PCP Lab Assignment 2

Submission Date: 6th Feb 2025

1. Matrix Multiplication Using Multithreading

Goal: Implement matrix multiplication using multithreading to divide the computation among multiple threads. Each thread will compute part of the resulting matrix in parallel. This assignment aims to help you understand how to use threads, share workloads, and handle data synchronization in a multi-threaded environment.

Problem Description: Given two matrices:

- **Matrix A** of size m×n
- **Matrix B** of size n×p

The resulting **Matrix C** will be of size m×p, where each element c[i][j] is computed as:

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot b[k][j]$$

For this assignment, assume m = n = p ≥ 8192.
To speed up the computation, use **multithreading** with the following approaches:

1. **Approach 1: Static Load Balancing (Row-wise)**
   Divide the rows of Matrix A across multiple threads. Each thread computes the corresponding rows of Matrix C.
2. **Approach 2: Static Load Balancing (Cyclic)**
   Use a cyclic approach where rows of Matrix A are assigned to threads in a round-robin fashion (e.g., Thread 0 computes rows 0, t, 2t, and so on, where t is the total number of threads).
3. **Approach 3: Dynamic Load Balancing**
   Use a shared counter to dynamically assign rows of Matrix A to threads during execution. This ensures better load balancing, particularly when some threads finish earlier than others. Synchronization techniques should be used to prevent race conditions.

Input:

- Two matrices A and B (loaded from a file).

Output:

- The resulting matrix C printed to the console and saved to a file.

Language: Implement in **C/C++** using OpenMP.

Report Guidelines

You must submit a report comparing the performance of the three approaches. Follow these steps:

1. **Run Performance Tests:**
    - Vary the number of threads: Use 2, 4, 8, 16, 32, and 64 threads.
    - Vary the size of the matrices: Test different sizes (e.g., 1024×1024, 2048×2048, 4096×4096, and 8192×8192).
2. **Measure Execution Time:**
    - Record the average time taken for each approach using a timer (e.g., gettimeofday() in C/C++, or System.nanoTime() in Java).
3. **Graphical Representation:**
    - **Graph 1:** The x-axis should represent the number of threads (2, 4, 8, 16, 32, 64), and the y-axis should show the average time taken by each thread for each approach.
    - **Graph 2:** The x-axis should represent the matrix size, and the y-axis should show the average time taken by each thread for each approach. Fix the thread as 8 or 16.
4. **Analysis:**
    - Provide a detailed analysis of the results.
    - Explain any anomalies observed, such as why certain thread counts or matrix sizes perform better or worse.

Deliverables

You need to submit the following items in a compressed file named: Prog1Assn1-<rollno>.zip

- **Source Code:**
    - The complete program code for all three approaches.
- **ReadMe File:**
    - A readme.txt file explaining how to compile and execute the program.
- **Report:**
    - A document containing:
        - Graphs comparing the performance of the three approaches.
        - Analysis of results, including explanations for observed performance trends.

2. Prime Number Count using Multithreading

Goal: Develop a **parallel program** using multithreading to count the number of prime numbers between 1 and 10^n, where n is a user-provided integer. The program should leverage multithreading to divide the workload efficiently, improving computation speed.

Problem Description:

- **Prime Checking**: A number x>1 is a prime if it is not divisible by any number 2 to sqrt{x}.

To speed up the computation, use **multithreading** with the following approaches:

1. **Approach 1: Static Load Balancing (Equal Range Division)**
   - Divide the range $[1,10^n]$ equally among t threads.
   - Each thread processes its assigned subrange and counts the primes independently.
2. **Approach 2: Static Load Balancing (Cyclic Distribution)**
   - Use a cyclic approach where threads work on non-contiguous indices in a round-robin fashion.
   - For example, Thread 0 computes primes for indices 1, t+1, 2t+1,..., and so on.
3. **Approach 3: Dynamic Load Balancing**
   - Use a shared counter to dynamically assign indices to threads during execution.
   - Each thread retrieves a unique index from the shared counter and checks if the number is prime.
   - Synchronization is required to ensure threads operate on distinct indices without race conditions.

Input:

- An integer n, entered by the user. n should be large (e.g., n≥6).

Output:

- The total number of prime numbers between 1 and 10^n.

Language: Implement in **C/C++** using OpenMP.

Report Guidelines

Performance Testing

- Run all three approaches multiple times with varying parameters:
  1. **Number of Threads**: Test with 2, 4, 8, 16, 32, and 64 threads.
  2. **Input Size (n)**: Vary n (e.g., 10^6, 10^7, 10^8).

Graphical Comparison

1. **Graph 1**:

- **X-axis**: Number of threads (2, 4, 8, 16, 32, 64).
- **Y-axis**: Average execution time (in milliseconds) taken by each thread for each approach.

2. **Graph 2**:
   - **X-axis**: vary the size of n.
   - **Y-axis**: Average execution time (in milliseconds) taken by each thread for each approach. Fix the thread as 8 or 16.

Analysis

- Compare and explain the performance of the three approaches.
- Discuss any anomalies or unexpected trends, such as diminishing returns with higher thread counts or large input sizes.

Deliverables

You need to submit the following items in a compressed file named: Prog2Assn1-<rollno>.zip

- **Source Code:**
  - The complete program code for all three approaches.
- **ReadMe File:**
  - A readme.txt file explaining how to compile and execute the program.
- **Report:**
  - A document containing:
    - Graphs comparing the performance of the three approaches.
    - Analysis of results, including explanations for observed performance trends.

Submission Details

- **Deadline**: Feb 6, 2025, 11:30 PM
- **Submission Platform**: Blackboard