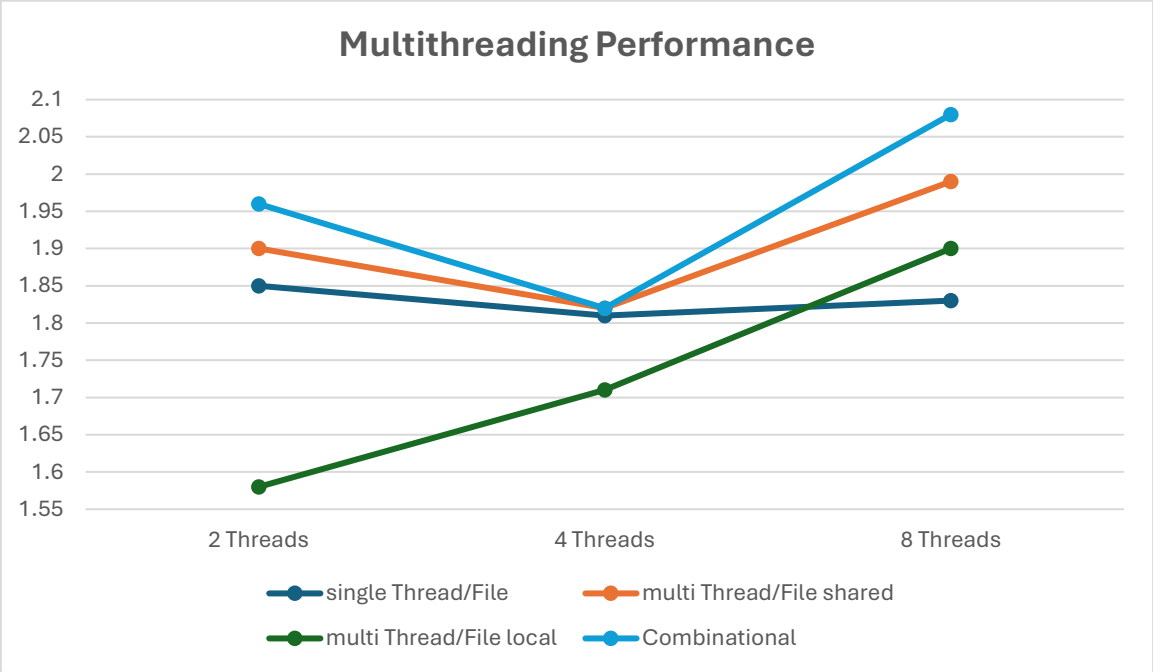
Mohammadreza Naziri

40030505

Analysis of Multithreading Performance



	single Thread/File	multi Thread/File shared	multi Thread/File local	Combinational
2 Threads	1.85	1.9	1.58	1.96
4 Threads	1.81	1.82	1.71	1.82
8 Threads	1.83	1.99	1.9	2.08

Introduction

This document analyzes the performance of three multithreading approaches applied to processing input files of significantly varying sizes, ranging from less than 1 MB to 900 MB. The performance of these approaches is compared to the sequential execution time of 1.93 seconds.

Sequential Baseline

In sequential mode, the task takes 1.93 seconds. Interestingly, executing the task sequentially using a single thread resulted in a slightly faster execution time of 1.83 seconds, likely due to system state variations rather than an inherent improvement.

Approach 1: One Thread Per File

In this approach, each thread processes a different file. For 2, 4, and 8 threads, each thread processes 4, 2, and 1 files, respectively. The results indicate that this approach is ineffective for this problem due to the following reasons:

- **Load Balancing:** Uneven file sizes lead to poor load distribution.
- **Granularity:** Granularity is not uniform because some files are significantly larger than others.

The execution time for this approach is dictated by the thread processing the largest file, combined with the overhead of thread creation. Since the largest file's size closely matches the total workload, the approach does not yield substantial performance benefits, producing results similar to the sequential execution.

Approach 2: Multiple Threads Per File

In this approach, multiple threads process a single file by dividing it into chunks. Threads work on different chunks simultaneously, potentially reducing the processing time of each file. However, the overhead of thread synchronization and joining must be considered.

This method can be implemented using two memory management schemes:

1. **Shared Memory:** Threads share the same memory space for processing.
2. **Local Memory:** Each thread works on a copy of its chunk, using its own local memory.

Profiling Results:

- **Shared Memory:** Using 4 threads per file yielded the best result. This improvement is likely due to a balance between thread creation/joining overhead and the dependencies between threads. However, memory interlock may be contributing to the performance degradation in shared memory, as the need for synchronization between threads can lead to contention and delays. This could be a factor in shared memory having worse performance compared to local memory in some cases.
- **Local Memory:** The best result was achieved with 2 threads per file, which also yielded the fastest overall execution time. This result is likely due to reduced data copying costs and lower thread overhead.

Key Observations:

- Unsuitable implementation can lead to worse performance than the sequential version.
- Shared memory slows down due to contention and locking.

Approach 3: Combination of Both Methods

This approach combines the previous two methods, where each thread processes a file, and each file is processed by multiple threads. For 8 input files, we tested configurations of 2, 4, and 8 threads per file. The best result was achieved with 4 threads per file.

Solution

A potential solution could involve splitting files into varying numbers of chunks. Larger files would be divided into more chunks, ensuring that all chunks are approximately the same size. This approach would increase concurrency and significantly improve efficiency.

General Observations

Due to the significant size differences between files, ranging from less than 1 MB to 900 MB, parallelizing the task effectively is challenging. Multithreading is generally beneficial for dividing tasks into smaller parts and executing them concurrently. However, this requires:

- Tasks of similar size for efficient load balancing.
- Minimal thread management overhead.

In cases like this, where file sizes vary widely, multithreading can lead to idle threads and increased overhead, reducing its effectiveness.

How to Run the Profiling for each implementation:

1. **Compile the Code:** Use the run.sh script to compile the source files (main.c and utilities.c) with profiling enabled.
2. **Run the Program:**
   Execute the compiled program images_processing to generate the profiling data.
3. **Generate the Profiling Report:**
   Use gprof to analyze the generated profiling data (gmon.out) and save the results to report.txt.
4. **View the Results:**
   Display the profiling results from report.txt.