(CS3006 – PDC) Report: Assignment 3

From: Muhammad Faheem

To: Sir Mateen Yaqoob

Table of Contents

Overview

This report details the problem-solving strategies, implementation decisions, and performance insights for three programming tasks related to parallel and distributed computing using OpenMP:

Q1: Box Blur Filter using OpenMP

Tasks:

1. Implement a box blur filter using a 3×3 kernel where each pixel is replaced by the average of its neighboring pixels.
2. Parallelize the filter operation for efficiency.
3. Compare execution times between sequential and parallel implementations.
4. Handle boundary conditions properly.

Q2: Parallel Treasure Hunt Simulation

Tasks:

1. Simulate a grid-based treasure hunt where each thread represents an adventurer.
2. Implement random movement and score accumulation mechanisms.
3. Introduce dynamic barriers and traps affecting adventurer scores.
4. Manage threads dynamically based on in-game events.

Q3: Conway's Game of Life with OpenMP

Tasks:

1. Implement a serial version of Conway's Game of Life on a 100x100 grid.
2. Parallelize row updates using OpenMP.
3. Implement both static and guided scheduling strategies.
4. Compare execution times and analyze the impact of removing critical sections.

Each section below describes the strategies and reasoning behind our implementation decisions and provides performance analysis outputs.

Box Blur Filter using OpenMP

- ## Problem Description
  - The task involves applying a 3×3 box blur filter to grayscale images of various sizes using OpenMP.
  - The goal is to compare execution times for serial and parallel implementations across different thread counts.

- ## Implementation Strategy and Decisions
  - **Data Partitioning:** Each thread processes a portion of the image using row-wise partitioning.
  - **Threading Decisions:** OpenMP parallel for loops are used with different scheduling strategies.
  - **Correctness and Verification:** Outputs are compared with sequential execution to ensure accuracy.

- ## Performance Analysis
  - Execution time is recorded for images of sizes 1000x1000, 2000x2000, 3000x3000, and 5000x5000 with varying thread counts.
  - Observations highlight improvements in execution time with parallelization but also show diminishing returns beyond a certain number of threads.

- Output

```
faheemgurkani@DESKTOP-AC39GPD: ~/22I-0485_BS-AI-B_PDC-Assignment3/src
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment3/src$ g++ -fopenmp -o q q1.cpp
faheemgurkani@DESKTOP-AC39GPD:~/22I-0485_BS-AI-B_PDC-Assignment3/src$ ./q
> Sequential Execution:
-> Time for 1000x1000 image: 0.560097 seconds.

> Parallel Execution:
-> Time for 1000x1000 image: 0.284805 seconds. Using 2 threads.
-> Time for 1000x1000 image: 0.159982 seconds. Using 4 threads.
-> Time for 1000x1000 image: 0.191158 seconds. Using 6 threads.
-> Time for 1000x1000 image: 0.168569 seconds. Using 8 threads.
-> Time for 1000x1000 image: 0.158477 seconds. Using 10 threads.
-> Time for 1000x1000 image: 0.175479 seconds. Using 12 threads.
-> Time for 1000x1000 image: 0.158794 seconds. Using 14 threads.
-> Time for 1000x1000 image: 0.164364 seconds. Using 16 threads.

--------------------------------------------------------------------------------

> Sequential Execution:
-> Time for 2000x2000 image: 2.3859 seconds.

> Parallel Execution:
-> Time for 2000x2000 image: 1.11081 seconds. Using 2 threads.
-> Time for 2000x2000 image: 0.6408 seconds. Using 4 threads.
-> Time for 2000x2000 image: 0.699918 seconds. Using 6 threads.
-> Time for 2000x2000 image: 0.708115 seconds. Using 8 threads.
-> Time for 2000x2000 image: 1.06678 seconds. Using 10 threads.
-> Time for 2000x2000 image: 0.914022 seconds. Using 12 threads.
-> Time for 2000x2000 image: 0.826871 seconds. Using 14 threads.
-> Time for 2000x2000 image: 0.902185 seconds. Using 16 threads.

--------------------------------------------------------------------------------

> Sequential Execution:
-> Time for 3000x3000 image: 5.52731 seconds.

> Parallel Execution:
-> Time for 3000x3000 image: 2.60335 seconds. Using 2 threads.
-> Time for 3000x3000 image: 1.62792 seconds. Using 4 threads.
-> Time for 3000x3000 image: 1.68299 seconds. Using 6 threads.
-> Time for 3000x3000 image: 1.54476 seconds. Using 8 threads.
-> Time for 3000x3000 image: 1.57477 seconds. Using 10 threads.
```

```
faheemgurkani@DESKTOP-AC39GPD: ~/22I-0485_BS-AI-B_PDC-Assignment3/src

> Sequential Execution:
-> Time for 3000x3000 image: 5.52731 seconds.

> Parallel Execution:
-> Time for 3000x3000 image: 2.60335 seconds. Using 2 threads.
-> Time for 3000x3000 image: 1.62792 seconds. Using 4 threads.
-> Time for 3000x3000 image: 1.68299 seconds. Using 6 threads.
-> Time for 3000x3000 image: 1.54476 seconds. Using 8 threads.
-> Time for 3000x3000 image: 1.57477 seconds. Using 10 threads.
-> Time for 3000x3000 image: 1.60125 seconds. Using 12 threads.
-> Time for 3000x3000 image: 1.73363 seconds. Using 14 threads.
-> Time for 3000x3000 image: 1.68168 seconds. Using 16 threads.

--------------------------------------------------------------------------------

> Sequential Execution:
-> Time for 5000x5000 image: 14.5847 seconds.

> Parallel Execution:
-> Time for 5000x5000 image: 7.29288 seconds. Using 2 threads.
-> Time for 5000x5000 image: 4.22535 seconds. Using 4 threads.
-> Time for 5000x5000 image: 4.44791 seconds. Using 6 threads.
-> Time for 5000x5000 image: 4.32319 seconds. Using 8 threads.
-> Time for 5000x5000 image: 4.06371 seconds. Using 10 threads.
-> Time for 5000x5000 image: 3.12533 seconds. Using 12 threads.
-> Time for 5000x5000 image: 3.06003 seconds. Using 14 threads.
-> Time for 5000x5000 image: 2.59503 seconds. Using 16 threads.

--------------------------------------------------------------------------------
```

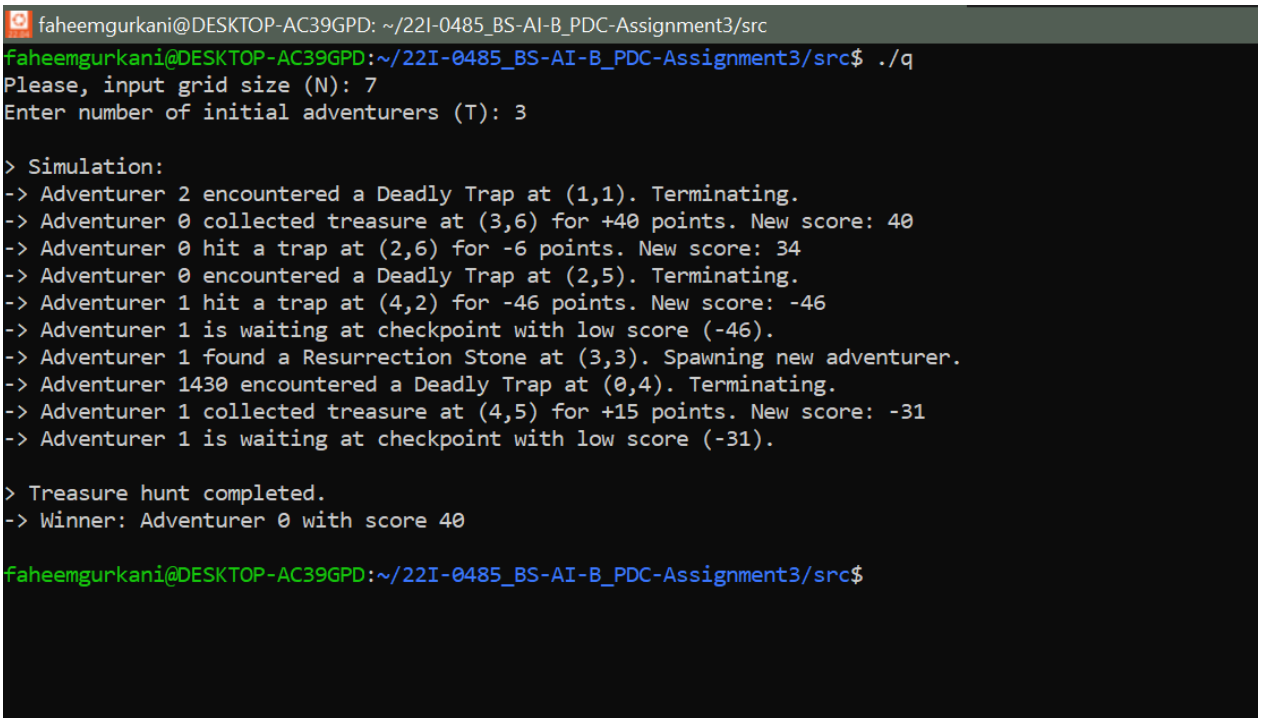
Parallel Treasure Hunt Simulation

- ## Problem Description
  - » Simulate a treasure hunt where adventurers (threads) explore a grid, collect treasures, and avoid traps.
  - » Implement random movements, barriers, and dynamic thread management.

- ## Implementation Strategy and Decisions
  - » **Threading Decisions:** Each adventurer is a separate thread, with movement and scoring handled using OpenMP dynamic scheduling.
  - » **Synchronization:** Barriers are implemented to ensure fair competition.
  - » **Correctness and Verification:** A controlled random seed is used for consistent simulation results.

- Performance Analysis
  - » The performance impact of dynamic scheduling vs. static scheduling is analyzed.
  - » The impact of increasing the number of adventurers (threads) is studied.

- Output



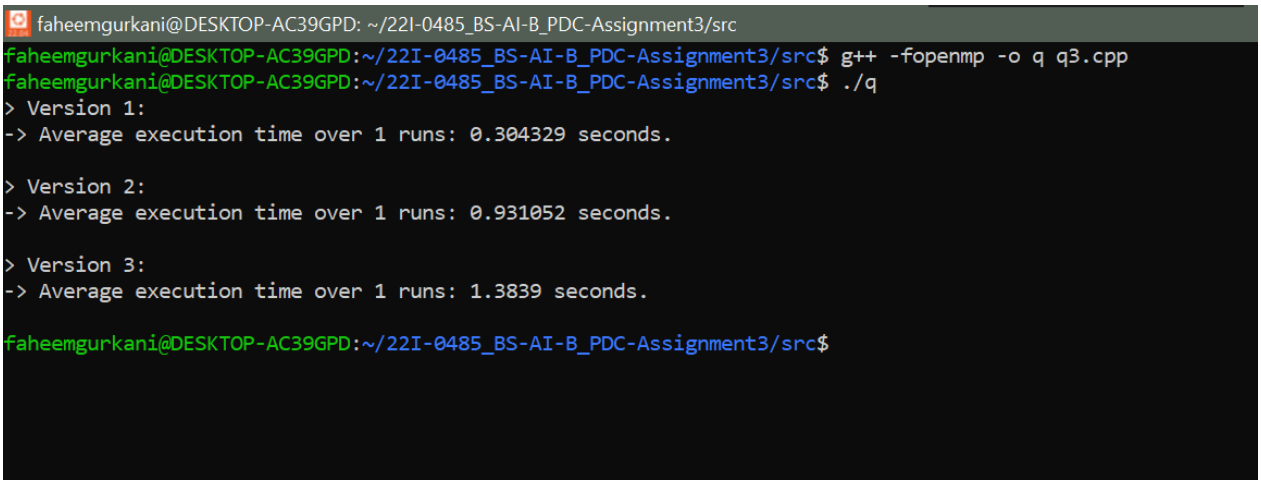
Conway's Game of Life with OpenMP

- Problem Description
  - » Implement a 100x100 grid-based simulation of Conway's Game of Life.
  - » Use OpenMP to parallelize row-wise updates and experiment with static and guided scheduling.

- Implementation Strategy and Decisions
  - » **Data Partitioning:** Each row is updated independently in parallel.
  - » **Threading Decisions:** Static and guided scheduling with different chunk sizes are explored.
  - » **Correctness and Verification:** Outputs are compared against the serial version.

- Performance Analysis
  - » Execution times for serial, static scheduling, and guided scheduling implementations are measured.
  - » Speedup calculations highlight the benefits of parallelization.
  - » The impact of removing the critical section is analyzed.

- Output



Conclusion

This report highlights the following key takeaways:

- **Problem Solving Strategies:** Each task was approached with an optimal parallelization strategy, leveraging OpenMP for efficient execution.
- **Implementation Decisions:** Data partitioning, scheduling strategies, and synchronization techniques were selected to ensure correctness and performance.
- **Performance Analysis:** Parallel implementations showed significant speedups over serial versions, with scheduling strategies impacting efficiency in different ways.