Assignment B - 7

Aim:

To implement parallel ODD-Even Merge Sort.

Problem Statement:

Perform concurrent ODD-Even Merge sort using HPC infrastructure (preferably BBB) using Python/ Scala/ Java/ C++.

Theory:

What is MPI:

MPI = 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.

MPI runs on virtually any hardware platform:

- Distributed Memory
- Shared Memory
- Hybrid

General MPI Program Structure:

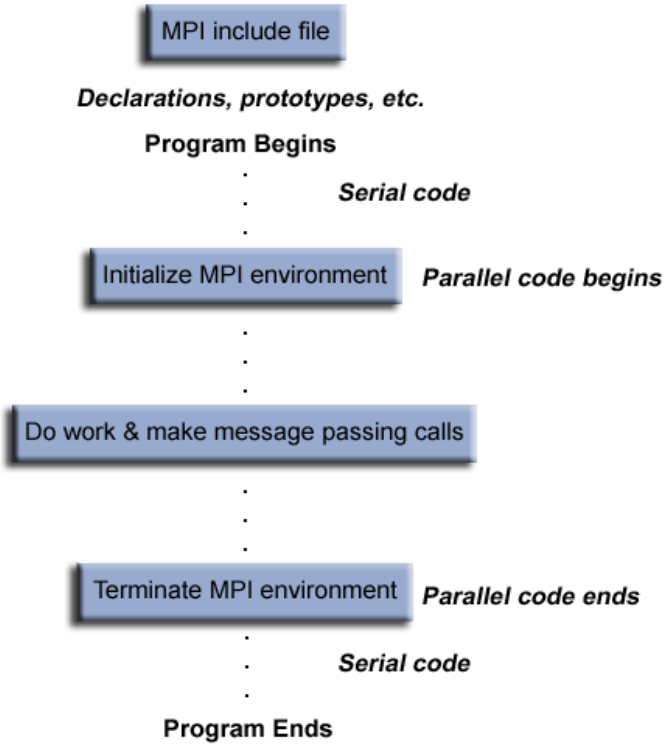

Figure 1: MPI Program Structure

Communicators and Groups:

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. MPI_COMM_WORLD - it is the predefined communicator that includes all of your MPI processes.

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).

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.

- **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.

  ```
  void MPI::Init(int& argc, char**& argv)
  void MPI::Init()
  ```

- **MPI::Comm.Get_size():** Get the number of processors this job is using.

  ```
  int MPI::COMM_WORLD.Get_size()
  ```

- **MPI::Comm.Get_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 number 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.

  ```
  int MPI::Comm.Get_rank()
  ```

- **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.

  ```
  void MPI::Finalize()
  ```

MPI Message Passing Routine Arguments:

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Table 1: MPI Message Passing Routine

Blocking sends	void MPI::Comm.Send(buffer,count,type,dest,tag)
Blocking receive	void MPI::Comm.Recv(buffer,count,type,source,tag,comm,status)

- **Communicator:** Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator COMM_WORLD is usually used.
- **Buffer:** Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: &var1
- **Data Count:** Indicates the number of data elements of a particular type to be sent.
- **Destination:** An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.
- **Data Type:** For reasons of portability, MPI predefines its elementary data types.
- **Source:** An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

Table 2: C++ Data Types

MPI::CHAR	signed char
MPI::INT	signed int
MPI::LONG	signed long int
MPI::SIGNED_CHAR	signed char
MPI::UNSIGNED_LONG	unsigned long int
MPI::FLOAT	float
MPI::C_BOOL	_Bool
MPI::BYTE	8 binary digits

- **Tag:** Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

- **Status:** For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG).

- **Request:** Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique “request number". The programmer uses this system assigned “handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request.

What is odd-even sort algorithm:

The odd even algorithm sorts a given set of n numbers where n is even in n phases. Each phase requires n/2 compare and exchange operations. It oscillates between odd and even phases successively.

Odd-Even Merging of Two Sorted Lists:

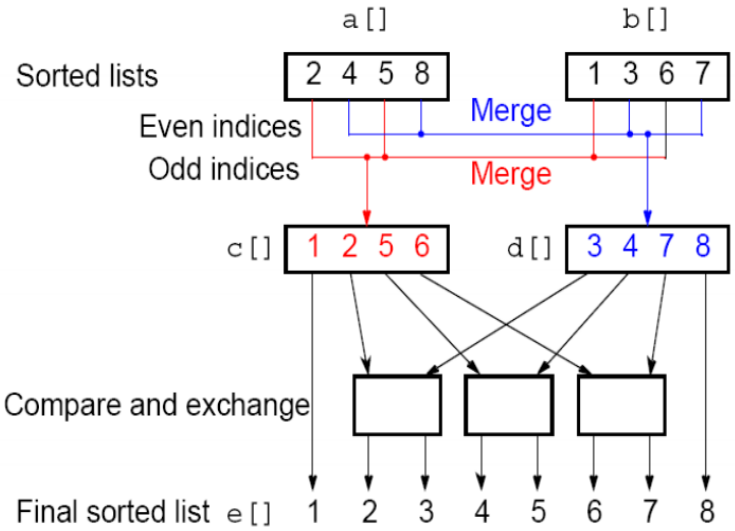

Figure 2: Merging

Logic:

Let {b1,b2,..,bn} be the sequence to be sorted. During the odd phase the elements with odd numbered subscripts are compared their neighbors on the right and exchanged if necessary. That is the elements {b1,b2},{b3,b4}..{bn-1,bn} are compared and exchanged, where n is odd.

During the even phase the elements with even numbered subscripts are compared with their neighbors on the right and exchanged if necessary. That is the elements {b2,b3},{b4,b5}..{bn-2,bn-1} are compared and exchanged, where n is even. After n phases the elements are sorted.

Example:

1. Let n=4 and a= {5,2,1,4}

2. According to the algorithm i varies from 1 to 2 The processors are P0, P1, P2 and P3. Let i=1

3. Since 0 is even, process P0 will compare even vertices with its successor and exchange if necessary. That is a becomes {2,5,1,4}.

4. P1 will compare odd vertices with its successor and exchange if necessary. That is a becomes{2,1,5,4} .

5. P2 will make a as {1,2,4,5} and P3 will retain a. Next i becomes 2 but no change in a. Hence the final sequence is {1,2,4,5}.

Conclusion:

Thus, we have studied and implemented a parallel ODD-Even Merge Sort program.

Mathematical Modeling:

Let S be the system that represents the ODD-Even Sort program.

Initially,

$$\mathbf{S} = \{\ \phi\ \}$$

Let,

$$\mathbf{S} = \{\ \mathbf{I, O, F}\ \}$$

Where,
I - Represents Input set
O - Represents Output set
F - Represents Function set

Input set - I:

The size of the input array.

Output set - O:

Sorted form of randomly generated array.

Function set - F:

$$\mathbf{F} = \{\ F_1, F_2, F_3\ \}$$

F_1 - Represents the function for implementing ODD-Even Sort.
$F_1(E_1, E_2 \ldots\ldots E_n) \rightarrow \{\ E_5, E_1 \ldots\ldots E_{n-3}\ \}$
Where,

- E_i: ith element of the input array.

F_2 - Represents the function for partitioning the array.
$F_2(E_1, E_2 \ldots\ldots E_n) \rightarrow \{\ E_0, E_2 \ldots\ldots E_{2n}\ \}$ or $\{\ E_1, E_3 \ldots\ldots E_{2n+1}\ \}$
Where,

- E_{2n}: even index elements of the input array.

- E_{2n+1}: odd index elements of the input array.

F_3 - Represents the function for merging the sorted array.
F_2({ E_0, E_2 ..... E_{2n} } U { E_1, E_3 ..... E_{2n+1} }) –> { E_0, E_1 ..... E_n }
Where,

- E_{2n}: even index elements of the input array.
- E_{2n+1}: odd index elements of the input array.

Finally,

S ={ I, O, F }