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:

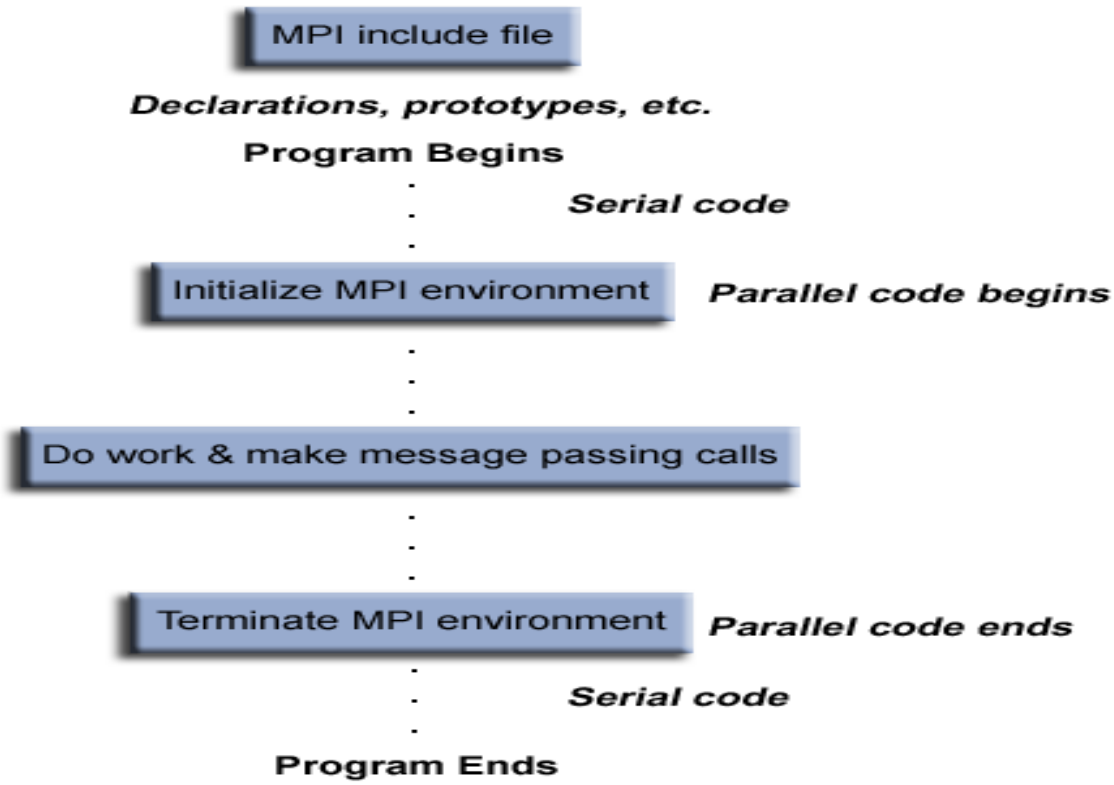

Figure 1: MPI Program Structure

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:

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

Table 1:MPI Message Passing Routine

· 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.

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

Table 2: C++ Data Types

· 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:

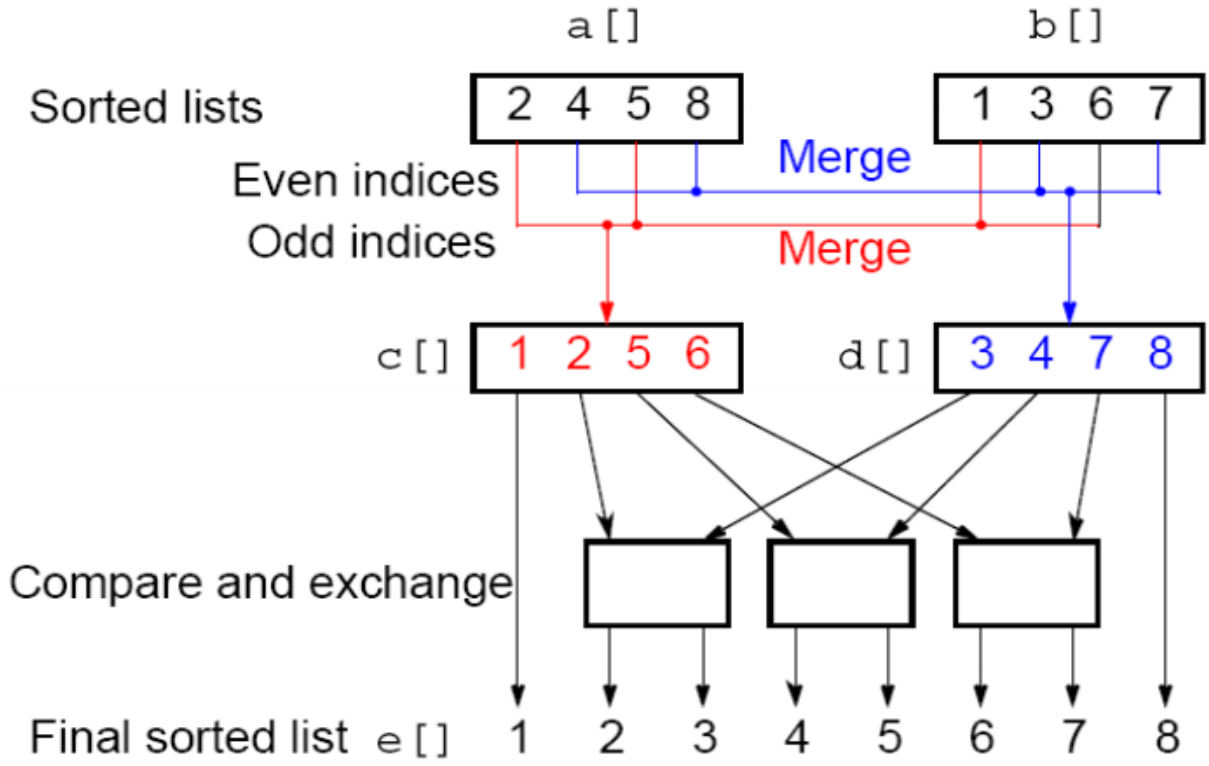

Figure 2: Merging

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.

White Box Testing:−

White Box Testing White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. White Box Testing (WBT) is also known as Code−Based Testing or Structural Testing. White box testing is the software testing method in which internal structure is being known to tester who is going to test the software. White box testing is also known as clear box testing, open box testing, logic driven testing or path driven testing or structural testing and glass box testing. The White−box testing is one of the best method to find out the errors in the software application in early stage of software development life cycle.
What do you verify in White Box Testing?
In the White box testing following steps are executed to test the software code:

- Basically verify the security holes in the code.

- Verify the broken or incomplete paths in the code.
- Verify the flow of structure mention in the specification document
- Verify the Expected outputs
- Verify the all conditional loops in the code to check the complete functionality of the
- Application.
- Verify the line by line or Section by Section in the code & cover the 100% testing.
- Above steps can be executed at the each steps of the STLC i.e. Unit Testing, Integration & System testing.

White Box Testing Techniques

Here are some white box testing techniques used in White Box Testing?

1. Statement Coverage: In this white box testing technique try to cover 100% statement coverage of the code, it means while testing the every possible statement in the code is executed at least once. Tools: To test the Statement Coverage the Cantata++ can be used as white box testing tool.

2. Decision Coverage: In this white box testing technique try to cover 100% decision coverage of the code, it means while testing the every possible decision conditions like if-else, for loop and other conditional loops in the code is executed at least once. Tools: To cover the Decision Coverage testing in the code the TCAT-PATH is used. This supports for the C, C++ and Java applications.

3. Condition Coverage: In this white box testing technique try to cover 100% Condition coverage of the code, it means while testing the every possible conditions in the code is executed at least once.

4. Decision/Condition Coverage: In this mixed type of white box testing technique try to cover 100% Decision/Condition coverage of the code, it means while testing the every possible Decisions/Conditions in the code is executed at least once.

5. Multiple Condition Coverage: In this type of testing we use to cover each entry point of the system to be execute once. In the actual development process developers are make use of the combination of techniques those are suitable for there software application.

How do you perform White Box Testing? Let take a simple website application. The end user is simply access the website, Login & Logout, this is very simple and day 2 days life example. As end users point of view user able to access the website from GUI but inside there are lots of things going on to check the internal things are going right or not, the white box testing method is used. To explain this we have to divide this part in two steps. This is all is being done when the tester is testing the application using White box testing techniques.

STEP 1) UNDERSTAND OF THE SOURCE CODE

STEP 2) CREATE TEST CASES AND EXECUTE

STEP 1) UNDERSTAND OF THE SOURCE CODE:- The first & very important steps is to understand the source code of the application is being test. As tester should know about the internal structure of the code which helps to test the application. Better knowledge of Source code will helps to identify & write the important test case which causes the security issues & also helps to cover the 100% test coverage. Before doing this the tester should know the programming language what is being used in the developing the application. As security of application is primary objective of application so tester should aware of the security concerns of the project which help in testing. If the tester is aware of the security issues then he can prevents the security issues like attacks from hackers & users who are trying to inject the malicious things in the application.
STEP 2) CREATE TEST CASES AND EXECUTE In the second step involves the actual writing of test cases based on Statement/Decision/Condition/Branch coverage & actual execution of test cases to cover the 100% testing coverage of the software application. The Tester will write the test cases by diving the applications by Statement/Decision/Condition/Branch wise. Other than this tester can include the trial and error testing, manual testing and using the Software testing tools as we covered in the previous article.
Types of White Box Testing:
1. Unit Testing
2. Integration Testing
1. Unit Testing What is a Unit?

Unit is a single smallest independent component.
Unit Testing is categorized as:
1. Execution Testing
2.Operations Testing
3. Mutation Testing
2. Integration Testing: After making independent components, these components are joined together to check if they are working fine together as one application. Example: Compose Mail and Sent Items. After composing and sending mails, mails are checked in sent items. Different modules should inter talk with each other.
Approaches in Integration Testing:

- Top Down Approach:
  When main module is ready and sub module is under development. Main module is dependent on sub module. Here, stubs comes into picture. Stubs are temporary programs which takes the place of module under development. Ready module is tested using Stubs.

- Bottom Up Approach: When Sub module is ready and main module is under development. Here, driver are the temporary programs which are used to test sub modules.

Mathematical Modeling:

Let S be the system that represents the ODD−Even Sort program.
Initially,
S ={ _ }
Let,
S ={ 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:
F = { F1, F2, F3 }
F1 − Represents the function for implementing ODD−Even Sort.
F1(E1, E2 ..... En) –> { E5, E1 ..... En−3 }
Where,
· Ei: ith element of the input array.
F2 − Represents the function for partitioning the array.
F2(E1, E2 ..... En) –> { E0, E2 ..... E2n } or { E1, E3 ..... E2n+1 }
Where,
· E2n: even index elements of the input array.

· E2n+1: odd index elements of the input array.
F3 − Represents the function for merging the sorted array.
F2({ E0, E2 ..... E2n } U { E1, E3 ..... E2n+1 }) –> { E0, E1 ..... En }
Where,
· E2n: even index elements of the input array.
· E2n+1: odd index elements of the input array.
Finally,
S ={ I, O, F }

UML:

Use Case Diagram:

Activity Diagram: