**11.3 IPC: MPI(C library for message passing between processes of different systems) Distributed memory programming**

**Objectives:**

1. To learn about IPC through MPI.

2. Use of IPC mechanism to write effective application programs.

3. configure cluster and experiment MPI program on it.

**Theory:**

Message Passing Interface (MPI) is a standardized and portable message-passing standard

designed by a group of researchers from academia and industry to function on a wide variety of

parallel computing architectures. 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 C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source 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.

The MPI interface is meant to provide essential virtual topology, synchronization, and

communication functionality between a set of processes (that have been mapped to

nodes/servers/computer instances) in a language-independent way, with language-specific syntax

(bindings), plus a few language-specific features. MPI programs always work with processes, but

programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or

mpiexec

**Program:**

//MPI PI calculation using area of circle.

#include <stdio.h>

#include<math.h>

#include <mpi.h>

#define N 1E7

#define d 1E-7

#define d2 1E-14

int main (int argc, char\* argv[])

{

int rank, size, error, i;

double pi=0.0, begin=0.0, end=0.0, result=0.0, sum=0.0, x2;

error=MPI\_Init (&argc, &argv);

//Get process ID

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank);

//Get processes Number

MPI\_Comm\_size (MPI\_COMM\_WORLD, &size);

//Synchronize all processes and get the begin time

MPI\_Barrier(MPI\_COMM\_WORLD);

begin = MPI\_Wtime();

//Each process calculates a part of the sum

for (i=rank; i<N; i+=size)

{

x2=d2\*i\*i;

result+=sqrt(1-x2);

}

//Sum up all results

MPI\_Reduce(&result, &sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

//Synchronize all processes and get the end time

MPI\_Barrier(MPI\_COMM\_WORLD);

end = MPI\_Wtime();

//Calculate and print PI

if (rank==0)

{

pi=4\*d\*sum;

printf("np=%2d; Time=%fs; PI=%lf\n", size, end-begin, pi);

}

error=MPI\_Finalize();

return 0;

}

**Output:**

pi = 3.1415

Time = 0.02554

**Conclusion:**

1. The MPI program run successfully with time less than sequential execution.

**References:**

[1] http://mpitutorial.com/tutorials/mpi-introduction/