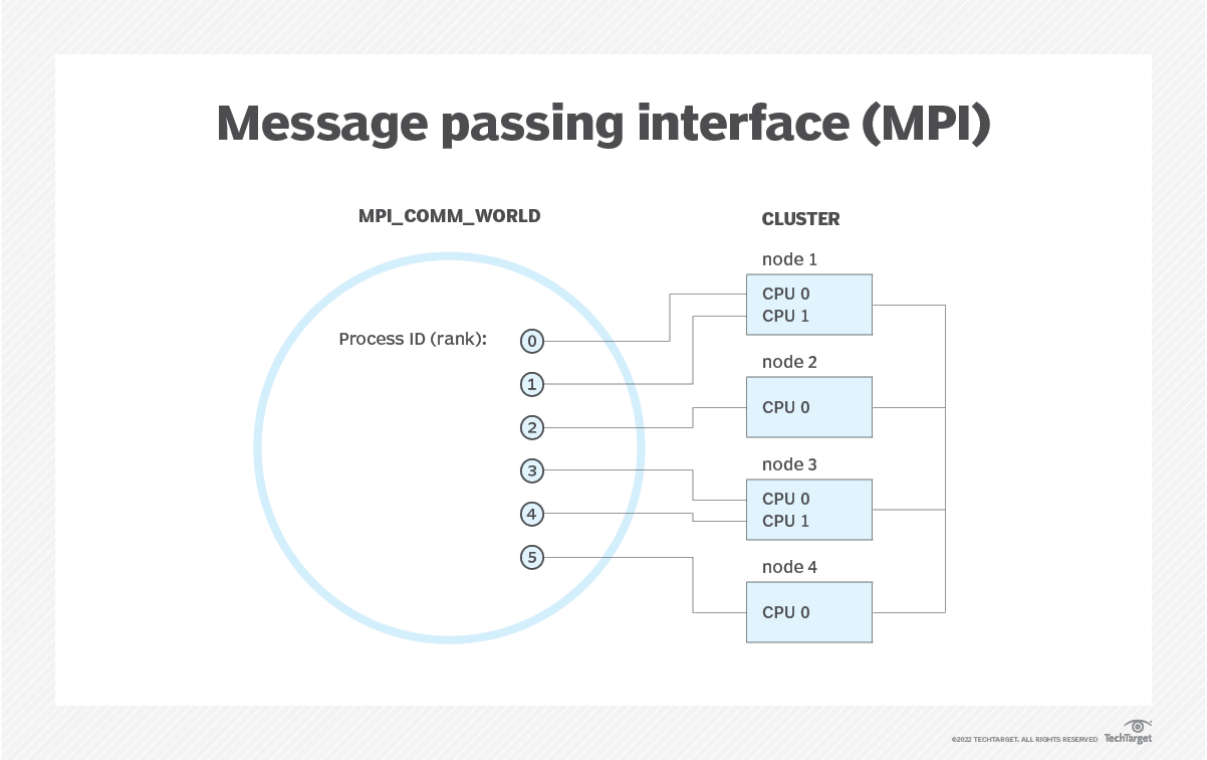
EXPERIMENT 1

AIM: Write Hello World using MPI

THEORY:
The Message Passing Interface (MPI) is a portable, standardized message-passing standard that functions on parallel computing architectures. The MPI system requires the syntax and semantics of library routines that can be used by a broad variety of users who are writing portable message-passing programs in C, C++, and Fortran.

There are many open-source MPI implementations that have aided in the development of the parallel software industry and the development of portable and scalable large-scale parallel applications.



Source:
https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI

JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 1

CPP IMPLEMENTATION :

CODE:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Get the rank of the process
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Print the message
    printf("Hello World! My rank is %d\n", my_rank);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

OUTPUT:

```
Build started...
1>------ Build started: Project: Project1, Configuration: Debug x64 ------
1>MPI.cpp
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Build started at 4:28 PM and took 04.288 seconds ==========
```

```
C:\Users\JARVIS\source\repos\Project1\x64\Debug>mpiexec -n 4 Project1.exe
Hello World! My rank is 1
Hello World! My rank is 0
Hello World! My rank is 2
Hello World! My rank is 3
```

JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 1

PYTHON IMPLEMENTATION:

CODE:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
        data = {'data' : 'Hello Junaid'}
else:
        data = None
data = comm.bcast(data, root=0)
print(data)
```

OUTPUT:

```
jarvis@jarvis-Inspiron-7591:~/Desktop$ mpiexec -n 4 python3 main.py
{'data': 'Hello Junaid'}
{'data': 'Hello Junaid'}
{'data': 'Hello Junaid'}
{'data': 'Hello Junaid'}
```

CONCLUSION: MPI is an important functionality that is used in distributed processing to achieve higher efficiency by sharing data between different processes. In this experiment, we have implemented the MPI mechanism using C++ and Python.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 2

EXPERIMENT 2

AIM: To implement the following:

1. Program to send data and receive data to/from processors using MPI
2. Program illustrating broadcast of data using MPI

THEORY:

Sending and receiving are the two foundational concepts of MPI. Almost every single function in MPI can be implemented with basic send and receive calls.

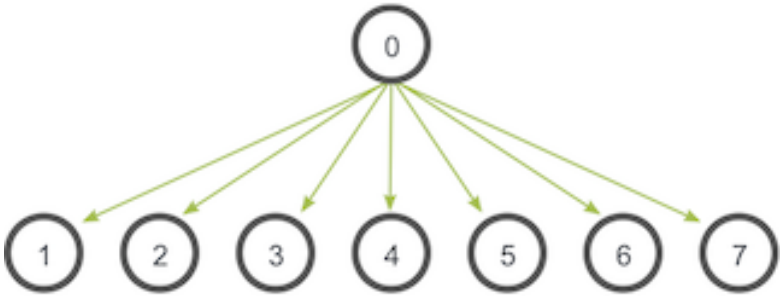
MPI's send and receive calls operate in the following manner. First, process A decides a message needs to be sent to process B. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.

Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as tags). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

The communication pattern of a broadcast looks like this:



In this example, process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.

1. **CODE:**

```
from mpi4py import MPI

def main():
  passes = 5
  id = ""
  rank = MPI.COMM_WORLD.Get_rank()
  comm = MPI.COMM_WORLD
  if rank == 0:
        id = "Player 0"
        next = 1
  else:
        id = "Player 1"
        next = 0

  if rank == 0:
        print(f"Number of Parallel Players is {comm.Get_rank()}")
        comm.send("Player 0 Initiates", dest = 1, tag = 0)

  while passes > 0:
        msg = comm.recv(source = next, tag = 0)
        print(f"{id} received : {msg}")
```

```
        comm.send(f"{id}'s {5 - passes + 1}th pass", dest = next, tag = 0)
        passes -= 1

 print("Passes over for ", id)

if __name__ == "__main__":
 main()
```

OUTPUT:

```
jarvis@jarvis-Inspiron-7591:~/Desktop$ mpiexec -n 4 python3 sendreceive.py
Number of Parallel Players is 0
Player 1 received : Player 0 Initiates
Player 1 received : Player 0's 1th pass
Player 0 received : Player 1's 1th pass
Player 0 received : Player 1's 2th pass
Player 0 received : Player 1's 3th pass
Player 1 received : Player 0's 2th pass
Player 1 received : Player 0's 3th pass
Player 0 received : Player 1's 4th pass
Player 1 received : Player 0's 4th pass
Player 0 received : Player 1's 5th pass
Passes over for  Player 0
Passes over for  Player 1
```

---

2. **CODE:**

master.py

```
from mpi4py import MPI
import random
random.seed(random.random())
```

```
def master(rank):
  steps = random.randint(1,10)
  print("Master Steps : ", steps)
  for i in range(steps):
    print(f"Master rank {rank} working for {i}th time")

def main():
  rank = MPI.COMM_WORLD.Get_rank()
  comm = MPI.COMM_WORLD
  master(rank)
  comm.Barrier()
  comm.bcast("Mission Successful", root = rank)

if __name__ == "__main__":
  main()
```

slave.py

```
from mpi4py import MPI
import random
random.seed(random.random())

def slave(rank):
  steps = random.randint(1,10)
  print("Slave Steps : ", steps)
  for i in range(steps):
    print(f"Slave rank {rank} working for {i}th time")

def main():
  rank = MPI.COMM_WORLD.Get_rank()
  comm = MPI.COMM_WORLD
  slave(rank)
  data = ""
  comm.Barrier()
  data = comm.bcast(data, root = 0)
  print(f"Slave Rank {rank} receives: {data}")
```

```
if __name__ == "__main__":
  main()
```

OUTPUT:

```
D:\SEM 7\HPC\EXPERIMENT 2>mpiexec /np 1 python master.py : /np 3 python
slave.py
Master Steps :  8
Master rank 0 working for 0th time
Master rank 0 working for 1th time
Master rank 0 working for 2th time
Master rank 0 working for 3th time
Master rank 0 working for 4th time
Master rank 0 working for 5th time
Master rank 0 working for 6th time
Master rank 0 working for 7th time
Slave Steps :  7
Slave rank 2 working for 0th time
Slave rank 2 working for 1th time
Slave rank 2 working for 2th time
Slave rank 2 working for 3th time
Slave rank 2 working for 4th time
Slave rank 2 working for 5th time
Slave rank 2 working for 6th time
Slave Rank 2 receives: Mission Successful
Slave Steps :  1
Slave rank 3 working for 0th time
Slave Rank 3 receives: Mission Successful
Slave Steps :  6
Slave rank 1 working for 0th time
Slave rank 1 working for 1th time
Slave rank 1 working for 2th time
Slave rank 1 working for 3th time
Slave rank 1 working for 4th time
Slave rank 1 working for 5th time
```

```
Slave Rank 1 receives: Mission Successful
```

CONCLUSION: Sending and receiving messages is an integral part of MPI. In this experiment, we have demonstrated a program that sends and receives data to/from processors. Along with this, we have also demonstrated data broadcast using MPI in Python.

EXPERIMENT 3

AIM: Implement a parallel program to demonstrate the cube of N number within a set range.

THEORY:
The term parallel programming may be used interchangeable with parallel processing or in conjunction with parallel computing, which refers to the systems that enable the high efficiency of parallel programming.

In parallel programming, tasks are parallelized so that they can be run at the same time by using multiple computers or multiple cores within a CPU. Parallel programming is critical for large scale projects in which speed and accuracy are needed. It is a complex task, but allows developers, researchers, and users to accomplish research and analysis quicker than with a program that can only process one task at a time.

Parallel programming works by assigning tasks to different nodes or cores. In High Performance Computing (HPC) systems, a node is a self-contained unit of a computer system contains memory and processors running an operating system. Processors, such as central processing units (CPUs) and graphics processing units (GPUs), are chips that contain a set of cores. Cores are the units executing commands; there can be multiple cores in a processor and multiple processors in a node.

With parallel programming, a developer writes code with specialized software to make it easy for them to run their program across on multiple nodes or processors. A simple example of where parallel programming could be used to speed up processing is recoloring an image. A developer writes the code to break up the overall task of to change the individual aspects of an image by segmenting the image into equal parts and then assigns the recoloring of each part to a different parallel task, each running on their own compute resources. Once the parallel tasks have completed, the full image is reassembled.

Parallel processing techniques can be utilized on devices ranging from embedded, mobile, laptops, and workstations to the world's largest supercomputers. Different

computer languages provide various technologies to enable parallelism. For C, C++ and Fortran, OpenMP, open multi-processing, provides a cross-platform API for developing parallel applications that enable running parallel tasks across cores of a CPU. When processes need to communicate between different computers or nodes, a technology such as MPI, message passing interface, is typically used. There are benefits to both models. Multiple cores on a single node share memory. Shared memory is typically faster for exchanging information than message passing between nodes over a network. However, there's a limit to how many cores a single node can have. As projects get larger, developers may use both types of parallelism together. One of the challenges that developers face though is properly decomposing their algorithm and parallelizing across multiple nodes and multiple cores for maximum performance and debugging their parallel application when it does not work correctly.

Parallel Programming Usage:
• Advanced graphics in the entertainment industry
• Applied physics
• Climate research
• Electrical engineering
• Financial and economic modeling
• Molecular modeling
• National defense and nuclear weaponry
• Oil and gas exploration
• Quantum mechanics

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int n = 20;
int a2[1000];
int tmp2;
int tmp;
int cntr;
int nR;
int prnt_cntr = 0;
```

```
int main(int argc, char* argv[]) {
	int pid, np, elements_per_process, n_elements_received; MPI_Status status;
	int index, i;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &pid);
	MPI_Comm_size(MPI_COMM_WORLD, &np);
	elements_per_process = n / np;
	if (pid == 0) {
		if (np > 1) {
			for (i = 1; i < np - 1; i++) {
				index = i * elements_per_process;
				MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
					MPI_COMM_WORLD);
				MPI_Send(&index, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
			}
			index = i * elements_per_process;
			int elements_left = n - index;
			MPI_Send(&elements_left, 1, MPI_INT, i, 0,
				MPI_COMM_WORLD);
			MPI_Send(&index, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
			int cube = 0;
			for (i = 0; i < elements_per_process; i++) {
				cube = (i) * (i) * (i);
				printf("%d^3=%d \n", prnt_cntr, cube);
				prnt_cntr += 1;
			}
			for (i = 1; i < np; i++) {
				MPI_Recv(&cntr, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
					&status);
				MPI_Recv(&a2, cntr, MPI_INT, i, 0, MPI_COMM_WORLD,
					&status);
				int sender = status.MPI_SOURCE;
				for (int j = 0; j < cntr; j++) {
					printf("%d^3 = %d \n", prnt_cntr, a2[j]);
					prnt_cntr += 1;
```

```
                                }
                        }
                }
        }
        else {
                MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Recv(&tmp2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                int partial_sum = 0;
                for (int i = 0; i < n_elements_received; i++) {
                        a2[i] = (tmp2 + i) * (tmp2 + i) * (tmp2 + i);
                        nR = n_elements_received;
                }
                MPI_Send(&nR, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                MPI_Send(&a2, nR, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        MPI_Finalize();
        return 0;
}
```

OUTPUT:

```
C:\Users\jarvis\source\repos\Exp3\x64\Debug>mpiexec —n 4 Experiment3.exe
0^3=0
1^3=1
2^3=8
3^3=27
4^3=64
5^3=125
6^3 = 216
7^3=343
8^3=512
9^3=729
10^3=1000
11^3=1331
12^3=1728
```

```
13^3=2197
14^3=2744
15^3=3375
16^3=4096
17^3=4913
18^3=5832
19^3=6859
```

CONCLUSION: MPI was used to teach the concept of parallel programming. A parallel program above showed how the program can be run in a parallel environment and compute the cube for a given set of values. Hence, parallelism improves the valuable time consumption and computes the results faster than any other way of computation.

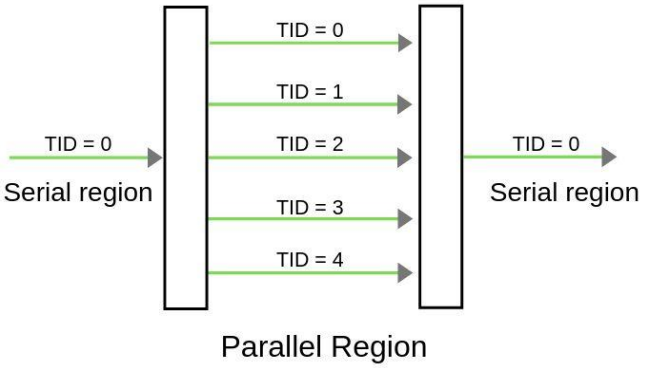
EXPERIMENT 4

AIM: Implement a parallel program to find area of a circle.

THEORY:
Parallel programming is the use of multiple resources, in this case, processors, to solve a problem. This type of programming takes a problem, breaks it down into a series of smaller steps, delivers instructions, and processors execute the solutions at the same time. It is also a form of programming that offers the same results as concurrent programming but, in less time, and with more efficiency. Many computers, such as laptops and personal desktops, use this programming in their hardware to ensure that tasks are quickly completed in the background.
This type of programming can be used for everything from science and engineering to retail and research. It's most common use from a societal perspective is in web search engines, applications, and multimedia technologies. Nearly every field in America uses this programming in some aspect, whether for research and development or for selling their wares on the web, making it an important part of computer science.
Parallel computing is the future of programming and is already paving the way to solving problems that concurrent programming consistently runs into. Although it has its advantages and disadvantages, it is one of the most consistent programming processes in use today.

Area of a circle is the region occupied by the circle in a two-dimensional plane. It can be determined easily using a formula, $A=r^2$ where r is the radius of the circle.
Any geometrical shape has its own area. This area is the region occupied the shape in a two-dimensional plane. Now we will learn about the area of the circle. So the area covered by one complete cycle of the radius of the circle on a two-dimensional plane is the area of that circle.

JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 4



CODE:

```
# include <stdio.h>
# include <math.h>
# include <mpi.h>

# define approx_val 2.19328059

# define N 32 /* Number of intervals in each processor */
double integrate_f(); /* Integral function */

double simpson();

double integrate_f(double x)
{
return 4.0 / (1.0 + x * x); /* compute and return value */

}

double simpson(int i, double y, double l, double sum)
{
/* store result in sum */
sum += integrate_f(y);
sum += 2.0 * integrate_f(y - l);
if (i == (N - 1))
sum += 2.0 * integrate_f(y + l);
return sum;
} /* simpson */
```

```
int main(int argc, char* argv[])
{
int Procs; /* Number of processors */
int my_rank; /* Processor number */
double total;
double exact_val_of_Pi, pi, y, processor_output_share[8], x1, x2, l, sum;
float area;
int i, radius = 4;
MPI_Status status;
/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);
/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* Find out how many processes are being used . */
MPI_Comm_size(MPI_COMM_WORLD, &Procs);
/* Each processor computes its interval */
x1 = ((double)my_rank) / ((double)Procs);
x2 = ((double)(my_rank + 1)) / ((double)Procs);
/* l is the same for all processes . */
l = 1.0 / ((double)(2 * N * Procs));
sum = 0.0;
for (i = 1; i < N; i++)
{
y = x1 + (x2 - x1) * ((double)i) / ((double)N);
/* call Simpson 's rule */
sum = (double)simpson(i, y, l, sum);
}
/* Include the endpoints of the intervals */
sum += (integrate_f(x1) + integrate_f(x2)) / 2.0;
total = sum;
/* Add up the integrals calculated by each process . */
if (my_rank == 0)
{
processor_output_share[0] = total;
/* source = i, tag = 0 */
for (i = 1; i < Procs; i++)
MPI_Recv(&(processor_output_share[i]), 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
&status);
```

```
}
else
{
/* dest = 0 , tag = 0 */
MPI_Send(&total, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
/* Add up the value of Pi and print the result . */
if (my_rank == 0)
{
pi = 0.0;
for (i = 0; i < Procs; i++)
pi += processor_output_share[i];
pi *= 2.0 * l / 3.0;
printf(" ------------------------------------------------\n");
printf("The computed Pi of the integral for %d grid points is %25.16e \n", (N * Procs), pi);

/* This is directly derived from the integeration of the formula . See
the report . */
#if 1
exact_val_of_Pi = 4.0 * atan(1.0);
area = exact_val_of_Pi * radius * radius;
# endif

#if 0
exact_val_of_Pi = 4.0 * log(approx_val);
area = exact_val_of_Pi * radius * radius;
# endif

// printf (" The error or the discrepancy between exact and computed value of Pi : %25.16
e\n" ,
// fabs (pi - exact_val_of_Pi ));
printf(" Area of circle is ( when radius of circle is 4): %0.4f \n", fabs(area));
printf(" ------------------------------------------------\n");
}
MPI_Finalize();
}
```

OUTPUT:

JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 4

```
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\JARVIS\source\repos\Experiment4\x64\Debug>mpiexec -n 1 Experiment4.exe
 -----------------------------------------------
The computed Pi of the integral for 32 grid points is    3.1415926535892149e+00
 Area of circle is ( when radius of circle is 4): 50.2655
 -----------------------------------------------

C:\Users\JARVIS\source\repos\Experiment4\x64\Debug>
```

CONCLUSION: MPI was used to teach the concept of parallel programming. A parallel program above showed how the program can be run in a parallel environment and compute the area of a circle by calculating the value of pi. Hence, parallelism improves the valuable time consumption and computes the results faster than any other way of computation.

EXPERIMENT 5

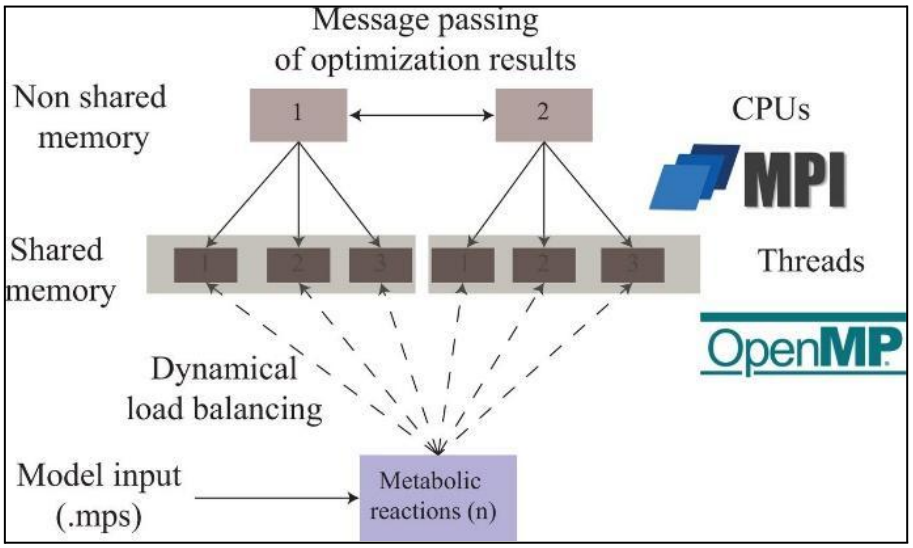
AIM: To implement a program to demonstrate balancing of workload on MPI Platform.

THEORY:
Load balancing is the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient. Load balancing can optimize the response time and avoid unevenly overloading some compute nodes while other compute nodes are left idle. Load balancing is the subject of research in the field of parallel computers.

Two main approaches exist: static algorithms, which do not consider the state of the different machines, and dynamic algorithms, which are usually more general and more efficient but require exchanges of information between the different computing units, at the risk of a loss of efficiency.

When splitting the workload across multiple processors, the speedup describes the achieved reduction of the runtime compared to the serial version. Obviously, the higher the speedup, the better. However, the effect of increasing the number of processors usually goes down at a certain point when they cannot be utilized optimally anymore. This parallel efficiency is defined as the achieved speedup divided by the number of processors used. It usually is not possible to achieve a speedup equal to the number of processors used as most applications have strictly serial parts (Amdahl's Law).

Load balancing is of great importance when utilizing multiple processors as efficiently as possible. Adding more processors creates a noticeable amount of synchronization overhead. Therefore, it is only beneficial if there is enough work present to keep all processors busy at the same time. This means tasks should not be assigned randomly as this might lead to serial behavior where only one processor is working while the others have already finished their tasks, e.g., at synchronization points (e.g., barriers) or at the end of the program.



CODE:

```
#include <iostream>
#include <ctime>
#include <mpi.h>
using namespace std;

double int_theta(double E) {
   double result = 0;
   for (int k = 0; k < 20000; k++)
      result += E * k;
   return result;
}

int main() {
   int n = 3500000;
   int counter = 0;
   time_t timer;
   int start_time = time(&timer);
   int myid, numprocs, k;
   double integrate, result;
   double end = 0.5;
```

```
   double start = -2.;
   double E;
   double factor = (end - start) / (n * 1.);
   integrate = 0;
   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   for (k = myid; k < n + 1; k += numprocs) {
      E = start + k * (end - start) / n;
      if ((k == 0) || (k == n))
         integrate += 0.5 * factor * int_theta(E);
      else
         integrate += factor * int_theta(E);
      counter++;
   }
   cout << "Process " << myid << " took " << time(&timer) - start_time << " s" << endl;
   cout << "Process " << myid << " performed " << counter << " computations" <<
endl;
   MPI_Reduce(&integrate, &result, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
   if (myid == 0)
      cout << "The Final Result is " << result << endl;
   MPI_Finalize();
   return 0;
}
```

OUTPUT:

```
C:\Users\JARVIS\source\repos\Experiment 5\x64\Debug>mpiexec -n 4 "Experiment
5.exe"
Process 2 took 87 s
Process 2 performed 875000 computations
Process 3 took 87 s
Process 3 performed 875000 computations
Process 1 took 87 s
Process 1 performed 875000 computations
```

```
Process 0 took 87 s
Process 0 performed 875001 computations
The Final Result is -3.74981e+08

C:\Users\JARVIS\source\repos\Experiment 5\x64\Debug>mpiexec -n 8 "Experiment
5.exe"
Process 2 took 40 s
Process 2 performed 437500 computations
Process 7 took 40 s
Process 7 performed 437500 computations
Process 3 took 40 s
Process 3 performed 437500 computations
Process 1 took 40 s
Process 1 performed 437500 computations
Process 4 took 40 s
Process 4 performed 437500 computations
Process 6 took 40 s
Process 6 performed 437500 computations
Process 0 took 40 s
Process 0 performed 437501 computations
Process 5 took 40 s
Process 5 performed 437500 computations
The Final Result is -3.74981e+08

C:\Users\JARVIS\source\repos\Experiment 5\x64\Debug>mpiexec -n 2 "Experiment
5.exe"
Process 0 took 173 s
Process 0 performed 1750001 computations
Process 1 took 173 s
Process 1 performed 1750000 computations
The Final Result is -3.74981e+08
```

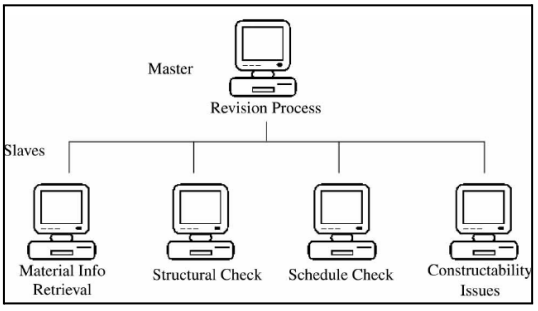
CONCLUSION: Load balancing is the process of spreading tasks between a set of nodes to have equal resource utilization across all. With MPI it is possible to distribute task subsets among a collection of nodes in a communication domain. Each node is thus executing the same operation but on different data. This ensures consistent response times for requests which scale downward as the number of nodes increases.

EXPERIMENT 6

AIM: To implement parallel programming for calculator application using directives of MPI.

THEORY:
Master/slave is a model of asymmetric communication or control where one device or process (the "master") controls one or more other devices or processes (the "slaves") and serves as their communication hub. In some systems, a master is selected from a group of eligible devices, with the other devices acting in the role of slaves.
Key Features:

1. The system comprises a master controller generating commands and receiving status signals and slave devices associated with the master controller.
2. Each slave receives commands, executes local commands responsive to the commands and generates status signals for the master controller.
3. Each slave has a communication arrangement for signals transmitted between it and the master controller.
4. The arrangement comprises a communication controller associated with the master controller.
5. The communication controller receives commands, transmits commands to each slave, receives status signals and provides information relating to the status signals to the master controller.
6. The controller has a communication link which transmits commands to each slave and the status signals to the controller.
7. The system allows local commands executed by the slaves to replace other commands directed by the master controller to the slave.
8. Further, each slave communicates independently with the master controller.



CODE:

```
#include < stdio.h >
#include < mpi.h >

#define send_data_tag 2001

int main(int argc, char** argv) {
   MPI_Status status;
   int ierr, my_id, num_procs;
   int a[2];
   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   switch (my_id) {
      case 0:
         scanf_s("%d %d", &a[0], &a[1]);
         for (int i = 1; i < num_procs; i++)
            ierr = MPI_Send(&a, 2, MPI_INT, i, send_data_tag, MPI_COMM_WORLD);
         break;
      case 1:
         ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
         printf("(Process %d) performs %d + %d = %d\n", my_id, a[0], a[1], (a[0] +
a[1]));
         break;
      case 2:
         ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
         printf("(Process %d) performs %d - %d = %d\n", my_id, a[0], a[1], (a[0] - a[1]));
         break;
      case 3:
         ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
```

```
        printf("(Process %d) performs %d * %d = %d\n", my_id, a[0], a[1], (a[0] * a[1]));
        break;
    case 4:
        ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
        printf("(Process %d) performs %d / %d = %d\n", my_id, a[0], a[1], (a[0] / a[1]));
        break;
    case 5:
        ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
        printf("(Process %d) performs %d %% %d = %d\n", my_id, a[0], a[1], (a[0] %
a[1]));
        break;
    default:
        ierr = MPI_Recv(&a, 2, MPI_INT, 0, send_data_tag, MPI_COMM_WORLD,
&status);
        printf("(Process %d) has no work to do \n", my_id);
  }
  ierr = MPI_Finalize();
}
```

OUTPUT:

```
C:\Users\JARVIS\source\repos\Experiment 6\x64\Debug>mpiexec -n 8 "Experiment
6.exe"
5 9
(Process 3) performs 5 * 9 = 45
(Process 1) performs 5 + 9 = 14
(Process 4) performs 5 / 9 = 0
(Process 2) performs 5 - 9 = -4
(Process 6) has no work to do
(Process 7) has no work to do
(Process 5) performs 5 % 9 = 5

C:\Users\JARVIS\source\repos\Experiment 6\x64\Debug>mpiexec -n 8 "Experiment
6.exe"
2 4
```

```
(Process 1) performs 2 + 4 = 6
(Process 3) performs 2 * 4 = 8
(Process 6) has no work to do
(Process 5) performs 2 % 4 = 2
(Process 2) performs 2 - 4 = -2
(Process 7) has no work to do
(Process 4) performs 2 / 4 = 0
```

CONCLUSION: Master slave architectures allow a distributed application with a single (or a few) servers handling request management. The rest of the network acts as slaves to the master server and executes commands on request. P2P links can be established between each master and slave using MPI. Thus, each node can act as a processing unit for the master executing different operations. This drastically reduces the processing and response time for a request and is directly correlated to the number of processors assigned per request.