Parallel and Distributed Processing Project
High Performance Computing

Name	Reg.No	Section
Aazmir Ahmed Bhatti	366399	PDP G2

Part-I: Getting to know the HPC Cluster Part-I is about finding out how the HPC cluster works. Create a map of all the compute nodes in the HPC cluster. Find out the CPU specs (number of cores etc.) of compute nodes. Check nodes that are mostly busy running long simulations. Rank nodes by availability and performance.

Part-II: Laplace Solver Implement Laplace Solver on HPC using MPI. You can test your implementation for correctness by using smaller arrays for comparison against single-threaded implementation. Progressively increase the size of your domain (2-D matrix) and make graphs to compare MPI performance with openmp implementation. For larger arrays, you will have to increase the number of epochs. Additionally, you will have to implement interprocess communication using MPI for sharing array data between neighbouring processes after each epoch. Do mention any performance bottlenecks you encounter.

Note 1: Please submit a report and explain your approach for MPI implementation includ- ing figure/graphs, problems faced and resolved/unresolved issues.
Note 2: Appropriately package your code so that the end-user can easily compile and run your code (makefile, cmake, VS build, etc.).

Note 3: HPC is a shared resource and users have a responsibility to use HPC responsibly. Avoid hogging HPC resources for too long by running unnecessarily long simulations. Keep track of RAM usage and don’t oversubscribe RAM as this can cause serious slowdowns. Avoid running simulations on the head node afrit.

Part-I

- Using a Script(.sh) file to run all the lscpu commands at once in head node to get a summary of all the nodes specifications

```
2. Node Summary (node_summary.csv)
--------------------------------
Node          Cores  Clock_Speed_MHz  CPU_Model  Uptime_Days  Load_Average  Longest_Process_Hours
compute-0-5   16     2267.000         100        14.07        2411.61
compute-0-6   16     1600.000         100        0.00         2411.58
compute-0-8   24     1600.000         100        0.01         2411.73
compute-0-9   16     2267.000         100        16.02        2412.24
compute-0-10  16     1600.000         100        0.10         2412.31
compute-0-11  16     1600.000         100        0.01         2412.27
compute-0-12  16     1600.000         100        0.00         2412.47
compute-0-13  16     1600.000         100        0.07         2412.53
compute-0-16  16     2267.000         101        13.00        2436.78
compute-0-17  16     1600.000         102        12.00        2455.85
compute-0-18  16     1600.000         102        0.00         2456.22
compute-0-19  16     2267.000         101        35.00        2435.81
compute-0-23  16     1600.000         100        0.00         2415.21
compute-0-24  16     1600.000         100        0.00         2415.16
compute-0-26  16     1600.000         100        0.00         2412.82
compute-0-27  16     1600.000         100        11.07        2415.01
```

Figure 1: (lscpu) of all nodes

- Using top to get an overall Ranking of all the Compute nodes in the Super Computer

```
2.1 Cluster Overview Statistics
------------------------------
Total Nodes Processed: 16
Total Cores Across Processed Nodes: 264
Average Uptime (Days): 100.4
Average Load Average (1-min): 6.33

3. Availability Ranking (Highest Uptime)
---------------------------------------
Nodes ranked by their uptime in days (highest first).
1. compute-0-18 (Uptime: 102 days)
2. compute-0-17 (Uptime: 102 days)
3. compute-0-19 (Uptime: 101 days)
4. compute-0-16 (Uptime: 101 days)
5. compute-0-9 (Uptime: 100 days)
6. compute-0-8 (Uptime: 100 days)
7. compute-0-6 (Uptime: 100 days)
8. compute-0-5 (Uptime: 100 days)
9. compute-0-27 (Uptime: 100 days)
10. compute-0-26 (Uptime: 100 days)
11. compute-0-24 (Uptime: 100 days)
12. compute-0-23 (Uptime: 100 days)
13. compute-0-13 (Uptime: 100 days)
14. compute-0-12 (Uptime: 100 days)
15. compute-0-11 (Uptime: 100 days)
16. compute-0-10 (Uptime: 100 days)
```

Figure 2 : (top) of all nodes

- The table ranks compute nodes based on availability (uptime) and performance (CPU clock speed), using data extracted from node_summary.txt as processed by the accompanying shell script. It provides a view of each node's performance, processing power, and workload history.

Full Ranking Table

Rank	Node	Uptime (Days)	Clock Speed (MHz)	Cores	Longest Process (hrs)
1	compute-0-18	102	1600	16	2456.22
2	compute-0-17	102	1600	16	2455.85
3	compute-0-19	101	2267	16	2435.81
4	compute-0-16	101	2267	16	2436.78
5	compute-0-9	100	2267	16	2412.24
6	compute-0-8	100	1600	24	2411.73
7	compute-0-6	100	1600	16	2411.58
8	compute-0-5	100	2267	16	2411.61
9	compute-0-27	100	1600	16	2415.01
10	compute-0-26	100	1600	16	2412.82
11	compute-0-24	100	1600	16	2415.16
12	compute-0-23	100	1600	16	2415.21
13	compute-0-13	100	1600	16	2412.53
14	compute-0-12	100	1600	16	2412.47
15	compute-0-11	100	1600	16	2412.27
16	compute-0-10	100	1600	16	2412.31

Part-II

Objective:

This task involved solving the 2D Laplace equation using three approaches: a serial implementation, an OpenMP-based shared memory parallel version, and an MPI-based distributed memory parallel version.

Methodology:

The Laplace equation was solved using the Jacobi iterative method on an n × n grid with fixed boundary conditions: +50 at the top, -50 at the bottom, and 0 elsewhere. Iterations continued until the maximum change across grid points was less than EPSILON = 0.001 or 5000 epochs were completed.

OpenMP-Serial-MPI:

- The Serial version used simple nested loops and was easy to implement but slow for large grids due to no parallelism.
- The OpenMP version parallelized grid updates using #pragma omp parallel for, improving performance on multi-core systems through shared memory.
- The MPI **version** split the grid across processes, used non-blocking communication for boundary exchange, and achieved better scalability for large problems despite some communication overhead.

Compilation:

```
mpic++ laplace.cpp -fopenmp -std=gnu++0x
```

Run:

```
mpirun -n 4  a.out
```

```
Kickstarted 13:09 13-Feb-2025
[user34@compute-0-8 ~]$ cd az/
[user34@compute-0-8 az]$ mpirun -n 4 a.out
Select execution mode (0: Serial, 1: OpenMP, 2: MPI): 1
Enter number of threads for OpenMP: 2
OpenMP GridSize: 100, Time: 0.424631 seconds
OpenMP GridSize: 500, Time: 17.6183 seconds
OpenMP GridSize: 700, Time: 34.6309 seconds
OpenMP GridSize: 1000, Time: 70.5187 seconds
[user34@compute-0-8 az]$ mpirun -n 4 a.out
Select execution mode (0: Serial, 1: OpenMP, 2: MPI): 2
MPI GridSize: 100, Time: 0.323383 seconds
MPI GridSize: 500, Time: 8.2036 seconds
MPI GridSize: 700, Time: 16.0857 seconds
MPI GridSize: 1000, Time: 32.7119 seconds
[user34@compute-0-8 az]$ mpirun -n 4 a.out
Select execution mode (0: Serial, 1: OpenMP, 2: MPI): 0
Serial GridSize: 100, Time: 0.730089 seconds
Serial GridSize: 500, Time: 31.6808 seconds
Serial GridSize: 700, Time: 62.3564 seconds
Serial GridSize: 1000, Time: 128.899 seconds
[user34@compute-0-8 az]$
```

Figure 3 : Running Laplace on Compute Node 8

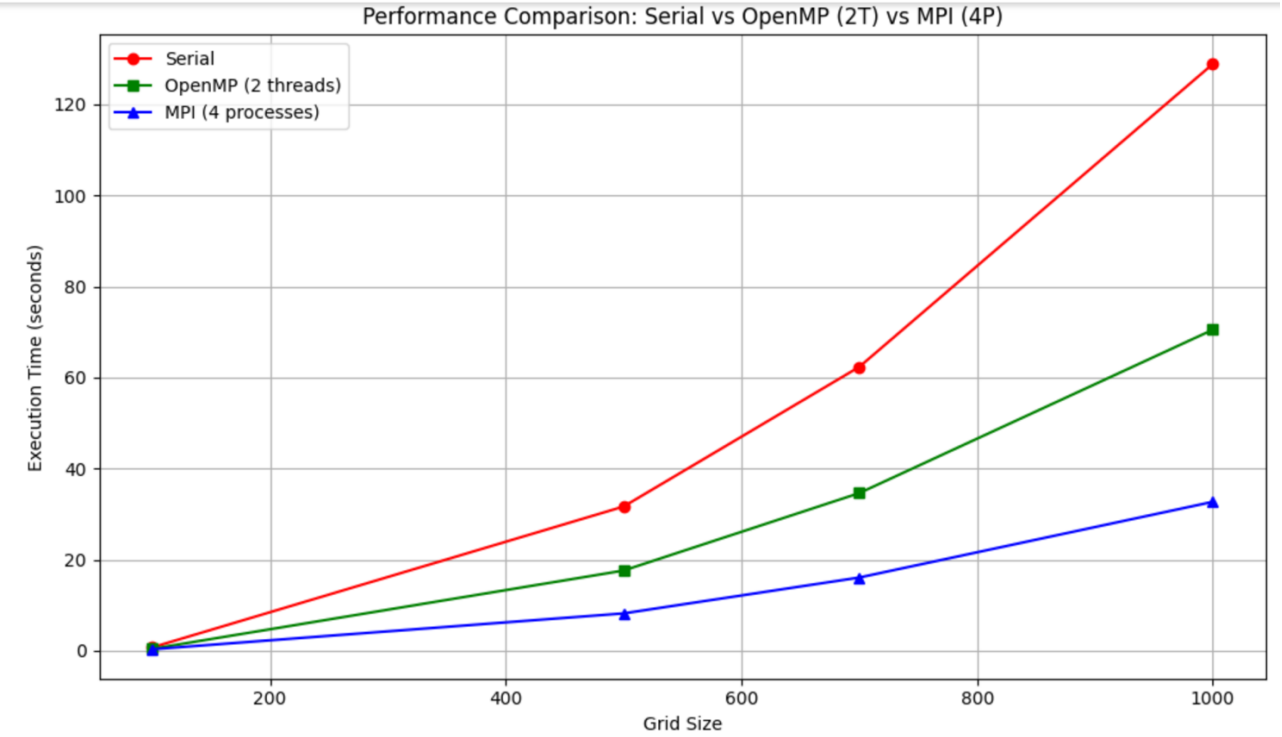

Figure 4 : Grid Size vs Execution time Laplace

Bottlenecks in MPI:

- For small grids, the MPI version can be slower because the time spent communicating between processes is more than the time saved by parallelizing.
- Network delays can slow down the program since processes need to exchange data frequently.
- If the grid is not evenly divided, some processes do more work than others, reducing overall speedup.

Problems Faced:

- Version issues while sending my files from host PC to HPC using **scp** command, causing files to not be sent to HPC.
- MPI Deadlock: Initially faced blocking communication issues, resolved using non-blocking MPI_Isend and MPI_Irecv with MPI_Waitall.
- High computation time in the serial version for large grid sizes taking too long in testing larger gridsizes.

Solution:

- Distribute grid rows more evenly across processes to avoid load imbalance and improve performance.
- Used a script to check Node Availability which output the results in a text file from which I extracted my data in table and the script gave outputs for figure 1 and 2.