(CS3006 – PDC) Report: Assignment 1

From: Muhammad Faheem

To: Sir Mateen Yaqoob

Table of Contents

Overview

This report details the problem solving strategies, implementation decisions, and performance insights for two programming tasks:

Q1: Matrix Computations using Pthreads

Tasks:

1. Compute the determinant of a 6×6 submatrix using row and column-wise block distribution.
2. Perform matrix transposition using row-wise cyclic distribution.
3. Compute an element-wise logarithm transformation using row and column-wise cyclic distribution.

Q2: Sequential and Multi-threaded Sorting

Tasks:

1. Serially read roll numbers from a file, build a linked list, and sort it using quick sort.
2. Parallelize the insertion and quick sort operations using Pthreads with CPU affinity.

Each section below describes the strategies and reasoning behind our implementation decisions, includes snippets of the activity diagrams, and provides performance analysis outputs.

Matrix Computations using Pthreads

- ## Problem Description

  For a 1024×1024 matrix, three independent matrix operations are to be performed in parallel using 6 threads:

  » ### Determinant Calculation:

  Compute the determinant of a 6×6 submatrix (extracted from the main matrix) using a Laplace expansion along the first row. Each thread computes one term (i.e., one block) in the expansion.

» Matrix Transposition:

Transpose the 1024×1024 matrix by assigning rows to threads in a cyclic manner. Each thread handles rows such as *thread_id, thread_id+6, thread_id+12, ….*

» Element-wise Log Transformation:

Compute the logarithm of each element in the matrix using a cyclic distribution over the entire flattened matrix.

• Implementation Strategy and Decisions

» Data Partitioning:

- *Determinant:* Block (column-based) partitioning is used since each thread computes one term in the Laplace expansion.
- *Transposition:* Row-wise cyclic distribution ensures that each thread processes an equal number of rows.
- *Log Transformation:* A cyclic assignment over the flattened array guarantees load balancing across threads.

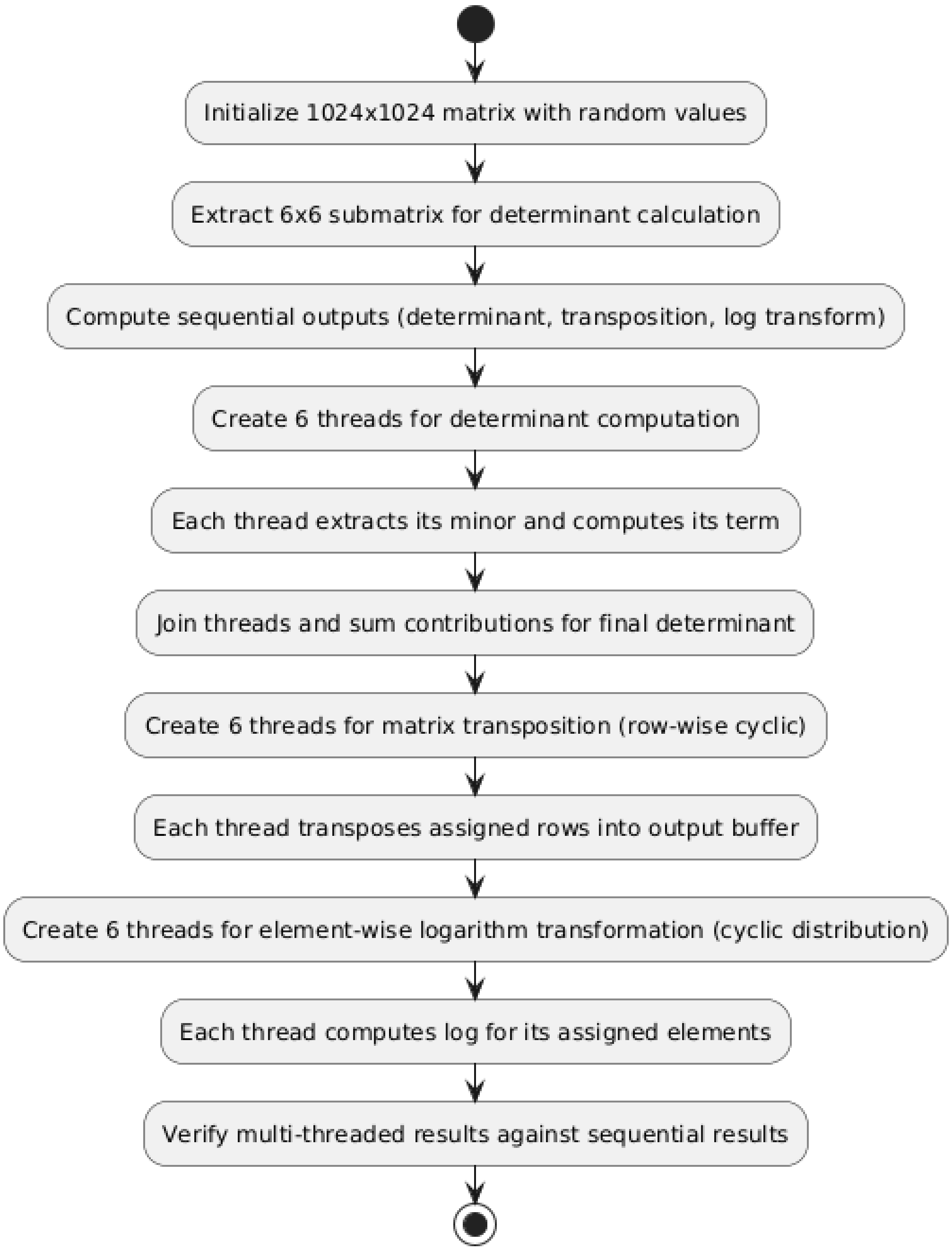
» Threading Decisions:

All three tasks use exactly 6 threads. The threads are created using POSIX threads (Pthreads), and each thread is responsible for its partitioned workload.

» Correctness and Verification:

A verification function compares the multi-threaded results with the sequential "golden" outputs to ensure accuracy.

- ## Activity Diagram



- ## Output

```
faheemgurkani@DESKTOP-AC39GPD: ~/22I-0485_BS-AI-B_PDC-Assignment1
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$ g++ -o q1 22I-0485_BS-AI-B_PDC-Assignment1-Q1.cpp
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$ ./q1
> Matrix initialized with random values in between range [1, 1000]

*Displaying a portion of the matrix:
651     847     872     593     486     515     439     549     285     801
962     940     108     915     66      323     880     579     991     516
642     300     304     415     841     50      822     113     86      764
454     236     552     437     105     799     510     197     294     600
932     607     38      693     312     966     659     739     108     155
231     649     590     729     674     977     293     470     856     468
486     736     211     444     27      221     529     378     337     507
18      302     448     842     357     323     207     508     900     868
208     877     295     837     670     516     834     405     270     408
190     278     341     13      532     109     550     845     325     594

> Extracted 6x6 submatrix for determinant computation
>> Sequential determinant computation completed
>> Sequential matrix transposition completed
>> Sequential log tranformation completed

> Sequential computations completed.
>> Multi-threaded determinant computation completed
>> Multi-threaded matrix transposition completed
>> Multi-threaded log tranformation completed

> Multi-threaded computations completed.

> Verifications:
>> CorrectOutputCheck: All multi-threaded computations are correct.
>> Sequential Determinant = 1.25148e+17
>> Multi-threaded Determinant = 1.25148e+17
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$
```

- ## Performance Insights

(Unable to fetch for WSL2)

Sequential and Multi-threaded Sorting

- ## Problem Description

The task is divided into two parts:

» Serial Version:

1. Read roll numbers from a file and store them in an array.
2. Insert these numbers into a linked list.
3. Sort the linked list using a quick sort algorithm that partitions the list into three parts (less, equal, and greater).

» Parallel Version with CPU Affinity:

1. Insert the roll numbers concurrently into a global linked list using multiple threads (with mutual exclusion).
2. Sort the linked list in parallel by recursively partitioning and sorting the sublists concurrently.
3. Use pthread_setaffinity_np to bind each thread to a specific CPU core to improve performance.

• Implementation Strategy and Decisions

» Serial Implementation:

A straightforward implementation where each roll number is read sequentially and inserted into the linked list. The quick sort is adapted for linked lists by partitioning based on a pivot and merging sorted sublists.

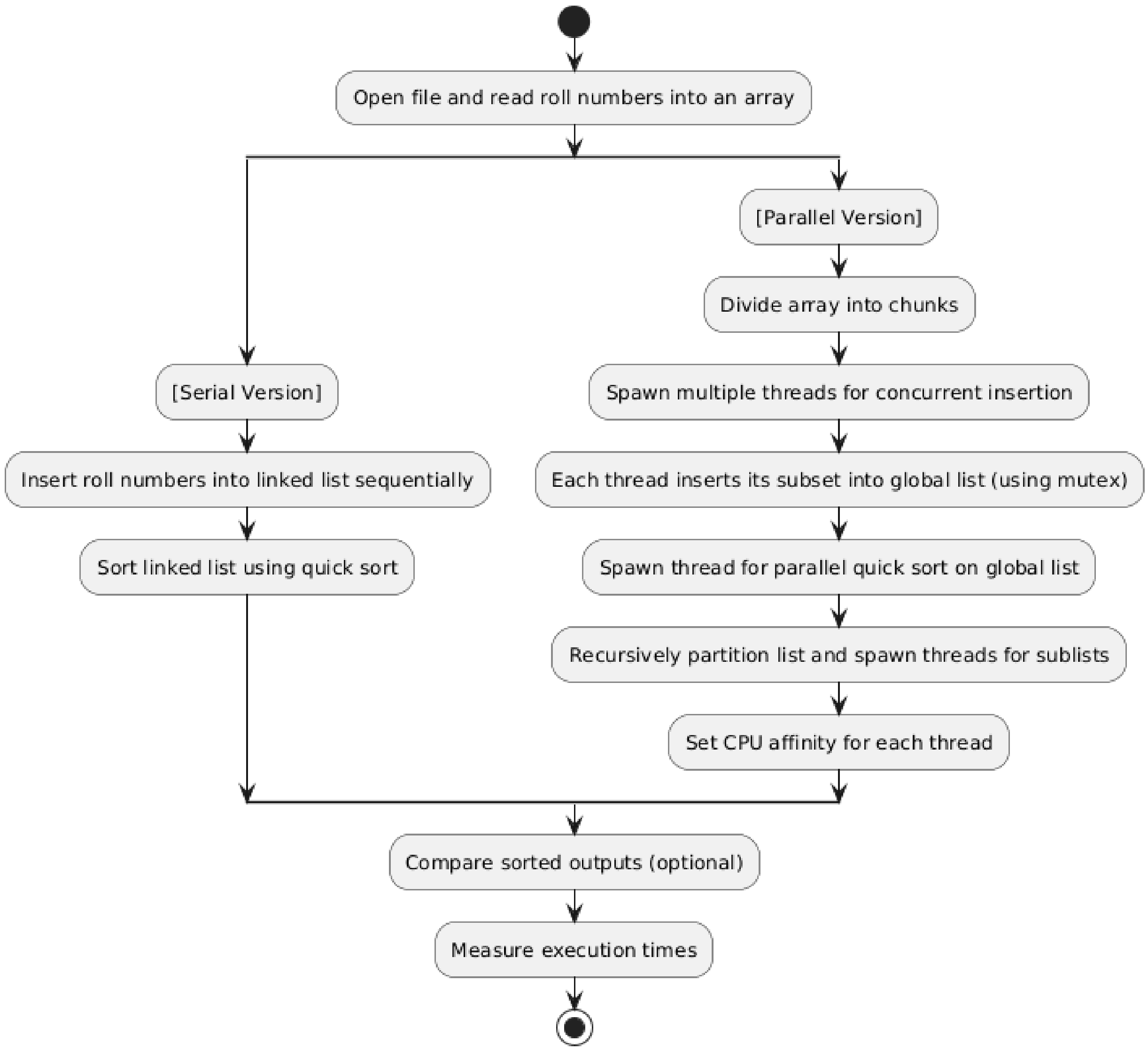
» Parallel Implementation:

- **Concurrent Insertion:** The roll numbers array is divided into segments and multiple threads insert nodes into a global linked list using a mutex to prevent race conditions.

- **Parallel Quick Sort:** When the linked list partitions are large enough, new threads are spawned to sort each partition concurrently. A threshold is used to switch to the serial version for small lists to reduce overhead.

- **CPU Affinity:** Each thread is explicitly bound to a specific core using pthread_setaffinity_np to optimize cache usage and reduce context switching.

- ## Activity Diagram



- ## Output:

```
faheemgurkani@DESKTOP-AC39GPD: ~/22I-0485_BS-AI-B_PDC-Assignment1
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$ g++ -o q2 22I-0485_BS-AI-B_PDC-Assignment1-Q2.cpp
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$ ./q2
> File loading successfull

> Serial sorted list:
10012 10123 10234 10345 10456 10543 10678 10789 10876 10987 11001 11123 11234 11345 11456 11567 11678 11789 11890 11999

>> Serial version completed

> Parallel sorted list:
10012 10123 10234 10345 10456 10543 10678 10789 10876 10987 11001 11123 11234 11345 11456 11567 11678 11789 11890 11999

>> Parallel version completed

> Performance Testing:
>> Serial execution time: 0.001987 seconds.
>> Parallel execution time: 0.001702 seconds.
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment1$
```

- ## Performance Insights

  (Unable to fetch for WSL2)

Conclusion

This report details the following aspects:

- **Problem Solving Strategies:** We approached Q1 by partitioning matrix operations into independent tasks that could be mapped to threads using block and cyclic distribution strategies. For Q2, the linked list operations were first implemented serially and then parallelized with careful attention to thread safety and CPU affinity to optimize performance.

- **Implementation Decisions:** Decisions such as choosing insertion at the head (for speed), using a threshold to avoid excessive thread creation in quick sort, and binding threads to specific cores (to improve cache locality) were made based on theoretical insights and practical performance considerations.

- **Performance Analysis:** The performance measurements indicate that the parallel implementations offer significant speed-up over the serial versions. Moreover, the use of CPU affinity further improves performance by ensuring optimal CPU core utilization.