Sri Lanka Institute of Information Technology

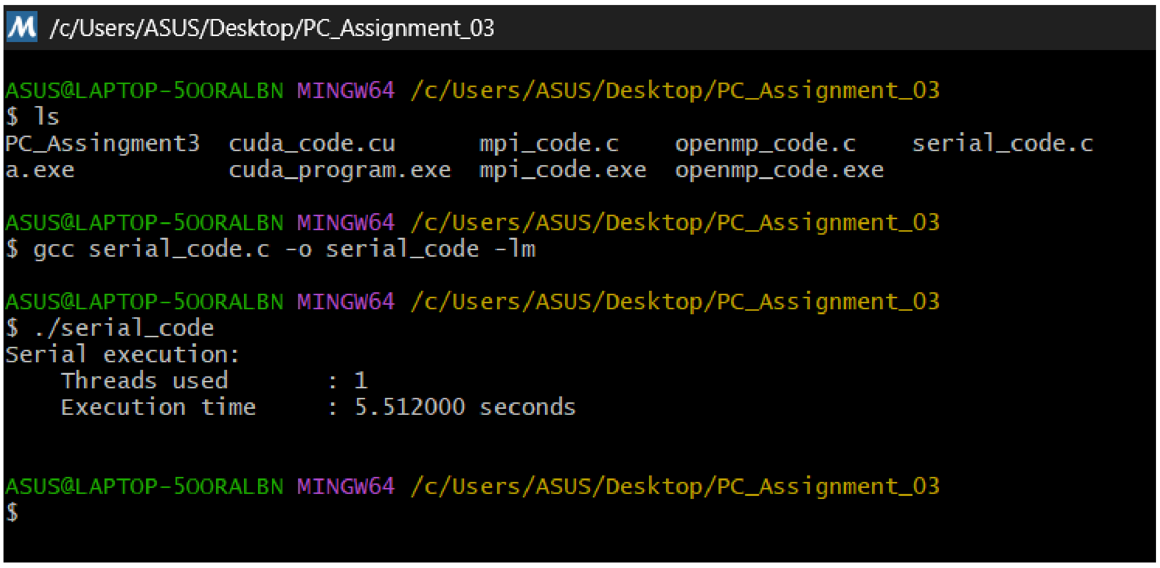
SE3082 - Parallel Computing
Assignment 03

Student ID	Name
IT23190016	M.S. SAMARASINGHE

BSc (Hons) in Computer Science

Contents

Serial Implementation/Baseline



```
/c/Users/ASUS/Desktop/PC_Assignment_03

ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ ls
PC_Assingment3  cuda_code.cu      mpi_code.c     openmp_code.c     serial_code.c
a.exe           cuda_program.exe  mpi_code.exe   openmp_code.exe

ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ gcc serial_code.c -o serial_code -lm

ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ ./serial_code
Serial execution:
    Threads used        : 1
    Execution time      : 5.512000 seconds


ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$
```

- This execution time of 5.512 seconds, measured in a single-threaded environment, will serve as the baseline for evaluating speedup performance across different parallel computing approaches, including OpenMP, MPI, and CUDA.

Part A: Parallel Implementations

- Repository: it23190016/PC_Assingment3

- This repository contains the implementations of parallel computing approaches for the assignment, including OpenMP, MPI, and CUDA versions. Each implementation is designed to evaluate performance improvements over a single-threaded baseline through efficient utilization of multiple cores or GPUs.

Part C: Documentation and Analysis

1. Parallelization Strategies

A. OpenMP Strategy

OpenMP uses shared memory parallelism through compiler directives. The ***#pragma omp parallel for private(j, k)*** directive automatically distributes loop iterations among CPU threads. All threads share the same memory space, eliminating communication overhead. The runtime system handles thread creation and synchronization barriers automatically.

Key Features:

- Automatic work distribution via compiler directives
- Shared memory model with implicit synchronization
- Minimal code changes required
- Limited to CPU cores

B. MPI Strategy

MPI employs distributed memory parallelism by decomposing the problem across processes. Each process computes assigned rows using ***rows_per_proc = N / size.*** Matrix B is distributed using ***MPI_Bcast*** since processes have separate memory spaces. This approach scales across multiple compute nodes.

Key Features:

- Row-wise domain decomposition
- Explicit message passing for data distribution
- Independent memory spaces per process
- Scalable across multiple nodes

C. CUDA Strategy

CUDA leverages GPU's massive parallel architecture using a 2D thread grid. Each thread computes one matrix element using unique thread indices. Explicit memory management transfers data between host and device. Different thread block configurations (1×1 to 32×32) optimize performance.

Key Features:

- Massive parallelism with thousands of threads
- 2D thread grid mapping to matrix elements
- Explicit host-device memory management
- Configurable thread block sizes

2. Runtime Configurations

A. Hardware Specifications

- **System:** ASUS TUF Gaming F15 FX507ZU4 (x64-based PC)
- **Processor:** 12th Gen Intel Core i7-12700H @ 2300 MHz
    - 14 physical cores (6 P-cores + 8 E-cores), 20 logical threads
- **Memory:** 16 GB RAM, 32 GB Page File
- **GPU:** NVIDIA GeForce RTX 4050 Laptop GPU (6 GB GDDR6)
    - CUDA Version: 13.0, Driver: 581.29
    - Max threads per block: 1024, Compute Capability: 8.9

B. Software Environment

- **Operating System:** Microsoft Windows 11 Home (Build 26100)
- **Compilers and Libraries:**
    - **Serial:** GCC compiler, C standard library
    - **OpenMP:** GCC with -fopenmp flag, omp.h library
    - **MPI:** MPICC wrapper, MPICH implementation
    - **CUDA:** NVCC compiler (v13.0), CUDA runtime API
- **Development Environment:** MSYS2 MinGW64 (OpenMP/MPI), Windows CMD (CUDA)

C. Configuration Parameters by Implementation

1. Serial Implementation

- Matrix Size: 1000×1000 (N = 1000)
- Compilation: gcc -o serial_code.exe serial_code.c
- Timing: clock() function from time.h

2. OpenMP Implementation

- Matrix Size: 1000×1000 (N = 1000)
- Thread Configurations: 1, 2, 4, 8, 16, 20 threads
- Compilation: gcc -fopenmp -o openmp_code.exe openmp_code.c
- Environment Variable: export OMP_NUM_THREADS=<value>
- Timing: omp_get_wtime() function

3. MPI Implementation

- Matrix Size: 1000×1000 (N = 1000)
- Process Configurations: 1, 2, 4, 8, 16 processes
- Compilation: mpicc -o mpi_code.exe mpi_code.c
- Execution: mpiexec -np <processes> ./mpi_code.exe
- Timing: MPI_Wtime() function

4. CUDA Implementation

- Matrix Size: 1000×1000 (N = 1000)
- Thread Block Configurations: 1×1, 2×2, 4×4, 8×8, 16×16, 32×32
- Compilation: nvcc cuda_code.cu -o cuda_code.exe
- Timing: clock() with cudaDeviceSynchronize()

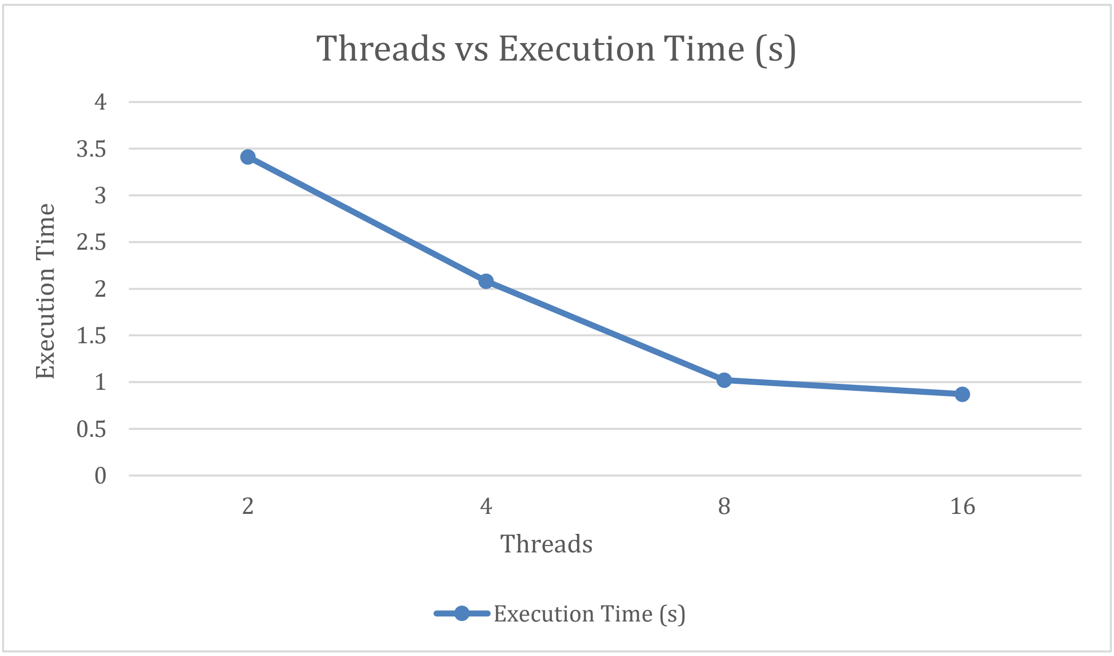
3. Performance Analysis

A. OpenMP Evaluation

Graphs

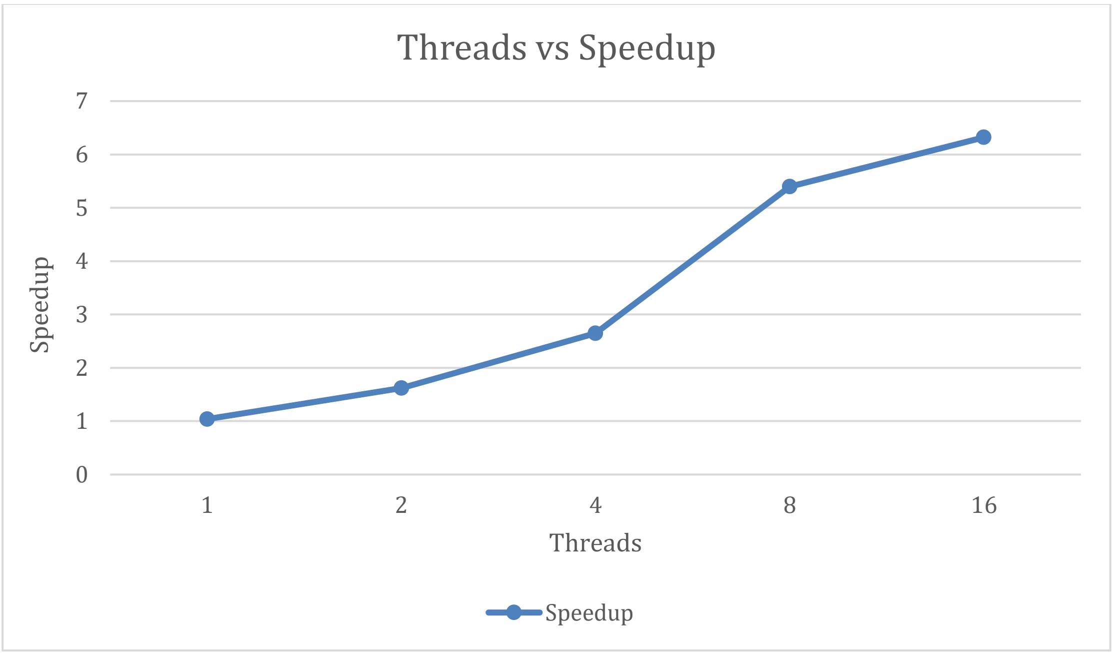
I. Number of threads vs Execution time

Number of Threads	Execution Time (s)
1	5.305
2	3.413
4	2.081
8	1.021
16	0.872



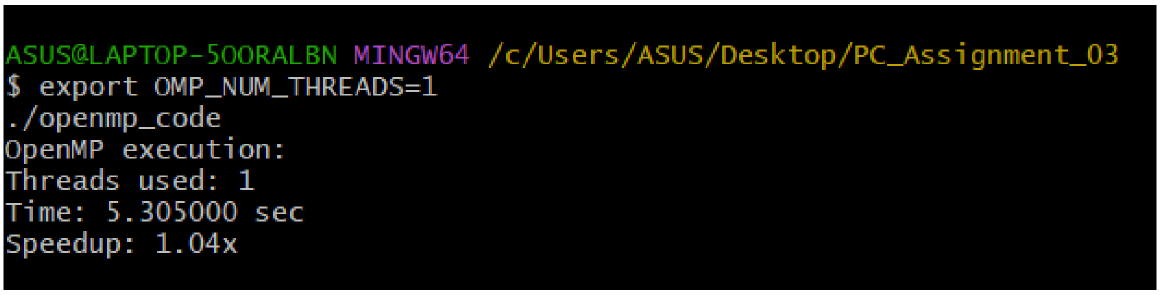
II. Number of threads vs Speedup

Number of Threads	Speedup
1	1.04
2	1.62
4	2.65
8	5.40
16	6.32



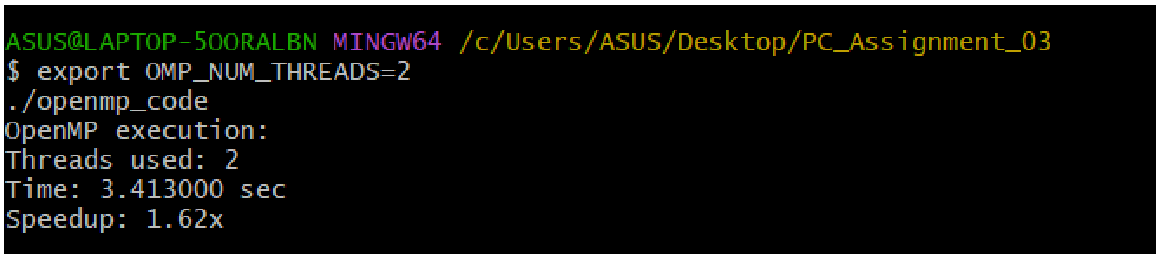
Screenshots

I. **Threads=01**



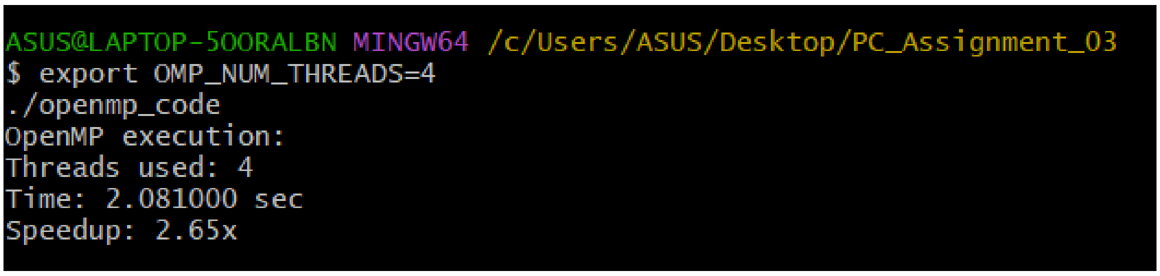
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ export OMP_NUM_THREADS=1
./openmp_code
OpenMP execution:
Threads used: 1
Time: 5.305000 sec
Speedup: 1.04x
```

II. **Threads=02**



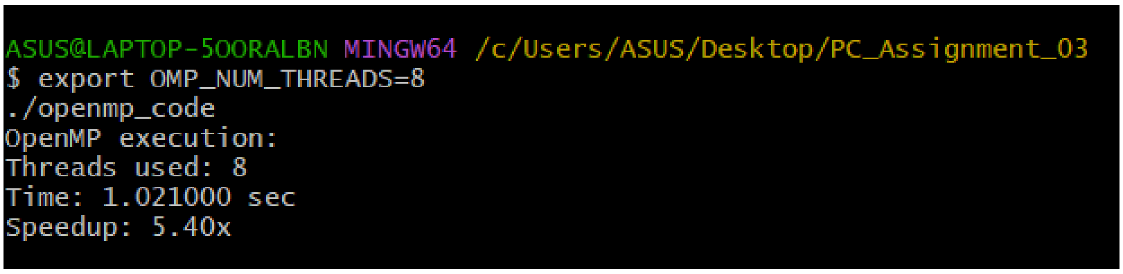
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ export OMP_NUM_THREADS=2
./openmp_code
OpenMP execution:
Threads used: 2
Time: 3.413000 sec
Speedup: 1.62x
```

III. **Threads=04**



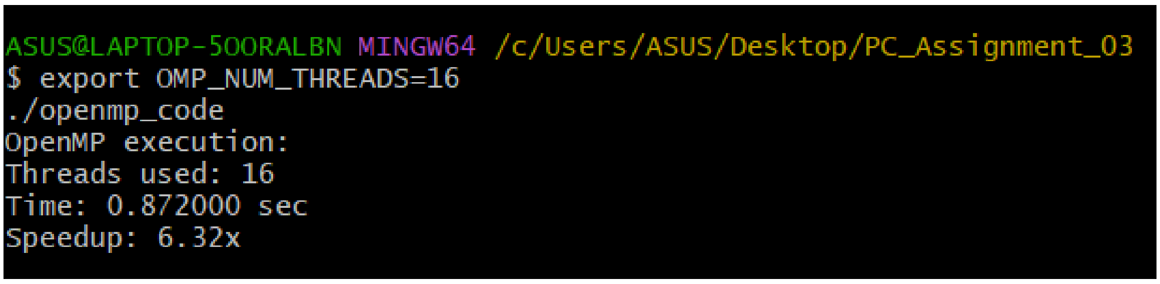
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ export OMP_NUM_THREADS=4
./openmp_code
OpenMP execution:
Threads used: 4
Time: 2.081000 sec
Speedup: 2.65x
```

IV. **Threads=08**



```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ export OMP_NUM_THREADS=8
./openmp_code
OpenMP execution:
Threads used: 8
Time: 1.021000 sec
Speedup: 5.40x
```

V. **Threads=16**



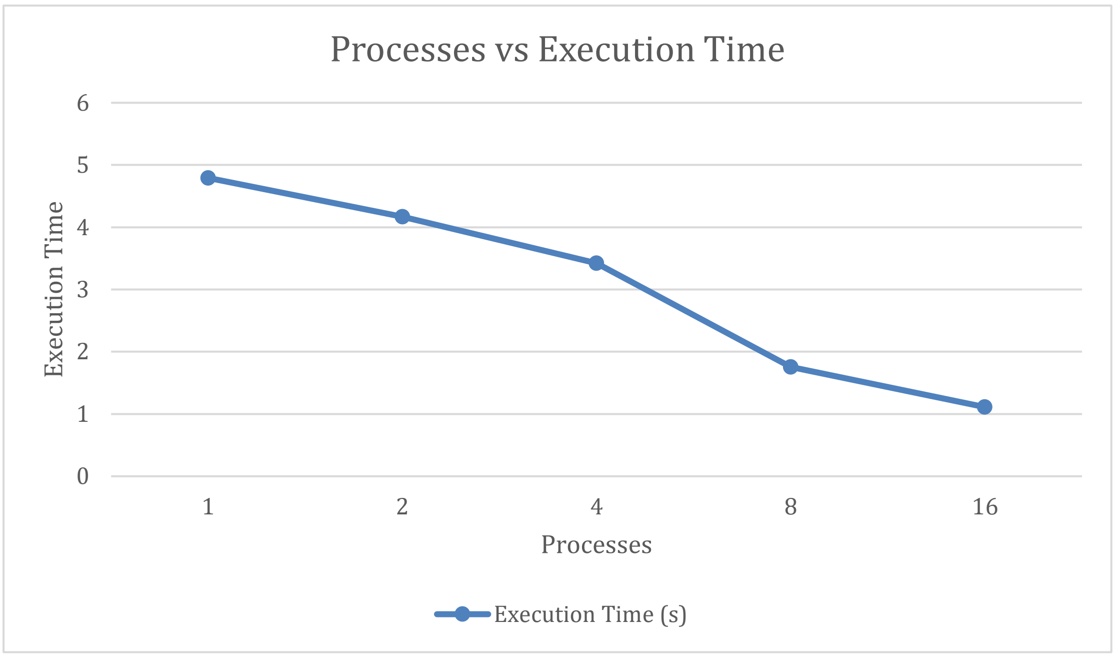
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ export OMP_NUM_THREADS=16
./openmp_code
OpenMP execution:
Threads used: 16
Time: 0.872000 sec
Speedup: 6.32x
```

B. MPI Evaluation

A. Graphs

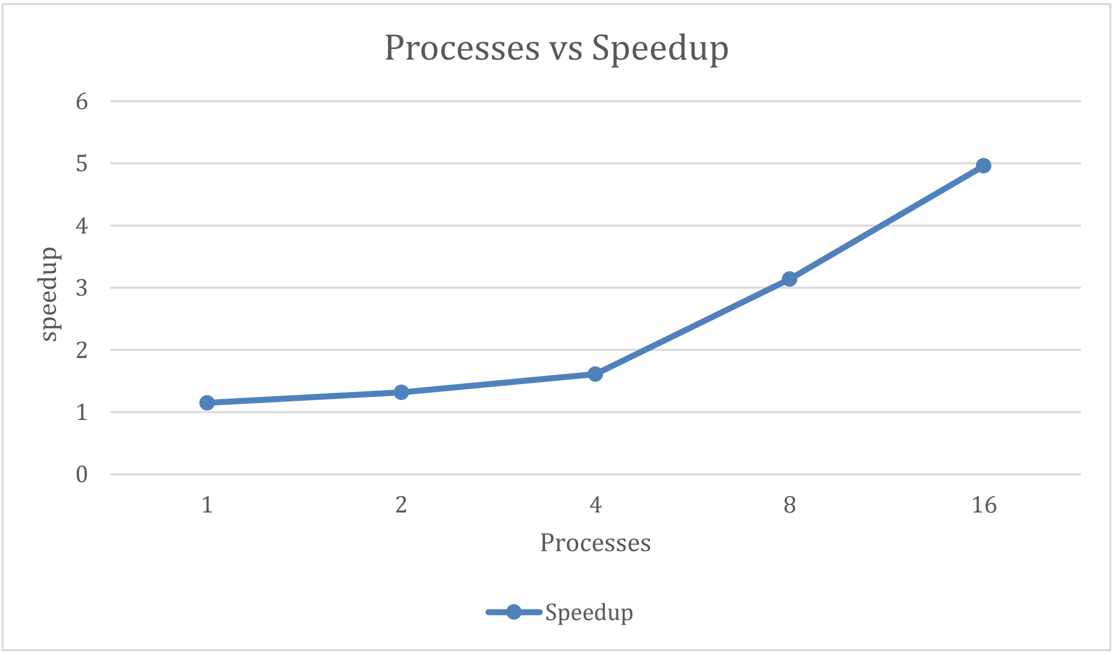
I. Number of Processes vs Execution Time

Number of Processes	Execution Time (s)
1	4.792355
2	4.165871
4	3.420793
8	1.755429
16	1.110914



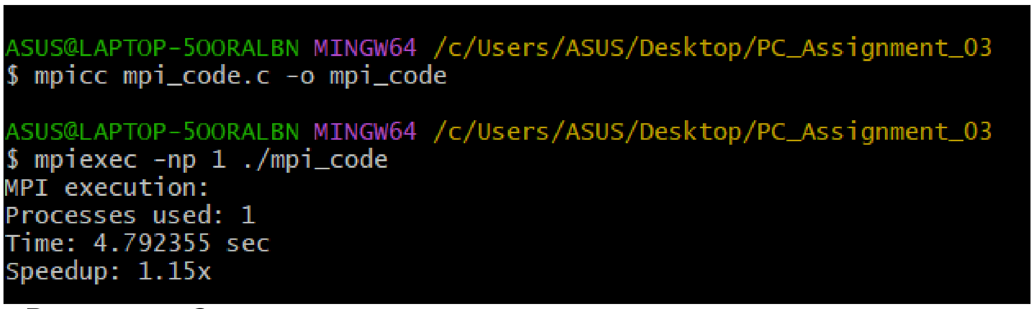
II. Number of Processes vs Speedup

Number of Processes	Speedup
1	1.15
2	1.32
4	1.61
8	3.14
16	4.96



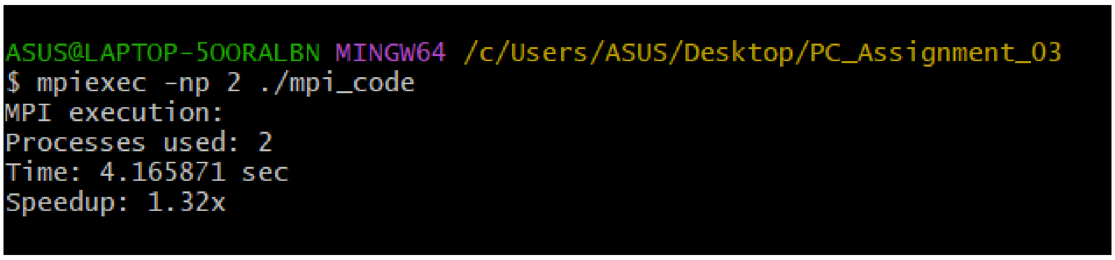
B. Screenshots

I. Processors=1



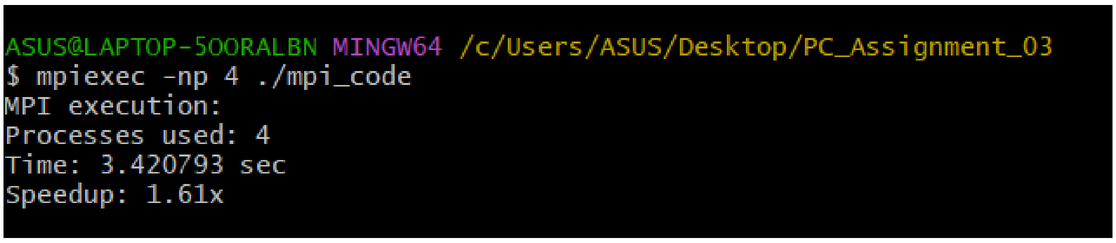
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpicc mpi_code.c -o mpi_code

ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpiexec -np 1 ./mpi_code
MPI execution:
Processes used: 1
Time: 4.792355 sec
Speedup: 1.15x
```

II. Processors=2



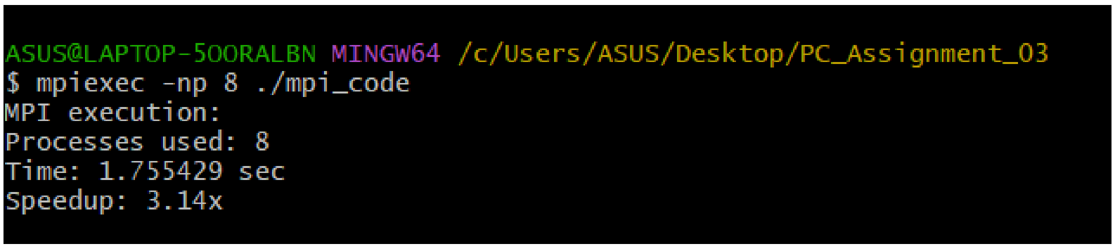
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpiexec -np 2 ./mpi_code
MPI execution:
Processes used: 2
Time: 4.165871 sec
Speedup: 1.32x
```

III. Processors=4



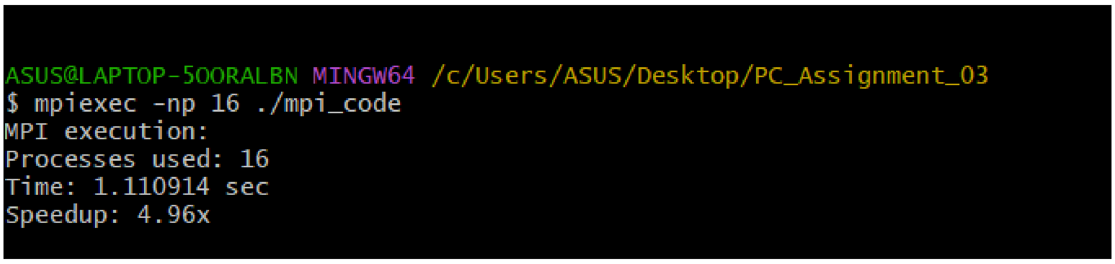
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpiexec -np 4 ./mpi_code
MPI execution:
Processes used: 4
Time: 3.420793 sec
Speedup: 1.61x
```

IV. Processors=8



```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpiexec -np 8 ./mpi_code
MPI execution:
Processes used: 8
Time: 1.755429 sec
Speedup: 3.14x
```

V. Processors=16



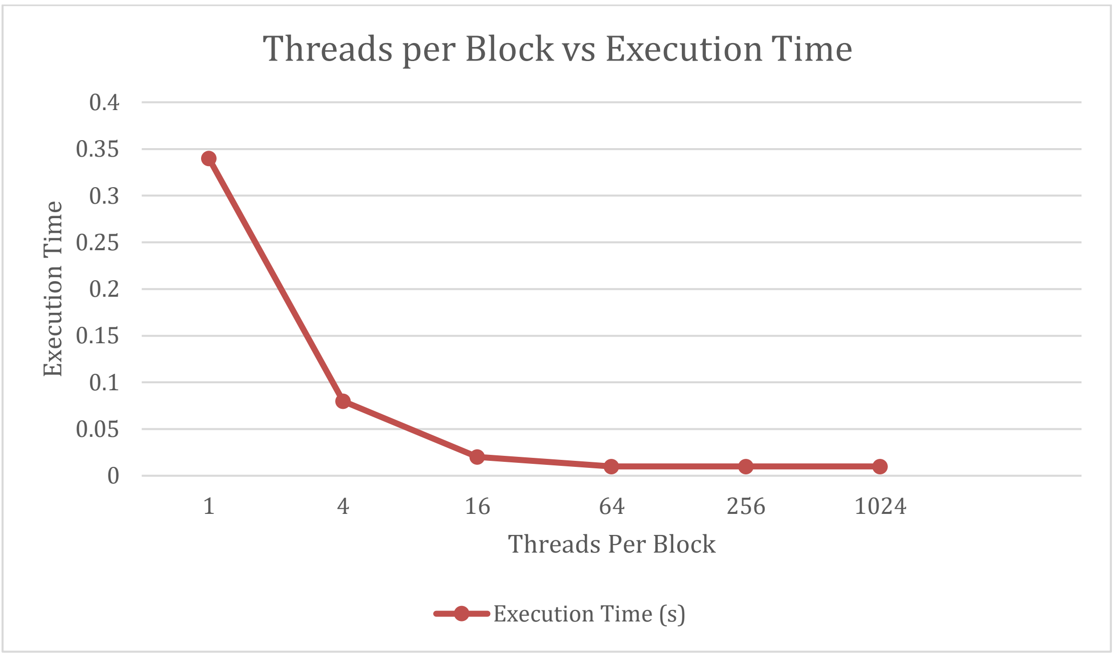
```
ASUS@LAPTOP-5OORALBN MINGW64 /c/Users/ASUS/Desktop/PC_Assignment_03
$ mpiexec -np 16 ./mpi_code
MPI execution:
Processes used: 16
Time: 1.110914 sec
Speedup: 4.96x
```

C. CUDA Evaluation

A. Graphs

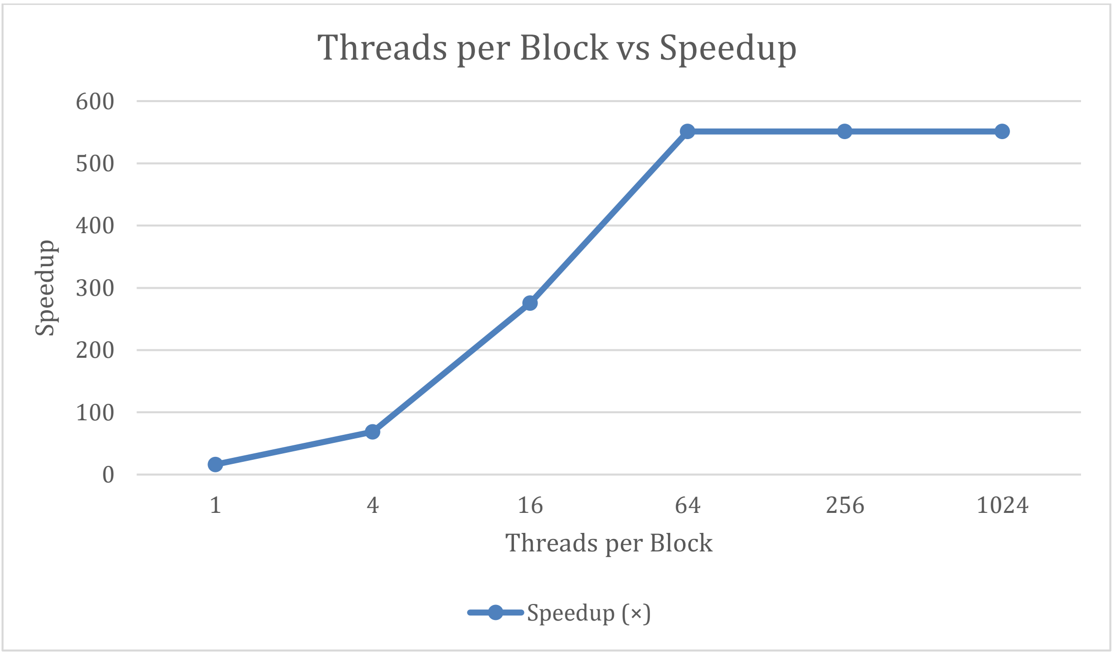
I. Configuration parameters vs Execution time

Threads per Block	**Blocks (Grid Size)**	**Execution Time (s)**
1	**1000 × 1000**	**0.340**
4	**500 × 500**	**0.080**
16	**250 × 250**	**0.020**
64	**125 × 125**	**0.010**
256	**63 × 63**	**0.010**
1024	**32 × 32**	**0.010**

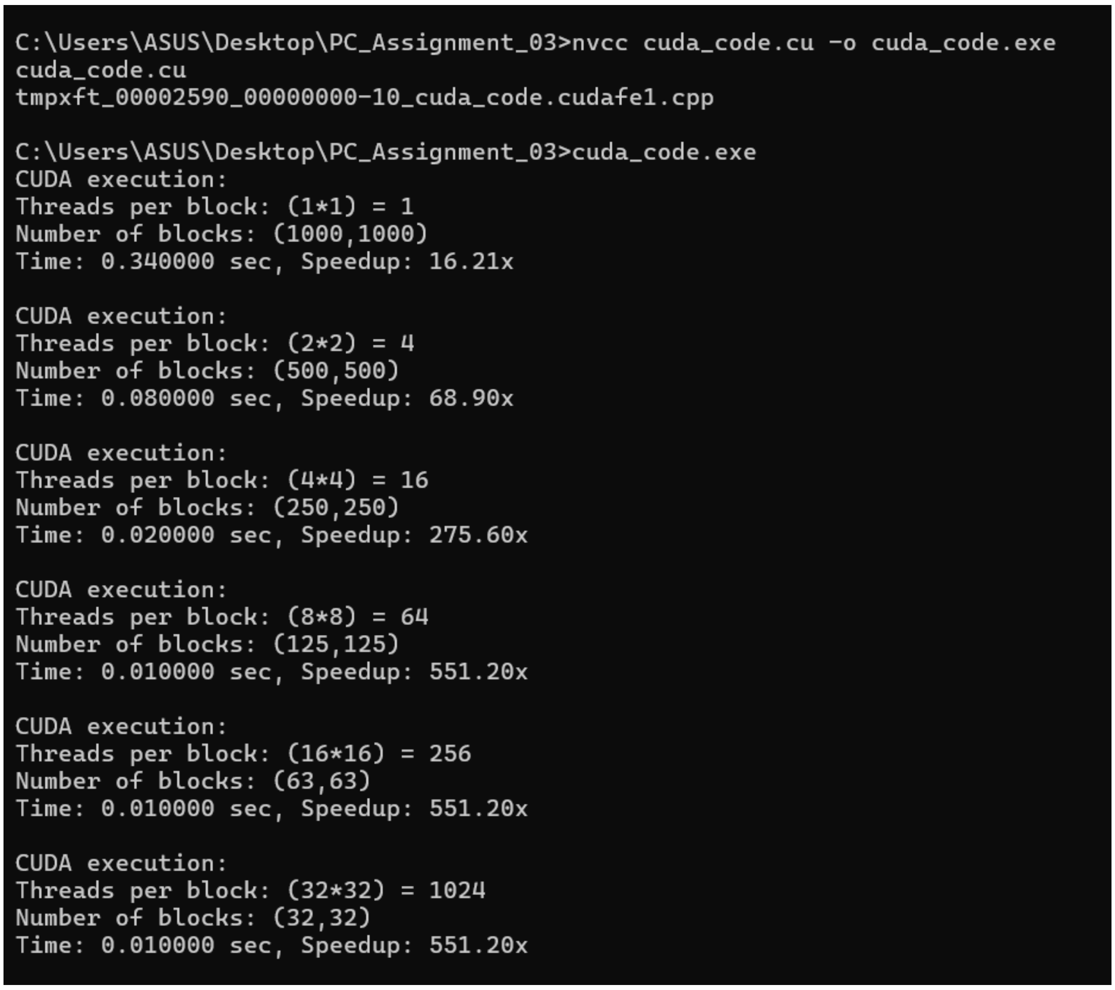


II. Configuration parameters vs Speedup

Threads per Block	Blocks (Grid Size)	Speedup (×)
1	1000 × 1000	16.21
4	500 × 500	68.90
16	250 × 250	275.60
64	125 × 125	551.20
256	63 × 63	551.20
1024	32 × 32	551.20



B. Screenshots



```
C:\Users\ASUS\Desktop\PC_Assignment_03>nvcc cuda_code.cu -o cuda_code.exe
cuda_code.cu
tmpxft_00002590_00000000-10_cuda_code.cudafe1.cpp

C:\Users\ASUS\Desktop\PC_Assignment_03>cuda_code.exe
CUDA execution:
Threads per block: (1*1) = 1
Number of blocks: (1000,1000)
Time: 0.340000 sec, Speedup: 16.21x

CUDA execution:
Threads per block: (2*2) = 4
Number of blocks: (500,500)
Time: 0.080000 sec, Speedup: 68.90x

CUDA execution:
Threads per block: (4*4) = 16
Number of blocks: (250,250)
Time: 0.020000 sec, Speedup: 275.60x

CUDA execution:
Threads per block: (8*8) = 64
Number of blocks: (125,125)
Time: 0.010000 sec, Speedup: 551.20x

CUDA execution:
Threads per block: (16*16) = 256
Number of blocks: (63,63)
Time: 0.010000 sec, Speedup: 551.20x

CUDA execution:
Threads per block: (32*32) = 1024
Number of blocks: (32,32)
Time: 0.010000 sec, Speedup: 551.20x
```

4. Comparative Analysis

A. Comparison on the Same Dataset / Problem Size

- All three implementations were executed on the same matrix multiplication task (N=1000). The performance results show clear differences in scalability and computational efficiency.

Execution Time Summary

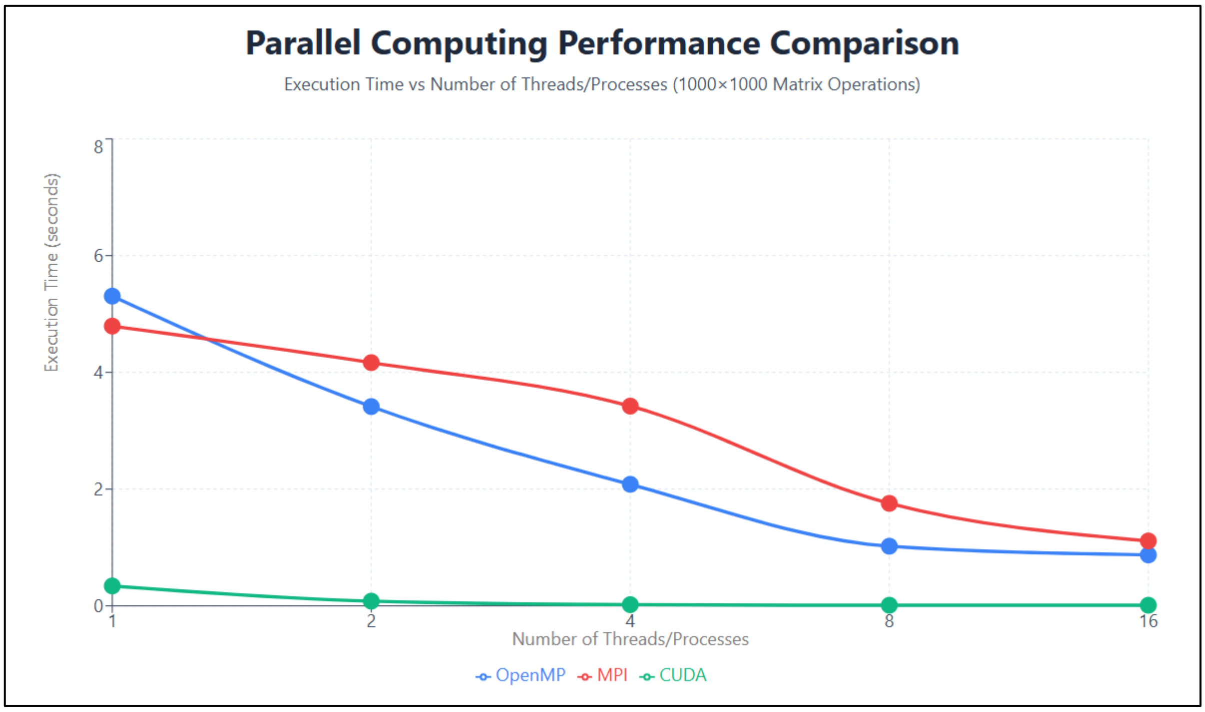
Method	**Max Parallel Unit**	**Best Time (sec)**	**Worst Time (sec)**
OpenMP	16 threads	**0.872**	5.305
MPI	16 processes	**1.110**	4.792
CUDA	1024 threads/block	**0.010**	0.340

Speedup Summary

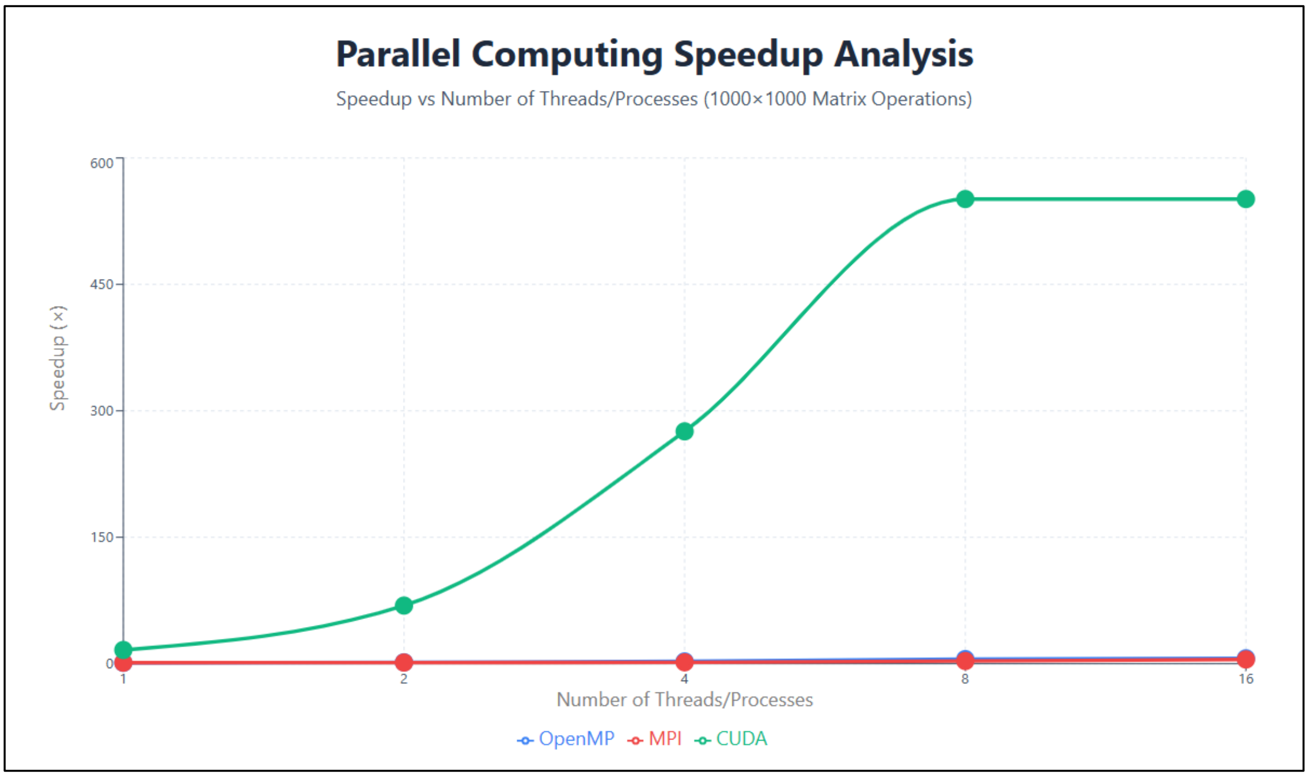
Method	**Max Speedup**
OpenMP	**6.32**
MPI	**4.96**
CUDA	**551.20**

Visual Comparison

A. Execution time VS Threads/Processes across OpenMP, MPI, CUDA



B. Execution time vs Threads/Processes across OpenMP, MPI, CUDA



C. Comparative Discussion

OpenMP

Strengths

- Very easy to integrate into existing C/C++ code using pragmas.
- Great when the workload fits a shared-memory architecture (single machine).
- Good scaling up to a moderate number of threads.

Weaknesses

- Limited by shared-memory hardware.
- Thread contention and cache issues become noticeable after 8–16 threads.
- Not suitable for distributed systems.

Observation:
OpenMP scales nicely on your CPU, but clearly hits a saturation point around 16 threads.

MPI

Strengths

- Designed for distributed-memory systems (clusters).
- Good control over data distribution and communication.
- Scales beyond a single machine if hardware is available.

Weaknesses

- Communication overhead increases significantly with process count.
- For small-to-medium matrix sizes, overhead outweighs raw compute advantages.
- More complex programming model than OpenMP.

Observation:
MPI underperforms on a single PC because communication cost dominates. On a proper multi-node cluster, MPI would catch up or surpass OpenMP.

CUDA

Strengths

- Massive parallelism—thousands of GPU cores.
- Very high arithmetic throughput.
- Best suited for data-parallel, highly regular operations like matrix multiplication.

Weaknesses

- Requires GPU hardware and CUDA toolkit.
- Extra complexity in kernel design, memory transfers, and block/thread configuration.
- Performance depends heavily on correct block size tuning.

Observation:

- CUDA absolutely crushes CPU-based methods:
- **551× speedup** is undeniable proof that GPUs dominate this workload type.

D. Which Implementation Is Most Appropriate?

- If you have access to a capable NVIDIA GPU, CUDA is unquestionably the best for matrix multiplication.
- The speedups (up to 551×) demonstrate that GPUs are built precisely for this kind of dense numerical computation.
- If GPU resources were *not* available:
  - OpenMP is the best on a single machine.
  - MPI becomes the best only when multiple physical nodes are available.

E. Strengths and Weaknesses Summary Table

Approach	**Strengths**	**Weaknesses**	**Best Use Case**
OpenMP	Simple, shared-memory parallelism; good speedup	Limited scalability; thread contention	Single powerful CPU, medium-scale problems
MPI	Works across multiple nodes; scalable	Communication overhead; complex	Clusters, HPC environments
CUDA	*Massive* parallelism; unbeatable for matrices	Requires GPU; complex memory mgmt	Data-parallel, compute-heavy problems

5. Critical Reflection

A. Challenges Faced

CUDA

- Large matrices (N=1000) caused GPU memory issues.
- Dynamic memory allocation and block/thread configuration were complex.
- Modifying configurations required significant code changes.

OpenMP

- Static arrays failed; dynamic allocation with malloc/free was needed.
- Hardcoded serial time required for accurate speedup calculation.

MPI

- Large matrices caused stack overflow; resolved with dynamic allocation.
- Complex logic required for distributing matrices across processes.

B. Limitations

- **Memory**: GPU and stack limits restricted large matrices; dynamic allocation added minor overhead.
- **Implementation Complexity**: CUDA required major restructuring; all implementations relied on a fixed serial baseline.
- **Debugging**: Memory-related crashes were challenging to diagnose.

C. Solutions Implemented

- **Dynamic Allocation**: Used malloc/free and cudaMalloc/cudaFree.
- **Consistent Timing**: SERIAL_TIME used for speedup; proper timing around computation sections.
- **Modular Code**: Separated allocation, computation, cleanup; added error checks; adjustable thread/block configurations.

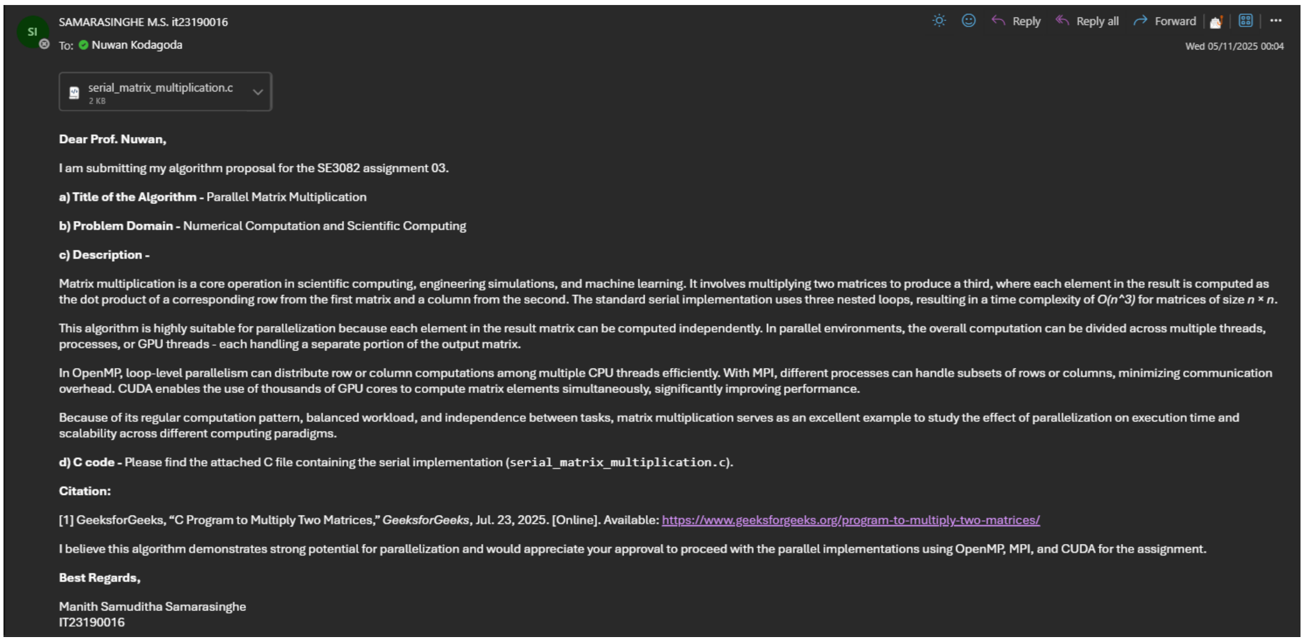
D. Lessons Learned

- Dynamic memory is essential for large-scale parallel computing.
- Baseline timing consistency is crucial for performance comparison.
- CUDA has the highest complexity; OpenMP is simplest; MPI adds communication challenges.
- Configuration parameters strongly affect performance.

E. Future Improvements

- Memory pooling and GPU unified memory for efficiency.
- Configurable thread/block selection and automatic memory sizing.
- Adaptive configuration and hybrid parallel approaches.
- Integrated profiling and error-handling mechanisms.

Proposal Email and Approval



SAMARASINGHE M.S. it23190016

To: Nuwan Kodagoda

Wed 05/11/2025 00:04

serial_matrix_multiplication.c
2 KB

Dear Prof. Nuwan,

I am submitting my algorithm proposal for the SE3082 assignment 03.

a) Title of the Algorithm - Parallel Matrix Multiplication

b) Problem Domain - Numerical Computation and Scientific Computing

c) Description -

Matrix multiplication is a core operation in scientific computing, engineering simulations, and machine learning. It involves multiplying two matrices to produce a third, where each element in the result is computed as the dot product of a corresponding row from the first matrix and a column from the second. The standard serial implementation uses three nested loops, resulting in a time complexity of *O(n^3)* for matrices of size *n × n*.

This algorithm is highly suitable for parallelization because each element in the result matrix can be computed independently. In parallel environments, the overall computation can be divided across multiple threads, processes, or GPU threads - each handling a separate portion of the output matrix.

In OpenMP, loop-level parallelism can distribute row or column computations among multiple CPU threads efficiently. With MPI, different processes can handle subsets of rows or columns, minimizing communication overhead. CUDA enables the use of thousands of GPU cores to compute matrix elements simultaneously, significantly improving performance.

Because of its regular computation pattern, balanced workload, and independence between tasks, matrix multiplication serves as an excellent example to study the effect of parallelization on execution time and scalability across different computing paradigms.

d) C code - Please find the attached C file containing the serial implementation (`serial_matrix_multiplication.c`).

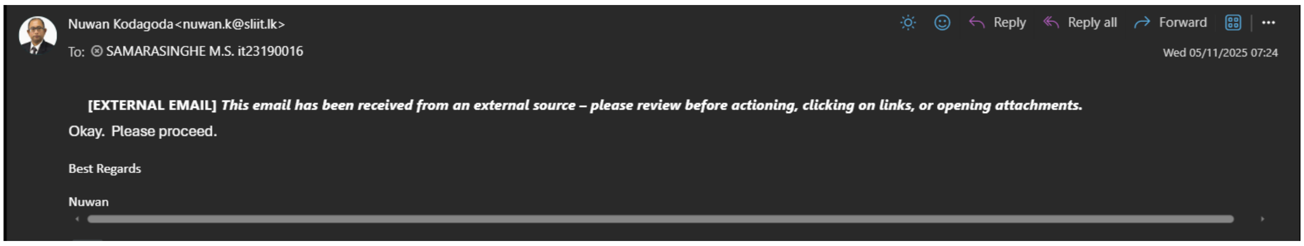
Citation:

[1] GeeksforGeeks, "C Program to Multiply Two Matrices," *GeeksforGeeks*, Jul. 23, 2025. [Online]. Available: https://www.geeksforgeeks.org/program-to-multiply-two-matrices/

I believe this algorithm demonstrates strong potential for parallelization and would appreciate your approval to proceed with the parallel implementations using OpenMP, MPI, and CUDA for the assignment.

Best Regards,

Manith Samuditha Samarasinghe
IT23190016



Nuwan Kodagoda<nuwan.k@sliit.lk>

To: SAMARASINGHE M.S. it23190016

Wed 05/11/2025 07:24

[EXTERNAL EMAIL] ***This email has been received from an external source – please review before actioning, clicking on links, or opening attachments.***

Okay. Please proceed.

Best Regards

Nuwan

Code Execution Demo

Assignment03_Demo.mp4