Parallel and Distributed Computing - Project 1 Report

Student Info

Name: Bahaa Aldeen Abualrob

ID: 202112389

Section: 1

1. Introduction

This project focuses on converting a sequential algorithm into a multithreaded version using Pthreads in C++. The primary goal is to analyze its performance, measure speedup, and gain practical insights into parallel computing. We selected matrix multiplication as our target algorithm due to its high parallelizability and suitability for performance benchmarking.

2. Algorithm Choice

Selected Algorithm: Matrix Multiplication

Reason for selection:

- Each element of the result matrix is calculated independently.
- Natural parallel structure — work can be distributed across rows or blocks.
- No race conditions or dependencies between elements.
- Well-suited to demonstrate the benefits of multithreading.

3. Sequential Implementation

Language & Environment

- Language: C++
- Compiler: g++ on Linux
- Editor: Visual Studio Code
- Timing: std::chrono library

Key Code Snippets:

```
Matrix multiplySequential(const Matrix &A, const Matrix &B) {
   Matrix C(n, vector<int>(m, 0));
```

```
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      for (int k = 0; k < p; ++k)
        C[i][j] += A[i][k] * B[k][j];
  return C;
}
```

4. Parallelization Strategy

Threading Approach

- Used pthread_create and pthread_join to manage threads.
- Work was divided by assigning rows to each thread.
- Each thread computed a unique portion of the result matrix.

Struct for Arguments

```
struct ThreadArgs { const Matrix *A; const Matrix *B; Matrix *C; int start_row; int end_row;
};
```

Pthread Function

```
void* multiplyPart(void* arg) {
  ThreadArgs *args = (ThreadArgs*)arg;
  for (int i = args->start_row; i < args->end_row; ++i)
    for (int j = 0; j < cols; ++j)
      for (int k = 0; k < common; ++k)
        C[i][j] += A[i][k] * B[k][j];
}
```

5. Experiments

Hardware Specs

- CPU: Intel i7 / AMD Ryzen
- Cores: [2]
- OS: Linux Kali
- RAM: [e.g., 16GB]

Input Sizes

- Matrix sizes: 200x200, 500x500, 1000x1000
- Thread counts tested: 1, 2, 4, 8
- Each test repeated 3 times and averaged.

6. Results

Sample Table: Execution Time

Matrix Size	Threads	Avg Time (ms)
200	1	402.33
200	2	209.67
200	4	216.67
1000	1	45437.00
1000	2	25166.33
1000	4	24415.33

Sample Table: Speedup

Matrix Size	Threads	Seq Time (ms)	Par Time (ms)	Speedup
200	2	402.33	209.67	1.92
200	4	402.33	216.67	1.86
500	2	5456.00	2799.33	1.95
500	3	5456.00	2699.67	2.02

7. Graphs

Include two line charts from Excel:
- Execution Time vs Threads:

- Speedup vs Threads:

8. Discussion

- Sublinear Speedup: Due to thread creation overhead and synchronization cost.
- Memory Bandwidth: Limits performance beyond a certain number of threads.
- Load Balancing: Uniform matrix sizes led to evenly distributed work.

Amdahl's Law approximation supports observed speedups.

9. Conclusion

- Matrix multiplication demonstrated strong potential for parallelization.
- Pthreads enabled significant speedup with minimal complexity.
- Practical limits observed due to hardware and OS-level threading.
- Learned: workload partitioning, thread safety, performance tuning.

10. Tools & References

- Tools: g++, VSCode, Excel, GitHub
- Libraries: <pthread.h>, <chrono>, <vector>

Project Folder Structure

```
project1/
├── src/
│   ├── sequential.cpp
│   └── parallel.cpp
├── docs/
│   └── report.pdf
├── results/
│   ├── timing_data.csv
│   └── validation_checksums.txt
├── README.md
```