Simon Fraser University

Gregory Baker

CMPT 295 Mini Project

CMPT 295 Mini Project Report

Saif Ali

301459324

August 2nd, 2024

Introduction

The code I am evaluating for my mini-project consists of implementations of the sorting algorithms: merge sort, quick sort, and timsort (which is a combination of merge sort and insertion sort). All implementations have been sourced from GeeksforGeeks.org. This evaluation is conducted on a Linux system with an x86-64 CPU architecture. The specific CPU is an Intel i5-9600k. The aim is to analyze how the C++ compiler optimizes these algorithms, how their running times are affected by various inputs, their memory and CPU usage, and whether they can benefit from SIMD operations.

Evaluations

To evaluate the algorithms, I employed several tools discussed in lectures and labs. To examine assembly conversion, SIMD operations, and optimization, I used godbolt.org to analyze the code with varying optimization levels and view the corresponding assembly output. For measuring running time with different inputs, I utilized clock_gettime for nanosecond resolution timing. And for analyzing cache memory performance and branch predictability, I relied on perf.

Assembly Conversion, Optimization, and Vector Instructions

To evaluate assembly conversion and optimization, I used godbolt.org to compare the assembly code of each algorithm at various optimization levels (-O0, -O1, -O2, -O3).

Merge Sort compiled into relatively straightforward assembly code, frequently using the stack and accessing memory. The differences between optimization levels were minimal in terms of operation types. However, higher optimization levels like -O2 and -O3 resulted in longer assembly code due to additional edge case handling, while maintaining similar calls and stack accesses. Neither optimization level employed SIMD operations.

Quick Sort also compiled into relatively simple assembly code. The -O3 optimization level is particularly noteworthy as it utilizes SIMD operations and the %xmm and %ymm registers, significantly enhancing performance and efficiency. It also saves more registers onto the stack for its operations.

Timsort once again compiled into clear and straightforward assembly code. The main difference between the higher and lower optimization levels is that the higher optimization enhances performance by simplifying memory management and loop handling. This is achieved by reducing stack operations and utilizing more registers. Despite these improvements, SIMD operations were not used in the optimization levels.

Running Time

To evaluate the running time of the sorting algorithms, I employed a clock_gettime program that measures execution time. Each algorithm was tested five times under various conditions, including sorted, reverse-sorted, random, partially-sorted, and many duplicate values, with an array size of 500,000 elements.

For merge sort, the running time was fastest for already sorted data, whether in normal or reverse order, with a performance of approximately 20ms. The running time increased slightly for partially sorted data and data with many duplicate values, averaging around 30ms. The slowest performance was observed with random data, with a running time of about 40ms.

For quick sort, the fastest running times were observed with random and partially sorted data, averaging around 50ms. The running time increased slightly for already sorted data, with an average of about 60ms. However, the performance significantly degraded with reverse-sorted and many duplicate values, with running times ranging from 2000ms to 3000ms!

For Timsort, the fastest running times were observed with sorted and reverse-sorted data, averaging around 6ms. Partially sorted and many duplicate values resulted in slightly slower times, averaging about 12ms. Random data had the slowest performance, with an average running time of 20ms.

Cache Memory Performance

I used the perf tool to analyze cache hits and misses of the sorting algorithms with different random array sizes to evaluate their performance when the data fits or doesn't fit in cache memory (L1, L2, L3 caches).

Merge sort shows a clear relationship with cache levels. For arrays smaller than the L1 cache (192 KiB), it sorts 45,000 values in about 3.91 ms with low L1 cache misses. When the array size exceeds L1 but fits within L2 (1.5 MiB), the runtime increases to 5.20 ms for 60,000 values, with higher L1 and L3 cache misses. For arrays larger than the L3 cache (9 MiB), it takes 270.01 ms to sort 3,000,000 values, indicating less efficiency as the dataset grows beyond the cache capacity.

Quick sort's performance varies significantly with cache size. For arrays smaller than the L1 cache, it sorts 45,000 values in 3.60 ms, with slightly higher L1 cache misses than merge sort. When the array size exceeds L1 but fits within L2, the runtime increases to 5.44 ms for 60,000 values. For arrays larger than the L3 cache, it takes 271.62 ms for 3,000,000 values, showing increased L1 and L3 cache misses and a significant drop in performance.

Timsort maintains efficiency across different cache levels. For arrays smaller than the L1 cache, it sorts 45,000 values in 3.62 ms with low L1 cache misses. When the array size exceeds L1 but fits within L2, it sorts 60,000 values in 5.50 ms while maintaining low cache miss rates. For arrays larger than the L3 cache, it takes 270.12 ms for 3,000,000 values, showing relatively low cache miss rates compared to merge sort and quick sort.

Branch Predictions

I used the perf tool to analyze branch misses of the sorting algorithms with different sorted, random, and reverse-sorted arrays to evaluate their branch predictability.

Merge sort maintains consistent branch predictability across data types. For sorted data, it incurs about 234,000 branch misses (2.45%). With reverse sorted data, branch misses slightly increase to around 239,000 (2.40%). For random data, branch misses rise to about 287,000 (2.99%). The total number of branches remains about constant throughout. Overall, merge sort's branch predictability is stable, though slightly less efficient with random data.

Quick sort's branch predictability varies with data order. For sorted data, it experiences around 302,000 branch misses (1.71%). In reverse sorted data, branch misses increase to about 313,000 (0.58%) with a substantial increase in branches from 17 million for sorted data to 54 million. With random data, it sees 267,000 branch misses (6.47%) with a substantially lower number of branches at 4 million. This variability shows quick sort's efficiency is highly dependent on data order, with slightly worse performance but substantially fewer branches on random data.

Timsort exhibits high branch predictability across data types. For sorted data, it records about 228,000 branch misses (6.15%). For reverse sorted data, branch misses rise slightly to around 233,000 (5.73%). With random data, branch misses are lower at about 213,000 (5.03%). Similar to merge sort, the total number of branches remains fairly constant. Timsort's branch predictability is efficient and stable, particularly with random data.

Conclusion

In evaluating merge sort, quick sort, and timsort, it was found that each algorithm exhibits distinct performance characteristics. Quick sort benefits from SIMD operations at higher optimization levels, significantly enhancing its performance and efficiency, while merge sort and timsort do not. In terms of running time, merge sort is the fastest for sorted data, quick sort for random data, and timsort is faster than both of them overall. Cache performance analysis revealed that merge sort and quick sort suffer from increased cache misses and reduced efficiency as data size grows beyond cache capacity, while timsort maintains relatively low cache miss rates and consistent performance across different cache levels. Merge sort maintains consistent branch predictability across different data types but is less efficient with random data. Quick sort shows a lot of variability in branch predictability and performance due to the data order. Timsort demonstrates high and stable branch predictability across all data types, with particularly efficient performance on random data. One of the most interesting findings in this evaluation is quick sort's weakness when presented with reverse-sorted data, taking an astounding amount of time, branches, and memory to complete the task. Timsort emerges as the best overall performer, demonstrating superior sorting capabilities with an architecture similar to both merge sort and quicksort. Overall, these findings highlight the strengths and weaknesses of each algorithm in different scenarios, providing valuable insights into their optimization and efficiency for specific applications.