CS633 Assignment Group Number 52

Vihaan Sapra - 231149
Nischay Agarwal - 230705
Yug Dharajiya - 230362
Pallav Rastogi - 230731
Aryamann Srivastava - 230211

Contribution

Vihaan Sapra - 231149
Nischay Agarwal - 230705
Yug Dharajiya - 230362
Pallav Rastogi - 230731
Aryamann Srivastava - 230211
Equal contribution By All members

1 Code Description

The code is organized into the following logical modules:

1. MPI initialization and argument parsing
2. Neighbor determination and memory allocation
3. Data initialization
4. Iterative communication and computation loop
5. Global maximum communication
6. Cleanup and finalization

1.1 MPI Initialization and Argument Parsing

Each process initializes MPI and obtains its rank and total number of processes:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &P);
```

The program parses 5 command line args: M, D1, D2, T, seed

1.2 Neighbor Determination

Each process determines wether they are either sender or reciever or both for both D1 and D2 distanced neighbours

```
int hasDx = (rank + Dx <= P - 1);
int preDx = (rank - Dx >= 0);
```

1.3 Memory Allocation

Buffers for recieving and sending data(seperate) is only allocated if they are reciever, sender for both D1 and D2 distanced neighbours

```
double *sendDx = hasDx ? malloc(M * sizeof(double)) : NULL;
double *recvDx = preDx ? malloc(M * sizeof(double)) : NULL;
```

1.4 Data Initialization

Random numbers generated(as given in problem statement)

1.5 Iterative Communication and Computation

The main workload is executed inside a loop running T iterations.In every iteration only the valid senders send data to their D1 and D2 neighbours and hence only the valid senders receive the values after computation from their D1 and D2 neighbours.

1.5.1 Comminication

For both D1 and D2, the code uses an alternating send/receive ordering based on:

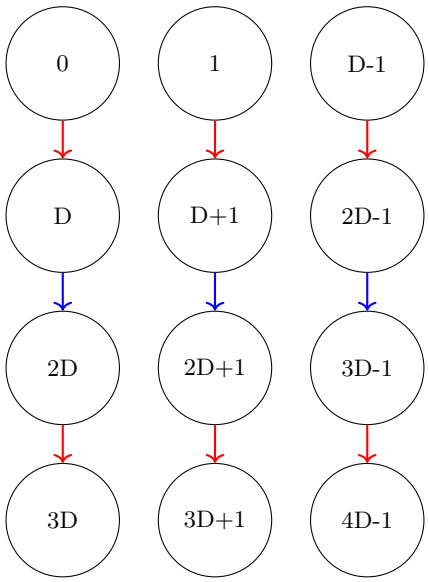
$$(rank/D) \bmod 2$$

```
if ((rank / D) % 2) {
    MPI_Recv(...);
    MPI_Send(...);
} else {
    MPI_Send(...);
    MPI_Recv(...);
}
```

This is similar to odd-even / even-odd communication which:

- Avoids deadlock
- Allows independent process pairs to communicate simultaneously
- Prevents serialization

Phase: $(rank/D) \bmod 2$ **based ordering**



Even group: Send
Odd group: Receive

Figure 1: Odd–even ordered MPI communication

1.5.2 Local Computation

Communication is done as per given in problem statement by all receivers.

Then all recievers communicate buffer to corresponding sender in similar fashion as described above.

1.6 Global Maximum

Each process computes local maxima:

```
double pair[2] = {-INFINITY, -INFINITY};
```

Non-root processes send their results to rank 0, which computes global maxima.

```
MPI_Recv(pair, 2, MPI_DOUBLE, r, 5, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Rank 0 also measures the total execution time using `MPI_Wtime()`.

1.7 Output

The root process prints max_D1, max_D2, Total execution time.

1.8 Memory Management and Finalization

All dynamically allocated buffers are freed, and MPI is finalized:

```
free(buf);
MPI_Finalize();
```

2 Code Compilation and Execution Instructions

Steps to Recreate our Results

chmod +x compile.sh
mkdir results
./compile.sh
sbatch job8.slurm
sbatch job16.slurm
sbatch job32.slurm

The results will be in the results folder [as csv in times_P($process_count)]. The standard output and error will be in the logs directory.

2.1 Compilation

The code is compiled using a shell script `compile.sh`. The script loads the required MPI module and compiles the program using `mpicc` with optimization level `-O3`.

```
#!/bin/bash
rm -f ./src
module purge
module load compiler/oneapi-2024/mpi/2021.12
mpicc -O3 src.c -lm -o src
```

Compilation is performed using:

```
./compile.sh
```

We just have to make sure the file has executing permsission which can be done by chmod +x compile.sh.

2.2 Execution Environment

All experiments were executed on the SLURM cluster using the CPU partition. The Intel MPI module `compiler/oneapi-2024/mpi/2021.12` was loaded for all runs.

Each configuration was executed using separate SLURM job scripts for:

- 8 processes (1 node)
- 16 processes (1 node)
- 32 processes (2 nodes)

For all experiments:

- $D1 = 2$
- $D2 = 4$
- $T = 10$
- $seed = 1000$
- $M \in \{262144,\ 1048576\}$
- 5 repetitions per configuration

2.3 Execution Script Structure

Each SLURM script:

- Allocates the required number of tasks
- Loads the MPI module
- Runs the program using `srun`
- Repeats each configuration 5 times
- Stores results in a CSV file

Example: 8 Processes (job8.slurm)

```
#SBATCH -N 1
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=16
#SBATCH -t 00:10:00
.
.
for M in "${M_LIST[@]}"; do
```

```
 for ((run=1; run<=REPEAT; run++)); do
  echo "P=8 run=$run M=$M"
   line=$(srun --mpi=pmi2 ./src $M $D1 $D2 $T $SEED)
       echo "8,$M,$run,$line" >> $CSV
      done
done
```

The 16-process and 32-process scripts are identical except for:

- `--ntasks=16` (1 node) for P=16
- `--ntasks=32` (2 nodes) for P=32

2.4 Job Submission

Jobs are submitted using:

```
sbatch job8.slurm
sbatch job16.slurm
sbatch job32.slurm
```

Each job generates a CSV file in results folder(create this folder for results to get stored) containing:

```
P, M, run, maxD1, maxD2, time (for the 5 runs)
```

These CSV files are then used to generate the timing boxplots shown in the Results section.

3 Results

3.1 Timing Results

All experiments were executed for:

- $P \in \{8, 16, 32\}$
- $M \in \{262144,\ 1048576\}$
- 5 repetitions per configuration

The recorded data was stored in a CSV file and visualized using boxplots. Figure 2 shows the distribution of execution time (in seconds) for each process count and data size.

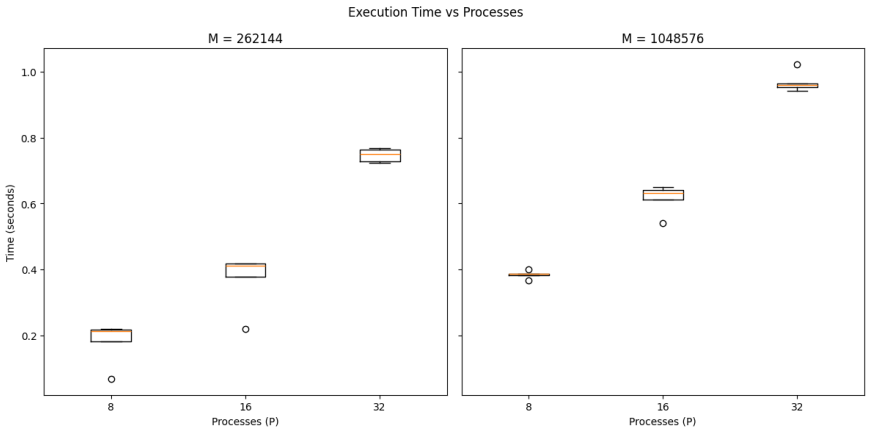


Figure 2: Execution time (seconds) for $P = 8, 16, 32$ processes.
Left: $M = 262144$. Right: $M = 1048576$.
Each boxplot represents 5 independent runs.

3.2 Observations and Analysis

1. Effect of Increasing M: For a fixed number of processes, execution time increases approximately linearly when M increases from 262144 to 1048576 (a factor of 4 increase in data size).

This confirms that both:

- Communication volume
- Local computation

scale proportionally with M.

2. Effect of Increasing P: Contrary to ideal strong scaling expectations, execution time increases as the number of processes increases:

P	Avg Time (M=262144)	Avg Time (M=1048576)
8	≈ 0.172 s	≈ 0.384 s
16	≈ 0.371 s	≈ 0.626 s
32	≈ 0.741 s	≈ 0.970 s

Instead of decreasing, runtime grows significantly with larger P.

3. Reason for Performance Degradation: This behavior is primarily due to communication overhead:

- The program uses blocking `MPI_Send` and `MPI_Recv`.

- Communication occurs in every iteration.
- The odd–even ordering introduces synchronization.
- Increasing P increases the total number of communicating pairs.
- For $P = 32$, communication spans multiple nodes, increasing network latency.

Because the workload per process decreases while communication frequency remains constant, the program becomes communication-bound at higher process counts.

4. Variability: For $M = 262144$, the variance is highest for $P = 8$ and significantly smaller for $P = 16$, with a moderate increase again at $P = 32$. This indicates that timing variability at lower process counts may arise from local scheduling effects.

For $M = 1048576$, the variance remains relatively small for $P = 8$ and $P = 16$, but increases for $P = 32$. The higher variability at $P = 32$ is consistent with inter-node communication overhead and network contention.

Overall, while variability exists, it remains small relative to the mean execution time, indicating that the implementation is stable and deterministic. The slight increase in variance at higher process counts is likely due to network latency, synchronization overhead from blocking communication, and shared cluster resource contention.

3.3 Summary

The implementation is correct and deterministic (identical maxD1 and maxD2 values across runs).
However, performance does not scale favorably with increasing process count due to:

- Blocking communication
- High synchronization cost
- Communication dominating computation at higher P

The results demonstrate that for this workload size, $P = 8$ provides the best performance.

4 Conclusions

In this assignment, we implemented the required sender–receiver communication workflow using only `MPI_Send` and `MPI_Recv`. The communication pattern was carefully designed using an odd–even ordering strategy based on $(rank/D)$ mod 2 to avoid deadlock while allowing independent process pairs to communicate concurrently.

The implementation correctly handles:

- Missing neighbors at domain boundaries
- Separate buffers for $D1$ and $D2$ communications
- Deterministic computation of local and global maxima
- Proper synchronization and final timing using `MPI_Wtime()`

Experimental results show that execution time increases with data size M, confirming that both communication volume and computation scale linearly with the input size. However, increasing the number of processes P does not yield performance improvement. Instead, runtime increases significantly for $P = 16$ and $P = 32$.

This behavior indicates that the implementation becomes communication-bound at higher process counts. Since communication occurs in every iteration using blocking operations, the overhead from synchronization and message latency grows with P. Additionally, inter node communication further increases the total execution time.

Overall, the program is functionally correct and well-structured, but its scalability is limited by the cost of repeated blocking communication. The results highlight an important principle in parallel programming: increasing the process count does not guarantee speedup when communication overhead outweighs computational savings.