Parallel Systems Project Report

Parallel Odd-Even Transposition Sort

-Atharva Mandhaniya
-Neeraj Kulkarni
Course: Parallel Systems

January 11, 2026

1 Introduction

Sorting is a fundamental operation in computer science. While algorithm design often focuses on minimizing total operation count (e.g., QuickSort with $\mathcal{O}(N \log N)$), parallel systems require algorithms that exhibit structural independence to utilize multiple processing units effectively.

This report analyzes the **Odd-Even Transposition Sort**, a parallel variation of the standard Bubble Sort. Unlike Bubble Sort, which processes the array sequentially, Odd-Even Transposition Sort decouples the comparison operations into two distinct, alternating phases. This structural change allows for a high degree of parallelism, making it suitable for implementation on shared-memory architectures using OpenMP.

2 Parallel Approach

The conceptual basis for parallelizing the algorithm lies in the data independence of comparisons within a single phase. The algorithm proceeds in N iterations (where N is the number of elements). Each iteration consists of two phases:

1. **Odd Phase:** Comparisons are performed for pairs $(A[i], A[i+1])$ where i is an odd integer $(1, 3, 5, \dots)$.
2. **Even Phase:** Comparisons are performed for pairs $(A[i], A[i+1])$ where i is an even integer $(0, 2, 4, \dots)$.

Justification for Parallelism: In the Odd Phase, the comparison at index $i = 1$ involves elements $A[1]$ and $A[2]$. The next comparison at $i = 3$ involves $A[3]$ and $A[4]$. Since these sets of memory addresses are disjoint, there is no race condition. Consequently, all comparisons within a phase can be executed simultaneously by P threads without explicit locking, provided a global synchronization barrier is enforced between phases.

3 Implementation

The implementation targets a shared-memory architecture using the **OpenMP** API. The C++ code utilizes the `#pragma omp parallel for` directive to distribute the loop iterations among threads.

Crucially, the algorithm requires strict synchronization. A barrier is implicitly provided by the OpenMP compiler at the end of each `parallel for` construct. This ensures that the Odd Phase completes globally before the Even Phase begins, preserving the correctness of the sort.

```cpp
void parallelOddEvenSort(std::vector<int>& arr, int n) {
    for (int i = 0; i < n; ++i) {
        // Phase 1: Odd Indices
        // Threads share the work of the loop.
        // 'static' schedule is chosen for load balancing.
        #pragma omp parallel for schedule(static)
        for (int j = 1; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
        // Implicit Barrier Here

        // Phase 2: Even Indices
        #pragma omp parallel for schedule(static)
        for (int j = 0; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
        // Implicit Barrier Here
    }
}
```

Listing 1: OpenMP Implementation Core

4 Theoretical Analysis

We analyze the asymptotic complexity with respect to the input size n and the number of processors p.

4.1 Work and Time Complexity

The serial version of the algorithm (Bubble Sort) performs $\mathcal{O}(n)$ passes, each doing $\mathcal{O}(n)$ comparisons.

$$W_S(n) = \Theta(n^2)$$

In the parallel formulation, the n comparisons in each phase are distributed among p processors. Assuming ideal load balancing, the time complexity per phase is $\Theta(n/p)$. With n total phases:

$$T_A(p, n) = n \times \Theta\left(\frac{n}{p}\right) = \Theta\left(\frac{n^2}{p}\right)$$

4.2 Cost Optimality

The cost $C_A(p, n)$ of a parallel algorithm is defined as the product of the parallel execution time and the number of processors:

$$C_A(p, n) = p \times T_A(p, n) = p \times \frac{n^2}{p} = \Theta(n^2)$$

Since the parallel cost $\Theta(n^2)$ matches the serial complexity $W_S(n)$, the algorithm is theoretically **cost-optimal**.

5 Experiments

The experiments were conducted on a MacBook Air (Apple Silicon) using the `g++` compiler (Apple Clang 15.0.0) with OpenMP enabled via `-Xpreprocessor -fopenmp` and the `-O3` optimization flag.

5.1 Strong Scaling

Strong scaling analyzes the speedup $S = T_S/T_A(p)$ when the problem size n is fixed and p varies. Table 1 presents results for $n = 40,000$.

n	p (Threads)	t (Time s)	Speedup (S)	Efficiency
40,000	1	2.7145	1.00x	100%
40,000	2	1.9733	1.38x	69%
40,000	4	2.0733	1.31x	32%
40,000	8	2.9517	0.92x	11%
40,000	16	4.9980	0.54x	3%

Table 1: Strong Scaling results ($n = 40,000$).

5.2 Weak Scaling

Weak scaling evaluates performance as the workload per processor is kept roughly constant. We increased n as p increased.

n	p (Threads)	t (Time s)
10,000	1	0.1241
14,140	2	0.4366
19,993	4	0.7316
28,271	8	2.2050
39,975	16	4.9893

Table 2: Weak Scaling results.

5.3 Discussion of Results

The experimental results exhibit a divergence from the theoretical linear speedup. While a speedup of **1.38x** was achieved at $p = 2$, performance degraded significantly as p increased further. This gap can be attributed to two primary factors:

1. Synchronization Overhead The Odd-Even Transposition Sort is a fine-grained parallel algorithm. It requires a global barrier synchronization after every phase. For $n = 40,000$, the threads must synchronize $2 \times 40,000 = 80,000$ times. As p increases, the latency of thread coordination at these 80,000 barriers accumulates. Eventually, the time spent waiting at barriers exceeds the computational time saved by parallelizing the comparisons.

2. Work Granularity At $n = 40,000$ and $p = 16$, each thread performs approximately $40,000/16 = 2,500$ comparisons between barriers. On modern CPUs, 2,500 integer comparisons take mere microseconds. The overhead of waking threads and context switching is often larger than this computation time, leading to the observed slowdown (Speedup < 1).

6 Conclusion

The Odd-Even Transposition Sort successfully demonstrates the principles of parallel algorithm design and correctness. It achieves cost-optimality in theory ($\Theta(N^2)$). However, experimental results highlight

the practical limitations of fine-grained parallelism on commodity multi-core hardware. The algorithm is best suited for scenarios where the cost of synchronization is low relative to the cost of computation, or for architectures like GPUs where barrier synchronization is hardware-accelerated.