ASSIGNMENT NO. 3

Problem Statement :

Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.

Tools/Environment :

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

Theory:

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. mpi Java is the near flexible Java binding for MPJ standards. Currently developers can produce more efficient and effective parallel applications using message passing.

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.h file 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:

int MPI_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:

int MPI_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:

int MPI_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.

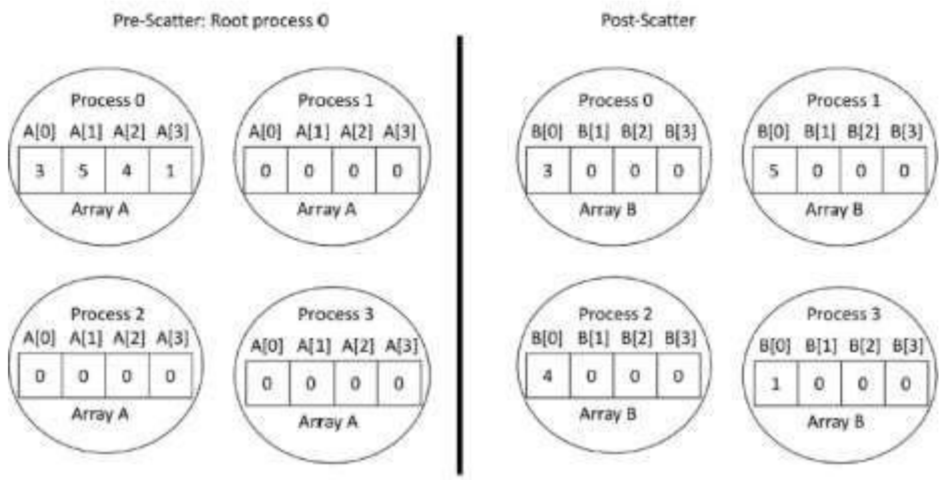
The receive command (MPI_Recv) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

int MPI_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 separate process.



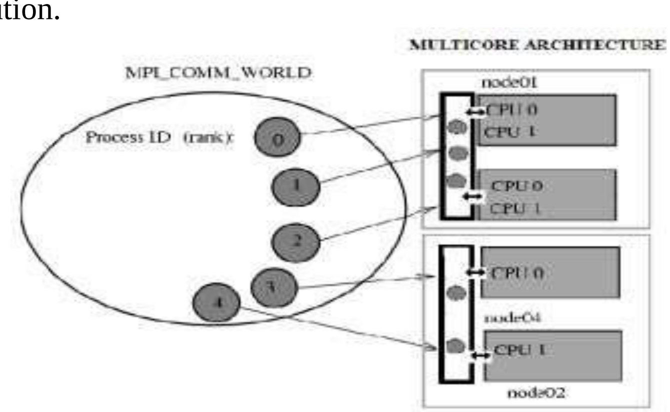
MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds.** The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing 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 the solution:

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 the solution:

1. For implementing the MPI program in multi-core environment, we need to install MPJ express library.
    a. Download MPJ Express (mpj.jar) and unpack it.
    b. Set MPJ_HOME and PATH environment variables:
    c. export MPJ_HOME=/path/to/mpj/
    d. export PATH=$MPJ_HOME/bin:$PATH
2. Write Sum parallel Java program and save it as Sum.java
3. Compile a simple Sum parallel Java program
4. Running MPJ Express in the Multi-core Configuration.

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.