# ASSIGNMENT NO. 03

**TITLE: MPI**

**AIM:** To 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:**

Students should be able to understand:-

- Basic concept of Message Passing Interface.
- Communication using MPI.
- Concept of OpenMPI.

**TOOLS / ENVIRONMENT:**

- **S/W:**
    - o Fedora/ Ubantu.
    - o Openmpi, Terminal, CC, editor
- **H/W:**
    - o Any basic configuration loaded machine (e.g. P IV )

**THEORY:**

**Introduction**

Message Passing Interface (MPI) is a standardized and portable message- passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

**Overview and Goals**

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the \classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development. The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows:

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processors, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

**An Interface Specification:**

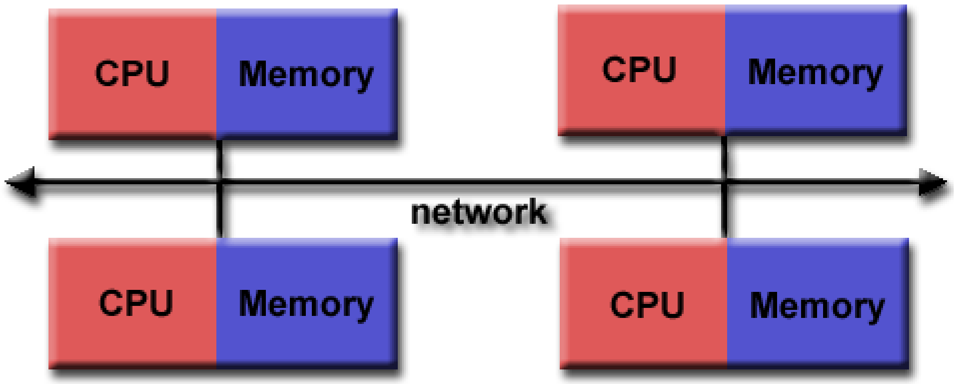
- M P I = Message Passing Interface
- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model: data is moved 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
- The MPI standard has gone through a number of revisions, with the most recent version being MPI 3.
- Interface specifications have been defined for C and Fortran90 language bindings:
    - C++ bindings from MPI-1 are removed in MPI-3

- MPI-3 also provides support for Fortran 2003 and 2008 features

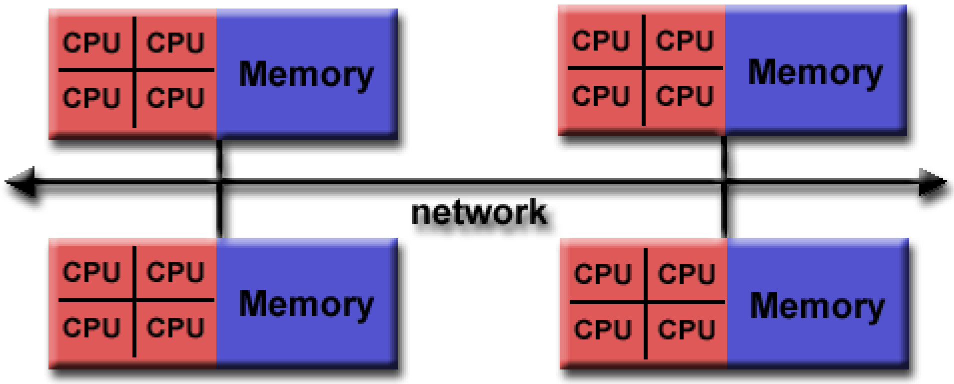
- Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

**Programming Model:**

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).




- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementers adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.




- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

**Concepts**

MPI provides a rich range of abilities. The following concepts help in understanding and providing context for all of those abilities and help the programmer to decide what functionality to use in their application programs. Four of MPI's eight basic concepts are unique to MPI-2.

**Communicator**

Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made. MPI understands single group intra-communicator operations, and bilateral inter-communicator communication. In MPI-1, single group operations are most prevalent. Bilateral operations mostly appear in MPI-2 where they include collective communication and dynamic in-process management. Communicators can be partitioned using several MPI commands. These commands include `MPI_COMM_SPLIT`, where each process joins one of several colored sub-communicators by declaring itself to have that color.

**Point-to-point basics**

A number of important MPI functions involve communication between two specific processes. A popular example is `MPI_Send`, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed. MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

**Collective basics**

Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the `MPI_Bcast` call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the `MPI_Reduce` call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. `MPI_Reduce` is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result. Other operations perform more sophisticated tasks, such as `MPI_Alltoall` which rearranges n items of data such that the nth node gets the nth item of data from each.

**Basic MPI types**

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SIGNED_CHAR | signed char |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_SHORT | signed short |
| MPI_UNSIGNED_SHORT | unsigned short |
| MPI_INT | signed int |
| MPI_UNSIGNED | unsigned int |
| MPI_LONG | Signed long |
| MPI_UNSIGNED_LONG | unsigned long |

| | |
|---|---|
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |

**Basic MPI Operations**

| | |
|---|---|
| MPI_Comm | the basic object used by MPI to determine which processes are involved in a communication |
| MPI_Status | the MPI_Recv operation takes the address of an MPI_Status structure as an argument (which can be ignored with MPI_STATUS_IGNORE). |
| MPI_Init | Initialize the MPI execution environment<br>int MPI_Init( int *argc, char ***argv ) |
| MPI_Comm_size | Determines the size of the group associated with a communicator<br>int MPI_Comm_size( MPI_Comm comm, int *size ) |
| MPI_Open_port | MPI_Open_port establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system.<br>MPI copies a system-supplied port name into port_name. port_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_PORT_NAME. |
| MPI_Comm_accept | MPI_Comm_accept establishes communication with a client. It is collective over the calling communicator. It returns an intercommunicator that allows communication with the client, after the client has connected with the MPI_Comm_accept function using the MPI_Comm_connect function. |
| MPI_Send | MPI_Send performs a standard-mode, blocking send. |
| MPI_Recv | This basic receive operation, MPI_Recv, is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started). |
| MPI_Comm_free | This operation marks the communicator object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intracommunicators and intercommunicators. |
| MPI_Close_port | MPI_Close_port releases the network address represented by port_name. |
| MPI_Finalize | This routine cleans up all MPI states. Once this routine is called, no MPI routine (not even MPI_Init) may be called, except for MPI_Get_version, MPI_Initialized, |

| | |
|---|---|
| | and `MPI_Finalized`. Unless there has been a call to `MPI_Abort`, you must ensure that all pending communications involving a process are complete before the process calls `MPI_Finalize`. If the call returns, each process may either continue local computations or exit without participating in further communication with other processes. |
| `MPI_Abort` | This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a UNIX or POSIX environment should handle this as a return errorcode from the main program or an `abort (errorcode)` |
| `MPI_Comm_disconnect` | `MPI_Comm_disconnect` waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to `MPI_COMM_NULL`. It is a collective operation. |

**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.

**Running Procedure:**
**Installation Sequence"-**

```
sudo yum install openmpi-devel

export PATH=$PATH:/usr/lib64/openmpi/bin
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:usr/lib64/openmpi/bin
```

Another method

```
>mkdir openmpi
>cd openmpi
>copy .gz file
>tar -xzvf openmpi-1.8.7.tar.gz
>cd openmpi-1.*
>./configure --prefix=$HOME/openmpi-1.8.7
>make all
>make install

#close terminal

#open terminal
>gedit .bashrc

#add following at the end of file

export PATH=$PATH:/home/student/openmpi-1.8.7/bin
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/student/openmpi-
1.8.7/bin

#save file and close gedit
#close terminal
```

**IMPLEMENTATION:**

(Students should write here the implementation for their program. Students should attach printout of their programs with commands used to run the programs. Also attach the proper outputs of programs.)

**Program Execution:-**

- compile
    - `mpicc server.c -o server`
    - `mpicc client.c -o client`
- run server
    - `mpirun -np 1 ./server`

    (it will display output similar to below (not necessarily the same) Server available at port:
    4290510848.0;tcp://192.168.1.101:35820;tcp://192.168.122.1:35820+4290510849.0;tcp://192.168.1 .101:40208;tcp://192.168.122.1:40208:300
    copy the port-string from the terminal output. we are going to supply this port-string as a first command line argument to the client

- open another terminal
    - ```
      mpirun -np 1 ./client
      '4290510848.0;tcp://192.168.1.101:35820;tcp://192.16
      8.122.1:35820+4290510849.0;tcp://192.168.
      1.101:40208;tcp://192.168.122.1:40208:300'
      ```

      (Don't forget to insert single quotes at the start & end of the port-string.)

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

**REFERENCES:** "**Distributed Systems: Concepts and Design**" by George Coulouris, J Dollimore and Tim Kindberg, Pearson Education, ISBN: 9789332575226, 5th Edition, 2017

**FAQ:**
1. What is MPI?
2. What features are included in MPI?
3. Why to use MPI?
4. What is communicator?
5. Explain point-to-point communication in MPI.
      ```