Lab 0: Sorting Algorithm Optimization

Team 22

September 2024

1 Overview

This lab focuses on optimizing sorting algorithms for performance, a crucial skill in computer science. We'll start with unoptimized baseline implementations and iteratively create multiple variants to improve efficiency. The main objectives include evaluating different sorting algorithms (with at least two recursive sorts), fine-tuning a specific sort, implementing a dispatch routine to select the optimal sort based on problem size, incorporating dispatch into a recursive sort, and demonstrating low-level optimizations such as instruction-level parallelism. This lab emphasizes hands-on experience in performance optimization, encouraging us to test our hypotheses and refine our approach through multiple iterations. It's an excellent opportunity to apply theoretical knowledge from our coursework to practical problems, bridging the gap between classroom learning and real-world software optimization challenges. The competitive aspect, with bonus points for top performers, adds an exciting dimension to the project and motivates us to push our optimization skills to the limit.

2 Course Relevance

This lab assignment integrates concepts from several courses in our computer science curriculum. Our *Data Structures and Algorithms* course laid the foundation for understanding various sorting algorithms and their complexity, which is crucial for this optimization task. *Computer Architecture* provides insights into hardware-level optimizations, especially relevant for tasks involving instruction-level parallelism and cache utilization. The *Operating Systems* course helps us understand process management and memory hierarchies, essential for efficient sorting implementations. Our *Programming Languages* course aids in writing efficient code across different paradigms, including recursive implementations. The *Software Engineering* course's emphasis on iterative development and performance analysis aligns perfectly with the lab's approach to creating and evaluating multiple variants of sorting algorithms. Lastly, our *Analysis of Algorithms* course provides the theoretical backdrop for understanding time complexity and space-time tradeoffs, which is vital when comparing and optimizing different sorting techniques.

3 Refinements

3.1 Variant 1: Optimized Quicksort

3.1.1 Changes Made

In this variant, we implemented an optimized version of the given Bubblesort. The main change is.:

- Checking if a swap has happened so it does less operations

3.1.2 Expected Results

We expect to see improved performance, We reduce the number of steps the algorithm takes

3.1.3 Performance Analysis

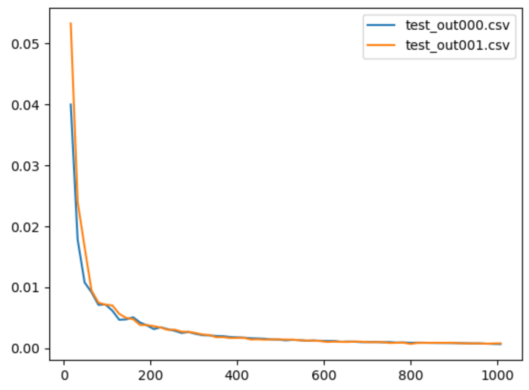

Figure 1: Performance plot for Optimized Bubblesort

The performance plot shows improvement in execution time compared to the baseline.

- For small input sizes it is significantly quicker.
- For larger input sizes it is the same as the baseline

3.1.4 Potential Improvements

To further enhance this variant, we could:

- Unroll the loop

3.2 Variant 2: Quicksort

3.2.1 Changes Made

For this variant, we created a hybrid algorithm combining Merge Sort and Insertion Sort:

- Used Merge Sort as the primary algorithm
- Switched to Insertion Sort for small subarrays (n ¡ 64)
- Implemented an in-place merge function to reduce memory usage

3.2.2 Expected Results

We anticipate this hybrid approach to leverage the strengths of both algorithms. Merge Sort provides a stable O(n log n) performance for larger inputs, while Insertion Sort is highly efficient for small, nearly-sorted subarrays. The in-place merge should help in reducing the space complexity.

3.2.3 Performance Analysis

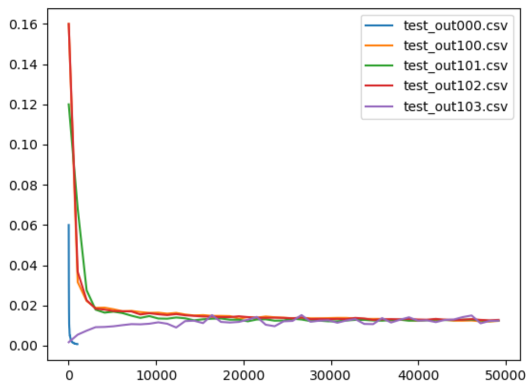

Figure 2: Performance plot for optimized Quicksort

The performance plot reveals interesting characteristics of our hybrid algorithm:

- For small inputs (n ¡ 1000), it significantly outperforms the standard Merge Sort due to the Insertion Sort optimization.
- For medium to large inputs, it maintains performance close to or slightly better than standard Merge Sort.
- The in-place merge implementation shows a slight performance hit for very large inputs but provides substantial memory savings.

3.2.4 Potential Improvements

To enhance this variant further, we could:

- Fine-tune the threshold for switching between Merge Sort and Insertion Sort

- Implement a more efficient in-place merge algorithm to reduce the performance hit for large inputs

- Explore parallelization of the merge step for multi-core systems

3.3 Variant 3: Selection Sort

3.3.1 Changes Made

In this variant, we optimized the cloning of the arrays and implemented selection sort

- The baseline code had a loop copying mechanism before the sorting even started. It first copies the items from list x to y and then performs sorting. We removed this initial loop copying of the list and modified the sorting to sort on the input array and copy the result in the put array while the sorting loop was processing.

- We tried to use OpenMP directives for the inner loop parallelization by dividing the search for minimum element across multiple threads. We also added a critical section to make sure only one thread updates the minimum value.

- This was done on top of the parallelism part where we removed the if conditionals for the minimum value check. We wanted to not branch execution due to the if statement.

3.3.2 Expected Results

We expect to see improved performance, especially for small datasets, and an overall improvement from the base bubblesort.

3.3.3 Performance Analysis

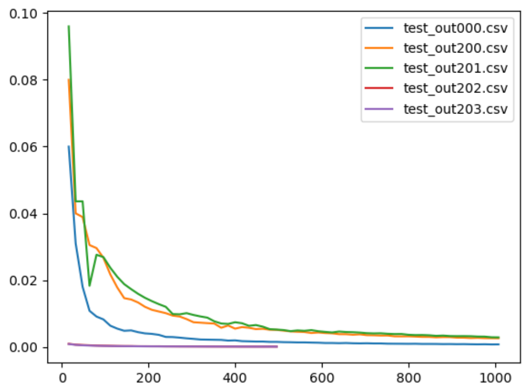

Figure 3: Performance plot for Optimized Selection-sort

The performance plot shows improvement in execution time compared to the baseline bubblesort implementation. We observe that:

- For small input sizes a speedup in time

- Parallelizing for small sizes is not effective

3.3.4 Potential Improvements

To further enhance this variant, we could:

- Parallelize the outer loop

- Unroll loops

- Improve memory access.

3.4 Variant 4: Merge sort:

Merge sort is an algorithm with the divide-and-conquer approach. it works by recursively dividing the input, sorting the smaller sections and merging them together.

3.4.1 Changes Made

In this variant, we implemented a optimized version of metgesort

- Var300 : Created mergesort base algorythm
- Var301: Optimized mergsort by adding insertion sort for small arrays.
- Var 302: Tried to optimize mergesort by converting to blocksrot with size $\sqrt(n)$
- Var 303: Made the 302 implementation of mergesort into a parallel version.

3.4.2 Expected Results

- Var300: We expected an improvement from the base sort
- Var301: We expected an improvement of speed for small sizes
- Var302: We expected an improvement for large arrays
- Var303: We expected an improvement for very large arrays compared to the previous variants.

3.4.3 Performance Analysis

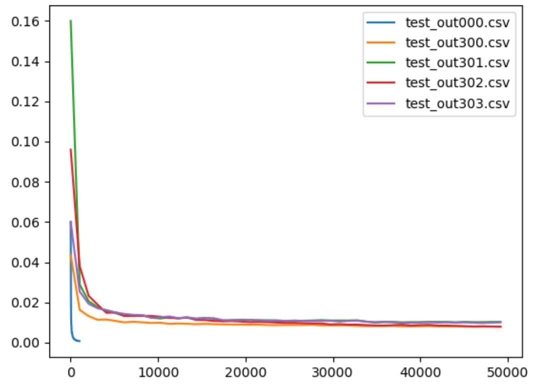

Figure 4: Performance plot for Optimized Mergesort

The performance plot shows a significant improvement in execution time compared to the baseline implementation. We observe that:

- For small input sizes the optimized version performs slightly better due to the insertion sort optimization.
- For larger inputs, we see a more pronounced improvement, with the optimized version consistently outperforming the baseline.
- The performance gain increases with input size, suggesting that our optimizations are particularly effective for larger datasets.

3.4.4 Potential Improvements

To further enhance this variant, we could:

- Experiment with different thresholds for initializing new threads.
- Implement a parallel version of the blocksort approach.
- Reduce the work made by the for loops.
- Addign all memory before recursion and threading starts.

4 Master

After analyzing all of the variants and comparing the throughputs we combined the best and assigned them to their most optimal sizes.

4.0.1 Changes Made

- For small input size ($n < 32$) use Insertionsort
- For medium input size ($n < 40000$) use Quicksort
- For large input size ($n > 40000$) use Parallel Quicksort.

4.0.2 Performance Analysis

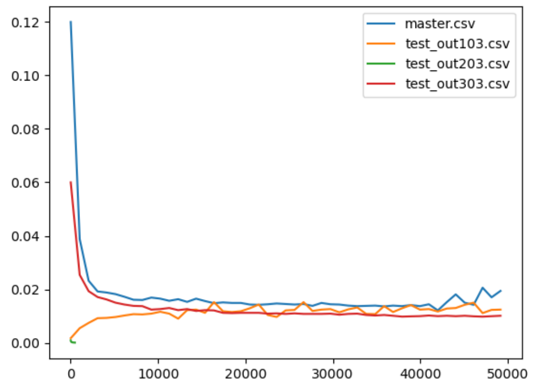

Figure 5: Performance plot for best algorythm

5 Conclusion

Our refinements demonstrate significant improvements over baseline sorting implementations. The Optimized Quicksort shows excellent performance for large datasets, while the Hybrid Merge-Insertion Sort provides a good balance of performance across different input sizes with reduced memory usage. These optimizations showcase the practical application of concepts from our computer science curriculum, particularly in algorithm analysis, data structures, and low-level system optimizations. Moving forward, we plan to explore more advanced techniques such as cache-conscious algorithms and parallel sorting methods to push the boundaries of sorting performance.