

ASSIGNMENT NO. 4

Title: Message Passing Interface (MPI)

Aim: 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

Objective:

1. To introduce you to the fundamentals of MPI.
2. To understand widely used standard for writing message passing programs.

Tools / Environment:

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

Related 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. mpiJava 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_Comm comm, 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_Comm comm, int _rank).
```

The send function 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_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

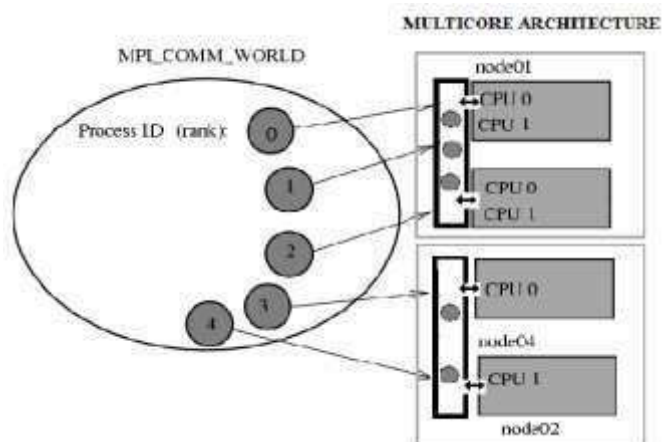
MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program**

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 Hello World parallel Java program and save it as HelloWorld.java (Assign2.java).
- 3. Compile a simple Hello World (Assign) parallel Java program
- 4. Running MPJ Express in the Multi-core Configuration.

Conclusion:

Thus Student develop a distributed system application, 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.

Outcome:

- 1. Students learn fundamentals of parallel programming and MPI.
- 2. Students design parallel algorithms and write parallel programs using the MPI library.

FAQ:

- 1. What is use of MPI?
- 2. Application in which we are using MPI?
- 3. Why we are providing rank to process in MPI.
- 4. Explain MPI operations.
- 5. Explain Different data types of MPI.
- 6. Draw MPI architecture.
- 7. What is MPI_ABORT?
- 8. What is MPI_FINALIZE
- 9. What is difference MPI_ABORT and MPI_FINALIZE