Assignment No-A4

1 Aim:

Write an **MPI** program and record the time taken to run on varying number of nodes.

2 Problem statement :

Write a program on an unloaded cluster for several different numbers of nodes and record the time taken in each case. Draw a graph of execution time against the number of nodes.

3 Learning Objectives:

1. To understand the concept of MPI.
2. To develop time and space efficient algorithms.

4 Learning Outcomes:

After successfully completing this assignment we would be able to successfully load the unloaded clusters and plot the execution time graph against the number of nodes.

5 Related Mathematics:

Let S be the solution perspective for the system to plot execution graphs against no. of nodes. S={s, e, i, o, f, DD, NDD, success, failure}
s={Initial state i.e. empty graph}

e={End state i.e. graph plotted against time axis for the execution against different nodes. }

i=Input of the system
={q1 | q1 is the position of 1st queen.}

o=Output of the system
={g1,g2,g3.....g9 | gi is the graph for corresponding execution time with every added node.}

DD={Deterministic data: It helps identifying the load store functions or assignment functions.}

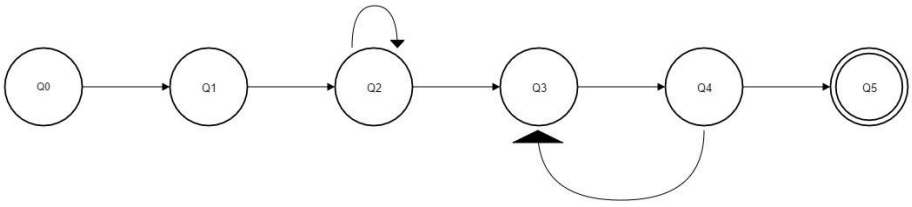
NDD={Non deterministic data: Data of the system S to be solved i.e the position of the 1st queen.}

Success={Desired output is obtained , the graph is obtained for additive nodes.}

Failure={Desired output is not obtained or is forced to exit due to system error.}

f = set of functions = {f1, f2 ,f3 , f4}
f1 = function to execute the console commands.
f2 = function to append the new node's execution with the previous ones.
f3 = function to align x-axis and y-axis and then plot the execution for the respective node. f4 = function to show the graph.

6 State Transition Diagram



q0:start state {deciding first queen's position}
q1:creation of master node q2:creation
of 9 slave nodes recursively.
q3:execute 8 Queens with slave nodes.

q4:plot corresponding execution time.
q5:show the final graph.

7 Concepts related theory:

7.1 MPI:

1. It is an abbreviation for *Message Passing Interface*.
2. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.
3. It 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.
4. 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.
5. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain.
6. These fostered the development of a parallel software industry, and encouraged development of portable and scalable largescale parallel applications.
7. 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.

   (a) *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 intracommunicator operations, and bilateral inter-communicator communication. In MPI-1, single group operations are most prevalent. Bilateral operations mostly appear in MPI2 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.

(b) *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-topoint 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.

(c) *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 n^{th} node gets the nth item of data from each.

(d) *Derived Datatypes:*
Many MPI functions require that you specify the type of data which is sent between processors. This is because these functions pass variable addresses, not defined types. If the data type is a standard one, such as *int*, *char*, *double*, etc., you can use predefined MPI datatypes such as **MPI INT**, **MPI CHAR**, **MPI DOUBLE**.

8. Some of the implementations of MPI are as follows:

(a) 'Classical' cluster and supercomputer implementation:
The MPI implementation language is not constrained to match the language or languages it seeks to support at runtime. Most implementations combine C, C++ and assembly language, and target C, C++, and Fortran programmers. Bindings are available for many other languages, including Perl, Python, R, Ruby, Java, and CL.

(b) Python:
MPI Python implementations include: pyMPI, mpi4py, pypar, MYMPI, and the MPI submodule in ScientificPython.

(c) OCaml:
The OCamlMPI Module implements a large subset of MPI functions and is in active use in scientific computing.

(d) Java:
Although Java does not have an official MPI binding, several groups attempt to bridge the two, with different degrees of success and compatibility. mpiJava API was a de facto MPI API for Java that closely followed the equivalent C++ bindings.

(e) Matlab:
There are a few academic implementations of MPI using Matlab. Matlab has its own parallel extension library implemented using MPI and PVM.

(f) R:

R implementations of MPI include Rmpi and pbdMPI, where Rmpi focuses on manager-workers parallelism while pbdMPI focuses on SPMD parallelism. Both implementations fully support Open MPI or MPICH2.

(g) Common Language Infrastructure:
The two managed Common Language Infrastructure (CLI) .NET implementations are Pure Mpi.NET and MPI.NET, a research effort at Indiana University licensed under a BSD-style license. It is compatible with Mono, and can make full use of underlying low-latency MPI network fabrics.

(h) Hardware implementations:
MPI hardware research focuses on implementing MPI directly in hardware, for example via processor-in-memory, building MPI operations into the microcircuitry of the RAM chips in each node. By implication, this approach is independent of the language, OS or CPU, but cannot be readily updated or removed.

(i) mpicc:
mpicc is a program which helps the programmer to use a standard C programming language compiler together with the Message Passing Interface (MPI) libraries, most commonly the OpenMPI implementation which is found in many TOP-500 supercomputers, for the purpose of producing parallel processing programs to run over computer clusters (often Beowulf clusters). The mpicc program uses a programmer's preferred C compiler and takes care of linking it with the MPI libraries.

7.2 Installations:

1. Execute command:
   sudo apt-get update

2. Install MPI 3. Install

   pyMPI, mpi4py

8 Algorithm:

1. Start.

2. Configure Master node.
3. Configure Slave node.
4. Send code to be executed from Master node.
5. Receive output from Slave nodes.
6. Display output.
7. Stop.

9 Conclusion:

Thus, we have successfully implemented an MPI program for calculating coverage.