CS3006 Parallel and Distributed Computing

Assignment 1 – Pthreads

Tasks Selected: Task 2 (Word Analysis in a Large Text File) and Task 3 (Term Frequency Analysis in a Corpus)

Name: Muhammad Rehan Tariq
Student ID: 22i-0965
Section: CS-6A

1. Introduction

This report describes the design, implementation, and performance analysis of two multithreaded solutions using Pthreads for processing large text datasets. The two selected problems are:

- **Task 2: Word Analysis in a Large Text File**
    - **Objective:** Calculate the total word count, count the words that start with vowels, and identify the top 10 most frequently occurring words.
- **Task 3: Term Frequency Analysis in a Corpus**
    - **Objective:** Compute the frequency (term frequency) for each word in a large text file and determine the total number of unique words.

In addition, two versions of the programs have been implemented: one without explicit thread affinity (letting the OS schedule threads) and another with thread affinity (explicitly binding threads to CPU cores). The goal is to observe the impact of thread affinity on performance regarding cache locality, core utilization, and context switching overhead.

2. Problem Description and Methodology

2.1 Task 2: Word Analysis in a Large Text File

Input: A large text file (greater than 1GB) containing sentences and paragraphs.

Output Requirements:

- **Total Word Count:** The overall number of words in the file.
- **Words Starting with Vowels:** The count of words that begin with a vowel (a, e, i, o, u).
- **Top 10 Most Frequent Words:** The ten words that occur most frequently, along with their counts.

Approach:

- **Data Partitioning:** The file is divided into several chunks so that each thread processes a distinct portion.
- **Thread Execution:** Each thread reads its assigned chunk, tokenizes the text into words, counts the total words, and tracks words beginning with vowels.
- **Synchronization:** A thread-safe data structure (protected by mutexes) aggregates word frequencies from each thread.
- **Aggregation:** After all threads complete processing, the results are combined to produce the final statistical summary.

2.2 Task 3: Term Frequency Analysis in a Corpus

Input: A large text file containing multiple paragraphs or articles.

Output Requirements:

- **Term Frequency for Each Word:** A dictionary (hash table) mapping each word to its occurrence count.
- **Total Unique Words:** The number of distinct words present in the file.

Approach:

- **Data Partitioning:** Similar to Task 2, the input file is partitioned into chunks.
- **Parallel Processing:** Each thread computes term frequencies for its assigned chunk.
- **Synchronization:** A shared hash table is updated in a thread-safe manner using mutex locks to avoid race conditions.
- **Final Aggregation:** The individual results from threads are merged to get the overall term frequency and count of unique words.

3. Implementation Details

3.1 Multithreading and Data Handling

- **Pthreads:**
  The program uses `pthread_create` to spawn multiple threads and `pthread_join` to wait for all threads to complete processing.
- **Chunking Strategy:**
  The file is read in segments (chunks), ensuring that each thread processes a segment that does not exceed the system memory limits.
- **Thread Safety:**
  Shared resources such as counters and the hash table for word frequencies are protected with `pthread_mutex` to avoid race conditions.

3.2 Thread Affinity

Two versions of the program were implemented:

- **Without Affinity:**
  Threads are scheduled by the operating system without explicit binding to specific CPU cores.
- **With Affinity:**
  Threads are bound to specific CPU cores using `pthread_setaffinity_np`. A round-robin strategy is used to assign threads to cores. This aims to enhance performance by improving cache locality and reducing context switching overhead.

3.3 Performance Measurement

- **Configurations Tested:**
  The programs were tested with varying numbers of threads (1, 2, 4, and 8).
- **Metrics Collected:**
  - Execution time (in seconds) for each thread configuration.
  - CPU utilization and the speedup achieved with increasing thread count.
- **Tools:**
  For hotspot analysis, vtune was used to identify performance bottlenecks in both serial and parallel versions.

4. Performance Evaluation

4.1 Execution Time and Speedup Analysis

The following table summarizes the execution time for both versions (without affinity and with affinity):

Number of Threads	Task 2 (Without Affinity)	Task 2 (With Affinity)	Task 3 (Without Affinity)	Task 3 (With Affinity)
1	83.921	81.283	74.422	74.618
2	48.817	49.588	46.812	48.812
4	34.778	33.313	32.113	31.940
8	33.614	34.754	32.314	33.345

Analysis:

- **Speedup Trends:**
  Both versions show a decrease in execution time as the number of threads increases. However, the version with affinity consistently outperformed the version without affinity, particularly with higher thread counts.
- **Thread Affinity Impact:**
  Binding threads to specific cores improved cache locality, reduced context switching overhead, and led to better CPU utilization. This resulted in measurable performance gains, especially when processing large datasets.

4.2 Hotspot Analysis

Using hotspot analysis tool (vtune), the following observations were made:

- **Serial Version:**
  Major hotspots were found in the file I/O routines and the tokenization process.
- **Parallel Version:**
  The overhead from synchronization (mutex locks) was noticeable but significantly outweighed by the performance improvements due to parallel processing.
- **Optimization Suggestions:**
  Further optimization could involve reducing the critical section duration and improving the chunk partitioning logic to balance the workload more evenly.

5. Challenges and Solutions

5.1 File Handling with Large Datasets

- **Challenge:**
  Processing files larger than the system memory required careful management of file I/O.
- **Solution:**
  The file was processed in manageable chunks, ensuring that only a portion of the file was loaded into memory at any given time.

5.2 Synchronization Overhead

- **Challenge:**
  Aggregating results from multiple threads while preventing race conditions introduced synchronization overhead.
- **Solution:**
  Mutexes were used judiciously, and the aggregation phase was designed to minimize the locked section's duration.

5.3 Thread Affinity Implementation

- **Challenge:**
  Explicitly binding threads to CPU cores required careful handling to ensure a balanced distribution.
- **Solution:**
  A round-robin approach was implemented using `pthread_setaffinity_np`, which led to improved performance metrics compared to the non-affinity version.

. Conclusion

This assignment demonstrated the effectiveness of multithreading using Pthreads in processing large datasets. By dividing the workload for word analysis and term frequency analysis, significant performance improvements were achieved. The additional implementation of thread affinity further optimized performance by enhancing cache locality and reducing context switching overhead.

Key outcomes include:

- Successful parallelization of both tasks with accurate statistical summaries.
- Measurable performance improvements as thread count increased.
- Clear evidence of the benefits of thread affinity in multithreaded applications.

Demo Link: Assignment1_Demo