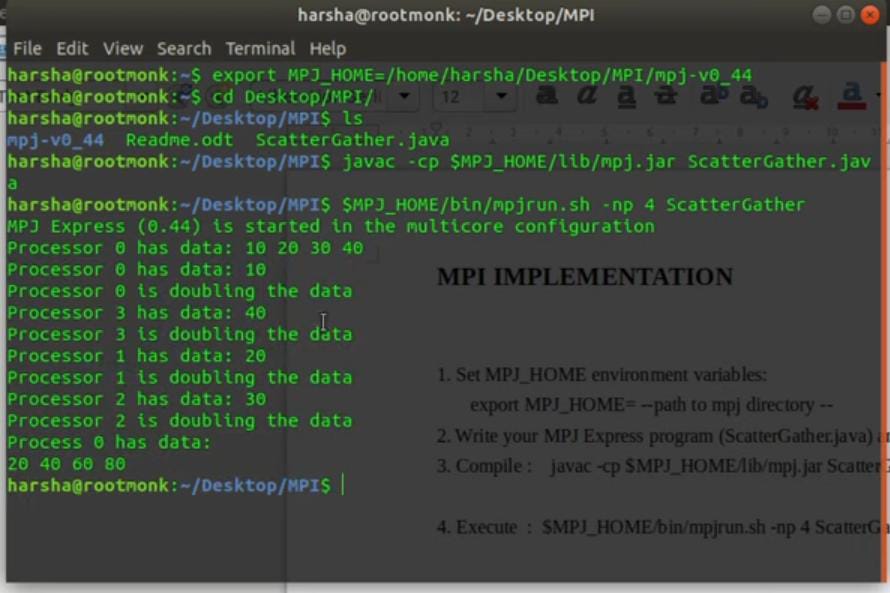
//Terminate MPI execution environment

MPI.Finalize();

}

}

**Output:**

****