Assignment 4: Multi-threading

Razi Haider

03 December 2023

1 Introduction

The objective of the file_processor_singlethreaded.c and file_processor_multithreaded.c implementations is to process a large dataset efficiently, with a focus on understanding the impact of single-threaded and multi-threaded approaches. The programs aim to read a dataset from a file, perform calculations such as sum, minimum, and maximum values, and measure the time taken for these operations. The single-threaded version provides a baseline for comparison, while the multi-threaded version leverages concurrent processing using POSIX Threads API to potentially enhance performance on multi-core systems.

2 Single Threaded Implementaion

2.1 Overview

The single-threaded implementation adopts a straightforward, sequential approach to dataset processing. Key operations include reading the dataset from a file, calculating the sum, minimum, and maximum values using a single thread, and measuring the time taken for these operations over multiple iterations.

2.2 Strengths

- **Simplicity:** The single-threaded approach is straightforward to understand.
- **Baseline Measurement:** Provides a baseline for comparison with the multi-threaded version.
- **Suitability:** Suitable for smaller datasets or tasks that may not benefit significantly from parallelization.

2.3 Weakness

- **Limited Parallelism:** Unable to leverage the potential parallelism offered by multi-core systems.
- **Performance Impact:** May face challenges in handling large datasets efficiently due to the lack of concurrent processing.

2.4 Code Analysis

- **Data Array:** A data array that Stores the dataset dynamically. It is dynamically allocated based on the dataset size and it can be manually resized if needed during data reading.

3 Multi Threaded Implementaion

3.1 Overview

The multi-threaded implementation employs the POSIX Threads API to process a large dataset concurrently. Key operations include dividing the dataset among worker threads, calculating the sum concurrently using

multiple threads, calculating the minimum and maximum values concurrently, and measuring the time taken for these operations over multiple iterations.

3.2 Strengths

- **Parallel Processing:** Leverages multiple threads for concurrent processing, potentially improving performance.
- **Thread Safety:** Implements mutexes for synchronization to ensure thread safety during shared variable updates.
- **Scalability:** Better scalability for large datasets on multi-core systems.

3.3 Weakness

- **Complexity:** Introduces complexity due to synchronization mechanisms and thread management.
- **Overhead:** Depending on the dataset characteristics, there might be overhead in thread creation and synchronization.

3.4 Code Analysis

- **Data Array:** A data array that Stores the dataset dynamically. It is dynamically allocated based on the dataset size and it can be manually resized if needed during data reading.
- **Mutex:** It ensures thread safety during shared variable updates. It Locks and unlocks to control access to critical sections.
- **Thread Data Structure:** It stores data for each thread, including a pointer to the data array and the start and end indices of the portion of the data array the thread will process.

4 Timing Comparison

For the tiny dataset, the single-threaded program outperforms the multi-threaded counterpart due to the overhead of thread creation, context switching, and synchronization outweighing the benefits of concurrent execution. The average elapsed time remains consistent up to 10 threads, after which there is a noticeable increase, indicating that thread management overhead supersedes execution time for such a small dataset. Similarly, in the case of small datasets, the single-threaded program performs better, with the average elapsed time starting to increase after 8 threads, showcasing the growing benefits of multi-threading with larger datasets. The medium dataset exhibits improved multi-threaded performance, with the average elapsed time gradually increasing as the number of threads rises, demonstrating good scalability. Beyond the optimal thread count aligned with the system's core capabilities, performance starts to degrade. Interestingly, the large dataset experiences a performance boost after 50 threads, remaining stable up to 100 threads before degradation, suggesting potential factors such as better cache utilization or underlying optimizations influencing the specific workload distribution.

5 Conclusion

Both implementations use a dynamic array to store the dataset, with the multi-threaded version introducing additional structures such as mutexes and a thread data structure. The choice of dynamic arrays allows for flexibility in handling datasets of varying sizes. The introduction of mutexes in the multi-threaded version ensures thread safety during concurrent operations, although it adds some complexity to the code. Understanding the characteristics and trade-offs of these data structures is essential for optimizing the performance of each implementation.