ASSIGNMENT 2

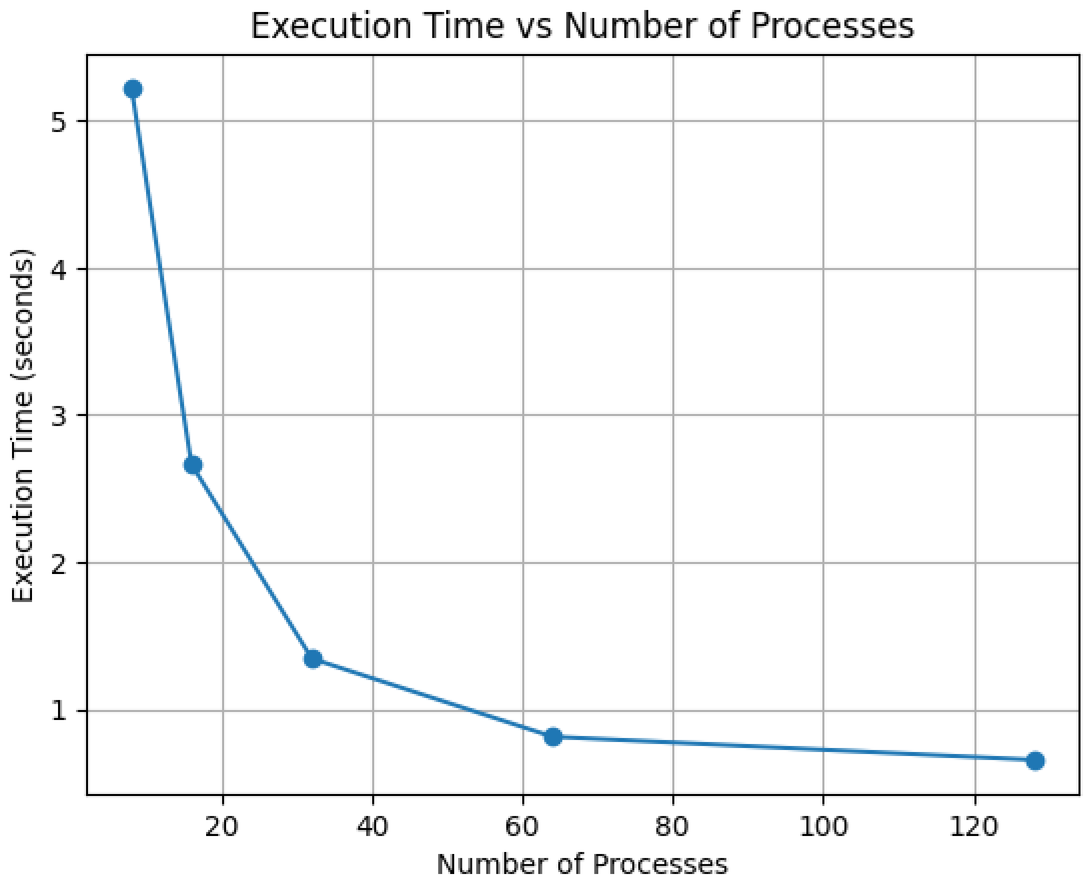
Name: Venkatesh Velidimalla

1. MPI Blocking Communication & Linear Reduction Algorithm

1.1

We get the following output on running pi_p2p_linear.c with 128 cores as outputs listed below : The plots have also been attached(Fig 1.1)

Output: (base) [vvelidi@node0159 intro-mpi]$ mpirun -n 8 ./pi_p2p_linear
pi=3.141601, t=5.221527
(base) [vvelidi@node0159 intro-mpi]$ mpirun -n 16 ./pi_p2p_linear
pi=3.141581, t=2.662387
(base) [vvelidi@node0159 intro-mpi]$ mpirun -n 32 ./pi_p2p_linear
pi=3.141577, t=1.343608
(base) [vvelidi@node0159 intro-mpi]$ mpirun -n 64 ./pi_p2p_linear
pi=3.141614, t=0.813296
(base) [vvelidi@node0159 intro-mpi]$ mpirun -n 128 ./pi_p2p_linear
pi=3.141620, t=0.655891



Number of Processes	Execution Time (seconds)
8	5.221527
16	2.662387
32	1.343608
64	0.813296
128	0.655891

1.2 Why MPI_Send and MPI_Recv are called "blocking "communication?

When employing MPI_Send() and MPI_Recv(), the communication process is blocking in nature, implying that these operations wait until the communication is entirely finished before allowing the program to proceed. This blocking behavior signifies that the buffer supplied to MPI_Send() becomes available for reuse either because MPI has stored it somewhere or the destination has received it. Similarly, MPI_Recv() concludes when the receive buffer is filled with accurate data.

In contrast, MPI_Isend() and MPI_Irecv() facilitate non-blocking communication. These functions return promptly, even if the communication is still in progress, without causing the program to halt. To determine the completion of the communication, one can utilize MPI_Wait() or MPI_Test(). These non-blocking operations provide a mechanism to continue with other tasks while concurrently monitoring the progress of communication.

1.3 What is the MPI function for timing?

In MPI programs, when you want to measure the time it takes for a specific part of your code to execute, you can use the MPI function called **MPI_Wtime()**. This handy function returns the elapsed time in seconds. So, by noting the time before and after a particular code segment, you can easily calculate how long that part took to run. It's a practical tool for assessing performance and conducting benchmarks in the realm of parallel and distributed computing.

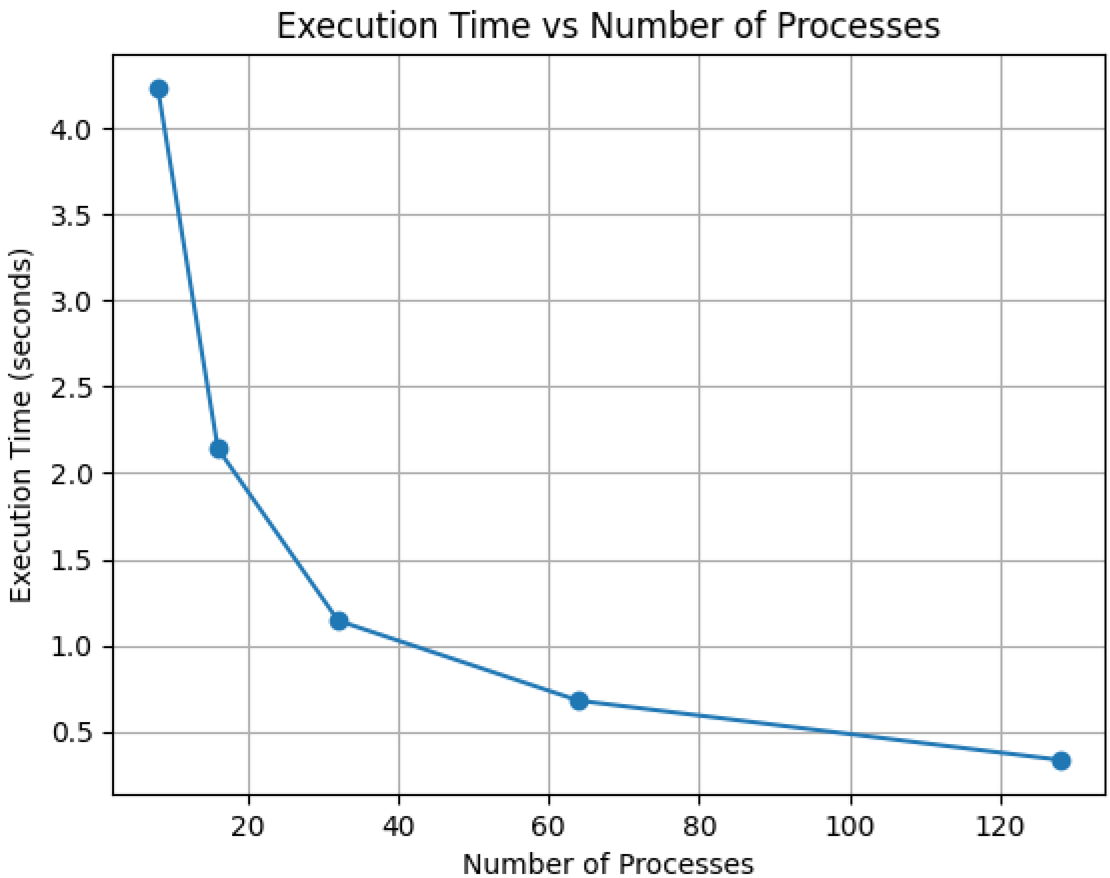
2. MPI Blocking Communication & Binary Tree Reduction Communication Algorithm.

2.1

The following code pi_p2p_tree.c implements binary tree algorithm with MPI Send/Recv. Below is the output:

```
[vvelidi@node0026 intro-mpi]$ mpirun -n 8 ./pi_p2p_tree
Estimated value of PI: 3.141597, Execution time: 4.231159 seconds
[vvelidi@node0026 intro-mpi]$ mpirun -n 16 ./pi_p2p_tree
Estimated value of PI: 3.141581, Execution time: 2.141729 seconds
[vvelidi@node0026 intro-mpi]$ mpirun -n 32 ./pi_p2p_tree
Estimated value of PI: 3.141572, Execution time: 1.143767 seconds
[vvelidi@node0026 intro-mpi]$ mpirun -n 64 ./pi_p2p_tree
Estimated value of PI: 3.141613, Execution time: 0.679758 seconds
[vvelidi@node0026 intro-mpi]$ mpirun -n 128 ./pi_p2p_tree
Estimated value of PI: 3.141620, Execution time: 0.336034 seconds
```

Number of Processes	**Execution Time (seconds)**
8	**4.231159**
16	**2.141729**
32	**1.143767**
64	**0.679758**
128	**0.336034**

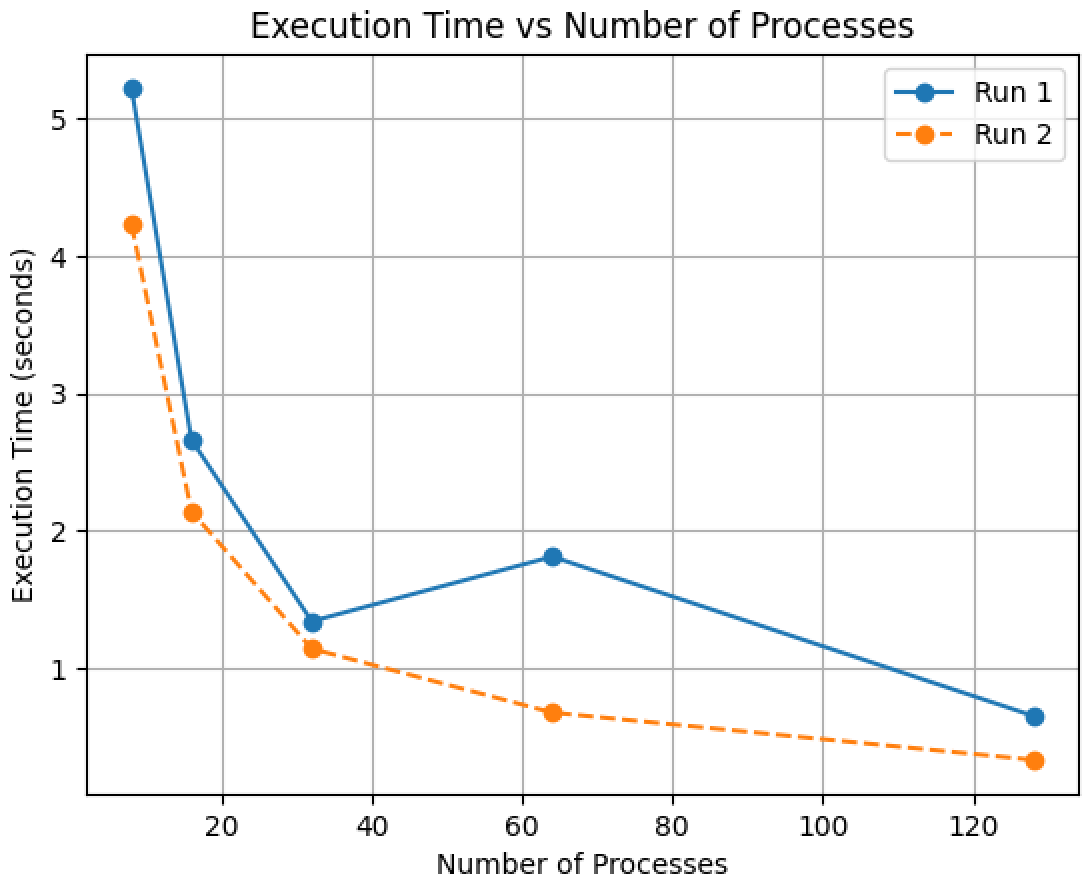


2.2

The following fig 2.1 shows the plot for process 8, 16, 32, 64, 128.

2.3

If we compare two plots shown the below figure 2.2



With increase in the processes linear reduction algorithm is taking more time compared to binary tree reduction.

3. MPI Collective: MPI_Gather

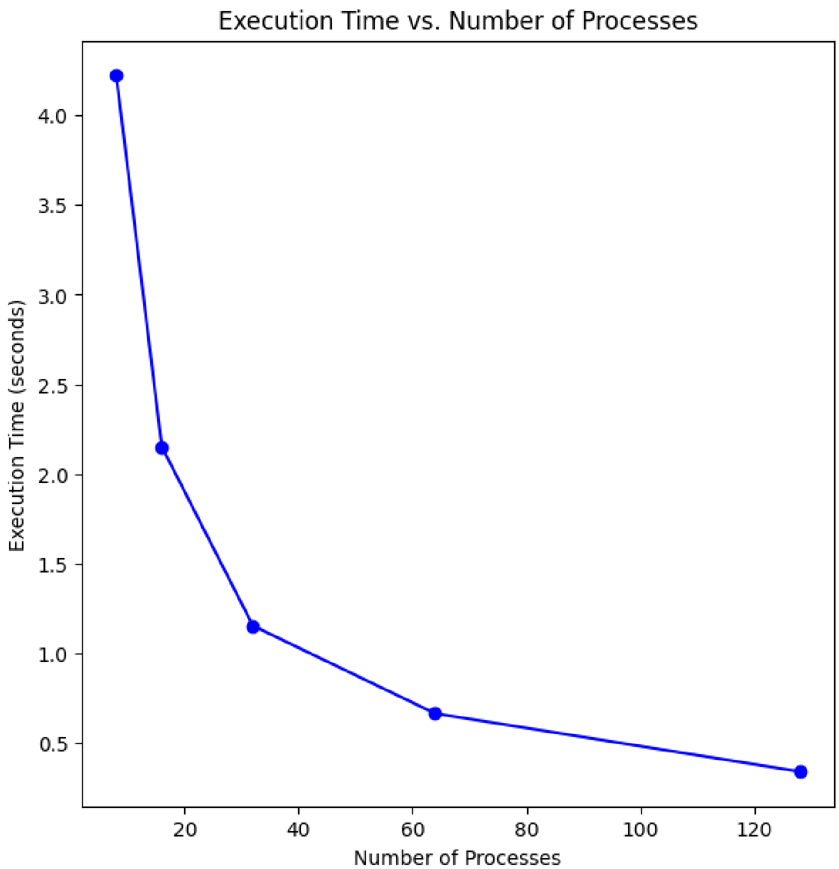
3.1

The following code pi_gather.c implements using MPI_Gather.

Below is the output:

```
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 8 ./pi_gather
pi=3.141604, t=4.220680
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 16
./pi_gather
pi=3.141565, t=2.150678
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 32
./pi_gather
pi=3.141574, t=1.153595
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 64
./pi_gather
pi=3.141609, t=0.665121
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 128
./pi_gather
pi=3.141620, t=0.341947
```

3.2



Number of Processes	Execution Time (seconds)
8	4.220680
16	2.150678
32	1.153595
64	0.665121
128	0.341947

3.3

In the context of MPI, communicating the final value of Pi to all processes involves leveraging the collective function named "broadcast." This MPI_Bcast function is quite handy for distributing information among processes. Notably, both the root process and other processes employ the same MPI_Bcast function, each playing its unique role.

Interestingly, when the root process initiates MPI_Bcast, it generously shares the data variable with all other processes. What adds to the intrigue is that, as each recipient process engages MPI_Bcast, the data variable gets filled with the valuable information transmitted by the root process. It's akin to a collaborative exchange where everyone is part of the communication loop!

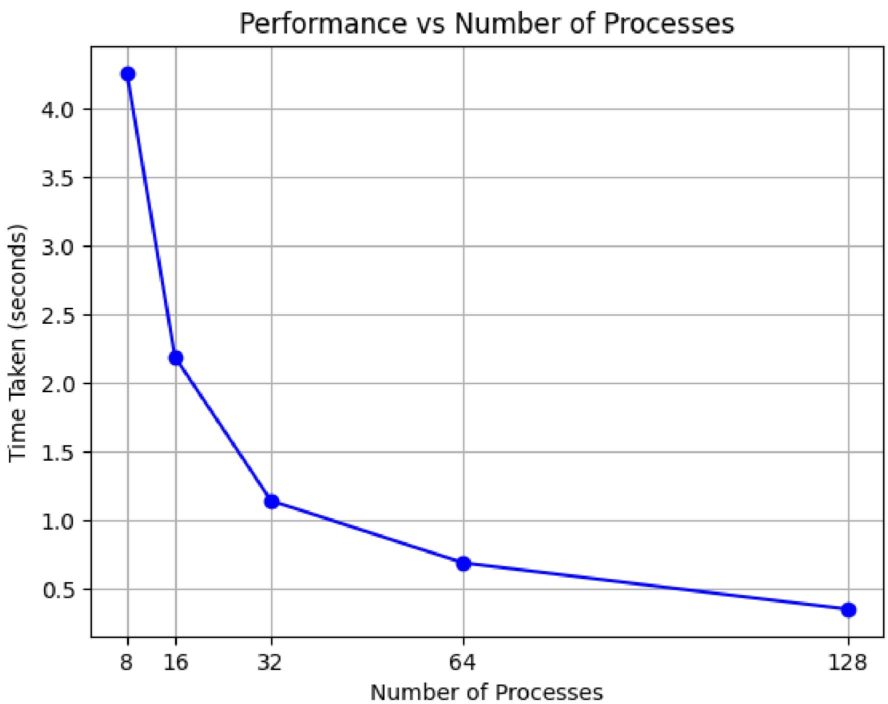
4. MPI Collective: MPI_Reduce

4.1

(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 8 ./pi_gather
pi=3.141604, t=4.258754
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 16 ./pi_gather
pi=3.141565, t=2.188433
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 32 ./pi_gather
pi=3.141574, t=1.140005
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 64 ./pi_gather
pi=3.141609, t=0.685930
(base) [vvelidi@node0026 intro-mpi]$ mpirun -np 128 ./pi_gather
pi=3.141620, t=0.348212

4.2

Plot and values are depicted below:



Number of Processes	Time Taken (seconds)
8	4.258754
16	2.188433
32	1.140005
64	0.685930
128	0.348212