APPENDIX

Author: Dr. Rutal Mahajan
Course: Data Structures & Algorithms (with Parallel Programming using OpenMP)
Term: Fall 2024
Affiliation: Utica University, New York, USA

Examples of Code samples demonstrated in the classrooms for explaining parallelization concepts along with data structures topics

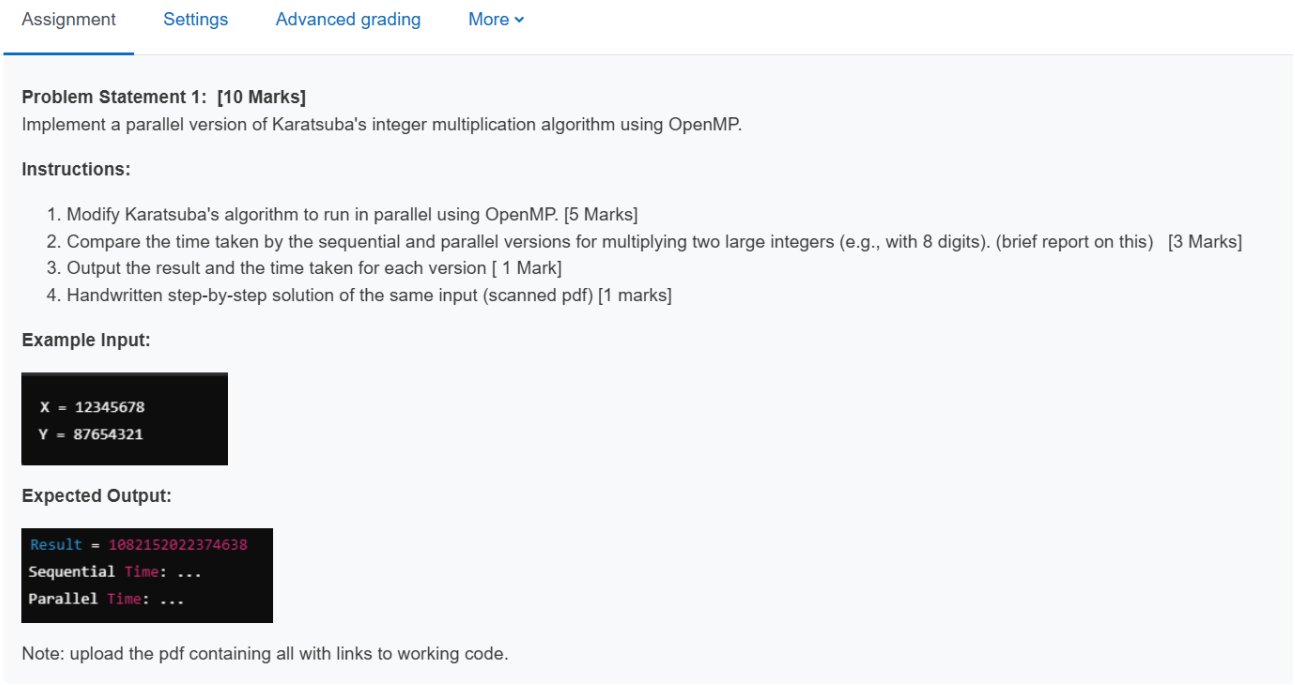
[1] Parallel version of "Hello world" code for class demonstration

[2] Parallelizing loops in C++

[3] Performing sum of 1000000 random numbers in sequential and parallel. The code writes the results in CSV files to analyze the time complexity of sequential code and parallel code.

Examples of assignments used in the classrooms for parallelization concepts along with data structures topics

[1] Assignment: introducing parallel loop concepts with basic data structures Arrays.



[2] **Assignment: Parallelizing Merge Sort**

Learning Objectives

- Understand the divide-and-conquer approach in sorting.
- Learn how to parallelize recursive calls using OpenMP tasks.

Problem Statement : Write a program to sort an array using Merge Sort. Then, parallelize the recursive sort calls using OpenMP tasks. Compare the performance of both versions.

Sequential Code Template (C++)

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

You are required to implement the Merge Sort algorithm in C++ in two versions:

1. A sequential version, using the standard recursive approach.
2. A parallel version, where the recursive calls are parallelized using OpenMP
   you should use **#pragma omp parallel, #pragma omp single,** and **#pragma omp sections** to parallelize the two recursive calls to mergeSort.

[3] Comparing the performance of different sorting algorithms in serial and parallel versions with best , worst and average cases for different sizes of inputs.
Sample Solution code

[4] Assignment: **Linear Search on Large Dataset**

[5] Conceptual Activity – Reasoning About Concurrent Data Structures

Title: *What Could Go Wrong? A Thought Experiment on Concurrent Stack and Queue Operations*

Objective: Introduce students to the conceptual challenges of sharing data structures across multiple "simultaneous" operations, without requiring multithreaded programming knowledge.

Context for Students:

In modern computing, multiple threads or processors often need to access the same data structure simultaneously. Imagine two users accessing a stack or queue at the same time. What problems might arise?

Although we're not covering actual concurrency or synchronization in this course, this short activity will help you start thinking about what happens when operations are interleaved in unexpected ways.

Instructions:

For each scenario below, assume two operations are being attempted "at the same time" (interleaved execution). Your task is to:

- Simulate the step-by-step effect of each operation.
- Identify if a race condition or correctness issue occurs.
- Suggest a real-world analogy or problem that this scenario could resemble**.**

Scenario 1: Two pop() Operations on a Stack

Initial stack: [10, 20, 30]

Thread A and Thread B both try to pop().

Guiding Questions:

- What might happen if both read the top as 30 and try to pop it?
- What value will remain? Will one pop be "lost"?

Scenario 2: One enqueue() and One dequeue() on a Queue

Initial queue: [A, B, C]

Thread A enqueues D, Thread B dequeues an item.

Guiding Questions:

- Could they interfere with each other's read/write pointers?
- What might go wrong with the front/back indicators?

Scenario 3: Stack Used as Shared Undo History

Imagine an undo stack shared by multiple tabs in a text editor. What could go wrong if two users press undo

at the same time?

Expected Outcome for Instructor: Students should be able to:

- Present simulated operation sequences.
- Identify problems (e.g., duplicated/removal conflict, skipped values).
- Gain an intuitive grasp that correct sequencing matters when sharing structures.

Instructor Notes:

- This assignment does not require threads, synchronization, or parallel code.
- It is a reasoning-based activity designed to:
    - Foster awareness of real-world complexity.
    - Provide a segue into why thread safety is important.
    - Encourage discussion about correctness and state consistency, even in a sequential mindset.

Optional Extension (non-assessed):

- Share logs or visuals from actual concurrent code on OnlineGDB to show real anomalies, just for curiosity.

[6] Assignment: **Breadth-First Search (BFS)**

Implement BFS sequentially and analyze opportunities for parallelization

[7] Assignment: **Stack Operations**

Implement basic stack operations and analyze potential parallel issues

[8] Assignment: **Queue Operations**

Implement basic queue operations and explore parallel concepts

[9] **Guidelines and Rubrics for Capstone Project**

Design a project based on the techniques learned (Module 3- Linear data structure and Module 4- non-linear data structure) in the **Data Structures and Algorithms (DSA)** course. The project should solve a real-time application problem by implementing the solution in **C++** with the integration of **parallel programming concepts using OpenMP**.

Guidelines

1. Identify a Real-Time Application: Choose an application where one or more data structures and algorithms learned in the course can be applied effectively.

Choose a Technique: Select an appropriate algorithmic technique (e.g., divide and conquer, graph traversal, Tree Traversal or searching and sorting) based on the application's requirements.

Integrate Parallel Programming:

Identify portions of your implementation that can benefit from parallel processing.

Use OpenMP to parallelize these sections. For instance:

- **Parallel Sorting:** Divide the array into parts and sort them concurrently.
- **Matrix Multiplication:** Split tasks among multiple threads.
- **Graph Algorithms:** Parallelize edge traversal or shortest path computations.

Design and Document Your Solution:

Clearly outline the problem and the proposed solution.
Implement your solution in **C++** with appropriate comments and documentation.

Test and Evaluate:

Test your program with sample inputs and evaluate the performance improvements achieved with parallelization.
Compare the sequential and parallel execution times and document your findings.

Rubrics

Criteria	Excellent (10)	Good (7-9)	Needs Improvement (4-6)	Poor (1-3)
Dataset Selection	Highly relevant, well-chosen dataset that aligns perfectly with the problem.	Relevant dataset, minor misalignment with the problem.	Dataset chosen is generic or less relevant.	No dataset used or irrelevant choice.
Problem Definition	Real-world application clearly defined and highly relevant.	Application identified but moderately relevant.	Application is generic or lacks real-world connection.	Problem poorly defined or irrelevant.
Algorithm Design	Efficient and innovative use of tree/linked list with optimal logic.	Appropriate algorithm, minor inefficiencies.	Algorithm chosen is basic or lacks relevance.	Algorithm missing or incorrectly applied.
Implementation (C++)	Code is well-structured, modular, and well-documented.	Code is functional with minor issues or lacks comments.	Code runs with significant errors or poor readability.	Code is incomplete or non-functional.
Parallel Programming (OpenMP)	Parallelization is highly effective and optimally used.	Parallelization implemented with minor inefficiencies.	Parallelization applied but with limited improvement.	Parallelization missing or incorrectly implemented.
Testing and Results	Thorough testing with detailed comparison of sequential and parallel execution.	Moderate testing with some comparison of sequential and parallel execution.	Minimal testing with limited performance analysis.	Testing incomplete or absent.
Documentation and Reporting	Comprehensive documentation with clear explanations and conclusions.	Adequate documentation but lacking some details.	Limited documentation with unclear explanations.	No or minimal documentation provided.