# ECE 361C Multicore Computing Project: Parallel CPU-based Sorting Algorithms

Suchir Vemulapalli
sv25773

Moustafa Neematallah
man3394

Srishaan Iyengar
ssi333

Daniel Xie
dx833

***Abstract*—Parallel computing has become increasingly important in addressing the computational challenges posed by large datasets and complex algorithms. In this paper, we investigate the performance of four parallel sorting algorithms—Merge Sort, Quick Sort, Bitonic Sort, and Odd-Even Sort—implemented on CPU-based architectures. To conduct our analysis, we develop parallel implementations of each algorithm using multithreading techniques to leverage the computational power of modern CPUs. We systematically vary both the number of threads employed in each algorithm's execution and the input size of the array, and compare the performance of each algorithm against each other. We also evaluate the performance differences between the sequential and parallel implementation of each algorithm.**

## I. Introduction

Sorting algorithms play a fundamental role in computer science, enabling efficient organization and retrieval of data across various applications and platforms. In recent years, the focus has shifted towards parallel sorting algorithms, leveraging the increasing prevalence of multi-core processors and distributed computing systems to enhance sorting efficiency. This paper evaluates several prominent parallel sorting algorithms, highlighting their design principles, parallelization strategies, and performance characteristics. The selected algorithms encompass a diverse range of parallel sorting techniques, including divide-and-conquer methods like Merge Sort and Quick Sort, as well as sorting networks such as Odd-Even Sort and Bitonic Sort. Through a meticulous examination of these algorithms, we aim to provide insights into their suitability for different parallel computing environments and workloads. The research methodology involves rigorous empirical testing on an 8-core machine, evaluating the runtime performance of each algorithm across varying input sizes and thread counts. By systematically analyzing the experimental results, we aim to elucidate the scalability, efficiency, and parallelization overheads associated with each algorithm.

## II. Divide-and-Conquer Sorting Algorithms

### *A. Merge Sort*

**Understanding Merge Sort through Divide-and-Conquer**
Merge Sort exemplifies the divide-and-conquer strategy in algorithm design. This method simplifies the sorting of an array by first splitting it into two halves, recursively sorting each separately, and then merging them back together. The merging step involves comparing elements from the two sorted halves, using two pointers, and methodically combining them into a new sorted array. This is done in linear time, showcasing efficient problem-solving through recursive decomposition and reintegration.

**Parallel Merging of Two Sorted Arrays**
To merge two sorted arrays efficiently in parallel, each element from arrays B and C calculates its precise position in the merged array D. This is achieved by determining its rank—the number of elements less than it in both arrays. Each element's rank in D is the sum of its position in its own array (as B and C are sorted) and the count of elements less than it in the other array, found using a binary search. This approach leverages parallel processing, enabling each element to independently find its spot in D, ideally suited for a Concurrent Read, Exclusive Write (CREW) PRAM model. Additionally, Parallel Merge Sort with merge powered by binary search takes $O(\log^2 n)$ time.

### *B. Quick Sort*

**Quick Sort and Divide-and-Conquer**
Like Merge Sort, Quick Sort naturally lends itself to divide-and-conquer. At invocation, Quick Sort first partitions the array such that all elements left of the pivot element are less than or equal to the pivot and all elements right of the pivot are greater than the pivot. Then, Quick Sort recursively sorts the left subarray and the right subarray. It is trivial to parallelize Quick Sort by performing both recursive subproblems in parallel. Like Sequential Quick Sort, the worst case happens if the pivot chosen at each step does not sufficiently partition the left and right subarrays. In the average case, Parallel Quick Sort with partitioning performed by prefix sum takes $O(\log^2 n)$ time.

## III. Sorting Networks

Sorting networks are a class of sorting algorithms that use comparators, which compares the elements at two indices and swaps them if needed. Many sorting algorithms that do not use other conditionals can be modeled using sorting networks, including some parallel sorting algorithms. They are oblivious algorithms in which the control flow of the algorithm will always proceed in the same manner regardless of the data. Odd-Even Sort and Bitonic Sort, as explained below, are examples of sorting network algorithms.

### *A. Odd-Even Sort (Brick Sort)*

Odd-Even Sort, also known as Brick Sort, is a simple sorting algorithm that operates by repeatedly comparing and swapping adjacent pairs of elements in a list until the entire list is sorted.

It is called "Odd-Even" because it alternates between two phases: the odd phase, where it compares and swaps elements at odd indices with their immediate greater indexed neighbor, and the even phase, where it does the same for elements at even indices. The odd and even phases continue in an alternating manner until there is an iteration where no swaps are made in either the odd or even phase, meaning the array is sorted. The Parallel Odd-Even Sort leverages parallelism by performing the compare and swap operations in parallel for odd indices during the odd phase and even indices during the even phase. The parallel time complexity of Odd-Even Sort is $O(n)$ because there can be at most $O(n)$ odd and even phases before the array is sorted, where an example of the worst case is a situation where $arr[0]$ is the largest element.

### *B. Bitonic Sort*

Bitonic Sort is a parallel sorting algorithm designed to efficiently sort sequences with a power-of-two length. The algorithm sorts the first half of the array in ascending order and the second half in descending order by recursively calling BitonicSort(L) and BitonicSort(R) on the left half (L) and right half (R) of the input array, respectively. This process creates a bitonic sequence since BitonicSort calls BitonicMerge which both performs pairwise comparisons, resulting in the lower half of the subarray containing elements smaller than those in the upper half, and then recursively calls BitonicMerge on both halves. The Bitonic Sort algorithm is then able to turn the newly formed bitonic sequence into a monotonic sequence by recursively applying BitonicMerge until it is called on one-element subarrays, which is the base case. Bitonic Sort is easily parallelized by making the recursive calls to BitonicSort(L) and BitonicSort(R) occur in parallel, performing the $n/2$ pairwise comparisons in BitonicMerge in parallel, and by making the two recursive calls to BitonicMerge occur in parallel. The parallel time complexity for Bitonic Sort is therefore modeled by $T(n) = T(n/2) + log(n) = O(log^2 n)$.

## IV. Results

We obtained runtime results by testing each algorithm on an 8-core machine. This first test evaluated the effectiveness of multi-threading. For each algorithm, we set thread counts of 1, 2, 4, and 8 with a randomly generated 20,000 element input array. We then obtained and graphed the median runtime result from five runs for each distinct thread count. For the Parallel Bitonic Sort, Parallel Merge Sort, and Parallel Quick Sort, the runtime data indicates expected behavior. As the thread count increases, the runtime of each algorithm decreases on the same size input array because the workload is divided among more processing cores, allowing for concurrent execution of sorting tasks. As the number of threads increases, each thread handles a smaller portion of the input array, reducing the overall computational load and consequently decreasing runtime. However, for Parallel Odd-Even Sort, the runtime data does not show improvement with increasing thread count. This is likely due to the fact that the overhead caused by creating and starting threads on the even and odd array indices is far more inefficient compared to only one thread performing the fast compare and swap operation. In the second test, we further evaluated the performance of each algorithm by subjecting them to a range of input array sizes, specifically 10, 100, 1000, 10000, and 100000 elements, while maintaining a fixed thread count of 8. This test aimed to assess the scalability and efficiency of the algorithms across varying input sizes under a consistent parallel processing environment. The runtime results obtained from this test provide insights into how each algorithm handles different workloads and how their performance scales with the size of the input array. The results for this test indicate that the as the array size increases, the parallel algorithm runtimes scale polylogarithmically, indicating they are relatively efficient. In the final test, we sought to compare the performance of the multithreaded implementations against their sequential counterparts. For the parallel algorithms, we utilized the data obtained in the second test. Subsequently, we subjected the sequential algorithms to the same data points, namely the sequential runtime for array sizes of 10, 100, 1000, and 10000. The data from the test leads to two key observations. Firstly, for both Parallel Merge Sort and Parallel Quick Sort, the runtime of the parallel implementation consistently outperforms the sequential counterpart as the array size increases, aligning with anticipated behavior. However, for Parallel Bitonic Sort and Parallel Odd-Even Sort, a contrasting trend emerges: the parallel runtime often surpasses that of the sequential implementation across most data points. This unexpected outcome is likely attributed to the overhead incurred from thread creation and management as well as potential inefficiencies within our parallel implementation, despite adhering closely to the pseudocode outlined in *Parallel Algorithms*.

## V. Conclusion

In conclusion, our study explored parallel sorting algorithms, focusing on the performance of Merge Sort, Quick Sort, Bitonic Sort, and Odd-Even Sort, tailored for CPU-based architectures, under varying thread counts and input array sizes. Through empirical testing and evaluation, we have demonstrated the effectiveness of parallelization techniques in enhancing sorting efficiency across a range of input sizes and thread counts. Our findings reveal that Parallel Merge Sort, Parallel Quick Sort, and Parallel Bitonic Sort exhibit significant runtime improvements with increasing thread counts, showcasing their suitability for parallel computing environments. Conversely, algorithms like Parallel Odd-Even Sort exhibit limited scalability due to the overheads associated with thread creation and synchronization and may require special parallel environments or implementation to achieve greater efficiency. Furthermore, the research underscores the importance of considering algorithmic design principles and parallelization strategies in the development of efficient parallel sorting algorithms. By leveraging parallelism judiciously and optimizing for specific hardware architectures, researchers can achieve substantial performance gains in sorting tasks.

## References

[1] *Parallel Algorithms*, Chapter 6: ”Sorting Algorithms”