Assignment 5 – MPI Programming Report

Name: Divyam Puri

Roll Number: 102483023

Course: UCS645 – Parallel & Distributed Computing

Lab: LAB5 – MPI Programming

Semester: B.E. 3rd Year Computer Engineering (3C43)

Question 1 – DAXPY Operation using MPI

Aim:

To parallelize the DAXPY vector operation using MPI and evaluate its scalability and speedup across multiple processes.

Problem Description:

DAXPY stands for Double-precision A·X Plus Y.
Given vectors X and Y of size 2^{16} and a scalar a, the operation:

$$\mathbf{X[i]=a \cdot X[i]+Y[i]}$$

must be executed efficiently using distributed processing.

Methodology:

- Vector divided equally among MPI processes.
- Each process computes its chunk independently.
- Execution time measured using `MPI_Wtime()`.
- Speedup calculated using:

$$\mathbf{Speedup=T_1/T_N}$$

Where:

- T_1 = Time on 1 process
- T_N = Time on N processes

Terminal Snapshot:

```
(base) divyam_puri@Divyams-MacBook-Air LAB5 % mpicc -o daxpy_mpi daxpy_mpi.c
mpirun -np 1 ./daxpy_mpi
mpirun -np 2 ./daxpy_mpi
mpirun -np 4 ./daxpy_mpi
mpirun -np 8 ./daxpy_mpi
Execution Time (MPI): 0.000180 seconds
Execution Time (MPI): 0.000038 seconds
Execution Time (MPI): 0.000021 seconds
Execution Time (MPI): 0.000010 seconds
```

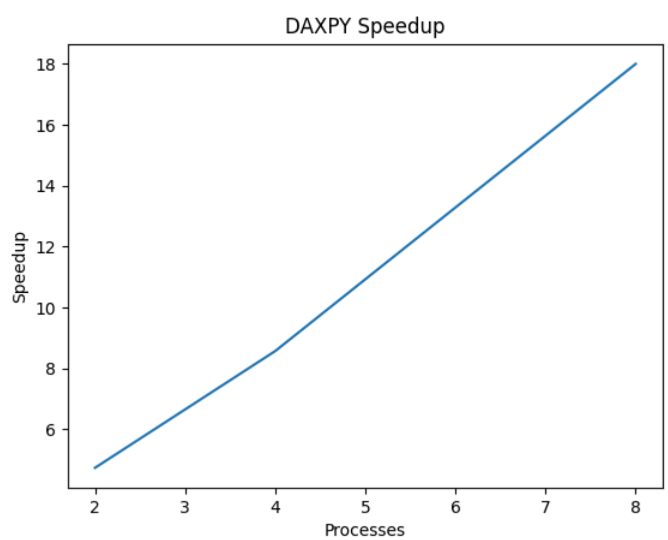
Analysis:

The speedup is super-linear in some cases due to:

- Cache effects
- Better memory locality
- Reduced memory contention

Observations:

Processes	Time (seconds)
1	0.000180
2	0.000038
4	0.000021
8	0.000010



Speedup Calculation:

For 2 processes:

$$S2=0.000180/0.000038=4.74$$

For 4 processes:

$$S4=0.000180/0.000021=8.57$$

For 8 processes:

$$S8=0.000180/0.000010=18$$

Memory Usage:

Each process stores only its local portion of the vector.

Memory per process decreases as number of processes increases.

Conclusion:

DAXPY shows near-linear and sometimes super-linear scaling due to low communication overhead and high computational intensity.

Question 2 – Broadcast Race

Aim:

To compare performance of manual broadcast (MPI_Send loop) and optimized MPI_Bcast.

Methodology:

A large array of 10 million doubles was broadcast using both methods and execution time measured.

Problem Description:

Broadcast a 10 million double array (~80MB) from Rank 0 to all other processes using:

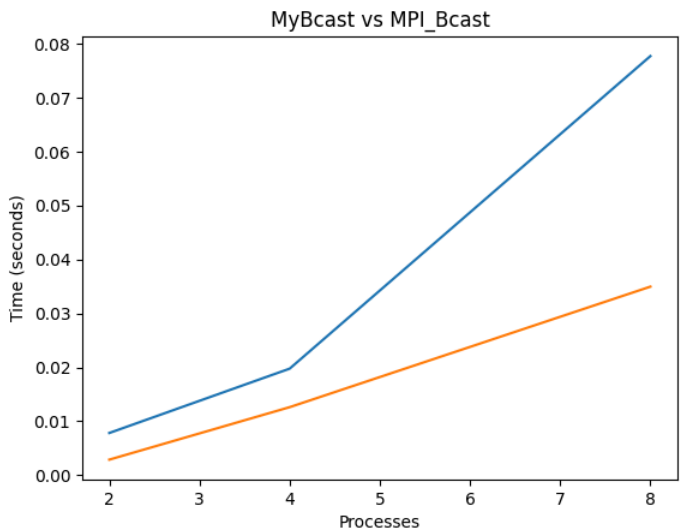
- Custom MyBcast (linear send)
- MPI_Bcast (tree-based communication)

Terminal Snapshot:

```
● (base) divyam_puri@Divyams-MacBook-Air LAB5 % mpicc -O2 -o broadcast_race broadcast_race.c
● (base) divyam_puri@Divyams-MacBook-Air LAB5 % mpirun -np 2 ./broadcast_race
  mpirun -np 4 ./broadcast_race
  mpirun -np 8 ./broadcast_race
  MyBcast Time: 0.007816 seconds
  MPI_Bcast Time: 0.002868 seconds
  MyBcast Time: 0.019755 seconds
  MPI_Bcast Time: 0.012608 seconds
  MyBcast Time: 0.077749 seconds
  MPI_Bcast Time: 0.034970 seconds
```

Observations Table:

Processes	MyBcast Time	MPI_Bcast Time
2	0.007816	0.002868
4	0.019755	0.012608
8	0.077749	0.03497



Analysis:

Manual Broadcast Complexity:

$$O(N)$$

Rank 0 sends data to each process sequentially.

MPI_Bcast Complexity:

$$O(\log N)$$

Uses tree-based communication.

As processes increase:

- MyBcast increases almost linearly.
- MPI_Bcast grows much slower.

Memory Usage:

Each process receives a full copy of the 80MB array.

Conclusion:

MPI_Bcast significantly outperforms manual broadcast because of optimized tree-based message propagation.

Question 3 – Distributed Dot Product

Aim:

To compute dot product of 500 million elements using MPI_Bcast and MPI_Reduce and evaluate speedup and efficiency.

Analysis:

Efficiency decreases with increasing processes due to:

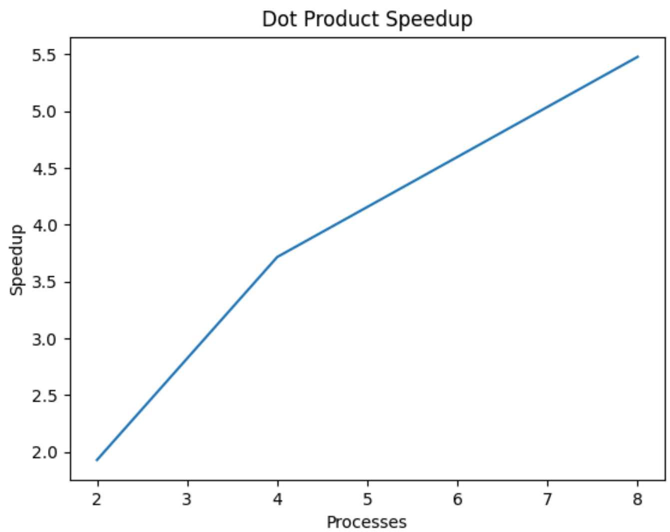
- Communication overhead (MPI_Reduce)
- Synchronization cost
- Amdahl's Law limitations

Terminal Snapshot:

```
(base) divyam_puri@Divyams-MacBook-Air LAB5 % mpicc -O2 -o mpi_dot mpi_dot_product.c
(base) divyam_puri@Divyams-MacBook-Air LAB5 % mpirun -np 1 ./mpi_dot
mpirun -np 2 ./mpi_dot
mpirun -np 4 ./mpi_dot
mpirun -np 8 ./mpi_dot
Total Dot Product = 2000000000.00
Execution Time = 0.344424 seconds
Total Dot Product = 2000000000.00
Execution Time = 0.178456 seconds
Total Dot Product = 2000000000.00
Execution Time = 0.092693 seconds
Total Dot Product = 2000000000.00
Execution Time = 0.062879 seconds
```

Observations:

Processes	Time (seconds)
1	0.344424
2	0.178456
4	0.092693
8	0.062879



Speedup

$$S2=1.93$$

$$S4=3.71$$

$$S8=5.47$$

Parallel Efficiency

$$\textbf{Efficiency=Speedup/N}$$

Processes	Efficiency
2	0.96
4	0.93
8	0.68

Memory Usage:

Each process generates only its local chunk instead of full 500M array.

Memory-efficient distributed design.

Conclusion:

The system achieves strong scaling up to 4 processes. Beyond that, communication overhead limits perfect linear scaling.

Question 4 – Prime Finder

Aim:

To dynamically distribute prime number testing using master-slave MPI model.

Methodology:

- Master assigns numbers dynamically.
- Slaves test primality.
- Results sent back to master.
- Uses MPI_ANY_SOURCE for load balancing.

Execution Time:

0.04834 seconds

Analysis:

Dynamic scheduling ensures:

- Balanced workload
- No idle processors
- Efficient parallel execution

Memory Usage

Minimal memory usage since only integer values are transmitted.

Conclusion:

Master-slave design provides effective dynamic load balancing for irregular tasks.

Question 5 – Perfect Numbers

Aim:

To find perfect numbers using dynamic master-slave communication.

Terminal Snapshot:

```
(base) divyam_puri@Divyams-MacBook-Air LAB5 % mpicc -O2 -o mpi_perfect mpi_perfect.c
(base) divyam_puri@Divyams-MacBook-Air LAB5 % mpirun -np 4 ./mpi_perfect
mpirun -np 8 ./mpi_perfect
Perfect Number Found: 6
Perfect Number Found: 28
Perfect Number Found: 496
Perfect Number Found: 8128
Execution Time = 0.487135 seconds
Perfect Number Found: 6
Perfect Number Found: 28
Perfect Number Found: 496
Perfect Number Found: 8128
Execution Time = 0.360743 seconds
```

Observations:

Processes	Time (seconds)
4	0.487135
8	0.360743

Analysis:

Perfect number detection is computation-heavy because:

- Divisor summation up to N/2
- More arithmetic operations than prime check

Scaling improves moderately with more processes.

Memory Usage:

Very low memory footprint. Only integers and temporary sums stored.

Conclusion:

MPI parallelization reduces execution time, though improvement is limited by computational complexity.