# COMP90025 Project 1
# Spell Checker — Identifying Words

Ziyu Wang (1560831)

## 1 Introduction

This project aims to turn a NUL-terminated ASCII document into a case-sensitive list of distinct `strcmp`-ordered words. In implementation, words are defined by `isalnum`, and the identifying pipeline consists of tokenization, sorting, deduplication, and packing phases. The final output includes both a contiguous words and a list of pointers to the starts of words with memory managed by a single allocation to ensure `free(*words)` releases all memory.

Given the potentially large input size, the parallelization is crucial for accelerating the execution and thus ensuring scalability. We first parallelize tokenization (Algorithm 2). The input is divided into evenly-sized blocks, and each thread scans one block independently to identify word boundaries. A prefix-sum offset is then computed to allow all threads write pointers to the beginning of words concurrently.

Since the most of the runtime is spent in sorting, this report would mainly focuses on the sorting evaluation. We compare three strategies:

1. **Sequential Quicksort** serves as baseline. The basic format described in Algorithm 1, using median-of-three pivot selection and Dijkstra's three-way partitioning.
2. **Parallel Quicksort** processes Quicksort recursive calls concurrently.
3. **Parallel Partition** further parallelizes the partition phase based on the Parallel Quicksort.

The basic form of a sequential quicksort algorithm includes three steps, as described in [1]:

```
Algorithm 1: Quicksort
  1. Finding a pivot
  2. Partition into two subarrays: s1 and s2
       •s1 contains all elements less than pivot
       •s2 contains all elements greater than pivot
  3. Recursive: repeat step 1. and step 2. for s1 and s2
     until subarray leave only one element
```

Since documents are typically contain a high frequency of duplicate words, such as 'a' and 'the', we use Dijkstra's three-way partitioning method to improve efficiency.

Intuitively, parallelizing the recursive calls in quicksort make the left and right sorts run concurrently. The parallel quicksort is shown in Algorithm 3. In actual implementation, we set a threshold of 40,000. If the input size falls below the threshold, the subtask is executed by the sequential quicksort to avoid the thread-setup overhead.

Since the sequential pivot comparison in the partition step is costly, parallel partition further parallelizes this step. Algorithm 4 is inspired by [1], follows the count-scan-scatter pattern. Each thread is assigned evenly sized sublist, and classifies each element is less than, equal to, or greater than the pivot. These counts of classification are aggregated to determine global offsets for prefix sum. Finally, each thread uses its offset to scatter elements into their positions in the output array.

For the unimplemented algorithm, we considered List 7 in [1]. However, it relies on the CRCW PRAM model, which assumes simultaneous writes to the same memory cell. This assumption would rise race condition in shared-memory architecture.

```
Algorithm 2: Tokenization
  Input: Document $doc$, length $n$, threads $T$
  Output: Pointers $ptrs$, total words $m$
  for $id \in [0 : T-1]$ do in parallel
      $b \leftarrow \lfloor id \times \frac{n}{T} \rfloor$;
      $e \leftarrow \lfloor (id+1) \times \frac{n}{T} \rfloor$;
      $prev \leftarrow isalnum(doc[b-1])$;
      $cnt \leftarrow 0$;
      for $i \in [b..e)$ do
          if word starts then $cnt{+}{+}$;
          else if word ends then $doc[i] \leftarrow \backslash 0$;
      end
      $starts[id] \leftarrow cnt$;
  end
  $m \leftarrow sum(starts)$;
  for $t = 1..T-1$ do
      $ps[t] \leftarrow ps[t-1] + starts[t-1]$;
  end
  for $id \in [0 : T-1]$ do in parallel
      $prev \leftarrow isalnum(doc[b-1])$;
      $out \leftarrow ps[id]$;
      for $i \in [b..e)$ do
          if word starts then $ptrs[out{+}{+}] \leftarrow \&doc[i]$;
      end
  end
  return $(ptrs, m)$;
```

**Algorithm 3:** Parallel Quicksort

```
Input: Array $A$, left index $L$, right index $R$
Output: Sorted array $A[L..R]$
$p \leftarrow median\_of\_three(A[L], A[(L+R)/2], A[R])$;
Divide $A$ into three arrays;
    $A[L..lt-1] < p$;
    $A[lt..gt] = p$;
    $A[gt+1..R] > p$;
for $subtasks$ do in parallel
    QuickSort$(A, L, lt-1)$;
    QuickSort$(A, gt+1, R)$;
end
```

**Algorithm 4:** Parallel Partition

```
Input: Array $A$, left index $L$, right index $R$, pivot
Output: Partition indices $(lt, gt)$
for $id \in [0:p)$ do in parallel
    $b \leftarrow id \times \lceil n/p \rceil$;
    $e \leftarrow (id+1) \times \lceil n/p \rceil$;
    for $i \in [b:e)$ do
        if $words[i] < pivot$ then $flags[i-L] \leftarrow -1$;
        $nL[id]$++;
        else if $words[i] > pivot$ then
          $flags[i-L] \leftarrow +1$;
        $nG[id]$++;
        else $flags[i-L] \leftarrow 0$;
        $nE[id]$++;
    end
end
$totalL = \sum nL$; $totalG = \sum nG$; $totalE = \sum nE$;
$psL[id]$: start offset for $< pivot$;
$psE[id]$: start offset for $= pivot$;
$psG[id]$: start offset for $> pivot$;
for $id \in [0:p)$ do in parallel
    $l \leftarrow psL[id]$; $e0 \leftarrow psE[id]$; $g \leftarrow psG[id]$;
    for $i \in [b:e)$ do
        if $flags = -1$ then $temp[l{+}{+}] = A[i]$;
        else if $flags = 0$ then
          $temp[totalL + (e0{+}{+})] = A[i]$;
        else $temp[(totalL + totalG) + (g{+}{+})] = A[i]$;
    end
end
for $k \in [0:n)$ do in parallel
    $A[L+k] = temp[k]$;
end
return $(L + totalL,\ L + totalL + totalE - 1)$;
```

## 2 Methodology

In this project, all implementations share the same overall pipeline consisting of tokenization, deduplication, and packing, while differing only in the sorting phase. We evaluated three sorting algorithms including (1) sequential quicksort, (2) parallel quicksort with sequential partition, and (3) parallel quicksort with parallel partition.

### 2.1 Experimental Setup

Experiments were conducted on the Spartan HPC system to eliminate hardware difference and guarantee fair comparisons between strategies. We specified the sapphire partition and the GCC/11.3.0 compiler.

There are two categories of datasets were tested:

1. **Small documents** includes `text1-ASCII.txt`, `text2-ASCII.txt`, and `text3-ASCII.txt`. These documents were selected to validate correctness and measure threads warm-up overhead on lightweight workloads.
2. **Large documents** includes `abstracts_head_10M.txt`, `abstracts_head_100M.txt`, and `abstracts_head_1G.txt`. These documents were used to evaluate algorithms speedup, efficiency and scalability of parallel strategies.

### 2.2 Correctness

The baseline implementation provided in the project skeleton serves as the gold standard reference in correction testing. The correctness requires both the hash values and the number of distinct words in the output to match the gold reference. Any deviation results in the output would be classified as incorrect.

### 2.3 Performance Measures

Performance was evaluated in four dimensions: (1) Wall-clock time, (2) Speedup and efficiency, (3) Memory usage, and (4) Parameter sensitivity. For the parallel implementations, the number of threads was varied from 1 to 30.

1. **Wall-clock time**: We measure both the overall execution time and the computation-only time, which excludes the file I/O time. For the latter, we record the runtime of subtasks, including tokenization, sorting, deduplication, and packing. This breakdown allows us to identify performance bottlenecks and facilitate the parallel strategies analysis.
2. **Speedup and efficiency**: Speedup is formally defined as

$$S(p) = \frac{T(1)}{T(p)} \quad \text{(times)}$$

, where T(1) is the runtime with one thread and T(p) is the runtime with p threads. Efficiency is formally defined as

$$E(p) = \frac{T(1)}{T(p) \times p} \quad (\%)$$

, which measures the average thread contribution.

3. **Memory usage**: Peak memory consumption was extracted from job logs to evaluate the memory requirement for each algorithm.
4. **Parameter sensitivity**: For the parallel partition algorithm, we test threshold of 30,000, 40,000, 50,000 in order to get the balance between thread warm-up cost and the benefit of parallel execution.

## 3 Experiments

For clarity, we denote the datasets `10M`, `100M`, and `1G` as shorthand for `abstracts_head_10M.txt`, `abstracts_head_100M.txt`, and `abstracts_head_1G.txt`, respectively. Similarly, we denote `T1`, `T2`, and `T3` for `text1-ASCII.txt`, `text2-ASCII.txt`, and `text3-ASCII.txt`. The implementations are referred to as *Sequential Quicksort* (sequential quicksort), *Parallel Quicksort* (parallel quicksort with sequential partition), and *Parallel Partition* (parallel quicksort with parallel partition).

Experiments were structured in two phases:

- **Correctness verification**: Results are compared against the golden reference.
- **Performance evaluation**: Metrics include runtime, speedup, efficiency, and memory usage.

### 3.1 Correctness Results

Table 1 summarizes the correctness results and shows that all implementations produced correct outputs.

| Dataset | Method | Rate | Total | Incorrect |
|---|---|---|---|---|
| 10M | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |
| 100M | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |
| 1G | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |
| T1 | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |
| T2 | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |
| T3 | Baseline | 1.0 | 1 | 0 |
| | Sequential Quicksort | 1.0 | 20 | 0 |
| | Parallel Quicksort | 1.0 | 30 | 0 |
| | Parallel Partition | 1.0 | 30 | 0 |

**Table 1.** Correctness verification

### 3.2 Performance Results

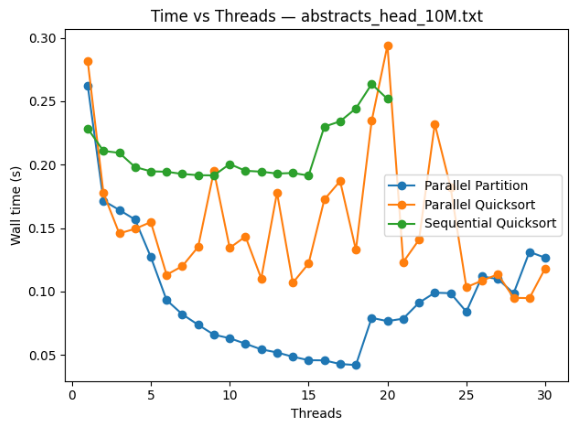
Performance results are summarized below:

1. **Runtime**
   - Overall: Table 2 summarizes the best execution times and Figure 1 illustrates their trends.
   - Breakdown: Table 3 decomposes the contributions of subtasks, and Figure 2 visualizes the percentage of each stages.
2. **Speedup and Efficiency**
   - Speedup: Figures 3 plots speedup of three algorithms, and Table 4 summarizes the maximum observed speedup.
   - Efficiency: Figures 4 shows efficiency trends as the number of thread increases.
3. **Memory**: Figure 5 demonstrates the memory usage trend and the usage ranges are summarized in Table 5.
4. **Threshold**: Figure 6 shows the performance trends for threshold of 30,000, 40,000, and 50,000.

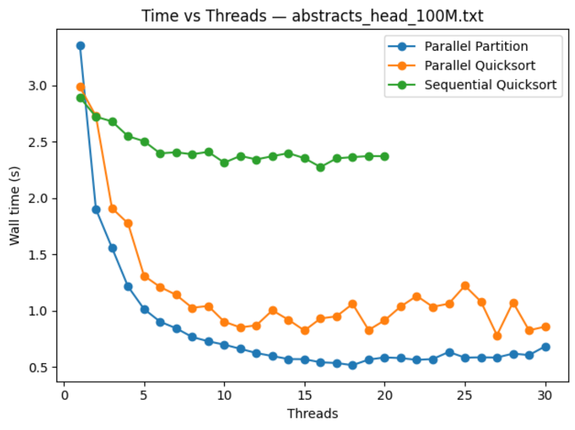
Table 2 reports the best execution time achieved by each algorithm across the datasets with their corresponding number of threads. Figure 1 presents the overall runtime trends as the thread count increases from 1 to 30 in the large-scale dataset category.

| Dataset | Method | Threads | Time (s) |
|---|---|---|---|
| 10M | Sequential Quicksort | - | 0.191442 |
| | Parallel Quicksort | 29 | 0.094851 |
| | Parallel Partition | 18 | 0.042137 |
| 100M | Sequential Quicksort | - | 2.274149 |
| | Parallel Quicksort | 27 | 0.781519 |
| | Parallel Partition | 18 | 0.515485 |
| 1G | Sequential Quicksort | - | 29.374119 |
| | Parallel Quicksort | 20 | 10.812595 |
| | Parallel Partition | 22 | 6.209646 |
| T1 | Sequential Quicksort | - | 0.014959 |
| | Parallel Quicksort | 10 | 0.011218 |
| | Parallel Partition | 12 | 0.006734 |
| T2 | Sequential Quicksort | - | 0.000168 |
| | Parallel Quicksort | 1 | 0.000146 |
| | Parallel Partition | 1 | 0.000171 |
| T3 | Sequential Quicksort | - | 0.000033 |
| | Parallel Quicksort | 1 | 0.000028 |
| | Parallel Partition | 1 | 0.000030 |

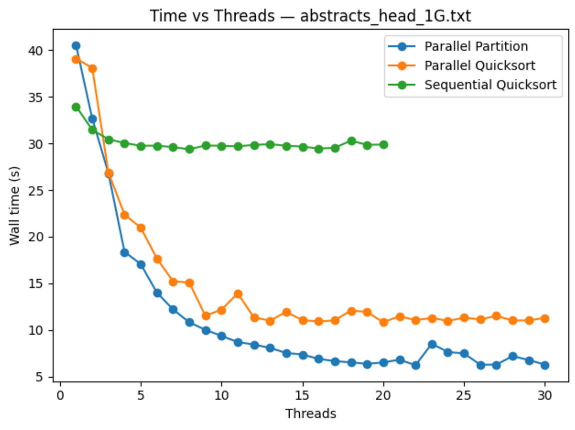
**Table 2.** Best execution time results



**(a)** 10M dataset



**(b)** 100M dataset



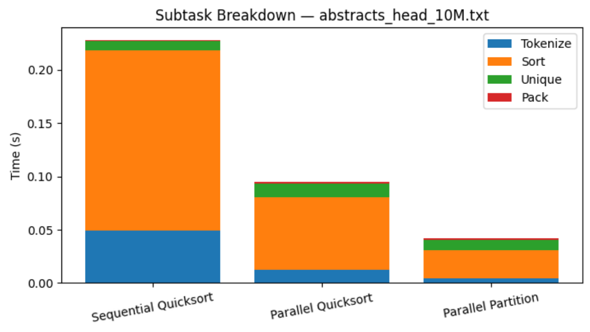
**(c)** 1G dataset

**Figure 1.** Execution time trends

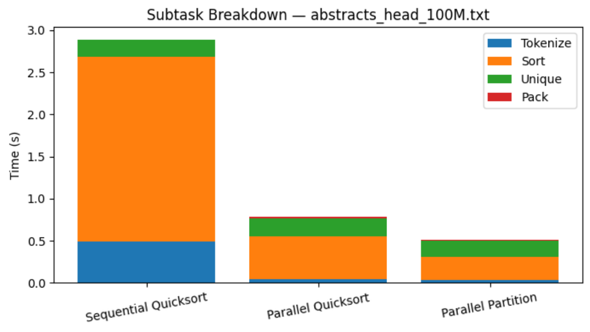
The subtask breakdown data are derived from Table 2, which reports the best execution time achieved with thread counts ranging from 1 to 30 for each algorithm. Based on these results, we decompose the total wall time into individual subtask contributions. The detailed result are presented in Table 3, with the corresponding visualization shown in Figure 2.

| Dataset | Method | Tokenize | Sort | Unique | Pack |
|---|---|---|---|---|---|
| 100M | Sequential Quicksort | 17.15 | 75.83 | 6.74 | 0.28 |
| | Parallel Quicksort | 5.97 | 64.25 | 27.65 | 2.12 |
| | Parallel Partition | 7.05 | 53.27 | 37.20 | 2.48 |
| 10M | Sequential Quicksort | 21.55 | 74.26 | 3.71 | 0.49 |
| | Parallel Quicksort | 13.25 | 71.89 | 12.96 | 1.90 |
| | Parallel Partition | 10.37 | 63.45 | 23.06 | 3.12 |
| 1G | Sequential Quicksort | 14.43 | 78.89 | 6.52 | 0.15 |
| | Parallel Quicksort | 3.89 | 64.94 | 30.04 | 1.13 |
| | Parallel Partition | 7.60 | 51.30 | 39.16 | 1.94 |

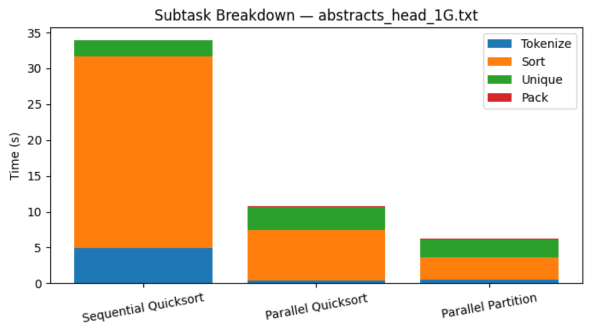
**Table 3.** Breakdown of execution time percentages (unit: %)



**(a)** 10M



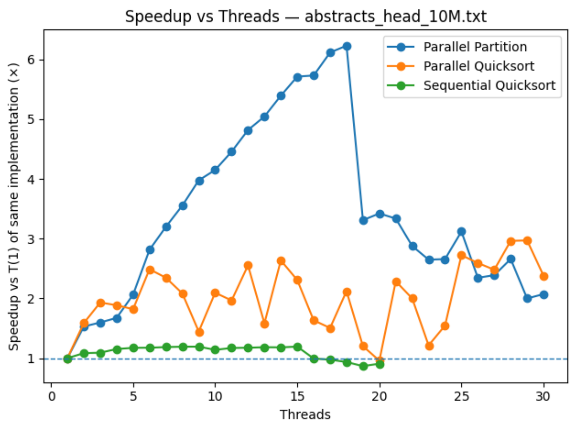
**(b)** 100M



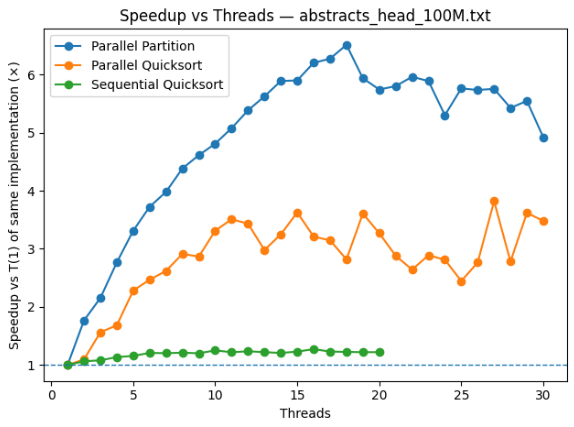
**(c)** 1G

**Figure 2.** Wall time breakdown by subtask

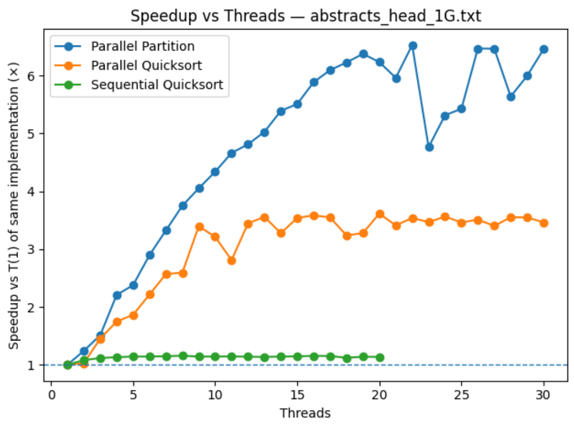
Figures 3 and 4 present the speedup and efficiency trends across configurations. In addition, Table 4 outlines the maximum speedup and its corresponding number of threads.



**(a)** 10M dataset

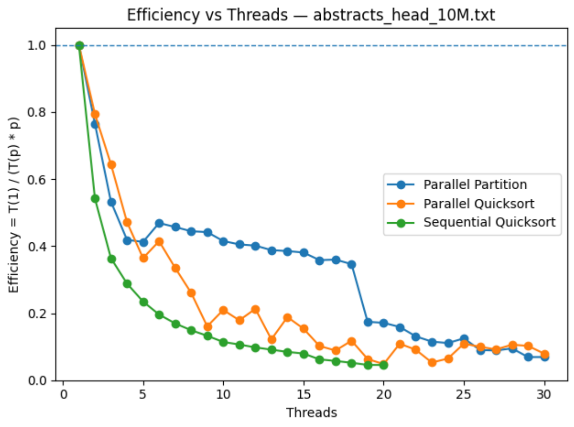


**(b)** 100M dataset

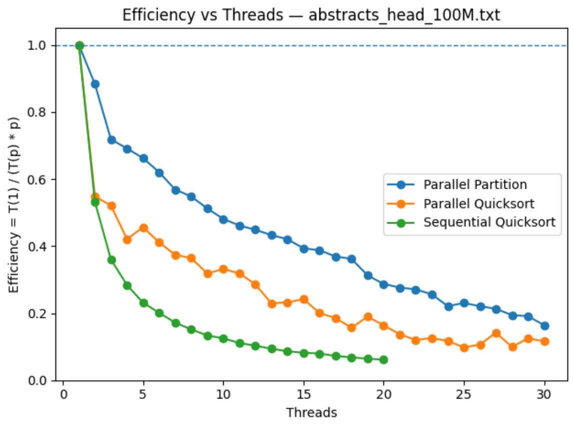


**(c)** 1G dataset

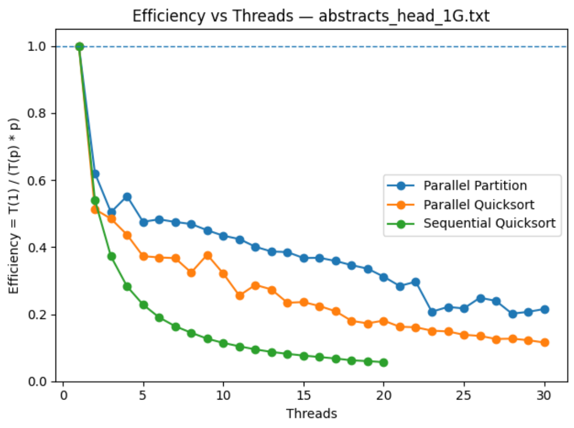
**Figure 3.** Speedup vs. Threads



**(a)** 10M dataset



**(b)** 100M dataset



**(c)** 1G dataset

**Figure 4.** Efficiency vs. Threads

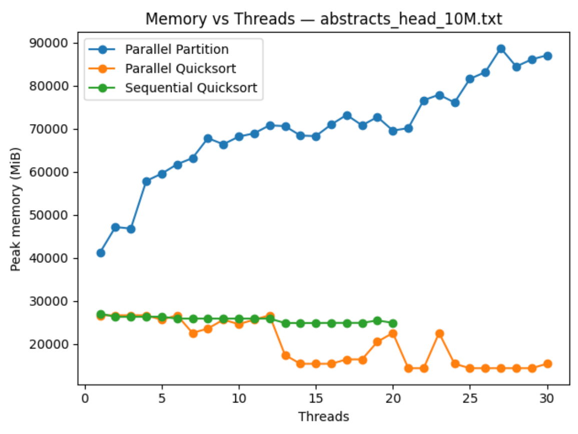
| Dataset | Method | Threads | Speedup |
|---|---|---|---|
| 10M | Parallel Quicksort | 29 | 2.969932 |
| | Parallel Partition | 18 | 6.228303 |
| 100M | Parallel Quicksort | 27 | 3.819171 |
| | Parallel Partition | 18 | 6.510669 |
| 1G | Parallel Quicksort | 20 | 3.615457 |
| | Parallel Partition | 22 | 6.529446 |

**Table 4.** Max speedup

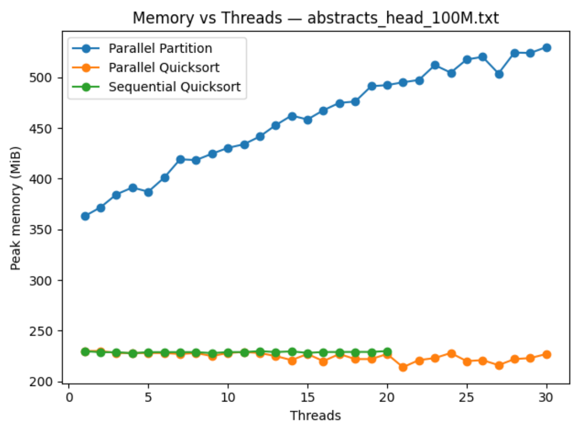
Table 5 summarizes the ranges of peak memory usage for each dataset and algorithm. The minimum and maximum values are measured with thread counts between 1 and 30. Figure 5 illustrates the corresponding memory usage trends.

| Dataset | Method | Mem Range (KiB) | Mean (KiB) |
|---|---|---|---|
| 10M | Sequential Quicksort | [24836, 27040] | 25631 |
| | Parallel Quicksort | [14336, 26624] | 20002 |
| | Parallel Partition | [41200, 88712] | 69851 |
| 100M | Sequential Quicksort | [233240, 235472] | 234445 |
| | Parallel Quicksort | [219136, 235520] | 230161 |
| | Parallel Partition | [371712, 542528] | 470066 |
| 1G | Sequential Quicksort | [2367548, 2368768] | 2367977 |
| | Parallel Quicksort | [2353152, 2368512] | 2361617 |
| | Parallel Partition | [4184644, 4513536] | 4340861 |

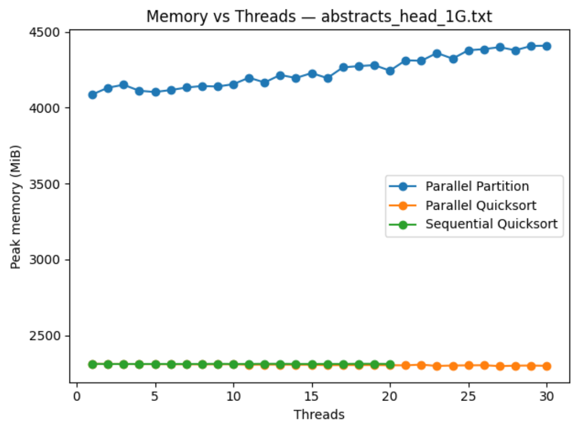
**Table 5.** Memory usage ranges and mean



**(a)** 10M dataset



**(b)** 100M dataset



**(c)** 1G dataset

**Figure 5.** Memory vs. Threads trends

## 4 Discussion and Conclusion

In this section, we first analyze the time and space complexities of the three implementations and then compare these theoretical inferences with the experimental results. Finally, we discuss scalability predictions and limitations, and draw a conclusion with a summary of key findings.

### 4.1 Complexity Analysis

**Notation**

Let $W(n)$ denote work, $T_\infty(n)$ denote the span, and $T_p(n)$ denote the parallel time on $p$ threads.

**Time complexity**

- Sequential Quicksort: $O(n \log n)$ in best/average case, $O(n^2)$ in the worst case.
- Parallel Quicksort: $W(n) = O(n \log n)$, $T_\infty(n) = O(n)$
- Parallel Partition: $W(n) = O(n \log n)$, $T_\infty(n) = O(\log^2 n)$

Time taken by sequential quicksort is written as $T(n) = T(k) + T(n-k-1) + n$, where the k depends on the pivot. In the best case, we split input roughly even while the worst case the list is highly imbalanced split. In implementation, we use median-of-three pivot selection to reduce the possibility of the worst case.

The total work of parallel quicksort is equivalent to the sequential cost. The span is the sum of linear partitions along the deepest branch, which can be expressed as $T_\infty = \Theta(n) + \Theta(n/2) + \cdots = \Theta(n)$.

For the parallel partition, one time partition consists of:

1. count: $W(n) = O(n)$, $T_\infty(n) = O(1)$
2. scan: $W(n) = O(n)$, $T_\infty(n) = O(\log n)$

3. scatter: $W(n) = O(n)$, $T_\infty(n) = O(1)$

Overall, each partition takes $W(n) = O(n)$ and $T_\infty(n) = O(\log n)$. Under $O(\log n)$ recursions, the total work and span would be $O(n \log n)$ and $O(\log^2 n)$, respectively.

**Space complexity**

- Sequential Quicksort: $O(n)$
- Parallel Quicksort: $O(n)$
- Parallel Partition: additional $O(n)$ space

Parallel partition requires an $O(n)$ flag array, $O(p)$ storage for prefix sums, and an $O(n)$ temporary output buffer for stable scatter. Therefore, this method involves extra $O(n)$ memory.

## 4.2 Experimental Analysis

**Correctness**

Table 1 shows all implementations outputs match the gold standard, which indicates the accuracy is not sacrificed for the speedup. We avoid the race condition by ensuring the subtasks operate on disjoint memory in both parallel quicksort and parallel partition. In parallel quicksort, tasks are only created after the non-overlapping partition boundaries are fixed, and `firstprivate` are used to ensure no overwrites across threads. In parallel partition, we use count-scan -scatter pattern to create exclusive write windows for each thread at the cost of $O(n)$ space.

**Runtime and speedup**

Table 2 suggests that sequential quicksort is preferable for the small datasets. For lightweight input, the advantage of parallel is offset by the overhead of thread creation. Besides, since we set a threshold of 40,000, at this scale, the parallel quicksort might create threads but fall below threshold quickly and revert to sequential processing, further diminishing marginal benefits.

For large-scale input, the parallel partition achieves the fastest runtime among all methods. Compared to parallel quicksort, parallel partition runs 2.25× faster on 10M, 1.52× faster on 100M, and 1.74× faster on 1G dataset. This advantages comes from a time-space trade-off. From Table 5, we can conclude parallel partition requires 1.8x memory on the 1G on average than parallel quicksort, and 2× memory on the 100M, which aligns with the theoretical expectation of an additional $O(n)$ memory.

Regarding runtime decomposition, Table 3 shows that the sorting phase takes the largest share of runtime in all strategies, whereas in the parallel partition implementation, the sort phase is reduced to the smallest proportion.

Theoretically, Amdahl's law bounds the speedup as

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

, where $f$ is the inherently sequential fraction. With infinite processors, the bound tends to $S_{\max}(p) = \frac{1}{f}$. Assuming sorting is the only parallelizable stage and sequential quicksort is the baseline, the theoretical upper bound and the observed speedups are shown in Table 6. The results imply that parallel partition implementation approaches Amdahl's law bound.

| Dataset | Amdahl's Law | Parallel Quicksort | Parallel Partition |
|---|---|---|---|
| 10M | 3.88 | 2.02 | 4.54 |
| 100M | 4.14 | 2.91 | 4.41 |
| 1G | 4.74 | 2.72 | 4.73 |

**Table 6.** Amdal's Upper Bound and Actual Speedup (unit: times)

## 4.3 Future Scaling Predictions

Strong scalability is defined as increasing the number of processors with fixed problem size while weak scalability considers both the problem size and processor count increase. From Figure 3, we can see that both parallel quicksort and parallel partition show overall trends of increasing speedup as the number of threads grows. For weak scalability, we pair (1 thread, 10M) with (10 threads, 100M), and (1, 100M) with (10, 1G) to test the speedup would remain or not. Table 7 shows parallel quicksort decreases from 3.13× to 2.45× while parallel partition maintains a more stable performance, with speedup slightly decreasing from 3.76× to 3.59×. This result suggests parallel partition has better weak scalability than Parallel Quicksort.

| Method | Pairing | $T(1, data1)$ | $T(10, data2)$ | Speedup |
|---|---|---|---|---|
| Parallel Quicksort | 10M → 100M | 0.281701 | 0.901287 | 3.13× |
| | 100M → 1G | 2.984755 | 12.173029 | 2.45× |
| Parallel Partition | 10M → 100M | 0.262442 | 0.697748 | 3.76× |
| | 100M → 1G | 3.356152 | 9.343491 | 3.59× |

**Table 7.** Weak scalability

However, Table 4 shows that the highest observed speedup for each parallel strategy does not occur at the maximum number of threads. From Figure 3, we also observe that the speedup begins to decline around 18–23 threads. From Amdahl's law perspective, the observation occurs because the partition algorithm still contains inherently sequential portions, which limit the benefits of strong scaling. In the case of parallel partition, this bottleneck is likely dominated by the scan phase.

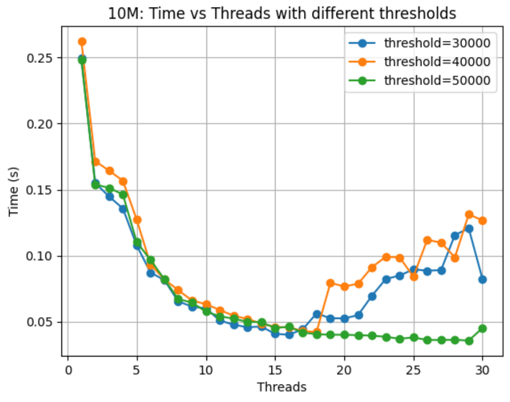
Moreover, the Brent's theorem outlines the scalability limit as:

$$T_p(n) = \Theta(\frac{W(n)}{p} + T_\infty(n))$$

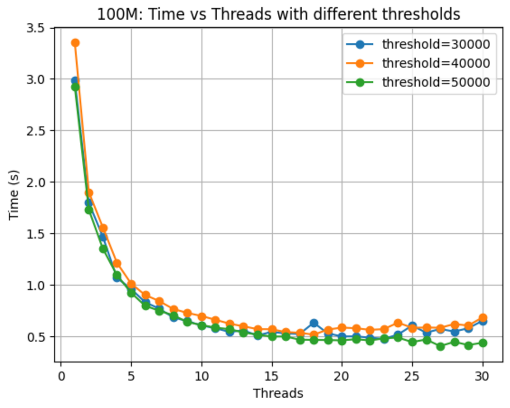
, where $W(n)$ is work and $T_\infty(n)$ is span.

For parallel partition,

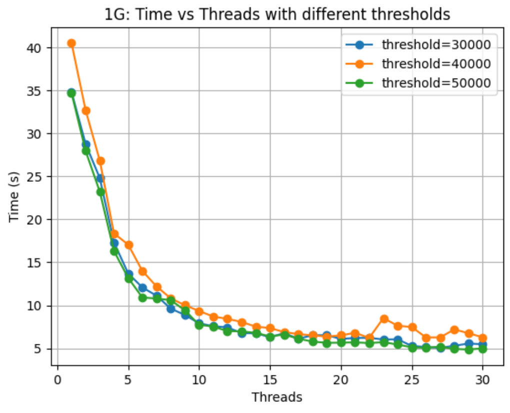
$$T_p(n) = \Theta(\frac{n \log n}{p} + \log^2 n)$$



**(a)** 10M dataset



**(b)** 100M dataset



**(c)** 1G dataset

**Figure 6.** Time vs. Threads with different thresholds

, indicates the optimal number of threads is bounded by

$$\frac{W(n)}{T_\infty(n)} = \Theta(\frac{n \log n}{log^2 n})$$

For 1G dataset, the critical point should be $\approx 3.3 \times 10^7$. This boundary indicates that the performance drop beyond 18–23 threads is not due to algorithmic limits.

On the software side, additional threads introduce synchronization barriers. When the problem size is not sufficiently large, the overhead of threads coordination can diminish the benefits, and therefore increasing runtime. On the hardware side, memory port number and bandwidth can become the bottleneck, preventing further acceleration of parallel strategies. Moreover, prior work [2] has shown that performance on NUMA systems can degrade significantly due to remote memory accesses and load imbalance. These factors may explain why speedup declines beyond 18–23 threads in our experiments.

### 4.4 Conclusion

This project evaluated three sorting strategies and analyzed both their theoretical complexities and their experimental behaviors on both small and large datasets. Since the parallel partition reduces the workload of the sorting subtask and demonstrates good weak scalability, it can be considered an effective implementation. Through performance evaluation, we identified bottlenecks from both software and hardware perspectives, including synchronization overheads and memory bandwidth limitations. For future work, we aim to optimize both algorithm design and take system-level factors into consideration in order to achieve sustainable scalability.

## References

[1] Jie Liu, Clinton Knowles, and Adam Brian Davis. 2005. A Cost Optimal Parallel Quicksorting and Its Implementation on a Shared Memory Parallel Computer. In *Parallel and Distributed Processing and Applications*, Yi Pan, Daoxu Chen, Minyi Guo, Jiannong Cao, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–502.

[2] Oussama Tahan. 2014. Towards Efficient OpenMP Strategies for Non-Uniform Architectures. arXiv:1411.7131 [cs.DC] https://arxiv.org/abs/1411.7131

## A Figures

Figures 6 shows different threshold in parallel partition.