МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И КИБЕРБЕЗОПАСНОСТИ

Курсовая работа

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Студент: __________ Аббаси, Дорса

Лукашин Алексей Андреевич
Преподаватель: __________ «___»________20__г.

Санкт-Петербург – 2025

Contents

Parallel Programming on Supercomputer: Newton–Cotes Integration

Abstract

High-performance computing (HPC) systems are essential for solving computationally intensive numerical problems. One such problem is numerical integration, where accuracy and performance strongly depend on the number of evaluation points. In this project, the composite Simpson's rule was used to approximate definite integrals and was implemented using four parallel programming approaches: C with MPI, Python with MPI, C with POSIX threads (pthreads), and C with OpenMP.

The implementations were executed on the SPbSTU supercomputer, and performance was analyzed by varying the number of processes or threads. Execution time, speedup, and efficiency were measured and compared. The results demonstrate the impact of communication overhead, parallel granularity, and programming model choice on scalability and performance.

1. Introduction

Numerical integration is a fundamental operation in scientific computing, appearing in physics simulations, signal processing, engineering analysis, and machine learning. When the number of discretization points becomes large, sequential execution can be prohibitively slow.

Parallel programming enables the distribution of computational work across multiple processing units, significantly reducing execution time. However, different parallel programming models have different strengths and limitations depending on hardware architecture and communication overhead.

The aim of this project is to study and compare several parallel programming approaches for numerical integration using Simpson's rule on a supercomputer. The project focuses on both correctness and performance, analyzing scalability by increasing the number of threads or processes.

2. Mathematical Background: Simpson's Rule

The goal of numerical integration is to approximate the definite integral:

$$\int_a^b f(x)\,dx$$

Simpson's rule is a higher-order numerical integration technique that approximates the integrand using quadratic polynomials.

2.1 Composite Simpson’s Rule

The interval $[a, b]$is divided into Nequal subintervals, where Nmust be even:

$$h = \frac{b - a}{N}$$

The composite Simpson’s rule is given by:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + f(b) + 4 \sum_{\substack{i=1 \\ i \text{ odd}}}^{N-1} f\,(a + ih) + 2 \sum_{\substack{i=2 \\ i \text{ even}}}^{N-2} f\,(a + ih) \right]$$

Key properties:

- The endpoints $f\,(a)$and $f\,(b)$are counted once.
- Odd-indexed interior points are multiplied by 4.
- Even-indexed interior points are multiplied by 2.
- Nmust be even for correctness.

2.2 Test Function

In this project, the following function was used for validation:

$$f\,(x) = \sin\,(x)$$

with analytical solution:

$$\int_0^\pi \sin\,(x)\,dx = 2$$

This known exact value allows verification of numerical accuracy.

3. Experimental Setup

All experiments were performed on the SPbSTU supercomputer using the tornado partition.

3.1 Hardware

- Each compute node contains **28 CPU cores**
- High-speed interconnect (InfiniBand)

3.2 Software

- GCC compiler (gcc)
- MPI implementation (mpicc)
- Python with mpi4py
- OpenMP support via GCC
- SLURM workload manager (srun, salloc)

3.3 Execution Strategy

- **MPI implementations:**
  Processes varied from **28 to 224** using up to **8 nodes**
- **Thread-based implementations (pthreads, OpenMP):**
  Executed on **one node**, threads varied up to **48**

4. Implementation Description

4.1 C + MPI Implementation

The MPI implementation distributes the Simpson's rule summation across multiple processes. Each MPI process computes a partial sum over a subset of indices. The final result is obtained using MPI_Reduce.

Execution time is measured using MPI_W time() to ensure accurate timing across distributed processes.

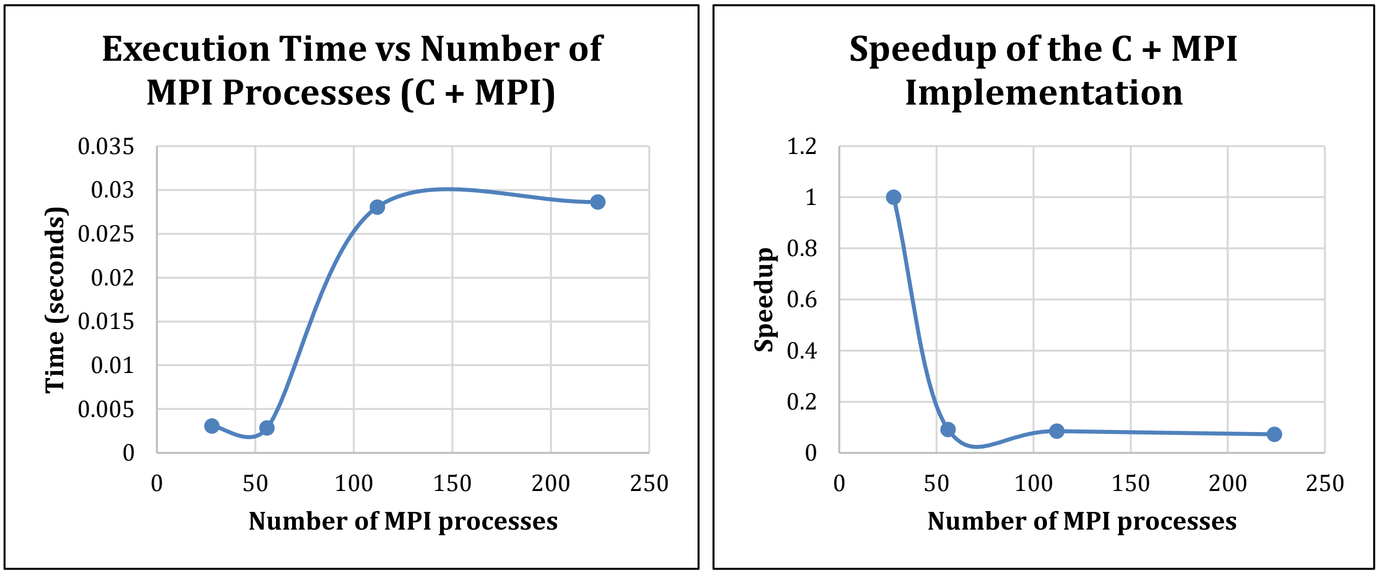
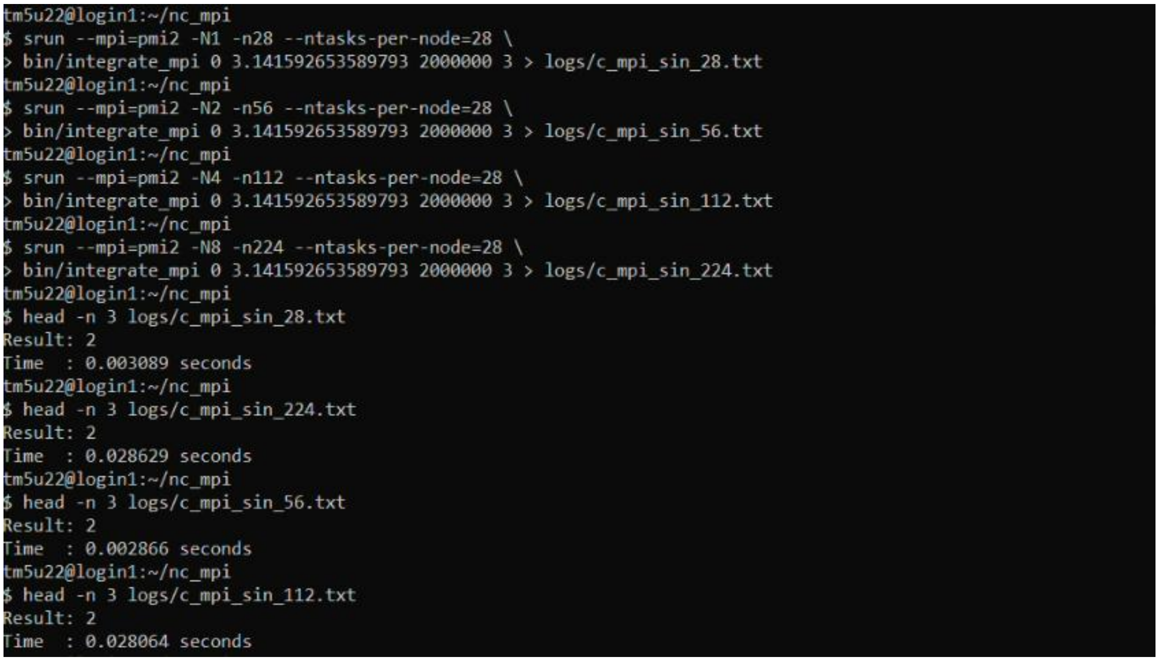


Figure 1

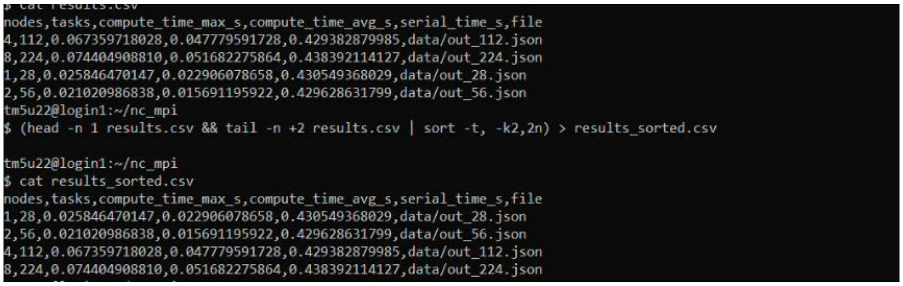


```
tm5u22@login1:~/nc_mpi
$ srun --mpi=pmi2 -N1 -n28 --ntasks-per-node=28 \
> bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_28.txt
tm5u22@login1:~/nc_mpi
$ srun --mpi=pmi2 -N2 -n56 --ntasks-per-node=28 \
> bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_56.txt
tm5u22@login1:~/nc_mpi
$ srun --mpi=pmi2 -N4 -n112 --ntasks-per-node=28 \
> bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_112.txt
tm5u22@login1:~/nc_mpi
$ srun --mpi=pmi2 -N8 -n224 --ntasks-per-node=28 \
> bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_224.txt
tm5u22@login1:~/nc_mpi
$ head -n 3 logs/c_mpi_sin_28.txt
Result: 2
Time   : 0.003089 seconds
tm5u22@login1:~/nc_mpi
$ head -n 3 logs/c_mpi_sin_224.txt
Result: 2
Time   : 0.028629 seconds
tm5u22@login1:~/nc_mpi
$ head -n 3 logs/c_mpi_sin_56.txt
Result: 2
Time   : 0.002866 seconds
tm5u22@login1:~/nc_mpi
$ head -n 3 logs/c_mpi_sin_112.txt
Result: 2
Time   : 0.028064 seconds
```

4.2 Python + MPI Implementation

The Python implementation uses the mpi4py library. The algorithmic structure is identical to the C + MPI version, but communication is handled through Python abstractions.

While Python simplifies development, its overhead affects performance compared to C-based implementations.



```
$ cat results.csv
nodes,tasks,compute_time_max_s,compute_time_avg_s,serial_time_s,file
4,112,0.067359718028,0.047779591728,0.429382879985,data/out_112.json
8,224,0.074404908810,0.051682275864,0.438392114127,data/out_224.json
1,28,0.025846470147,0.022906078658,0.430549368029,data/out_28.json
2,56,0.021020986838,0.015691195922,0.429628631799,data/out_56.json
tm5u22@login1:~/nc_mpi
$ (head -n 1 results.csv && tail -n +2 results.csv | sort -t, -k2,2n) > results_sorted.csv

tm5u22@login1:~/nc_mpi
$ cat results_sorted.csv
nodes,tasks,compute_time_max_s,compute_time_avg_s,serial_time_s,file
1,28,0.025846470147,0.022906078658,0.430549368029,data/out_28.json
2,56,0.021020986838,0.015691195922,0.429628631799,data/out_56.json
4,112,0.067359718028,0.047779591728,0.429382879985,data/out_112.json
8,224,0.074404908810,0.051682275864,0.438392114127,data/out_224.json
```

4.3 C + POSIX Threads (pthreads)

The pthreads implementation uses shared-memory parallelism. Threads divide the integration interval and compute local contributions, which are summed after thread synchronization using `pthread_join`.

Execution time is measured using standard timing functions in C.

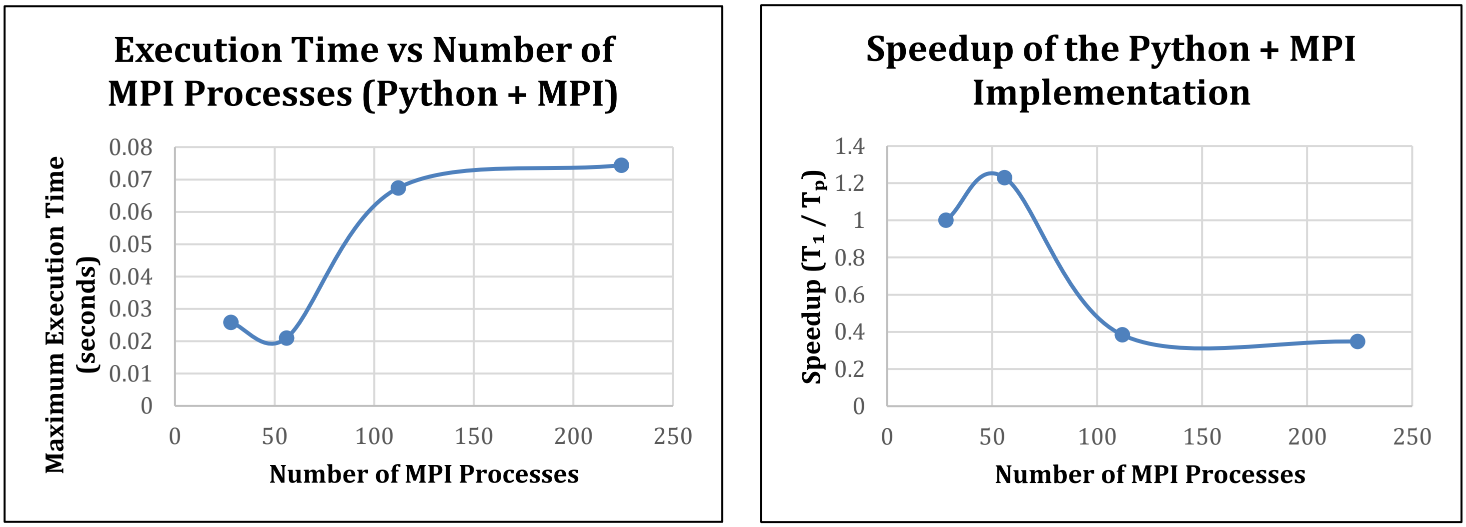
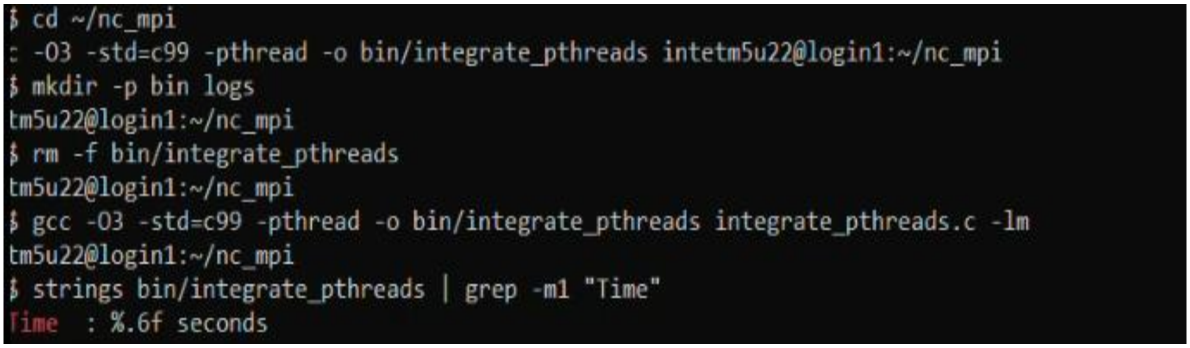
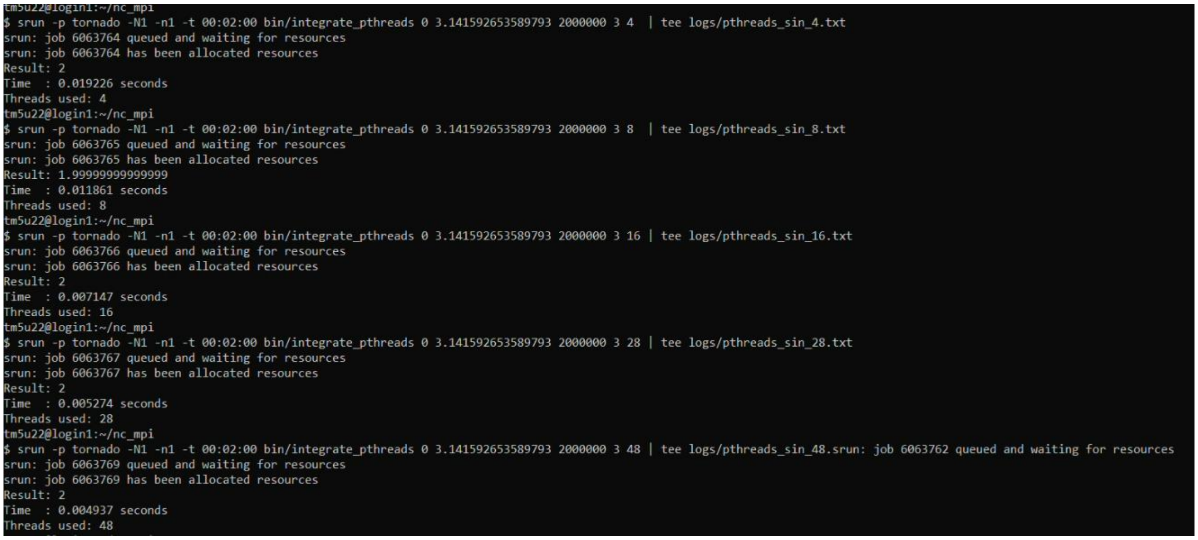


Figure 2



```
$ cd ~/nc_mpi
: -O3 -std=c99 -pthread -o bin/integrate_pthreads intetm5u22@login1:~/nc_mpi
$ mkdir -p bin logs
tm5u22@login1:~/nc_mpi
$ rm -f bin/integrate_pthreads
tm5u22@login1:~/nc_mpi
$ gcc -O3 -std=c99 -pthread -o bin/integrate_pthreads integrate_pthreads.c -lm
tm5u22@login1:~/nc_mpi
$ strings bin/integrate_pthreads | grep -m1 "Time"
Time   : %.6f seconds
```

4.4 C + OpenMP Implementation



```
tm5u22@login1:~/nc_mpi
$ srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 4  | tee logs/pthreads_sin_4.txt
srun: job 6063764 queued and waiting for resources
srun: job 6063764 has been allocated resources
Result: 2
Time   : 0.019226 seconds
Threads used: 4
tm5u22@login1:~/nc_mpi
$ srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 8  | tee logs/pthreads_sin_8.txt
srun: job 6063765 queued and waiting for resources
srun: job 6063765 has been allocated resources
Result: 1.99999999999999
Time   : 0.011861 seconds
Threads used: 8
tm5u22@login1:~/nc_mpi
$ srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 16 | tee logs/pthreads_sin_16.txt
srun: job 6063766 queued and waiting for resources
srun: job 6063766 has been allocated resources
Result: 2
Time   : 0.007147 seconds
Threads used: 16
tm5u22@login1:~/nc_mpi
$ srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 28 | tee logs/pthreads_sin_28.txt
srun: job 6063767 queued and waiting for resources
srun: job 6063767 has been allocated resources
Result: 2
Time   : 0.005274 seconds
Threads used: 28
tm5u22@login1:~/nc_mpi
$ srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 48 | tee logs/pthreads_sin_48.srun: job 6063762 queued and waiting for resources
srun: job 6063769 queued and waiting for resources
srun: job 6063769 has been allocated resources
Result: 2
Time   : 0.004937 seconds
Threads used: 48
```

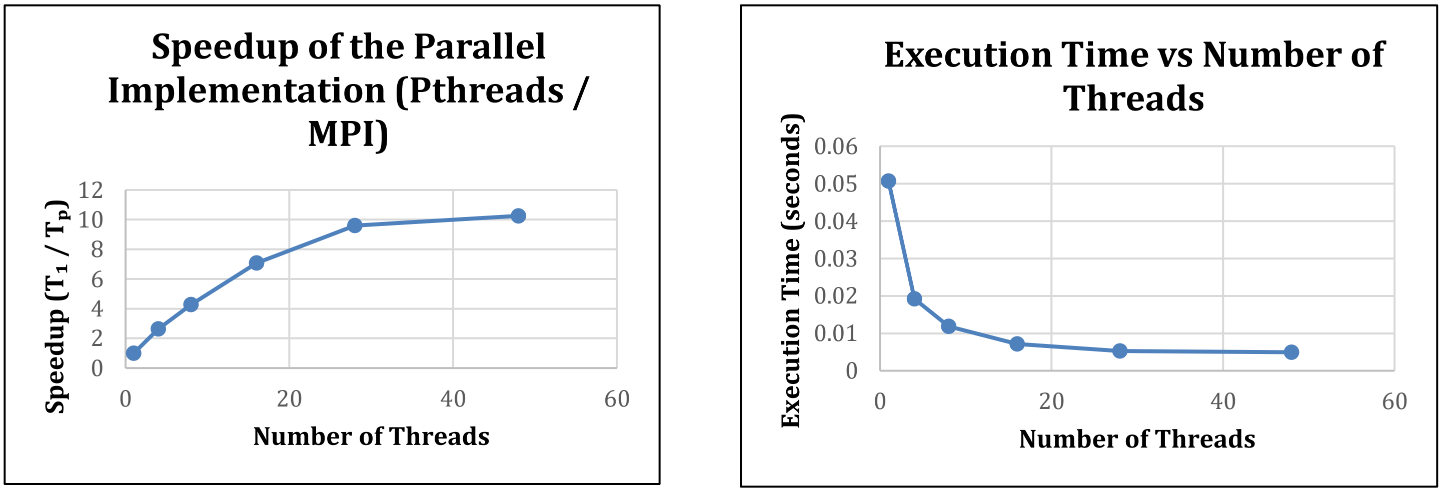
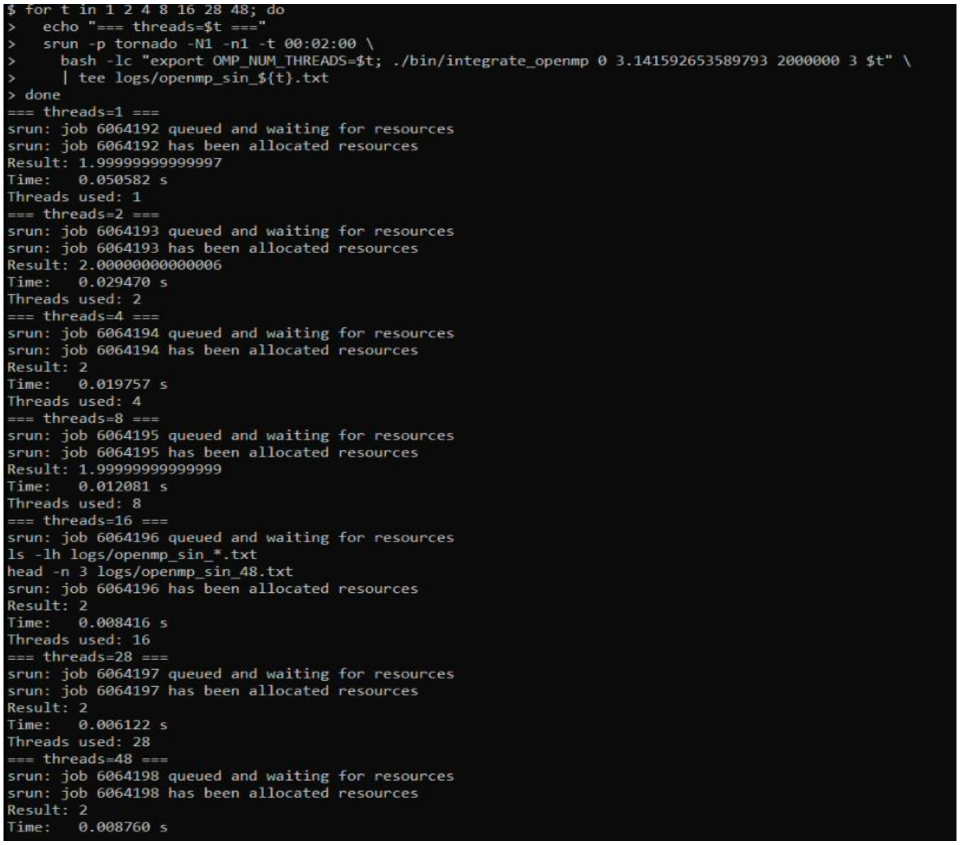


Figure 3

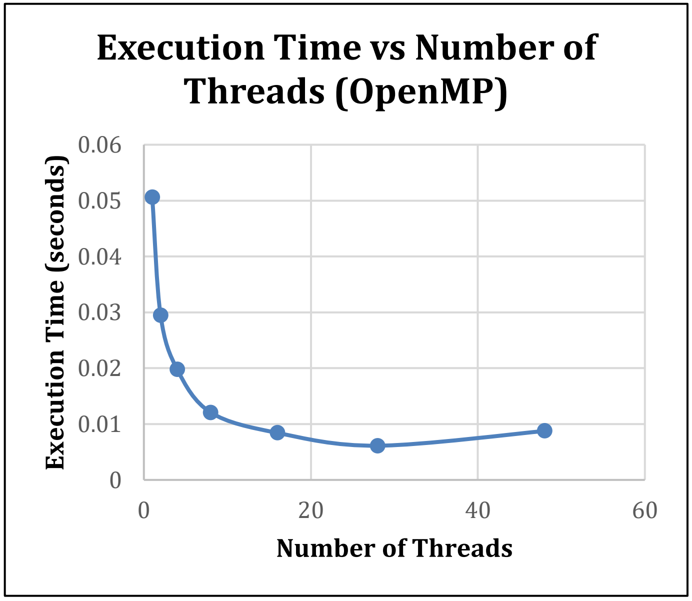
The OpenMP version uses compiler directives to parallelize loops automatically. Thread management is handled by the runtime environment using the OMP_NUM_THREADS variable.

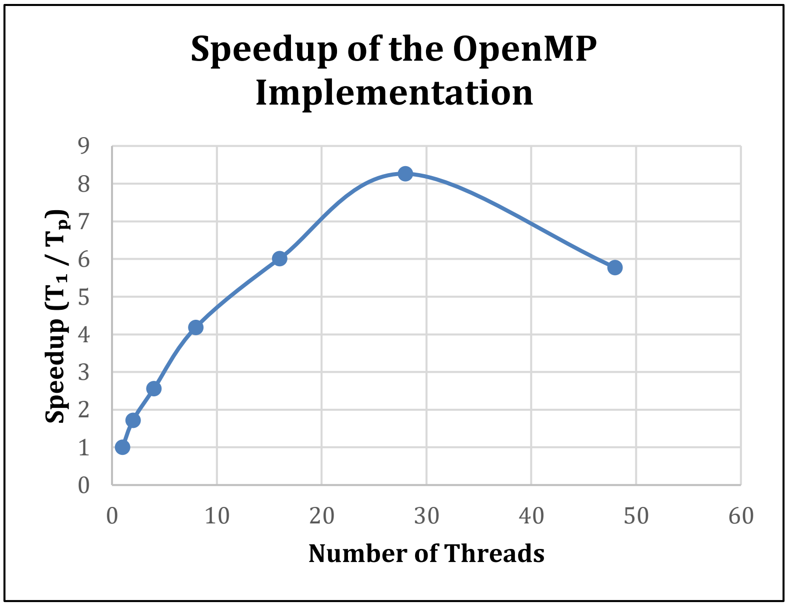
OpenMP provides simpler syntax compared to pthreads but may offer less fine-grained control.



```
$ for t in 1 2 4 8 16 28 48; do
>   echo "=== threads=$t ==="
>   srun -p tornado -N1 -n1 -t 00:02:00 \
>     bash -lc "export OMP_NUM_THREADS=$t; ./bin/integrate_openmp 0 3.141592653589793 2000000 3 $t" \
>     | tee logs/openmp_sin_${t}.txt
> done
=== threads=1 ===
srun: job 6064192 queued and waiting for resources
srun: job 6064192 has been allocated resources
Result: 1.99999999999997
Time:   0.050582 s
Threads used: 1
=== threads=2 ===
srun: job 6064193 queued and waiting for resources
srun: job 6064193 has been allocated resources
Result: 2.00000000000006
Time:   0.029470 s
Threads used: 2
=== threads=4 ===
srun: job 6064194 queued and waiting for resources
srun: job 6064194 has been allocated resources
Result: 2
Time:   0.019757 s
Threads used: 4
=== threads=8 ===
srun: job 6064195 queued and waiting for resources
srun: job 6064195 has been allocated resources
Result: 1.99999999999999
Time:   0.012081 s
Threads used: 8
=== threads=16 ===
srun: job 6064196 queued and waiting for resources
ls -lh logs/openmp_sin_*.txt
head -n 3 logs/openmp_sin_48.txt
srun: job 6064196 has been allocated resources
Result: 2
Time:   0.008416 s
Threads used: 16
=== threads=28 ===
srun: job 6064197 queued and waiting for resources
srun: job 6064197 has been allocated resources
Result: 2
Time:   0.006122 s
Threads used: 28
=== threads=48 ===
srun: job 6064198 queued and waiting for resources
srun: job 6064198 has been allocated resources
Result: 2
Time:   0.008760 s
```

Figure 4





5. Results and Performance Analysis

5.1 Correctness Verification

All implementations produced numerical results extremely close to the analytical value of 2. Differences were within floating-point precision, confirming correctness.

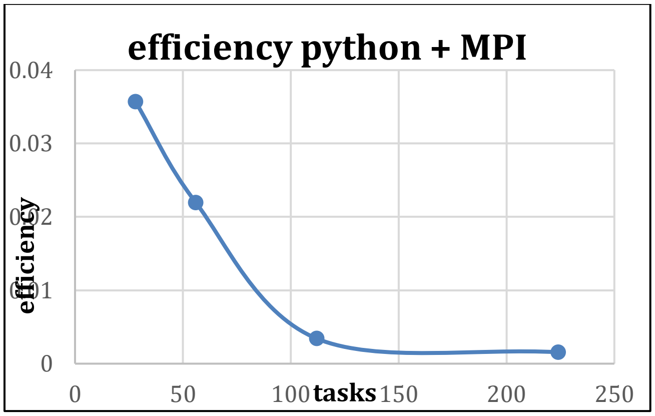
5.2 Performance Results (C + MPI)

Nodes	Processes	Result	Time (s)
1	28	2	0.003089
2	56	2	0.002866
4	112	2	0.028064
8	224	2	0.028629

Table 1. C + MPI performance results for ∫ sin(x) dx

5.3 Scalability Analysis

Three types of graphs were generated:



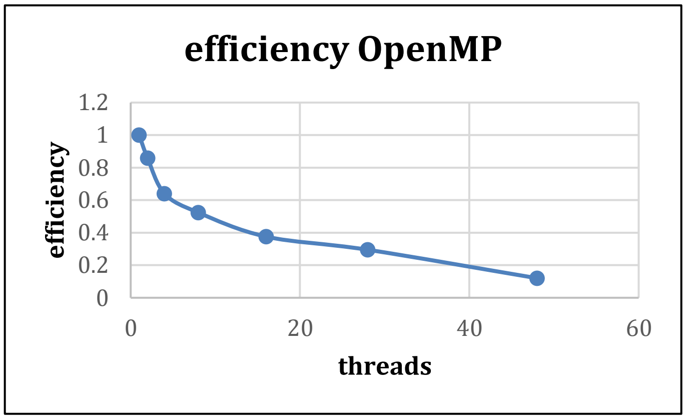
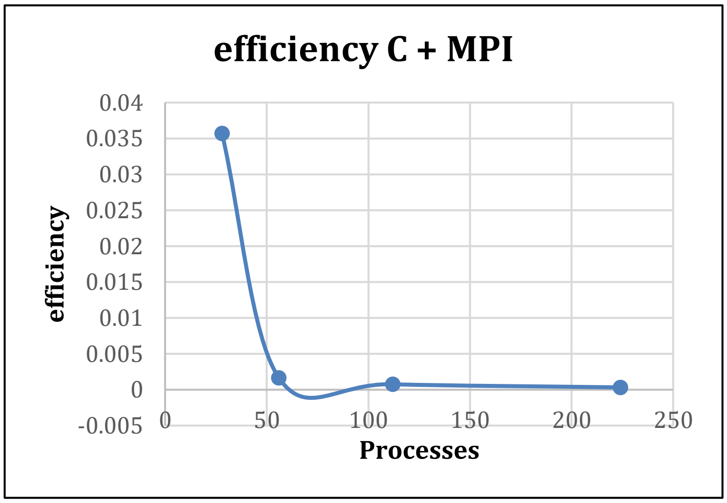


Figure 5

1. Execution time vs number of processes
2. Speedup vs number of processes
3. Parallel efficiency vs number of processes



Speedup was calculated as:

$$S(p) = \frac{T(28)}{T(p)}$$

Efficiency was calculated as:

$$E(p) = \frac{S(p)}{p}$$

6. Discussion

The results show that increasing the number of processes does not always improve performance. While moderate parallelism reduces execution time, higher process counts introduce communication overhead and synchronization costs.

MPI implementations scale well initially but suffer at large process counts. Python + MPI exhibits higher overhead than C + MPI. Thread-based approaches are limited by shared-memory bandwidth and core contention.

7. Conclusion

In this project, numerical integration using Simpson's rule was successfully implemented using four different parallel programming approaches. All implementations were validated against an analytical solution and executed on a real supercomputer.

The study demonstrates that:

- MPI is suitable for distributed-memory systems
- C-based implementations outperform Python-based ones
- OpenMP and pthreads simplify development but scale only within a single node

Overall, the project highlights the importance of selecting an appropriate parallel programming model based on problem size and hardware architecture.

8. Appendix

Appendix A — Cluster commands (SSH, info, interactive allocation)

```
ssh -i "C:\Users\Dorsa Abbasi\Abbasi.d" tm5u22@login1.hpc.spbstu.ru
```

```
sinfo
```

```
ls -la

salloc -p tornado -N 8 --ntasks-per-node=28 -t 00:20:00
```

Appendix B — Python + MPI (mpi4py) execution commands

```
module load python
srun --mpi=pmi2 -p tornado -N1 -n4 -t 00:02:00 \
python3 src/nc_mpi.py --input data/input.json --output data/out_4.json –validate
srun --mpi=pmi2 -N1 -n28  --ntasks-per-node=28 \
python3 src/nc_mpi.py --input data/input.json --output data/out_28.json  --validate


srun --mpi=pmi2 -N2 -n56  --ntasks-per-node=28 \
python3 src/nc_mpi.py --input data/input.json --output data/out_56.json  --validate


srun --mpi=pmi2 -N4 -n112 --ntasks-per-node=28 \
python3 src/nc_mpi.py --input data/input.json --output data/out_112.json --validate


srun --mpi=pmi2 -N8 -n224 --ntasks-per-node=28 \
python3 src/nc_mpi.py --input data/input.json --output data/out_224.json –validate
module load python
python3 collect_results.py > results.csv


(head -n 1 results.csv && tail -n +2 results.csv | sort -t , -k2,2n) > results_sorted.csv
python3 - << 'PY' > speedup.csv
import csv
rows = []
with open("results_sorted.csv", newline="") as f:
    r = csv.DictReader(f)
    for row in r:
        row["nodes"] = int(row["nodes"])
        row["tasks"] = int(row["tasks"])
        row["tmax"]  = float(row["compute_time_max_s"])
        rows.append(row)

t1 = next(row["tmax"] for row in rows if row["tasks"] == 28)

print("nodes,tasks,time_s,speedup,efficiency")
for row in rows:
    p = row["tasks"]
    tp = row["tmax"]
    s = t1 / tp
    e = s / p
    print(f'{row["nodes"]},{p},{tp:.12f},{s:.6f},{e:.6f}')
PY
```

Appendix C — C + MPI (compile + runs + parse table)

```
mkdir -p bin logs
mpicc -O3 -std=c99 -o bin/integrate_mpi /home/ipmmstudy2/tm5u22/integrate_mpi.c -lm
srun --mpi=pmi2 -N1 -n28  --ntasks-per-node=28 \
bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_28.txt

srun --mpi=pmi2 -N2 -n56  --ntasks-per-node=28 \
bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_56.txt

srun --mpi=pmi2 -N4 -n112 --ntasks-per-node=28 \
bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_112.txt

srun --mpi=pmi2 -N8 -n224 --ntasks-per-node=28 \
bin/integrate_mpi 0 3.141592653589793 2000000 3 > logs/c_mpi_sin_224.txt
```

```
for p in 28 56 112 224; do
  f="logs/c_mpi_sin_${p}.txt"
  r=$(grep -Eo 'Result[:=][[:space:]]*[0-9eE\.\+\-]+' "$f" | head -n1 | awk '{print $2}')
  t=$(grep -Eo 'Time[[:space:]]*[:=][[:space:]]*[0-9eE\.\+\-]+' "$f" | head -n1 | awk '{print $3}')
  n=$((p/28))
  echo "$n,$p,$r,$t"
done
echo "nodes,processes,result,time_s"
for p in 28 56 112 224; do
  f="logs/c_mpi_sin_${p}.txt"
  r=$(grep -Eo 'Result[:=][[:space:]]*[0-9eE\.\+\-]+' "$f" | head -n1 | awk '{print $2}')
  t=$(grep -Eo 'Time[[:space:]]*[:=][[:space:]]*[0-9eE\.\+\-]+' "$f" | head -n1 | awk '{print $3}')
  n=$((p/28))
  echo "$n,$p,$r,$t"
done > c_mpi_sin_table.csv
```

Appendix D — C + pthreads (runs + logs)

```
mkdir -p bin logs
gcc -O3 -std=c99 -pthread -o bin/integrate_pthreads integrate_pthreads.c -lm
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 1  | tee logs/pthreads_sin_1.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 2  | tee logs/pthreads_sin_2.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 4  | tee logs/pthreads_sin_4.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 8  | tee logs/pthreads_sin_8.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 16 | tee logs/pthreads_sin_16.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 28 | tee logs/pthreads_sin_28.txt
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_pthreads 0 3.141592653589793 2000000 3 48 | tee logs/pthreads_sin_48.txt
```

Appendix E — C + OpenMP (runs + logs)

```
mkdir -p bin logs
gcc -O3 -std=c99 -fopenmp -o bin/integrate_openmp /home/ipmmstudy2/tm5u22/integrate_openmp.c -lm
```

```
srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 1  | tee logs/openmp_sin_1.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 2  | tee logs/openmp_sin_2.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 4  | tee logs/openmp_sin_4.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 8  | tee logs/openmp_sin_8.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 16 | tee logs/openmp_sin_16.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 28 | tee logs/openmp_sin_28.txt

srun -p tornado -N1 -n1 -t 00:02:00 bin/integrate_openmp 0 3.141592653589793 2000000 3 48 | tee logs/openmp_sin_48.txt
```

Appendix F — Copy results back to your PC (scp) Run in **Windows PowerShell** (not inside SSH

```
scp -i "C:\Users\Dorsa Abbasi\Abbasi.d" tm5u22@login1.hpc.spbstu.ru:~/nc_mpi/results_sorted.csv .

scp -i "C:\Users\Dorsa Abbasi\Abbasi.d" tm5u22@login1.hpc.spbstu.ru:~/nc_mpi/speedup.csv .
```