Parallelizing Merge Sort: Serial vs. OpenMP vs. POSIX Threads

Jonathan McKesey, Julian Cruzet

April 16, 2025

Abstract

This project explores the parallelization of the merge sort algorithm through three different approaches: serial, OpenMP, and POSIX threads. Merge sort, a divide-and-conquer algorithm, is known for its stable and efficient sorting properties in its serial form, but its performance can be significantly improved in parallel computing environments. In this study, we compare the execution time and scalability of merge sort when parallelized using OpenMP, a high-level parallel programming model, and POSIX threads, a lower-level threading model. The objective is to assess the performance gains achieved by each parallelization technique and understand their respective strengths and limitations in handling large datasets. The results will provide insights into the trade-offs between ease of implementation, portability, and performance in parallel sorting algorithms.

1 Implementation Details

1.1 Sequential Merge Sort

The sequential implementation follows a classic recursive merge sort approach. A major challenge during the implementation was managing the memory allocation for large arrays. Initially, recursive calls push data on the stack, resulting in segmentation faults for array sizes larger than three million. This issue was resolved by switching to heap allocations, which slightly slowed down the allocation step but preserved the overall intent of benchmarking the sorting performance.

1.2 OpenMP

OpenMP is a high-level API for multi-platform shared-memory parallel programming in C. The OpenMP version uses `#pragma omp parallel` to distribute recursive calls across multiple threads. To fine-tune performance, the minimum task size (min_task) was chosen to be 100'000 after empirical testing, this number was chosen because sequentially the algorithm performs better on average hence it offered the best trade-off between overhead and concurrency.

1.3 POSIX Threads(pthread)

While developing the pthread version, an initial attempt was made to build a custom threadpool implementation to manage worker threads. However, due to the complexity and debugging overhead, this approach was abandoned in favor of limiting thread creation based on recursion depth. This simple model allows manageable concurrency while preventing excessive thread spawning and synchronization issues.

2 Parallel Architecture Methodology

In designing the parallel architecture, we made the following key considerations:

- **Thead Pool Management:** Instead of manually building a thread pool using pthreads, OpenMP's built-in thread pool was used. OpenMP initializes its thread pool on the first `#pragma omp parallel` call and reuses the threads efficiently.
- **Recursion Depth Cap:** For the pthread implementation, thread creation was limited by recursion depth to prevent thread explosion, memory exhaustion, and performance degradation. This also ensured predictable resource usage.
- **Minimum Task Size:** A cutoff threshold of a size of 100'000 was determined experimentally to yield the best performance. This avoided unnecessary parallelism overhead for small sub arrays

- **Stack vs. Heap Allocation** Given the large array sizes, heap allocation were used for recursive subarrays to avoid stack overflow, a common issue when pushing large data onto the call stack in C.

These decisions were influenced by both hardware constraints and the desire to keep implementation complexity manageable. OpenMP's simplicity allowed for more stable scaling, while pthreads offered unrestricted freedom into lower-level control mechanisms.

2.1 Parallel Performance limitations

A key bottleneck in the parallel merge sort algorithm lies in the merge step. While the divide step can be parallelized, the merge step requires careful coordination and involves memory-intensive operations that are not trivially parallelizable. As a result despite using up to all 16 logical cores available, the parallel versions achieves only around a 2 times speedup compared to the sequential baseline. This phenomenon aligns with **Amdahl's Law**, which states that the maximum speedup of a program using multiple processors is limited by the serial portion of the program. Since merging is inherently sequential in this implementation, it constrains overall scalability.

3 Benchmarking setup

All benchmarks were conducted on the following hardware and software configuration:

- 16-core AMD Ryzen 5 3600
- 16 GB RAM (8 GB allocated to WSL)
- WSL 2.0 environment
- GCC with -O3 optimizations

4 Performance Analysis

Multiple benchmarks were conducted using array sizes ranging from 1,000 to 50,000,000. Each configuration was executed 10 times, and the average elapsed time was recorded.

4.1 Execution Time Comparison

As shown in the figure below, OpenMP consistently outperformed pthreads and the sequential version for large datasets.

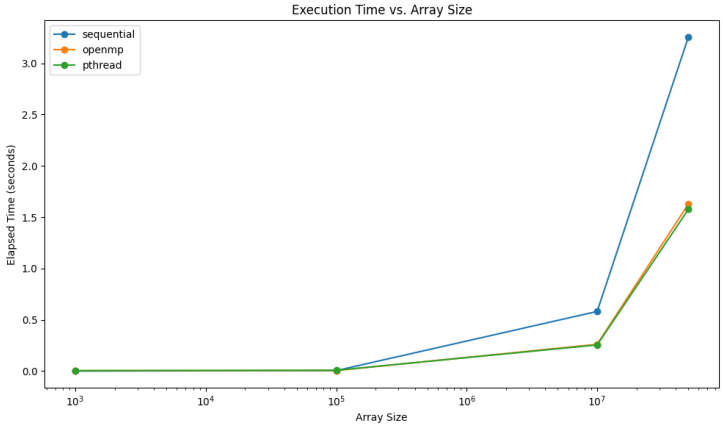

Figure 1: Elapsed Time vs Execution Method for Various Array Sizes

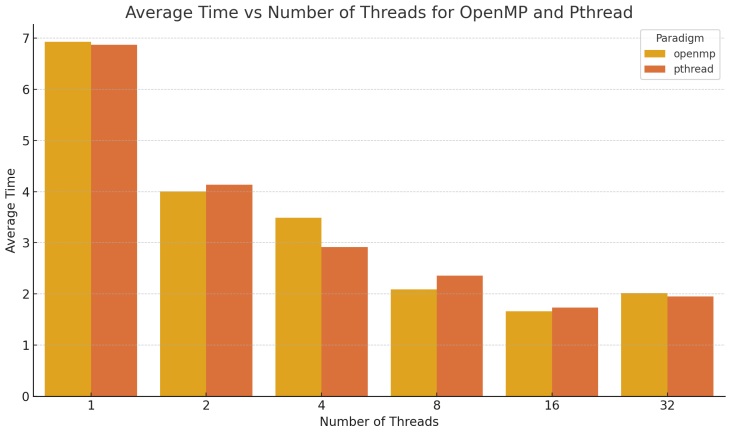

Figure 2: Execution Time vs # of Threads, 50M element Array

4.2 Thread Count Scaling

The graph illustrates the effect of increasing thread count on performance. OpenMP demonstrates good scalability up to the number of physical cores.

5 Lessons Learned

- Heap allocation is necessary for large arrays to avoid stack overflow.
- Optimal task size is crucial; an array of size 100'000 minimized execution time for OpenMP.

- Writing a custom thread pool from scratch in C is challenging and may not be worth the effort in a constrained academic setting.
- OpenMP offers easier syntax, and better portability.
- Pthreads provide more control but require meticulous management of thread lifecycle and shared memory.
- The merge step remains a significant bottleneck in parallel merge sort.
- Amdahl's Law is an important consideration in evaluating the scalability of parallel algorithms.

6 Conclusion

Parallel merge sort significantly improves performance on large datasets when implemented with OpenMP or pthreads. OpenMP provides a simpler, more stable, and more scalable solution, making it preferable for general-purpose use. Pthreads, while powerful, introduce complexity that may not yield substantial performance gains without fine-tuned management. These findings underscore the importance of choosing the right parallel framework based on project constraints and programmer experience. The results also reaffirm that while parallelization improves throughput, serial bottlenecks such as merging must be addressed to realize full scalability.

7 Bonus: Radix Sort

Given the nature of the input array being uniforemely distributed 32-bit integers, radix sort provides an excellent fit due to its ability to process data based on individual digits. The linear time complexity of radix sort makes it suitable for large datasets. As an bonus to this project, we implemented both a sequential radix sort and a parallelized version utilizing OpenMP to asses the performance improvements offered by parallelism and a better suited algorithm.

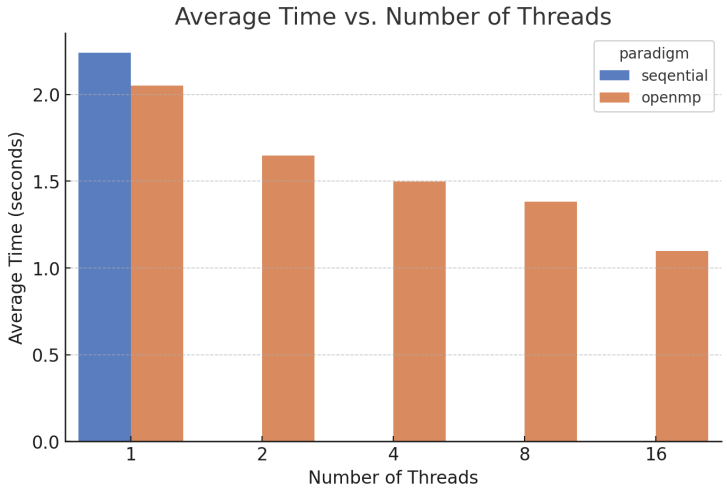

Figure 3: Execution Time vs # of Threads, 50M element Array

From the figure, it is evident that the parallelization of radix sort with OpenMP significanlty reduces execution time. Which 16 threads, the parallelized version of radix sort nearly doubles in speed compared to the sequential approach making it highly efficient solution for large-sacle data sorting tasks. Comparing radix sort to other algorithms, such as merge sort, we observe that radix sort outperforms merge sort in this specific case due to the large volume of integers being sorted. Howeverm it's important to note that radix sort has limitations. For example, it may not perform as well with other data types such as long integers, floating-pint values or strings, where it can suffer from reduced efficiency and memory overhead due to the need for multiple passes over the data.