Beyond O(n²): Scaling Bubble Sort with Parallelism

Jay Joshi
NMIMS
Navi Mumbai, India
joshi.r.jay@gmail.com

Muskaan Singh
NMIMS
Navi Mumbai, India
smk.muskaan@gmail.com

Prof. Variza Negi
NMIMS
Navi Mumbai, India
Variza.Negi@nmims.edu

***Abstract*—Bubble Sort is a simple but inefficient sorting algorithm. This paper presents a comparative analysis of several Bubble Sort implementations in Java, including the traditional sequential version and two parallelized approaches (using Java threads and the Fork/Join framework). The goal is to determine how optimizations like parallelism and multithreading can improve Bubble Sort's performance. We implemented and evaluated a sequential Bubble Sort, a parallel version using Java's Fork/Join (with one-level and recursive variants), and a multithreaded version using an Executor Service. The experiments were carried out on varying input sizes to measure execution times. The results show that while the basic Bubble Sort is slow for large inputs, performance improves significantly with multithreading. In particular, the fully optimized Fork/Join implementation achieves the best performance on large datasets, greatly outperforming the sequential algorithm. We summarize the performance trade-offs and recommend the optimized parallel technique for handling larger input sizes.**

***Index Terms*—Bubble Sort, Java, Fork/Join Framework, Multithreading, Parallel Computing, Executor Service, Performance Analysis**

I. Introduction

Sorting is the process of arranging data in a specific order (usually ascending or descending). It is a fundamental operation in computer science because many tasks (such as searching, merging datasets, and data analysis) rely on sorted data [1]. Efficient sorting enables faster lookup and processing of information in various applications, from databases to algorithms that require sorted input.

Bubble Sort is one of the simplest sorting algorithms taught in introductory courses. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. Despite its simplicity, Bubble Sort is known to be highly inefficient for large data sets. Its worst-case time complexity is O(n²), and even in the best case it requires adjacent comparisons throughout the array (unless an early termination check is added) [1]. In practical terms, Bubble Sort performs far more comparisons and swaps than more advanced algorithms like Quicksort or Merge Sort, making it impractical for sorting large arrays.

Given Bubble Sort's poor performance, a natural question arises: Can we optimize Bubble Sort using modern computing techniques? One approach is to take advantage of parallelism and multithreading. Modern processors have multiple cores and support concurrent threads, suggesting that dividing the sorting work among multiple threads might speed it up. If the workload can be split and run in parallel, we could potentially mitigate Bubble Sort's slowness. The motivation for this research is to explore whether parallel computing can significantly improve the execution time of bubble sort and how different parallel implementations compare.

In this paper, we investigate three approaches to accelerating Bubble Sort using Java: a basic sequential version, a parallel version using Java's Fork/Join framework, and a multithreaded version using an Executor Service. We further refine the Fork/Join approach with an optimized variant. Our thesis is that while naive Bubble Sort is extremely slow, careful use of multithreading and parallel algorithm design can yield substantial performance gains, potentially making Bubble Sort viable for larger inputs. The remainder of the paper discusses the background of classification algorithms, the methodology of our implementations, the experimental results, the analysis of those results, and the conclusions with future work suggestions.

Prof. Variza Negi supervised this research work.

II. Background: Overview of sorting algorithms

Sorting algorithms have varying efficiencies in terms of time (speed) and space (memory usage). Table I provides a brief comparison of common sorting algorithms, highlighting their time complexity (best, average, and worst cases) and space complexity:

TABLE I
Comparison of Sorting Algorithm Complexities

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quicksort	O(n log n)	O(n log n)	O(n²)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)

The best case O(n) for Bubble Sort assumes an optimized implementation that can detect a sorted array early and terminate. A naive implementation without a swap check would also be O(n²) in the best case [1].

As shown in the table, comparison-based sorting algorithms like Merge Sort, Quicksort, and Heap Sort have much better average and worst-case complexities O(n log n) than Bubble Sort O(n²). This means that for large n, Bubble Sort will perform orders of magnitude more operations. However, Bubble Sort has the advantage of a simple implementation and O(1) extra space (it sorts in-place). It also has a predictable access pattern (repeated swaps of adjacent elements).

A. How Bubble Sort Works

In a classic Bubble Sort, we iterate through the array multiple times. In each pass, adjacent elements are compared and swapped if they are out of order. This "bubbles up" the largest element to the end of the list on the first pass, the second-largest on the next pass, and so on. The algorithm continues until a pass finds no swaps, indicating the array is sorted. Pseudo-code for the standard Bubble Sort is given below:

```
BubbleSort(arr):
n = length(arr)
for i from 0 to n-2 do
    // Invariant: after i-th pass, last i elements are in correct
    place
    for j from 0 to n-i-2 do
        if arr[j] ¿ arr[j+1] then
            swap arr[j] and arr[j+1]
        end if
    end for
end for
```

This simple approach has well-known problems. Firstly, it is slow: even if the array is partially sorted, Bubble Sort will still compare many adjacent pairs repeatedly. A minor optimization is to track if any swap was made in an iteration; if not, the array is already sorted and the algorithm can terminate early. This can improve the best-case performance to linear time for an already sorted array. However, the worst-case remains quadratic. Secondly, Bubble Sort's iterative adjacent swapping is inherently sequential – each swap depends on previous comparisons, which makes it challenging to parallelize. The next section discusses methodology for introducing parallelism into Bubble Sort to address these issues.

III. Methodology

To evaluate the impact of parallelism on Bubble Sort, we implemented four versions of the algorithm in Java:

1) Serialized Bubble Sort (Sequential) – the standard single-threaded implementation.
2) Parallel Bubble Sort (Fork/Join 1) – a parallel version using Java's Fork/Join framework with a simple two-task split.
3) Multithreaded Bubble Sort (ExecutorService) – a multithreaded implementation using a thread pool (ExecutorService).
4) Fork/Join Optimized Bubble Sort (Fork/Join 2) – an improved Fork/Join implementation with recursive splitting and efficient merging.

Each implementation is described below with its approach and pseudo-code. We then outline the experimental setup used to compare their performance.

A. Serialized Bubble Sort (Single-Threaded)

The serialized version uses the classic Bubble Sort algorithm described earlier. It runs on a single thread and serves as the baseline for performance comparison. In Java, this is implemented with nested loops over the array. The pseudo-code is identical to the one given in the Background section. This algorithm has a time complexity of O(n²) and will typically be the slowest for large inputs, but it has no multithreading overhead.

B. Parallelized Bubble Sort (Fork/Join 1)

The first parallel approach uses Java's Fork/Join framework to divide the sorting work into two tasks. The array is split into two halves, each of which is sorted in parallel using Bubble Sort, and then the two sorted halves are merged. We used Java's ForkJoinPool to execute these tasks concurrently. The pseudo-code for this approach is:

```
ParallelBubbleSort_FJ1(arr):
n = length(arr)
if n ¡= threshold then
    BubbleSort(arr) // small enough to sort directly
else
    mid = n / 2
    // Fork two tasks to sort left and right halves in parallel
    fork ParallelBubbleSort_FJ1(arr[0 .. mid-1])
    fork ParallelBubbleSort_FJ1(arr[mid .. n-1])
    join both tasks // wait for halves to be sorted
    // Merge step: combine two sorted halves
    for i from 0 to n-2 do
            // bubble merge of the two halvesfor j from 0 to
            n-2 do  arr[j] ¿ arr[j+1]
            swap arr[j], arr[j+1]
        end if
    end for
end for
end if
```

In our implementation, we set a threshold (e.g., 1000 elements) below which the task does not fork further and just sorts sequentially. Fork/Join 1, however, performs only one level of parallelism (splitting into two halves once). After both halves are sorted, it merges them by essentially performing another bubble sort over the entire array. This merge step is the same as a full Bubble Sort pass and has O(n²) complexity in the worst case, which can dominate the performance. The expectation was that sorting two halves in parallel might reduce the overall time compared to sorting the whole array sequentially, but the inefficient merging could negate some benefits. This approach demonstrates a straightforward but naive way to parallelize Bubble Sort.

C. Multithreaded Bubble Sort (ExecutorService)

The multithreaded implementation uses a fixed thread pool (Java ExecutorService) to sort different parts of the array concurrently. Instead of using recursion, we divide the array into p segments (where p is the number of threads, chosen based on available processor cores) upfront. Each segment is assigned to a separate thread which performs Bubble Sort on that portion of the array. After all threads complete sorting their segments, a merging phase merges the sorted segments into one fully sorted array.

Pseudo-code for the multithreaded approach:

```
ParallelBubbleSort_Executor(arr, p):
n = length(arr)
segmentSize = ceil(n / p)
// Submit sorting tasks for each segment
for t from 0 to p-1 do
    start = t * segmentSize
    end = min(n-1, (t+1)*segmentSize - 1)
    submit BubbleSort(arr[start .. end]) // sort segment in a thread
end for
wait for all p threads to finish
// Merge phase: iteratively merge sorted segments
current_list = arr[0 .. segmentSize-1] // sorted segment 1
for each next segment s in arr do
    current_list = merge(current_list, s) // merge two sorted lists
    // (merge is linear time combining of two sorted subarrays)
end for
arr = current_list (fully sorted array)
```

We used an ExecutorService to manage the threads and a fixed number of threads equal to the available processor cores (e.g., 8 threads on an 8-core machine). Each thread sorts roughly n/p elements with Bubble Sort. The merging of p sorted subarrays is done sequentially by repeatedly merging two sorted lists at a time (similar to p-1 successive merge operations). The merging step is much more efficient than Bubble Sort's swapping approach: merging two sorted arrays of total length m takes O(m) time. In this implementation, because the segments are sorted independently, the final merge essentially transforms the process into something akin to Merge Sort (with an initial segmentation by index rather than recursion). The multithreaded approach avoids deep recursion and gives more control over thread management at the cost of writing a custom merge routine.

D. Fork/Join Optimized Bubble Sort (Fork/Join 2)

The second Fork/Join implementation improves upon the first by employing a divide-and-conquer strategy more fully. It uses the Fork/Join framework recursively to split the array into smaller subarrays until a threshold size, and crucially, it merges the sorted subarrays in a more optimal way (linear merge) instead of a full quadratic pass. The algorithm can be seen as a hybrid of Bubble Sort (for small pieces) and Merge Sort (for combining results). The pseudo-code is:

```
ParallelBubbleSort_FJ2(arr, start, end):
size = end - start + 1
if size ¡= threshold then
    BubbleSort(arr[start .. end]) // sort directly when small
else
    mid = (start + end) / 2
    fork ParallelBubbleSort_FJ2(arr, start, mid)
    fork ParallelBubbleSort_FJ2(arr, mid+1, end)
    join both subtasks // wait for both halves sorted
    merge(arr, start, mid, end) // linear merge of two sorted halves
end if
```

Like Fork/Join 1, we used a threshold (1000 elements) to decide when to stop splitting further. Fork/Join 2 will recursively fork tasks until subarrays of size 1000 are reached, which are sorted by the standard Bubble Sort. This greatly reduces the workload of each sorting task. After sorting two halves, it merges them by traversing both sorted halves and combining them into a sorted whole (similar to the merge step in Merge Sort). This merge is O(n) for two subarrays of total length n, which is significantly faster than the $O(n^2)$ bubble-merge used in Fork/Join 1. The Fork/Join 2 approach therefore transforms the algorithm's overall complexity: the divide-and-conquer approach yields an approximate time complexity closer to $O(n^2 / k + n \log k)$ (where k is the number of subtasks created) rather than the pure $O(n^2)$ of Bubble Sort. In practice, as we will see, this optimized approach achieves the best performance among the Bubble Sort variants tested.

E. Experimental Setup

All implementations were coded in Java and tested across two different systems to observe how hardware architecture influences performance.

1) System A: Intel-Based Desktop:

- **Device:** Desktop PC
- **Processor:** Intel® Core™ i7 (Quad-Core)
- **RAM:** 16 GB DDR4
- **Operating System:** Windows 10
- **Java Version:** OpenJDK 1.8
- **Architecture:** x86_64, traditional multi-threading support

2) System B: Apple Silicon MacBook:

- **Device:** MacBook Air (2024)
- **Processor:** Apple M3 chip (8-core CPU: 4 performance + 4 efficiency cores)
- **RAM:** 16 GB Unified Memory
- **Operating System:** macOS Sequoia
- **Java Version:** Java SE 23.0.2 (build 23.0.2+7-58)
- **Architecture:** ARM64, optimized for concurrency and memory sharing

3) Execution Environment Details::

- The **Fork/Join** implementations utilized Java's `ForkJoinPool.commonPool()` (which

automatically uses all available processors –8 on the MacBook), while the **ExecutorService** was configured with a fixed thread-pool size of 8 for multithreaded sorting.
- Random input arrays were generated for each test with dataset sizes of: 100, 500, 1000, 5000, 10000, 50000, 100000.
- A **fixed seed** was used to ensure consistent random number generation across all tests for fairness in comparison.
- Each algorithm was tested using **five trial runs** per dataset size, and the **average execution time** was recorded after discarding the first run to allow for JVM warm-up.
- Time was recorded using `System.nanoTime()`.
- To avoid performance skew from task overheads, we used a **threshold of 1000 elements** for Fork/Join splits—determined through empirical testing as the crossover point where parallelism outpaces task division cost.

IV. Results and Analysis

We collected the average execution times of each Bubble Sort implementation across the range of input sizes. Table II summarizes the results:

TABLE II
Average Execution Time (in seconds) vs. Input Size for Different Bubble Sort Implementations

Array Size	Sequential	Odd-Even	Bitonic	Multi-threaded	Fork/Join (v1)	Fork/Join (v2)	Winner (Fastest)
100	0.0001	0.0001	0.0000	0.0020	0.0010	0.0002	Bitonic (2.98×)
500	0.0005	0.0004	0.0001	0.0027	0.0037	0.0006	Bitonic (6.59×)
1,000	0.0009	0.0008	0.0002	0.0032	0.0074	0.0010	Bitonic (5.60×)
5,000	0.0096	0.0078	0.0010	0.0160	0.0936	0.0029	Bitonic (9.23×)
10,000	0.0336	0.0265	0.0022	0.0466	0.2214	0.0033	Bitonic (15.44×)
50,000	1.3390	0.6205	0.0083	1.0577	1.5218	0.0063	Fork/Join v2 (214.2×)
100,000	7.4839	3.0794	0.0179	4.5340	3.6647	0.0110	Fork/Join v2 (678.2×)

Fig. 1 illustrates the performance trends of the implementations (execution time in logarithmic scale vs input size).

A. Performance Analysis of Bubble Sort Implementations

Based on the experimental results, we provide a detailed comparison of six Bubble Sort variations evaluated across array sizes from 100 to 100,000 elements. Each implementation was tested for performance trends and scalability.

1) Sequential Bubble Sort:

- **Growth Pattern:** Demonstrates a clear $\mathcal{O}(n^2)$ time complexity.
- **Efficiency:** Performs efficiently only for very small input sizes (≤ 500).
- **Scalability:** Execution time grows rapidly as the input size increases.
- **Use Case:** Primarily suitable for educational demonstrations or extremely small datasets.

2) Odd-Even Sort (Parallelized):

- Shows minor improvement over Sequential.
- Still exhibits O(n^2) growth.
- Bottlenecked by lack of optimized merge.
- Better than Sequential on mid-sized inputs (1k-10k).

3) Bitonic Sort:

- Fastest for inputs: 100, 500, 1,000, 5,000, and 10,000.
- Outperforms all others by 3× to 15× in these ranges.
- Naturally parallel, benefits from low branching and predictability.
- Best choice for all inputs below 50,000.

4) Multithreaded Bubble Sort (ExecutorService):

- Excellent performance after 5,000+ elements.
- Utilizes 8-thread concurrency effectively.
- Thread overhead hurts performance on small arrays.
- Good balance of simplicity and scalability.

5) Fork/Join 1 (Full Merge):

- Underperforms across all input sizes.
- Only marginal improvement at 100k.
- Merging with full bubble sort nullifies gains.
- Not recommended without algorithmic merge improvement.

6) Fork/Join 2 (Optimized Merge):

- Best performance for 50k and 100k inputs.
- 0.0063s for 50k, 0.0110s for 100k vs 7.48s sequential.
- Low overhead with recursive task splitting.
- Not optimal for small arrays due to setup cost.

Input Size	Recommended Sort	Justification
100 - 5,000	Bitonic Sort	Fastest and most consistent for small inputs.
10,000	Bitonic / Fork/Join 2	Bitonic still slightly faster.
50,000	Fork/Join 2	Best scaling, lowest overhead.
100,000	Fork/Join 2	Dominates all others in performance.

7) Final Recommendations by Input Size:

V. Discussion

The experiment highlights how each Bubble Sort variant behaves under different conditions, and it provides insight into the interplay between algorithmic complexity and parallel execution:

A. Small vs. Large Inputs

For small input sizes (a few hundred elements or less), the overhead of multithreading clearly outweighs any gains. The sequential Bubble Sort often finished fastest for n ¡ 1000 in our results. This is because the absolute runtime is so low that the fixed cost of creating threads or tasks (which can be milliseconds) is more than the time to just sort the data in one thread. As input size grows, the O(n^2) cost starts to dominate and the parallel methods catch up and eventually surpass the sequential time. We observed a turning point around a few thousand elements where multithreading became beneficial.

B. Thread Overhead vs. Benefit

The multithreaded and Fork/Join methods show the classic trade-off in parallel computing. Using multiple threads introduces overhead in terms of thread management, synchronization (waiting for threads to finish), and combining results. If the task (sorting) is too small, these overheads make the parallel version slower (as seen with 100 or 500 elements). However, as the task size increases, the *benefit* of dividing work among threads begins to outweigh the overhead. For example, sorting 100,000 elements took ∼7.48 seconds sequentially, but only ∼0.08 seconds with 8 threads, despite some overhead.

C. Merging Strategy Impact

One of the key lessons from our Fork/Join implementations is the impact of the merging method on overall performance. In Fork/Join 1, after sorting subarrays in parallel, the merge was done by a bubble sort merging (essentially re-sorting the entire array). This negated much of the parallel sorting advantage, since merging took $O(n^2)$ time in the worst case. In Fork/Join 2, we replaced that with a linear merge, which only takes $O(n)$ time, preserving the advantage gained from parallel sorting. The difference between Fork/Join 1 and Fork/Join 2's results demonstrates that **parallelism alone is not enough**—one must also adapt the algorithm to handle the divided data efficiently.

D. Scalability and Efficiency

The parallel efficiency (speedup relative to number of threads) of the multithreaded approaches can degrade if too many threads are used or if load is imbalanced. We used 8 threads for up to 100k data, which worked well; using more threads than cores would likely not help. In Fork/Join 2, the work-stealing mechanism of ForkJoinPool dynamically balances tasks across cores, which helps maintain high CPU utilization. The Executor approach partitioned the data evenly among threads, which also resulted in balanced load. The study by Saadeh and Qatawneh (2019) similarly found that parallel bubble sort runs faster as processor count increases, but the efficiency per processor is higher when using fewer processors [2].

E. Algorithmic Alternatives

It is worth noting that while our focus was on Bubble Sort, there are other algorithms and variations that inherently perform better. For instance, *odd-even sort* (also called odd-even transposition sort) is a parallel variation of bubble sort that swaps pairs in alternating phases and is designed for parallel processors [4]. *Bitonic sort* is another parallel algorithm (often used in hardware or GPU sorting networks) with $O(n \log^2 n)$ complexity [5]. In preliminary tests, we observed that a well-implemented Bitonic sort could outperform our Bubble Sort variants at certain sizes.

VI. Conclusion and Future Work

In this paper, we performed a comparative analysis of Bubble Sort implementations in Java, examining how sequential, parallel, and multithreaded techniques affect performance. We confirmed that the basic Bubble Sort is impractically slow for large inputs due to its $O(n^2)$ time complexity. However, by applying parallel computing techniques, we were able to *significantly* improve Bubble Sort's performance. Among the implementations tested, the Fork/Join optimized Bubble Sort (Fork/Join 2) was the clear winner for large data sets, achieving speedups of several hundred-fold over the sequential version on a 100k array.

In conclusion, **we recommend the Fork/Join divide-and-conquer approach (Fork/Join 2) for anyone attempting to parallelize Bubble Sort on large arrays**. It provides the best performance among the tested techniques by combining multithreading with an improved algorithmic strategy. For small arrays, a sequential sort or minimal threading is preferable, as overhead can dwarf the runtime.

Despite the improvements, it is important to recognize that even the optimized Bubble Sort is essentially emulating a merge sort in the end. For practical purposes, one would typically use algorithms like mergesort or quicksort (or Java's built-in Arrays.sort which uses Dual-Pivot Quicksort, or Arrays.parallelSort which uses a parallel mergesort) for large data, as they are simpler and more reliably efficient [3].

A. Future Work

There are several avenues to explore beyond this work:

- **Parallel Merge Optimizations:** Further improve the merging phase in Fork/Join 2 by implementing parallel merging of subarrays. For example, one could merge multiple pairs of subarrays concurrently in a tree-based fashion to utilize threads even during merging.
- **GPU Acceleration:** Investigate Bubble Sort on GPUs using CUDA or OpenCL. GPUs can run thousands of threads in parallel, and algorithms like odd-even sort or bitonic sort (which relate to bubble sort) are suitable for GPU architecture [6].
- **Hybrid Algorithms:** We could integrate the Bubble Sort approach with other algorithms. For instance, use a threshold at which the algorithm switches from bubble sort to a more efficient method (similar to how introsort switches to heapsort at a certain recursion depth) [7].
- **Memory and Cache Optimizations:** Analyze and improve cache usage in parallel Bubble Sort. Techniques like blocking (tiling the array) could be applied to bubble sort to improve cache hits when dealing with large arrays, especially in parallel context [8].

The code developed for this study can be made available for further research and verification. The full source code and scripts for running the benchmarks are hosted in a GitHub repository, which interested readers can refer to for implementation details or to reproduce the results.

Acknowledgment

The authors would like to thank the Department of Computer Science at NMIMS for providing the computational resources necessary for this research.

References

[1] D. Parekh, "Sorting Algorithms: Slowest to Fastest," *Built In*, Dec. 14, 2022. [Online]. Available: https://builtin.com/machine-learning/fastest-sorting-algorithm

[2] R. Saadeh and M. Qatawneh, "Performance Evaluation of Parallel Bubble Sort Algorithm on Supercomputer IMAN1," *International Journal of Computer Science & Information Technology*, vol. 11, no. 3, pp. 39–48, June 2019.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[4] K. E. Batcher, ”Sorting networks and their applications,” in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 307–314, 1968.

[5] T. P. Buzbee and K. M. Chandy, ”Bitonic Sort: A Parallel Sorting Algorithm,” *Journal of the ACM*, vol. 19, no. 4, pp. 618–624, 1972.

[6] N. Satish, M. Harris, and M. Garland, ”Designing Efficient Sorting Algorithms for Manycore GPUs,” in *Proc. IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, 2009, pp. 1–10.

[7] D. R. Musser, ”Introspective Sorting and Selection Algorithms,” *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[8] A. LaMarca and R. E. Ladner, ”The Influence of Caches on the Performance of Sorting,” *Journal of Algorithms*, vol. 31, no. 1, pp. 66–104, 1999.