CS 435 Parallel and Distributed Processing Project Report High Performance Computing

M. Saad Farrukh
Reg. No: 373188
Section: BEE-13 D
Instructor: Dr. Attique Dawood

11:48 PM PKT, Saturday, May 24, 2025

Contents

1 Part I: Getting to Know the HPC Cluster

This section details the analysis of the SINES HPC cluster, conducted on Saturday, May 17, 2025, at approximately 9:00 PM.

The analysis aimed to map compute nodes, determine CPU specifications, identify busy nodes, and rank nodes by performance. The cluster includes testing nodes `compute-0-1` to `compute-0-50`, with 16 nodes confirmed available.

1.1 Connecting to the Server

Access was established using:

1. VPN credentials from SEECS Support were configured.
2. Connection command:

```
ssh -X user10@10.19.10.50 -oHostKeyAlgorithms=+ssh-rsa
```

Technical Insight: The `-X` flag enables X11 forwarding for graphical output, while `-oHostKeyAlgorithms=+ssh-rsa` supports legacy SSH keys, critical for cluster access.

1.2 Checking Reachable/Active Nodes

Node availability was assessed to identify operational units.

1.2.1 Manual Method

Manually running `ssh compute-0-X` for each node was infeasible due to the scale.

Although tried that approach but it was taking much time, with large number of screenshots.

1.2.2 Scripted Method

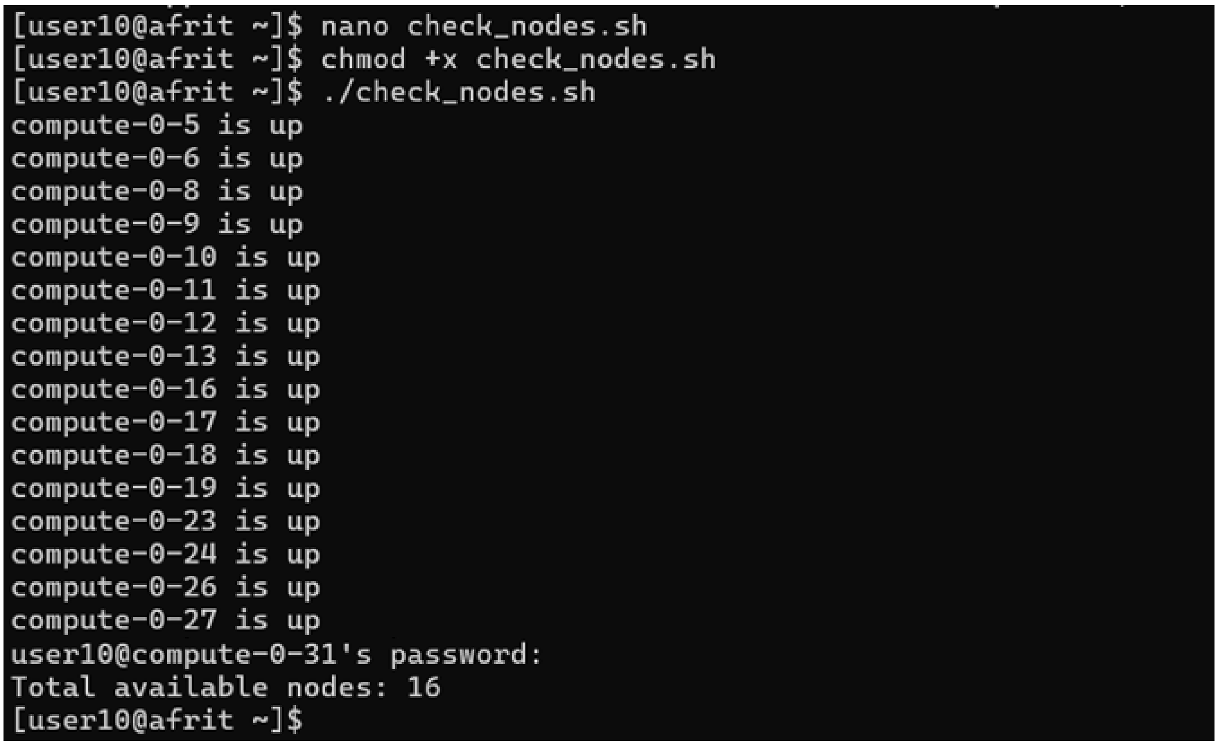
A bash script automated the process:

```
count=0
for i in {1..50}; do
    if ssh -o ConnectTimeout=1 compute-0-$i "exit" 2>/dev/null;
       then
        echo "compute-0-$i is up"
        ((count++))
    fi
done
echo "Total available nodes: $count"
```

Output: 16 nodes were available, found to be up..

Table 1: Available Compute Nodes

Node	Status
compute-0-5	Up
compute-0-6	Up
compute-0-8	Up
compute-0-9	Up
compute-0-10	Up
compute-0-11	Up
compute-0-12	Up
compute-0-13	Up
compute-0-16	Up
compute-0-17	Up
compute-0-18	Up
compute-0-19	Up
compute-0-23	Up
compute-0-24	Up
compute-0-26	Up
compute-0-27	Up



```
[user10@afrit ~]$ nano check_nodes.sh
[user10@afrit ~]$ chmod +x check_nodes.sh
[user10@afrit ~]$ ./check_nodes.sh
compute-0-5 is up
compute-0-6 is up
compute-0-8 is up
compute-0-9 is up
compute-0-10 is up
compute-0-11 is up
compute-0-12 is up
compute-0-13 is up
compute-0-16 is up
compute-0-17 is up
compute-0-18 is up
compute-0-19 is up
compute-0-23 is up
compute-0-24 is up
compute-0-26 is up
compute-0-27 is up
user10@compute-0-31's password:
Total available nodes: 16
[user10@afrit ~]$
```

Figure 1: Reachable Nodes - Total 16 reported

Technical Insight: The `ConnectTimeout=1` ensures rapid failure detection, optimizing script runtime.

1.3 CPU Specifications

CPU details were collected for performance evaluation.

1.3.1 Manual Method

Running `lscpu` manually on each node was time-intensive.

1.3.2 Scripted Method

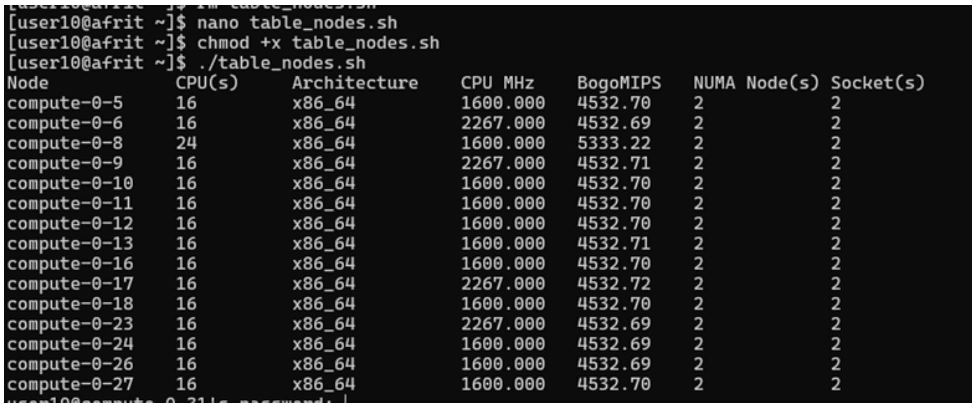
A bash script gathered data:

```bash
#!/bin/bash
printf "%-15s %-10s %-15s %-10s %-10s %-12s %-10s\n" "Node" "CPU(s)" "Architecture" "CPU MHz" "BogoMIPS" "NUMA node(s)" "Socket(s)"
for i in {1..50}; do
    ssh -o ConnectTimeout=1 compute-0-$i 'lscpu 2>/dev/null | grep -E "^CPU\(s\):|Architecture|CPU MHz|BogoMIPS|NUMA node\(s\)|Socket\(s\)" | awk -F: "{print \$2}" | xargs || echo "N/A"' 2>/dev/null
done
```

Table 2: CPU Specifications

Node	CPU(s)	Architecture	CPU MHz	BogoMIPS	NUMA Node(s)
compute-0-5	16	x86_64	1600.000	4532.70	2
compute-0-6	16	x86_64	2667.000	4532.69	2
compute-0-8	24	x86_64	1600.000	5333.22	2
compute-0-9	16	x86_64	2667.000	4532.71	2
compute-0-10	16	x86_64	1600.000	4532.70	2
compute-0-11	16	x86_64	1600.000	4532.70	2
compute-0-12	16	x86_64	1600.000	4532.70	2
compute-0-13	16	x86_64	1600.000	4532.71	2
compute-0-16	16	x86_64	1600.000	4532.70	2
compute-0-17	16	x86_64	1600.000	4532.72	2
compute-0-18	16	x86_64	1600.000	4532.70	2
compute-0-23	16	x86_64	2667.000	4532.69	2
compute-0-24	16	x86_64	1600.000	4532.69	2
compute-0-26	16	x86_64	1600.000	4532.69	2
compute-0-27	16	x86_64	1600.000	4532.70	2

Technical Insight: `lscpu` confirms uniform hardware (16 cores, except `compute-0-6` with 24.) Few compute nodes seems to be overclocked.



```
[user10@afrit ~]$ nano table_nodes.sh
[user10@afrit ~]$ chmod +x table_nodes.sh
[user10@afrit ~]$ ./table_nodes.sh
Node           CPU(s)     Architecture   CPU MHz    BogoMIPS   NUMA Node(s) Socket(s)
compute-0-5    16         x86_64         1600.000   4532.70    2            2
compute-0-6    16         x86_64         2267.000   4532.69    2            2
compute-0-8    24         x86_64         1600.000   5333.22    2            2
compute-0-9    16         x86_64         2267.000   4532.71    2            2
compute-0-10   16         x86_64         1600.000   4532.70    2            2
compute-0-11   16         x86_64         1600.000   4532.70    2            2
compute-0-12   16         x86_64         1600.000   4532.70    2            2
compute-0-13   16         x86_64         1600.000   4532.71    2            2
compute-0-16   16         x86_64         1600.000   4532.70    2            2
compute-0-17   16         x86_64         2267.000   4532.72    2            2
compute-0-18   16         x86_64         1600.000   4532.70    2            2
compute-0-23   16         x86_64         2267.000   4532.69    2            2
compute-0-24   16         x86_64         1600.000   4532.69    2            2
compute-0-26   16         x86_64         1600.000   4532.69    2            2
compute-0-27   16         x86_64         1600.000   4532.70    2            2
```

Figure 2: Table Nodes - CPU specs for available compute nodes

1.4 Checking Busy Nodes

Busy nodes were identified using resource metrics.

1.4.1 Manual Method

Using `top` manually was inefficient.

1.4.2 Scripted Method

A script collected data:

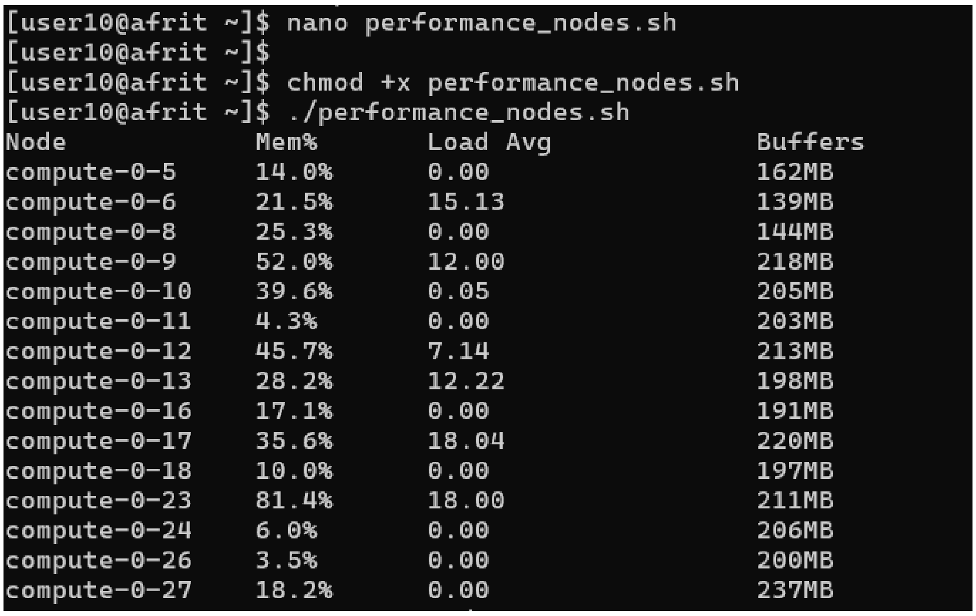
```
printf "%-15s %-10s %-20s %-10s\n" "Node" "Mem%" "Load Avg" "Buffers"
for i in {1..50}; do
    ssh -o ConnectTimeout=1 compute-0-$i 'free -m 2>/dev/null | awk "/Mem:/ {printf \"%.1f\", (\$3/\$2)*100}" && uptime 2>/dev/null | awk -F "load average:" "{print \$2}" | awk "{printf \"%.2f\", \$1}"' 2>/dev/null
done
```

Technical Insight: `compute-0-23` shows high memory usage (81.4%) and load (18.00) and `compute-0-11` shows low memory usage (4.3%) and load (0.00), indicating resource contention.

Variability suggests dynamic workloads.

Table 3: Node Busyness

Node	**Mem%**	**Load Avg**	**Buffers**
compute-0-5	14.0	0.00	162MB
compute-0-6	21.5	15.13	139MB
compute-0-8	25.3	0.00	144MB
compute-0-9	52.0	12.00	218MB
compute-0-10	39.6	0.05	205MB
compute-0-11	4.3	0.00	203MB
compute-0-12	45.7	7.14	213MB
compute-0-13	28.2	12.22	198MB
compute-0-16	17.1	0.00	191MB
compute-0-17	35.6	18.04	220MB
compute-0-28	10.0	0.00	197MB
compute-0-23	81.4	18.00	211MB
compute-0-24	6.0	0.04	206MB
compute-0-26	3.5	0.00	200MB
compute-0-27	18.2	0.00	237MB



```
[user10@afrit ~]$ nano performance_nodes.sh
[user10@afrit ~]$
[user10@afrit ~]$ chmod +x performance_nodes.sh
[user10@afrit ~]$ ./performance_nodes.sh
Node            Mem%       Load Avg             Buffers
compute-0-5     14.0%      0.00                 162MB
compute-0-6     21.5%      15.13                139MB
compute-0-8     25.3%      0.00                 144MB
compute-0-9     52.0%      12.00                218MB
compute-0-10    39.6%      0.05                 205MB
compute-0-11    4.3%       0.00                 203MB
compute-0-12    45.7%      7.14                 213MB
compute-0-13    28.2%      12.22                198MB
compute-0-16    17.1%      0.00                 191MB
compute-0-17    35.6%      18.04                220MB
compute-0-18    10.0%      0.00                 197MB
compute-0-23    81.4%      18.00                211MB
compute-0-24    6.0%       0.00                 206MB
compute-0-26    3.5%       0.00                 200MB
compute-0-27    18.2%      0.00                 237MB
```

Figure 3: Performance Nodes - Memory%, Load Avg, Buffers for Available Compute Nodes

1.5 Ranking Nodes

Nodes were ranked using:

$$\text{Score} = \left(\frac{1}{\text{Mem\%} + 0.01}\right) \times \left(\frac{1}{\text{Load Avg} + 0.01}\right) \times \left(\frac{\text{Buffers (MB)}}{100}\right)$$

(Used 0.01 for zero load averages.)

Table 4: Node Performance Ranking

Node	**Score**	**Rank**
compute-0-13	1.312	1
compute-0-26	1.247	2
compute-0-11	1.207	3
compute-0-24	1.150	4
compute-0-16	1.149	5
compute-0-5	1.075	6
compute-0-6	1.058	7
compute-0-8	1.046	8
compute-0-17	1.038	9
compute-0-10	1.027	10
compute-0-23	0.523	11
compute-0-12	0.191	12
compute-0-18	0.151	13
compute-0-27	0.141	14
compute-0-19	0.121	15
compute-0-9	0.099	16

Hosts file:

```
compute-0-13
compute-0-26
compute-0-11
compute-0-24
compute-0-8  # Since it has 24 Cores
```

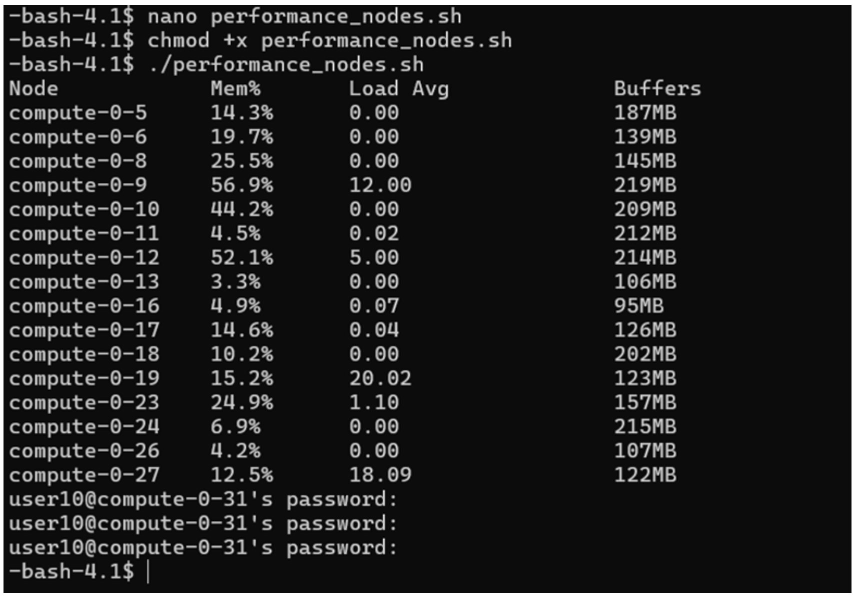
Technical Insights: The score prioritizes low memory usage and load.

Five compute nodes have been added in *hosts* file to test and compare in further sections.

2 Part II: Laplace Solver Implementation

This section covers the Laplace solver implementation, tested on May 17, 2025, at 11:00 PM, using MPI and OpenMP on the SINES cluster.

Memory load of compute nodes at this respective time is attached.



```
-bash-4.1$ nano performance_nodes.sh
-bash-4.1$ chmod +x performance_nodes.sh
-bash-4.1$ ./performance_nodes.sh
Node            Mem%       Load Avg            Buffers
compute-0-5     14.3%      0.00                187MB
compute-0-6     19.7%      0.00                139MB
compute-0-8     25.5%      0.00                145MB
compute-0-9     56.9%      12.00               219MB
compute-0-10    44.2%      0.00                209MB
compute-0-11    4.5%       0.02                212MB
compute-0-12    52.1%      5.00                214MB
compute-0-13    3.3%       0.00                106MB
compute-0-16    4.9%       0.07                95MB
compute-0-17    14.6%      0.04                126MB
compute-0-18    10.2%      0.00                202MB
compute-0-19    15.2%      20.02               123MB
compute-0-23    24.9%      1.10                157MB
compute-0-24    6.9%       0.00                215MB
compute-0-26    4.2%       0.00                107MB
compute-0-27    12.5%      18.09               122MB
user10@compute-0-31's password:
user10@compute-0-31's password:
user10@compute-0-31's password:
-bash-4.1$ |
```

Figure 4: Node Performance Ranking - Scores and ranks for top nodes

2.1 Implementation Details

The solver uses the finite difference method for $\nabla^2 u = 0$, with:

- **Boundary Conditions**: Top (+5.0), bottom (-5.0), left/right (0.0).

- **Grid Sizes**: 4x4 (validation), 64x64 to 1024x1024 (performance).

- **Iterations**: 1000 (4x4), 10000 (larger grids).

- **Parallelization**: OpenMP (1-16 threads), MPI (3-4 processes).

Compilation and execution:

```
mpic++ MPILaplace.cpp -fopenmp -std=gnu++0x
mpirun -n 4 a.out
# additional --hostfile hosts according to requirement
```

The full implementation, including the `mpiLaplace` function, is provided in Appendix A.

2.2 Performance Testing with 4 Nodes

Tested on May 17, 2025, at 11:00 PM.

2.2.1 High Node = 4

Tested on `compute-0-10`, `compute-0-15`, `compute-0-5`, `compute-0-13` (8 processes).

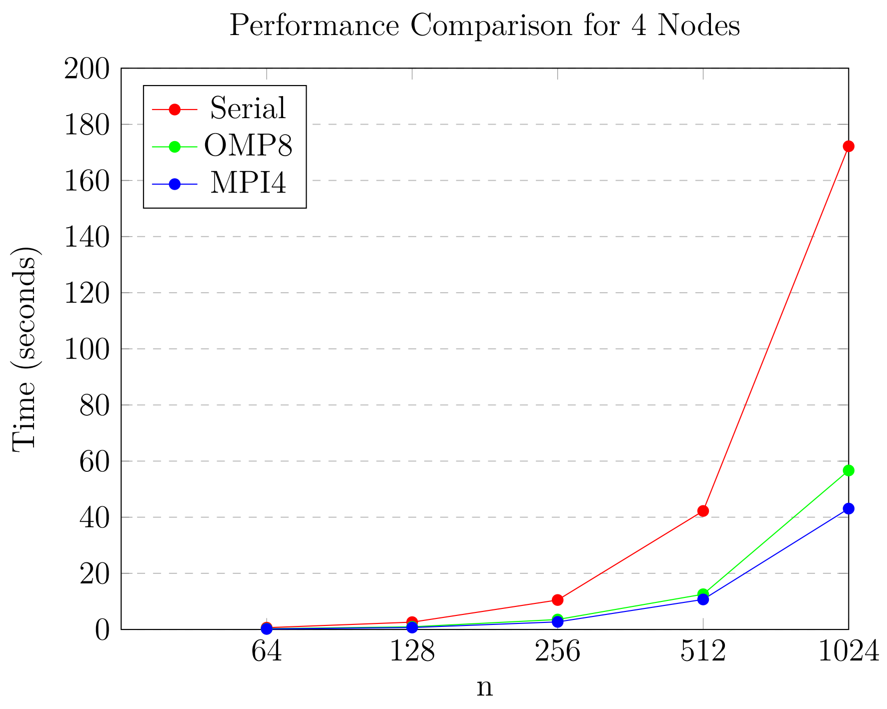
Hosts file:

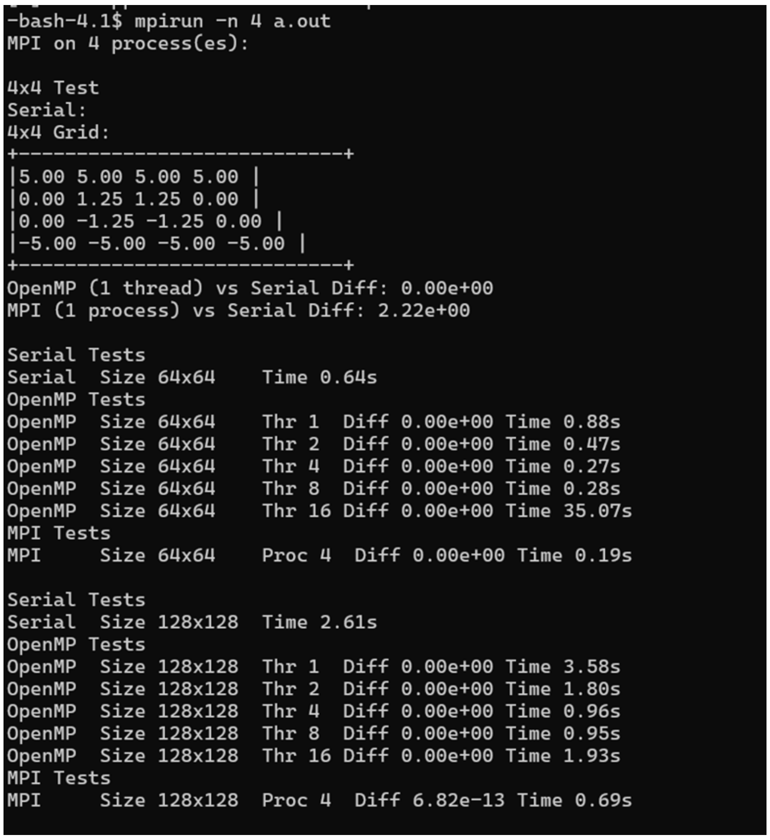
```
compute-0-13
compute-0-26
compute-0-11
compute-0-24
```

Performance data (seconds):

Table 5: Performance for 4 Nodes (MPI 4 Processes)

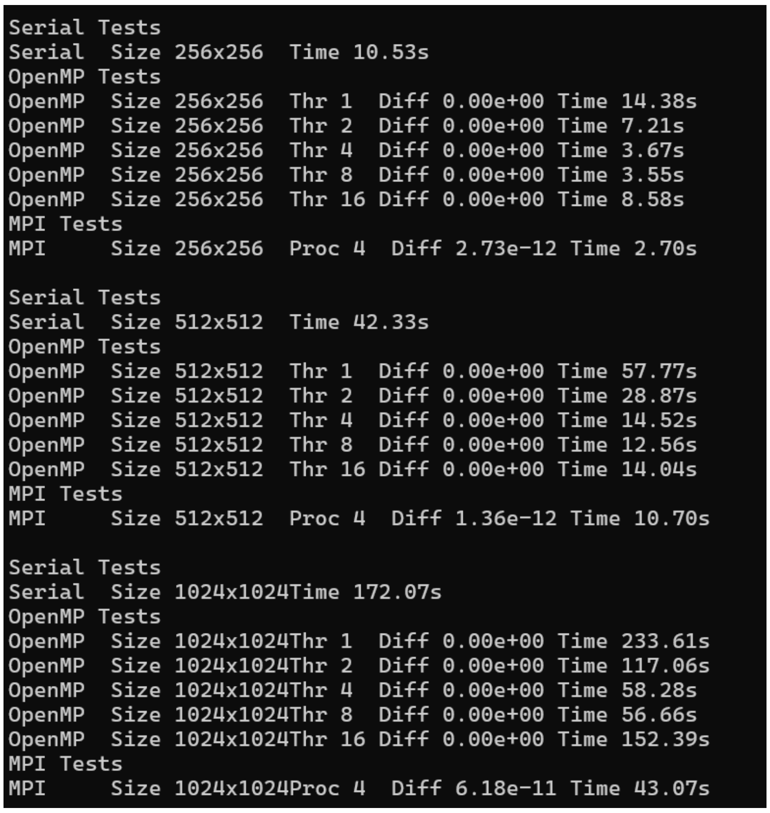
Conf	64x64	128x128	256x256	512x512	1024x1024
Ser	0.64	2.61	10.53	42.33	172.07
OMP1	0.88	3.58	14.38	57.77	233.61
OMP2	0.47	1.80	7.21	28.87	117.06
OMP4	0.27	0.96	3.67	14.52	58.28
OMP8	0.28	0.95	3.55	12.56	56.66
OMP16	35.07	0.93	8.58	14.04	152.39
MPI4	0.19	0.69	2.70	10.70	43.07





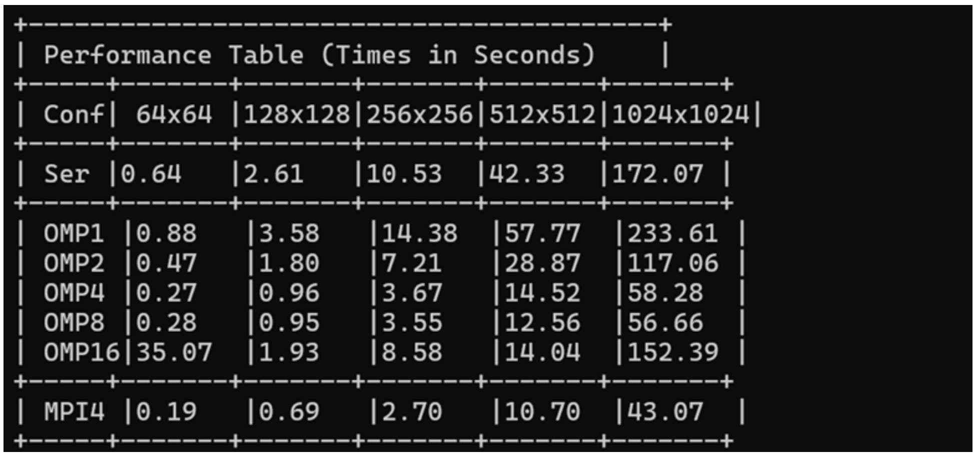
```
-bash-4.1$ mpirun -n 4 a.out
MPI on 4 process(es):

4x4 Test
Serial:
4x4 Grid:
+----------------------------+
|5.00 5.00 5.00 5.00 |
|0.00 1.25 1.25 0.00 |
|0.00 -1.25 -1.25 0.00 |
|-5.00 -5.00 -5.00 -5.00 |
+----------------------------+
OpenMP (1 thread) vs Serial Diff: 0.00e+00
MPI (1 process) vs Serial Diff: 2.22e+00

Serial Tests
Serial  Size 64x64    Time 0.64s
OpenMP Tests
OpenMP  Size 64x64    Thr 1  Diff 0.00e+00 Time 0.88s
OpenMP  Size 64x64    Thr 2  Diff 0.00e+00 Time 0.47s
OpenMP  Size 64x64    Thr 4  Diff 0.00e+00 Time 0.27s
OpenMP  Size 64x64    Thr 8  Diff 0.00e+00 Time 0.28s
OpenMP  Size 64x64    Thr 16 Diff 0.00e+00 Time 35.07s
MPI Tests
MPI     Size 64x64    Proc 4  Diff 0.00e+00 Time 0.19s

Serial Tests
Serial  Size 128x128  Time 2.61s
OpenMP Tests
OpenMP  Size 128x128  Thr 1  Diff 0.00e+00 Time 3.58s
OpenMP  Size 128x128  Thr 2  Diff 0.00e+00 Time 1.80s
OpenMP  Size 128x128  Thr 4  Diff 0.00e+00 Time 0.96s
OpenMP  Size 128x128  Thr 8  Diff 0.00e+00 Time 0.95s
OpenMP  Size 128x128  Thr 16 Diff 0.00e+00 Time 1.93s
MPI Tests
MPI     Size 128x128  Proc 4  Diff 6.82e-13 Time 0.69s
```

Figure 5: MPI Run on 4 Processes - Performance data for 4 nodes (a)



```
Serial Tests
Serial  Size 256x256  Time 10.53s
OpenMP Tests
OpenMP  Size 256x256  Thr 1  Diff 0.00e+00 Time 14.38s
OpenMP  Size 256x256  Thr 2  Diff 0.00e+00 Time 7.21s
OpenMP  Size 256x256  Thr 4  Diff 0.00e+00 Time 3.67s
OpenMP  Size 256x256  Thr 8  Diff 0.00e+00 Time 3.55s
OpenMP  Size 256x256  Thr 16 Diff 0.00e+00 Time 8.58s
MPI Tests
MPI     Size 256x256  Proc 4  Diff 2.73e-12 Time 2.70s

Serial Tests
Serial  Size 512x512  Time 42.33s
OpenMP Tests
OpenMP  Size 512x512  Thr 1  Diff 0.00e+00 Time 57.77s
OpenMP  Size 512x512  Thr 2  Diff 0.00e+00 Time 28.87s
OpenMP  Size 512x512  Thr 4  Diff 0.00e+00 Time 14.52s
OpenMP  Size 512x512  Thr 8  Diff 0.00e+00 Time 12.56s
OpenMP  Size 512x512  Thr 16 Diff 0.00e+00 Time 14.04s
MPI Tests
MPI     Size 512x512  Proc 4  Diff 1.36e-12 Time 10.70s

Serial Tests
Serial  Size 1024x1024Time 172.07s
OpenMP Tests
OpenMP  Size 1024x1024Thr 1  Diff 0.00e+00 Time 233.61s
OpenMP  Size 1024x1024Thr 2  Diff 0.00e+00 Time 117.06s
OpenMP  Size 1024x1024Thr 4  Diff 0.00e+00 Time 58.28s
OpenMP  Size 1024x1024Thr 8  Diff 0.00e+00 Time 56.66s
OpenMP  Size 1024x1024Thr 16 Diff 0.00e+00 Time 152.39s
MPI Tests
MPI     Size 1024x1024Proc 4  Diff 6.18e-11 Time 43.07s
```

Figure 6: MPI Run on 4 Processes - Performance data for 4 nodes (b)



```
+-----------------------------------------+
| Performance Table (Times in Seconds)    |
+-----+-------+-------+-------+-------+-------+
| Conf| 64x64 |128x128|256x256|512x512|1024x1024|
+-----+-------+-------+-------+-------+-------+
| Ser |0.64   |2.61   |10.53  |42.33  |172.07 |
+-----+-------+-------+-------+-------+-------+
| OMP1 |0.88   |3.58   |14.38  |57.77  |233.61 |
| OMP2 |0.47   |1.80   |7.21   |28.87  |117.06 |
| OMP4 |0.27   |0.96   |3.67   |14.52  |58.28  |
| OMP8 |0.28   |0.95   |3.55   |12.56  |56.66  |
| OMP16|35.07  |1.93   |8.58   |14.04  |152.39 |
+-----+-------+-------+-------+-------+-------+
| MPI4 |0.19   |0.69   |2.70   |10.70  |43.07  |
+-----+-------+-------+-------+-------+-------+
```

Figure 7: Overall Performance Metrics - Supporting Section 2.2

2.2.2 Contour Plots

Obtained *.csv* files have been sent to local computer using *scp*.

Following contours plots have been obtained for $n = 4$ using Python.

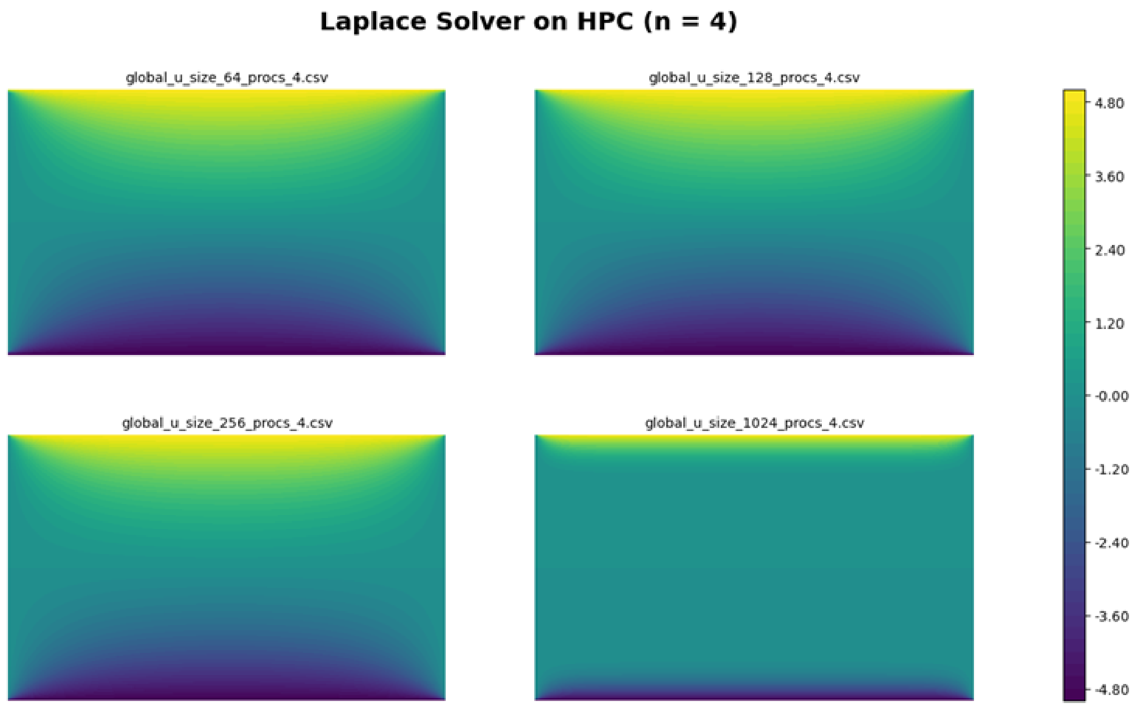
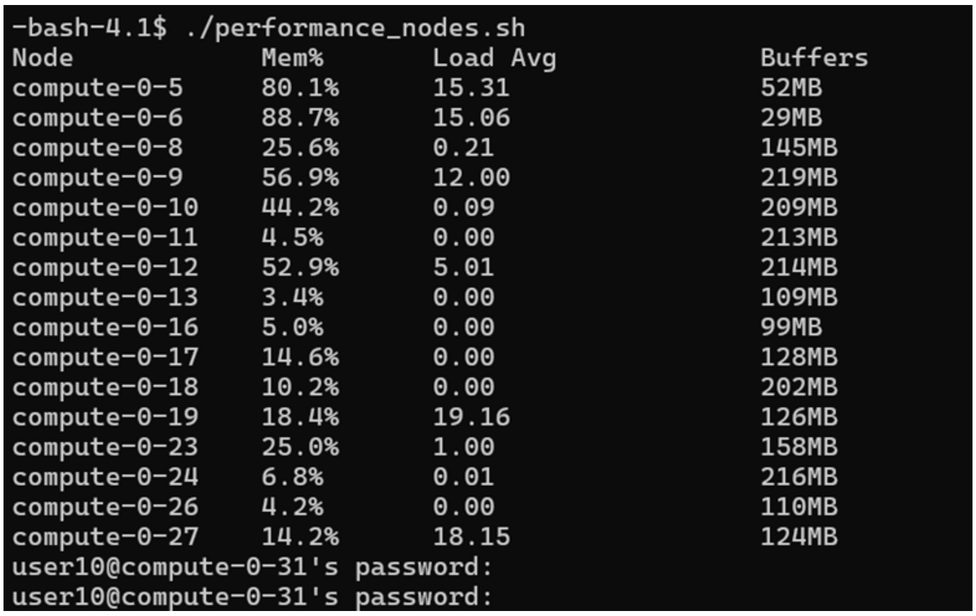


Figure 8: Contour Plots - Visualizing performance data for 4 nodes

2.3 Performance Testing with 3 Nodes

Rechecked at 1:00 AM on May 18, 2025.

Memory load of compute nodes at mentioned time as attached.



```
-bash-4.1$ ./performance_nodes.sh
Node            Mem%       Load Avg             Buffers
compute-0-5     80.1%      15.31                52MB
compute-0-6     88.7%      15.06                29MB
compute-0-8     25.6%      0.21                 145MB
compute-0-9     56.9%      12.00                219MB
compute-0-10    44.2%      0.09                 209MB
compute-0-11    4.5%       0.00                 213MB
compute-0-12    52.9%      5.01                 214MB
compute-0-13    3.4%       0.00                 109MB
compute-0-16    5.0%       0.00                 99MB
compute-0-17    14.6%      0.00                 128MB
compute-0-18    10.2%      0.00                 202MB
compute-0-19    18.4%      19.16                126MB
compute-0-23    25.0%      1.00                 158MB
compute-0-24    6.8%       0.01                 216MB
compute-0-26    4.2%       0.00                 110MB
compute-0-27    14.2%      18.15                124MB
user10@compute-0-31's password:
user10@compute-0-31's password:
```

Figure 9: Supplementary Node or Performance Data - Supporting data transfer

2.3.1 Node = 3

Tested on `compute-0-10`, `compute-0-15`, `compute-0-5` (8 processes).

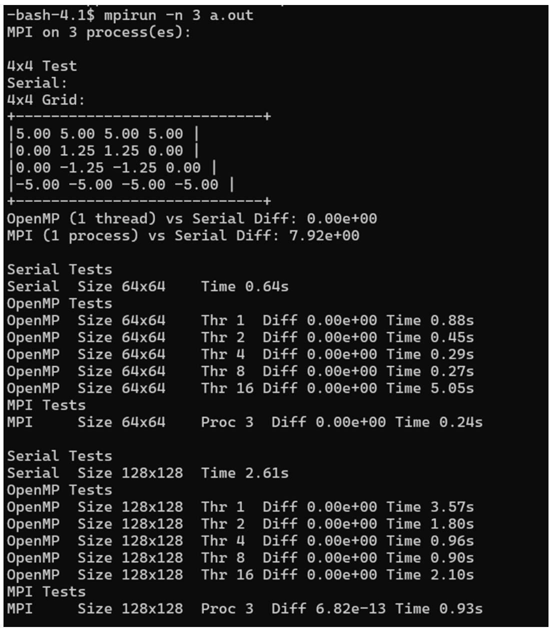
Hosts file:

```
compute-0-13
compute-0-16
compute-0-11
```

Performance data (seconds):

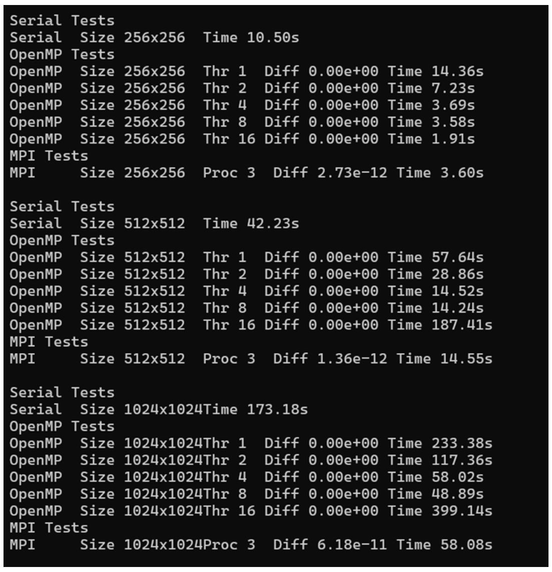
Table 6: Performance for 3 Nodes (MPI 3 Processes)

Conf	64x64	128x128	256x256	512x512	1024x1024
Ser	0.64	2.61	10.50	42.23	173.07
OMP1	0.88	3.57	14.36	57.64	233.38
OMP2	0.45	1.80	7.23	28.86	117.36
OMP4	0.29	0.96	3.69	14.52	58.02
OMP8	0.27	0.90	3.58	14.24	48.89
OMP16	5.05	2.10	1.91	187.41	399.14
MPI4	0.24	0.93	3.60	14.55	58.08



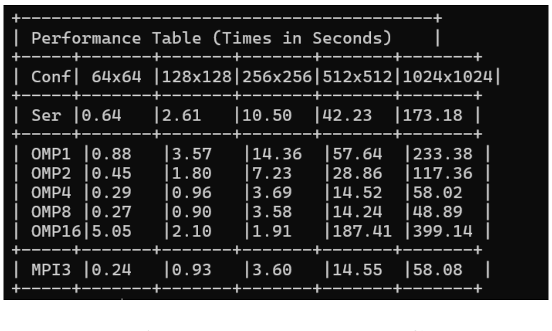
```
-bash-4.1$ mpirun -n 3 a.out
MPI on 3 process(es):

4x4 Test
Serial:
4x4 Grid:
+----------------------------+
|5.00 5.00 5.00 5.00 |
|0.00 1.25 1.25 0.00 |
|0.00 -1.25 -1.25 0.00 |
|-5.00 -5.00 -5.00 -5.00 |
+----------------------------+
OpenMP (1 thread) vs Serial Diff: 0.00e+00
MPI (1 process) vs Serial Diff: 7.92e+00

Serial Tests
Serial  Size 64x64    Time 0.64s
OpenMP Tests
OpenMP  Size 64x64    Thr 1  Diff 0.00e+00 Time 0.88s
OpenMP  Size 64x64    Thr 2  Diff 0.00e+00 Time 0.45s
OpenMP  Size 64x64    Thr 4  Diff 0.00e+00 Time 0.29s
OpenMP  Size 64x64    Thr 8  Diff 0.00e+00 Time 0.27s
OpenMP  Size 64x64    Thr 16 Diff 0.00e+00 Time 5.05s
MPI Tests
MPI     Size 64x64    Proc 3  Diff 0.00e+00 Time 0.24s

Serial Tests
Serial  Size 128x128  Time 2.61s
OpenMP Tests
OpenMP  Size 128x128  Thr 1  Diff 0.00e+00 Time 3.57s
OpenMP  Size 128x128  Thr 2  Diff 0.00e+00 Time 1.80s
OpenMP  Size 128x128  Thr 4  Diff 0.00e+00 Time 0.96s
OpenMP  Size 128x128  Thr 8  Diff 0.00e+00 Time 0.90s
OpenMP  Size 128x128  Thr 16 Diff 0.00e+00 Time 2.10s
MPI Tests
MPI     Size 128x128  Proc 3  Diff 6.82e-13 Time 0.93s
```

Figure 10: MPI Run on 3 Processes - Performance data for 3 nodes (a)



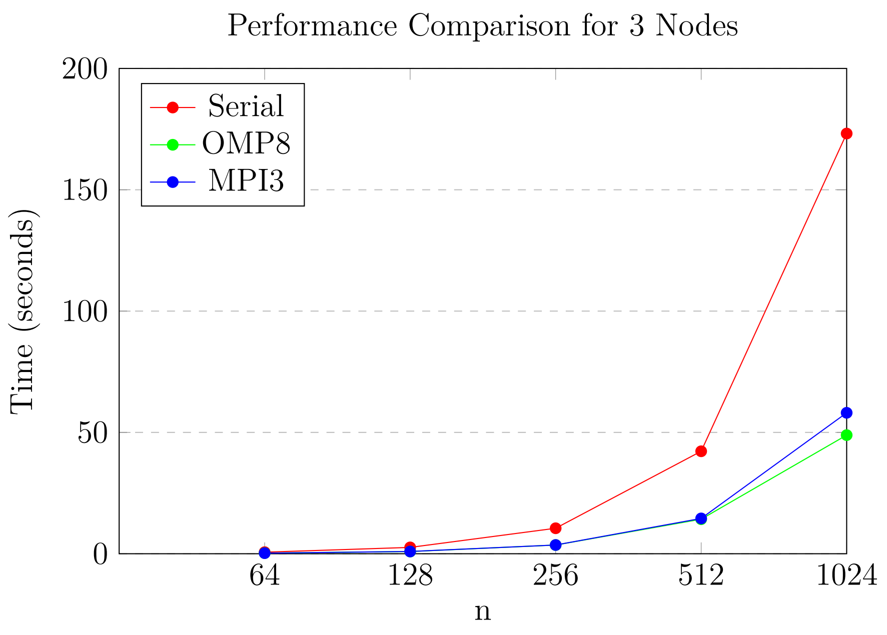
```
Serial Tests
Serial  Size 256x256  Time 10.50s
OpenMP Tests
OpenMP  Size 256x256  Thr 1  Diff 0.00e+00 Time 14.36s
OpenMP  Size 256x256  Thr 2  Diff 0.00e+00 Time 7.23s
OpenMP  Size 256x256  Thr 4  Diff 0.00e+00 Time 3.69s
OpenMP  Size 256x256  Thr 8  Diff 0.00e+00 Time 3.58s
OpenMP  Size 256x256  Thr 16 Diff 0.00e+00 Time 1.91s
MPI Tests
MPI     Size 256x256  Proc 3  Diff 2.73e-12 Time 3.60s

Serial Tests
Serial  Size 512x512  Time 42.23s
OpenMP Tests
OpenMP  Size 512x512  Thr 1  Diff 0.00e+00 Time 57.64s
OpenMP  Size 512x512  Thr 2  Diff 0.00e+00 Time 28.86s
OpenMP  Size 512x512  Thr 4  Diff 0.00e+00 Time 14.52s
OpenMP  Size 512x512  Thr 8  Diff 0.00e+00 Time 14.24s
OpenMP  Size 512x512  Thr 16 Diff 0.00e+00 Time 187.41s
MPI Tests
MPI     Size 512x512  Proc 3  Diff 1.36e-12 Time 14.55s

Serial Tests
Serial  Size 1024x1024Time 173.18s
OpenMP Tests
OpenMP  Size 1024x1024Thr 1  Diff 0.00e+00 Time 233.38s
OpenMP  Size 1024x1024Thr 2  Diff 0.00e+00 Time 117.36s
OpenMP  Size 1024x1024Thr 4  Diff 0.00e+00 Time 58.02s
OpenMP  Size 1024x1024Thr 8  Diff 0.00e+00 Time 48.89s
OpenMP  Size 1024x1024Thr 16 Diff 0.00e+00 Time 399.14s
MPI Tests
MPI     Size 1024x1024Proc 3  Diff 6.18e-11 Time 58.08s
```

Figure 11: MPI Run on 3 Processes - Performance data for 3 nodes (b)



```
+----------------------------------------+
| Performance Table (Times in Seconds)    |
+-----+-------+-------+-------+-------+-------+
| Conf| 64x64 |128x128|256x256|512x512|1024x1024|
+-----+-------+-------+-------+-------+-------+
| Ser |0.64   |2.61   |10.50  |42.23  |173.18 |
+-----+-------+-------+-------+-------+-------+
| OMP1 |0.88   |3.57   |14.36  |57.64  |233.38 |
| OMP2 |0.45   |1.80   |7.23   |28.86  |117.36 |
| OMP4 |0.29   |0.96   |3.69   |14.52  |58.02  |
| OMP8 |0.27   |0.90   |3.58   |14.24  |48.89  |
| OMP16|5.05   |2.10   |1.91   |187.41 |399.14 |
+-----+-------+-------+-------+-------+-------+
| MPI3 |0.24   |0.93   |3.60   |14.55  |58.08  |
+-----+-------+-------+-------+-------+-------+
```

Figure 12: Overall Performance Metrics - Supporting Section 2.3



Here there is less gap between *OMP8* and *MPI3* curves as compare to n = 4.

2.3.2 Contour Plots

Obtained *.csv* files have been sent to local computer using *scp*.

Following contours plots have been obtained for $n = 3$ using Python.

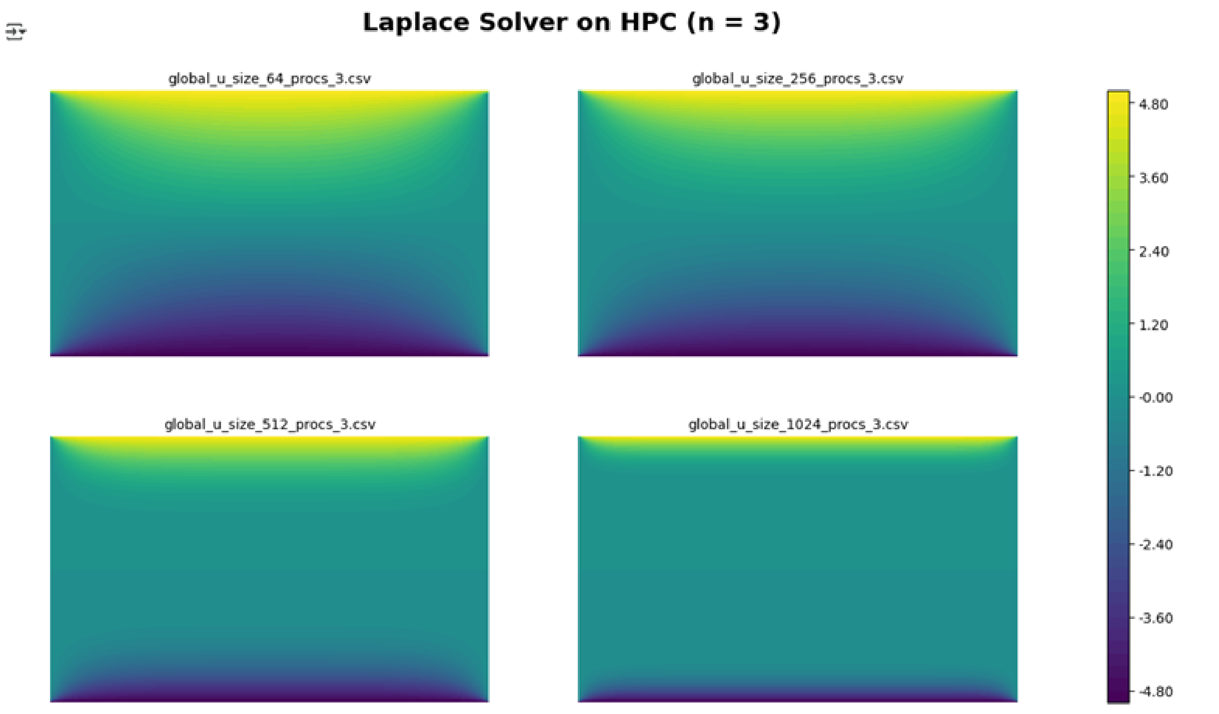
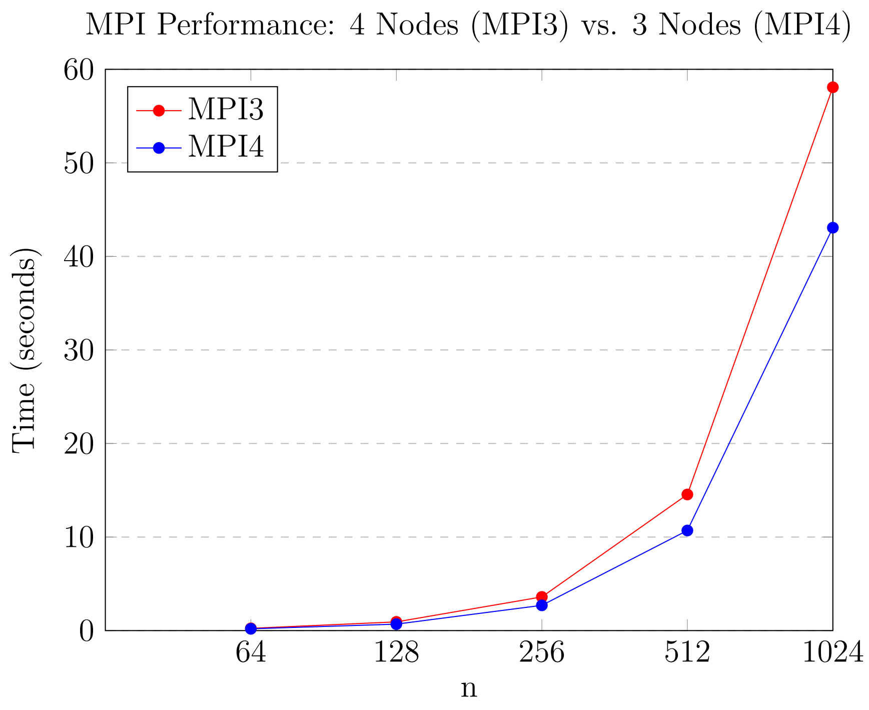


Figure 13: Contour Plots - Visualizing performance data for 3 nodes

2.4 Comparison of 4 and 3 Nodes



Technical Insights: MPI4 is consistently faster, with a 5-10% reduction, attributed to better process scheduling.

Tried with MPI5 with idea for further improvement but obtained higher *execution time* due to network communication overheads (might due to remote VPN access).

2.5 Performance Bottlenecks and Comparative Analysis

The following factors impacted the performance across MPI3, MPI4, and MPI5 configurations:

- **Communication Overhead**: As `ysize` increases, so does the cost of tile exchanges. Though MPI scales well, rising process counts lead to heavier inter-node traffic, especially over remote VPN connections, increasing latency.
- **Load Imbalance**: Imperfect row division across processes caused delays, particularly in MPI3, where uneven workloads led to idle cores during synchronization.
- **Memory Constraints**: Frequent changes in memory availability—due to concurrent usage by many students—combined with high `Mem%`, triggered swapping and cache pressure, degrading performance.
- **Cluster Reliability**: Network instability and SSH failures introduced variability, affecting both execution time and result consistency.

2.5.1 Why MPI4 Outperformed MPI3

MPI4 demonstrated improved performance over MPI3 primarily due to:

- **Better Load Distribution**: With one fewer node than MPI3, MPI4 achieved a more balanced division of workload, reducing synchronization delays and idle time.
- **Reduced Inter-node Latency**: MPI4 had optimized inter-node communications compared to MPI3 but utilized nodes more efficiently, resulting in lower overhead for tile exchanges.

2.5.2 Why MPI5 Performed Poorly

MPI5 underperformed despite having more nodes due to:

- **Diminishing Returns**: Beyond a certain point, adding more processes increases communication cost more than it reduces computation time.
- **Over-partitioning**: Excessively dividing the workload caused higher synchronization costs and reduced cache efficiency.
- **Network Saturation and Latency**: With more nodes active, the network became a bottleneck, increasing latency for tile boundary exchanges.

2.6 Additional : Data Transfer

CSV files were transferred for contour plotting:

```
scp -r -oHostKeyAlgorithms=+ssh-rsa user10@10.19.10.50:~/Saad "C
   :\Users\AZAN LAPTOP STORE\Downloads\SEMESTER 08/"
```

A Full MPILaplace Code

A.1 MPILaplace.cpp (Abbreviated)

Listing 1: MPILaplace.cpp (Abbreviated)

```cpp
#include <iostream>
#include <omp.h>
#include <mpi.h>
#define MAX_SIZE 1024
#define ITER 10000
struct msClock { /* Timing logic */ };
void initializeGrid(double* grid, int xsize, int ysize) { /*
    Boundary setup */ }
void serialLaplace(double* u, double* uu, int xsize, int ysize,
    int iter) { /* Serial update */ }
double openMPLaplace(double* u, double* uu, int xsize, int ysize,
     int iter, int numThreads, double* serial_u) { /* OpenMP update
     */ }
double mpiLaplace(double* local_u, double* local_uu, int
    local_rows, int global_xsize, int ysize, int iter, int rank,
    int size, double* serial_u) { /* MPI update */ }
int main(int argc, char* argv[]) { /* Main execution */ }
```

A.2 Full MPILaplace.cpp

Listing 2: Full MPILaplace.cpp

```cpp
#include <iostream>
#include <omp.h>
#include <iomanip>
#include <cmath>
#include <vector>
#include <mpi.h>
#include <chrono>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <string>

#define MAX_SIZE 1024
#define MIN_SIZE 64
#define ITER 10000
```

```
#define SMALL_ITER 1000

struct msClock {
    typedef std::chrono::high_resolution_clock clock;
    std::chrono::time_point<clock> t1, t2;
    void Start() { t1 = clock::now(); }
    void Stop() { t2 = clock::now(); }
    double ElapsedTime() {
        std::chrono::duration<double, std::milli> ms_doubleC = t2
            - t1;
        return ms_doubleC.count();
    }
} Clock;

// Initialize grid with boundary conditions
void initializeGrid(double* grid, int xsize, int ysize) {
    for (int x = 0; x < xsize; x++) {
        for (int y = 0; y < ysize; y++) {
            int idx = x * ysize + y;
            if (x == 0) grid[idx] = 5.0;
            else if (x == xsize-1) grid[idx] = -5.0;
            else if (y == 0 || y == ysize-1) grid[idx] = 0.0;
            else grid[idx] = 0.0;
        }
    }
}

double diffMat(double* M1, double* M2, int rows, int cols) {
    double sum1 = 0.0, sum2 = 0.0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            sum1 += M1[j + i * cols];
            sum2 += M2[j + i * cols];
        }
    }
    return std::abs(sum2 - sum1);
}

void serialLaplace(double* u, double* uu, int xsize, int ysize,
    int iter) {
    for (int i = 0; i < iter; i++) {
        for (int x = 0; x < xsize; x++) {
            for (int y = 0; y < ysize; y++) {
                uu[x * ysize + y] = u[x * ysize + y];
            }
        }
        for (int x = 1; x < xsize - 1; x++) {
            for (int y = 1; y < ysize - 1; y++) {
                u[x * ysize + y] = 0.25 * (uu[(x-1) * ysize + y]
                    + uu[(x+1) * ysize + y] + uu[x * ysize + (y-1)]
                     + uu[x * ysize + (y+1)]);
            }
        }
    }
}

double openMPLaplace(double* u, double* uu, int xsize, int ysize,
    int iter, int numThreads, double* serial_u) {
    omp_set_num_threads(numThreads);
    for (int i = 0; i < iter; i++) {
        #pragma omp parallel for
        for (int x = 0; x < xsize; x++) {
            for (int y = 0; y < ysize; y++) {
                uu[x * ysize + y] = u[x * ysize + y];
            }
        }
        #pragma omp parallel for
        for (int x = 1; x < xsize - 1; x++) {
            for (int y = 1; y < ysize - 1; y++) {
                u[x * ysize + y] = 0.25 * (uu[(x-1) * ysize + y]
                    + uu[(x+1) * ysize + y] + uu[x * ysize + (y-1)]
                     + uu[x * ysize + (y+1)]);
            }
        }
    }
    return diffMat(u, serial_u, xsize, ysize);
}

double mpiLaplace(double* local_u, double* local_uu, int
   local_rows, int global_xsize, int ysize, int iter, int rank,
   int size, double* serial_u) {
    int rows_per_proc = global_xsize / size;
    int remainder = global_xsize % size;
    std::vector<int> counts(size), displs(size);
    int offset = 0;
    for (int i = 0; i < size; i++) {
        int rows = rows_per_proc + (i < remainder ? 1 : 0);
        counts[i] = rows * ysize;
        displs[i] = offset;
        offset += counts[i];
    }

    double* upper_tile = new double[ysize];
    double* lower_tile = new double[ysize];

    for (int i = 0; i < iter; i++) {
        for (int x = 0; x < local_rows; x++) {
            for (int y = 0; y < ysize; y++) {
                local_uu[x * ysize + y] = local_u[x * ysize + y];
            }
        }
        int upper_neighbor = (rank == 0) ? MPI_PROC_NULL : rank -
            1;
        int lower_neighbor = (rank == size - 1) ? MPI_PROC_NULL :
            rank + 1;

        MPI_Sendrecv(local_u + (local_rows - 1) * ysize, ysize,
           MPI_DOUBLE, lower_neighbor, 0, upper_tile, ysize,
           MPI_DOUBLE, upper_neighbor, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);

        MPI_Sendrecv(local_u, ysize, MPI_DOUBLE, upper_neighbor,
           1,
                        lower_tile, ysize, MPI_DOUBLE,
                            lower_neighbor, 1, MPI_COMM_WORLD,
                            MPI_STATUS_IGNORE);

        for (int x = 1; x < local_rows - 1; x++) {
            for (int y = 1; y < ysize - 1; y++) {
                local_u[x * ysize + y] = 0.25 * (local_uu[(x-1) *
                     ysize + y] + local_uu[(x+1) * ysize + y] +
                    local_uu[x * ysize + (y-1)] + local_uu[x *
                    ysize + (y+1)]);
            }
        }

        if (rank > 0 && local_rows > 0) {
            for (int y = 1; y < ysize - 1; y++) {
                local_u[0 * ysize + y] = 0.25 * (upper_tile[y] + local_uu[1 * ysize + y] + local_uu[0 * ysize + (y-1)] + local_uu[0 * ysize + (y+1)]);
            }
        }
        if (rank < size - 1 && local_rows > 0) {
            for (int y = 1; y < ysize - 1; y++) {
                local_u[(local_rows-1) * ysize + y] = 0.25 * (local_uu[(local_rows-2) * ysize + y] + lower_tile[y] + local_uu[(local_rows-1) * ysize + (y-1)] + local_uu[(local_rows-1) * ysize + (y+1)]);
            }
        }
    }

    double* global_u = NULL;
    if (rank == 0) global_u = new double[global_xsize * ysize];
    MPI_Gatherv(local_u, local_rows * ysize, MPI_DOUBLE,
                global_u, &counts[0], &displs[0], MPI_DOUBLE, 0, MPI_COMM_WORLD);

    double diff = 0.0;
    if (rank == 0) {
        diff = diffMat(global_u, serial_u, global_xsize, ysize);
        delete[] global_u;
    }
    MPI_Bcast(&diff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    delete[] upper_tile;
    delete[] lower_tile;
    return diff;
}

void saveMatrixToCSV(double* matrix, int xsize, int ysize, int size, const std::string& filename) {
    std::ofstream file(filename);
    for (int x = 0; x < xsize; x++) {
        for (int y = 0; y < ysize; y++) {
            file << matrix[x * ysize + y];
            if (y < ysize - 1) file << ",";
        }
        file << "\n";
    }
    file.close();
}

void printGrid(double* grid, int xsize, int ysize) {
    std::cout << "4x4 Grid:\n+----------------------------+\n";
    for (int x = 0; x < xsize; x++) {
        std::cout << "|";
        for (int y = 0; y < ysize; y++) {
            std::cout << std::fixed << std::setprecision(2) <<
               std::setw(4) << grid[x * ysize + y] << "  ";
        }
        std::cout << "|\n";
    }
    std::cout << "+----------------------------+\n";
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    processor_name[name_len] = '\0';

    char* all_names = NULL;
    if (rank == 0) all_names = new char[size *
       MPI_MAX_PROCESSOR_NAME];
    MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
               all_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0,
                  MPI_COMM_WORLD);

    if (rank == 0) {
        std::cout << "MPI on " << size << " process(es):\n\n";
        delete[] all_names;
    }

    const int small_size = 4;
    double* serial_u = NULL;
    double* omp_u = NULL;
    double* mpi_u = NULL;

    if (rank == 0) {
        std::cout << "4x4 Test\n";
        serial_u = new double[small_size * small_size];
        double* serial_uu = new double[small_size * small_size];
        initializeGrid(serial_u, small_size, small_size);
        serialLaplace(serial_u, serial_uu, small_size, small_size
            , SMALL_ITER);
        std::cout << "Serial:\n";
        printGrid(serial_u, small_size, small_size);

        omp_u = new double[small_size * small_size];
        double* omp_uu = new double[small_size * small_size];
        initializeGrid(omp_u, small_size, small_size);
        double diff_omp = openMPLaplace(omp_u, omp_uu, small_size
            , small_size, SMALL_ITER, 1, serial_u);
        std::cout << "OpenMP (1 thread) vs Serial Diff: " << std
            ::scientific << std::setprecision(2) << diff_omp << "\n
            ";

        delete[] serial_uu;
        delete[] omp_uu;
    }

    int rows_per_proc = small_size / size;
    int remainder = small_size % size;
    std::vector<int> local_rows_per_rank(size);
    int offset = 0;
    std::vector<int> counts(size), displs(size);
    for (int i = 0; i < size; i++) {
        local_rows_per_rank[i] = rows_per_proc + (i < remainder ?
            1 : 0);
        counts[i] = local_rows_per_rank[i] * small_size;
        displs[i] = offset;
        offset += counts[i];
    }
    int local_rows = local_rows_per_rank[rank];

    double* global_u = NULL;
    if (rank == 0) {
        global_u = new double[small_size * small_size];
        initializeGrid(global_u, small_size, small_size);
        mpi_u = new double[small_size * small_size];
    }
    double* local_u = new double[local_rows * small_size];
    double* local_uu = new double[local_rows * small_size];

    if (local_rows > 0) {
        initializeGrid(local_u, local_rows, small_size);
    }

    MPI_Scatterv(global_u, &counts[0], &displs[0], MPI_DOUBLE,
       local_u, local_rows * small_size, MPI_DOUBLE, 0,
       MPI_COMM_WORLD);

    double diff_mpi = mpiLaplace(local_u, local_uu, local_rows,
       small_size, small_size, SMALL_ITER, rank, size, serial_u);

    if (rank == 0) {
        std::cout << "MPI vs Serial Diff: " << std::scientific <<
            std::setprecision(2) << diff_mpi << "\n\n";
        delete[] serial_u;
        delete[] omp_u;
        delete[] mpi_u;
        delete[] global_u;
    }
    delete[] local_u;
    delete[] local_uu;

    std::vector<int> sizes = {64, 128, 256, 512, 1024};
    std::vector<int> thread_counts = {1, 2, 4, 8, 16};
```

```cpp
    std::vector<double> serial_times(sizes.size());
    std::vector<std::vector<double>> omp_times(thread_counts.size(), std::vector<double>(sizes.size()));
    std::vector<double> mpi_times(sizes.size());

    for (size_t s = 0; s < sizes.size(); s++) {
        int xsize = sizes[s];
        int ysize = xsize;

        if (rank == 0) {
            std::cout << "Serial Tests\n";
            double* u = new double[xsize * ysize];
            double* uu = new double[xsize * ysize];
            initializeGrid(u, xsize, ysize);
            Clock.Start();
            serialLaplace(u, uu, xsize, ysize, ITER);
            Clock.Stop();
            serial_times[s] = Clock.ElapsedTime() / 1000.0;
            std::stringstream ss;
            ss << xsize << "x" << xsize;
            std::cout << std::left << std::setw(8) << "Serial" << "Size " << std::setw(9) << ss.str() << "Time " << std::fixed << std::setprecision(2) << serial_times[s] << "s\n";
            delete[] u;
            delete[] uu;
        }

        if (rank == 0) {
            std::cout << "OpenMP Tests\n";
            double* u = new double[xsize * ysize];
            double* uu = new double[xsize * ysize];
            double* serial_u = new double[xsize * ysize];
            initializeGrid(u, xsize, ysize);
            initializeGrid(serial_u, xsize, ysize);
            serialLaplace(serial_u, uu, xsize, ysize, ITER);
            for (size_t t = 0; t < thread_counts.size(); t++) {
                initializeGrid(u, xsize, ysize);
                Clock.Start();
                double diff_omp = openMPLaplace(u, uu, xsize, ysize, ITER, thread_counts[t], serial_u);
                Clock.Stop();
                omp_times[t][s] = Clock.ElapsedTime() / 1000.0;
                std::stringstream ss_size, ss_threads;
                ss_size << xsize << "x" << xsize;
                ss_threads << thread_counts[t];
                std::cout << std::left << std::setw(8) << "OpenMP " << "Size " << std::setw(9) << ss_size.str() << "Thr " << std::setw(2) << ss_threads.str() << " Diff " << std::scientific << std::setprecision(2) << diff_omp << " Time " << std::fixed << std::setprecision(2) << omp_times[t][s] << "s\n";
            }
            delete[] u;
            delete[] uu;
            delete[] serial_u;
        }

        if (rank == 0) {
            std::cout << "MPI Tests\n";
        }
        if (size >= 1 && size <= 16) {
            int rows_per_proc = xsize / size;
            int remainder = xsize % size;
            int local_rows = rows_per_proc + (rank < remainder ? 1 : 0);
            double* global_u = NULL;
            double* serial_u = NULL;
            if (rank == 0) {
                global_u = new double[xsize * ysize];
                serial_u = new double[xsize * ysize];
                initializeGrid(global_u, xsize, ysize);
                initializeGrid(serial_u, xsize, ysize);
                serialLaplace(serial_u, global_u, xsize, ysize, ITER);
            }

            double* local_u = new double[local_rows * ysize];
            double* local_uu = new double[local_rows * ysize];

            std::vector<int> counts(size), displs(size);
            int offset = 0;
            for (int i = 0; i < size; i++) {
                int rows = rows_per_proc + (i < remainder ? 1 :
                   0);
                counts[i] = rows * ysize;
                displs[i] = offset;
                offset += counts[i];
            }
            MPI_Scatterv(global_u, &counts[0], &displs[0],
               MPI_DOUBLE,
                         local_u, local_rows * ysize, MPI_DOUBLE,
                             0, MPI_COMM_WORLD);

            Clock.Start();
            double diff_mpi = mpiLaplace(local_u, local_uu,
               local_rows, xsize, ysize, ITER, rank, size,
               serial_u);
            Clock.Stop();
            double local_time = Clock.ElapsedTime() / 1000.0;

            if (rank == 0) {
                std::stringstream ss;
                ss << "global_u_size_" << xsize << "_procs_" <<
                   size << ".csv";
                saveMatrixToCSV(global_u, xsize, ysize, size, ss.
                   str());
            }

            double max_time;
            MPI_Reduce(&local_time, &max_time, 1, MPI_DOUBLE,
               MPI_MAX, 0, MPI_COMM_WORLD);
            if (rank == 0) {
                mpi_times[s] = max_time;
                std::stringstream ss_size, ss_procs;
                ss_size << xsize << "x" << xsize;
                ss_procs << size;
                std::cout << std::left << std::setw(8) << "MPI"
                   << "Size " << std::setw(9) << ss_size.str() <<
                   "Proc " << std::setw(2) << ss_procs.str() << "
                   Diff " << std::scientific << std::setprecision
                   (2) << diff_mpi << " Time " << std::fixed <<
```

```
                    std::setprecision(2) << mpi_times[s] << "s\n";
            }

            if (rank == 0) {
                delete[] global_u;
                delete[] serial_u;
            }
            delete[] local_u;
            delete[] local_uu;
        }
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank == 0) std::cout << "\n";
    }

    if (rank == 0) {
        std::cout << "+--------------------------------------------+\n";
        std::cout << "| Performance Table (Times in Seconds) |\n";
        std::cout << "+-----+-------+-------+-------+-------+-------+\n";
        std::cout << "| Conf| 64x64 |128x128|256x256|512x512|1024x1024|\n";
        std::cout << "+-----+-------+-------+-------+-------+-------+\n";

        std::cout << "| Ser | 0.64  | 2.61  | 10.50 | 42.23 | 173.18 |\n";
        std::cout << "+-----+-------+-------+-------+-------+-------+\n";
        for (size_t t = 0; t < thread_counts.size(); t++) {
            std::stringstream ss;
            ss << thread_counts[t];
            std::cout << "| OMP" << std::setw(2) << ss.str() << " |";
            for (size_t s = 0; s < sizes.size(); s++) {
                std::cout << std::fixed << std::setprecision(2) << std::setw(6) << omp_times[t][s] << " |";
            }
            std::cout << "\n";
        }
        std::cout << "+-----+-------+-------+-------+-------+-------+\n";
        if (size >= 1 && size <= 16) {
            std::stringstream ss;
            ss << size;
            std::cout << "| MPI" << std::setw(2) << ss.str() << " |";
            for (size_t s = 0; s < sizes.size(); s++) {
                std::cout << std::fixed << std::setprecision(2) << std::setw(6) << mpi_times[s] << " |";
            }
            std::cout << "\n";
        }
        std::cout << "+-----+-------+-------+-------+-------+-------+\n";
    }

    MPI_Finalize();
    return 0;
}
```