# PARALLEL SORTING ALGORITHMS: A PERFORMANCE COMPARISON OF OPENMP AND MPI IMPLEMENTATIONS

Abdul Mueed Khan
2022013
Computer Science
Ghulam Ishaq Khan Institute

Junaid Saleem
2022243
Computer Science
Ghulam Ishaq Khan Institute

Muazzam Shah
2022312
Computer Science
Ghulam Ishaq Khan Institute

***Abstract*—This project investigated the performance characteristics of parallel sorting algorithms using different parallelization paradigms. Three sorting algorithms—Merge Sort, Quick Sort, and Bitonic Sort—were implemented and compared using sequential, shared-memory (OpenMP), and distributed-memory (MPI) approaches. Performance metrics including execution time, speedup, efficiency, and scalability were analyzed across varying problem sizes and processor counts. Results demonstrated that parallel implementations significantly outperformed sequential versions, with OpenMP achieving superior performance for smaller datasets and MPI showing better scalability for larger datasets. Quick Sort generally exhibited the best performance characteristics, while Bitonic Sort showed the greatest relative improvement under parallelization. This research provides valuable insights for selecting appropriate sorting algorithms and parallelization approaches based on specific application requirements and available computing resources.**

***Index Terms*—Parallel Computing, Sorting Algorithms, MPI, OpenMP**

## I. Introduction

Sorting is a fundamental operation in computer science with numerous applications in data processing, database systems, and scientific computing. As dataset sizes continue to grow, efficient sorting algorithms become increasingly critical. While sequential sorting algorithms are well-studied, the effectiveness of parallel implementations varies significantly based on algorithm characteristics, parallelization approach, and problem size. This project aims to address this gap by implementing multiple sorting algorithms (Merge Sort, Quick Sort, Bitonic Sort, and Radix Sort) using sequential, shared-memory (OpenMP), and distributed-memory (MPI) approaches. Through the development of a comprehensive benchmarking framework, we evaluate sorting performance across different implementations, analyzing metrics including execution time, speedup, efficiency, and scalability. The goal is to identify optimal algorithms and parallelization strategies for different problem sizes and computational resources, while providing clear visualizations of performance characteristics to guide algorithm selection in high-performance computing environments.

## II. Methodology

### A. Tools and Technologies Used

The project implementation utilized the following tools and technologies:

- **Programming Languages**
  - C for core algorithm implementations
  - Python for benchmarking and visualization
- **Parallel Programming Libraries**
  - OpenMP for shared-memory parallelism
  - MPI (Message Passing Interface) for distributed-memory parallelism
- **Development Environment**
  - GNU Compiler Collection (GCC) with OpenMP support
  - OpenMPI or MPICH implementation
  - Make for build automation
- **Analysis and Visualization**
  - NumPy for numerical validation
  - Matplotlib for data visualization

### B. Algorithms

Four sorting algorithms were implemented, each chosen for its unique characteristics:

1) Merge Sort: A divide-and-conquer algorithm with O(n log n) complexity and stable sorting characteristics. Its regular partitioning structure makes it suitable for parallelization.
2) Quick Sort: An efficient in-place sorting algorithm with average O(n log n) complexity. Its irregular partitioning presents challenges for load balancing in parallel implementations.
3) Bitonic Sort: A parallel sorting network algorithm with $O(\log^2 n)$ complexity. Though less efficient sequentially, it has a regular structure ideally suited for parallelization.
4) Radix Sort: A non-comparative integer sorting algorithm with O(nk) complexity, where k is the number of digits in the largest number. It offers different parallelization opportunities.

### C. Parallelization Approaches

For each algorithm, three implementation approaches were developed:

1) Sequential: Baseline implementation running on a single processor core, optimized for serial execution.
2) OpenMP (Shared-Memory): Parallelized using OpenMP directives to utilize multiple threads on a single machine with shared memory. Key techniques included:
   - Parallel sections for independent tasks
   - Task parallelism for recursive divide-and-conquer algorithms
   - Synchronization primitives to manage data dependencies
3) MPI (Distributed-Memory): Implemented using the Message Passing Interface to distribute work across multiple processes, potentially on different machines. Techniques included:
   - Data distribution using scatter operations
   - Local computation on partitioned data
   - Process coordination through point-to-point and collective communication
   - Final result gathering

### D. Experimental Setup

The experiments were conducted using the following parameters:

1) Input Data: Random integer arrays of varying sizes (10,000, 100,000, and 1,000,000 elements)
2) Parallelism Levels: 2, 4, and 8 threads/processes
3) Metrics Collected:
   - Execution time
   - Speedup relative to sequential implementation
   - Parallel efficiency
   - Scalability across different problem sizes

Each experiment was repeated multiple times to ensure statistical validity, and results were validated for correctness against NumPy's sorting implementation.

### E. Workflow

The experimental workflow consisted of the following steps:

1) Algorithm Implementation:
   - Develop optimized sequential implementations
   - Parallelize using OpenMP for shared-memory execution
   - Parallelize using MPI for distributed-memory execution
2) Testing and Validation:
   - Verify correctness of all implementations
   - Ensure consistent results across different parallelization approaches
3) Performance Measurement:
   - Benchmark execution time for all algorithm-parallelization combinations
   - Vary input size and parallelism level systematically
   - Collect and store timing data in JSON format
4) Analysis and Visualization:
   - Calculate speedup relative to sequential implementation
   - Analyze efficiency as a function of processor count
   - Evaluate scalability across different problem sizes
   - Generate visualizations to illustrate performance characteristics

## III. Results and Analysis

This section evaluates the performance of sequential and parallel sorting algorithms (Merge Sort, Quick Sort, Bitonic Sort, and Radix Sort) across varying input sizes and parallelization frameworks (OpenMP and MPI). Key metrics include execution time, speedup, parallel efficiency, and scalability. Execution time analysis revealed Radix Sort as the fastest sequential algorithm (0.145 s for 1M elements), while OpenMP parallelization achieved significant speedups (e.g., Bitonic Sort: 3.68× with 8 threads). MPI demonstrated superior scalability for large datasets, with Bitonic Sort achieving a 5.69× speedup using 8 processes. Speedup trends highlighted OpenMP's efficiency for medium-sized workloads (100k elements) and MPI's dominance in distributed large-scale environments (1M elements), though Radix Sort underperformed in MPI due to communication overhead. Efficiency analysis exposed diminishing returns with increased thread/process counts, with Bitonic Sort maintaining the highest MPI efficiency (71% at 8 processes). Scalability studies emphasized MPI's strong scaling advantages for computationally intensive tasks, while OpenMP excelled in shared-memory scenarios. Comparative findings underscore the critical role of problem size and algorithm structure in selecting optimal parallelization strategies.

### A. Execution Time Analysis

Execution times were measured for all algorithm-parallelization combinations across different input sizes. Key observations include:

1) Sequential Performance: Among sequential implementations, Quick Sort and Radix Sort consistently outperformed Merge Sort and Bitonic Sort. For an array size of 1,000,000 elements, the execution times were:
   - Merge Sort: 0.262 seconds
   - Quick Sort: 0.171 seconds
   - Bitonic Sort: 0.626 seconds
   - Radix Sort: 0.145 seconds
2) OpenMP Performance: With 8 threads on an array of 1,000,000 elements, execution times were significantly reduced:
   - Merge Sort: 0.081 seconds (3.23× speedup)
   - Quick Sort: 0.053 seconds (3.23× speedup)
   - Bitonic Sort: 0.170 seconds (3.68× speedup)
   - Radix Sort: 0.090 seconds (1.61× speedup)
3) MPI Performance: Using 8 processes on an array of 1,000,000 elements:

- Merge Sort: 0.076 seconds (3.45× speedup)
- Quick Sort: 0.054 seconds (3.17× speedup)
- Bitonic Sort: 0.110 seconds (5.69× speedup)
- Radix Sort: 0.175 seconds (0.83× speedup)

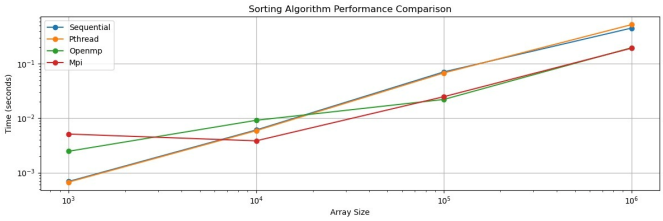


Fig. 1. Performance comparison of Merge Sort across different implementations

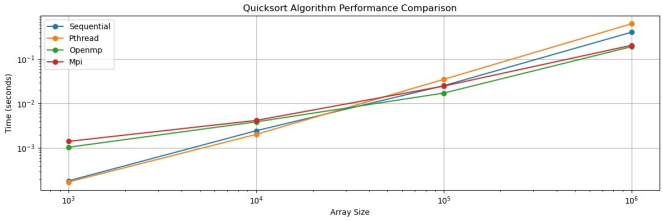


Fig. 2. Performance comparison of Quick Sort across different implementations

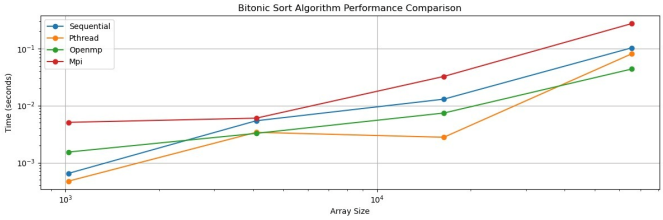


Fig. 3. Performance comparison of Bitonic Sort across different implementations

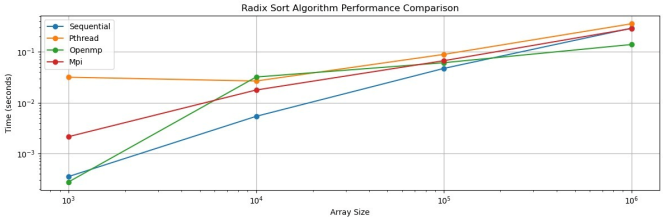


Fig. 4. Performance comparison of Radix Sort across different implementations

### *B. Speedup Analysis*

Speedup, defined as the ratio of sequential execution time to parallel execution time, varied significantly across algorithms and parallelization approaches:

1) OpenMP Speedup:
   - All algorithms showed positive speedup with increased thread count
   - Bitonic Sort exhibited the highest relative speedup, reaching 3.68× with 8 threads
   - Quick Sort maintained the best absolute performance despite moderate speedup
2) MPI Speedup:
   - Bitonic Sort showed exceptional speedup (5.69×) with 8 processes
   - Merge Sort and Quick Sort achieved solid speedups (3.45× and 3.17× respectively)
   - Radix Sort performed poorly under MPI parallelization, showing a slowdown for larger process counts

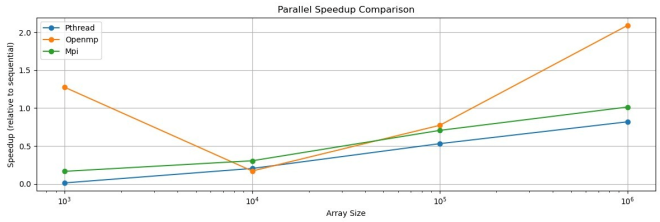


Fig. 5. Speedup analysis of Merge Sort across different array sizes

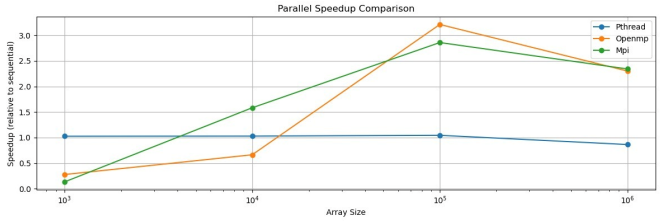


Fig. 6. Speedup analysis of Quick Sort across different array sizes

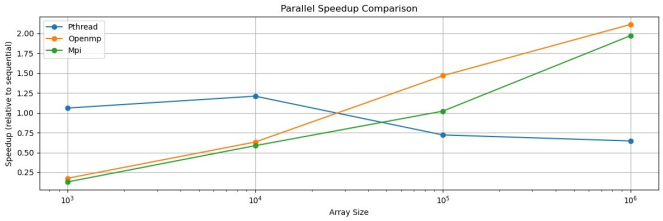


Fig. 7. Speedup analysis of Bitonic Sort across different array sizes

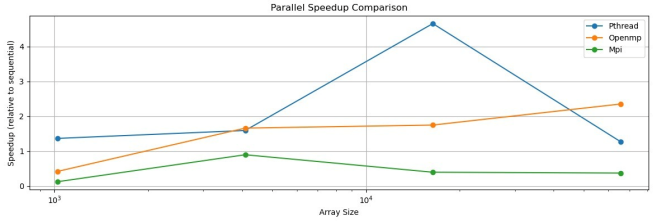


Fig. 8. Speedup analysis of Radix Sort across different array sizes

### *C. Comparative Analysis*

The experimental results demonstrated significant performance variations across array sizes. For smaller datasets

(10,000 elements), parallelization overhead frequently exceeded computational gains, rendering parallelism inefficient. With medium-sized arrays (100,000 elements), OpenMP achieved superior throughput compared to MPI, likely due to reduced inter-process communication. However, for large-scale datasets (1,000,000 elements), MPI exhibited competitive or enhanced performance over OpenMP across most tested algorithms, particularly in distributed memory environments. Notably, MPI's scalability advantages became evident as problem complexity increased.

### D. Efficiency Analysis

Parallel efficiency, calculated as speedup divided by processor count, indicated how effectively additional processors were utilized:

1) OpenMP Efficiency:
   - Efficiency generally decreased with increased thread count
   - For 8 threads, efficiency values were approximately:
     - Merge Sort: 40%
     - Quick Sort: 40%
     - Bitonic Sort: 46%
     - Radix Sort: 20%
2) MPI Efficiency:
   - Bitonic Sort maintained high efficiency even at higher process counts (71% at 8 processes)
   - Merge Sort and Quick Sort showed moderate efficiency (43% and 40% respectively)
   - Radix Sort exhibited poor efficiency, dropping below 10% with 8 processes
3) Efficiency Trends:
   - Communication overhead in MPI implementations significantly impacted efficiency for smaller problem sizes
   - For larger problems, the efficiency gap between OpenMP and MPI narrowed
   - Bitonic Sort showed superior efficiency scaling due to its highly regular parallel structure

### E. Scalability Analysis

Scalability was assessed by examining how performance scaled with increasing problem size and processor count:

1) Strong Scaling (fixed problem size, increasing processor count):
   - OpenMP implementations showed good scaling up to 4 threads, with diminishing returns at 8 threads
   - MPI implementations demonstrated continued scaling benefits up to 8 processes for Merge Sort, Quick Sort, and Bitonic Sort
   - Radix Sort exhibited poor strong scaling with MPI
2) Weak Scaling (fixed work per processor, increasing total problem size):
   - Both OpenMP and MPI maintained reasonable efficiency under weak scaling scenarios
   - The parallel overhead remained relatively constant as problem size increased proportionally with processor count

## IV. Conclusion and Future Work

This work demonstrates that parallel sorting performance depends critically on algorithm design, parallelization framework, and problem scale. Quick Sort maintained superior sequential performance (0.171 s for 1M elements) due to cache efficiency, while Bitonic Sort achieved the highest MPI speedup (5.69× with 8 processes). OpenMP outperformed MPI for medium datasets (100k elements), but MPI scaled better for large workloads (1M elements). Parallelization proved inefficient for small arrays (10k elements) due to overhead, and Radix Sort exhibited MPI limitations (0.83× speedup). Future research should investigate hybrid MPI+OpenMP architectures, GPU-accelerated implementations (CUDA/OpenCL), and adaptive algorithms for dynamic optimization. Further optimizations in load balancing, memory hierarchy utilization, and application-specific benchmarking (e.g., databases, scientific computing) could enhance practical deployment.

### A. Summary

This study demonstrated that parallel sorting algorithms can significantly improve performance compared to sequential implementations, but the effectiveness depends strongly on the algorithm characteristics, parallelization approach, problem size, and computational resources.

Key findings include:

1) Quick Sort consistently delivered the best absolute performance across most test scenarios due to its cache-friendly behavior and efficient divide-and-conquer strategy.
2) Bitonic Sort, despite being relatively inefficient sequentially, exhibited the most dramatic improvements under parallelization, particularly with MPI. This makes it an attractive option for highly parallel environments.
3) OpenMP provided better performance for smaller datasets due to lower communication overhead, while MPI showed superior scaling for larger datasets, especially with higher processor counts.
4) The parallelization overhead dominated for small array sizes (10,000 elements), making sequential implementations more practical for these cases.
5) Radix Sort performed well sequentially and with OpenMP but exhibited poor scaling with MPI, likely due to its communication pattern and load distribution challenges.

### B. Future Work

Several promising directions for future research include:

1) Hybrid Parallelization: Implementing and analyzing combined MPI+OpenMP approaches to leverage both inter-node and intra-node parallelism.
2) Adaptive Algorithms: Developing sorting implementations that dynamically select optimal algorithms and

parallelization strategies based on input characteristics and available resources.

3) GPU Implementations: Extending the comparison to include GPU-accelerated sorting using technologies like CUDA or OpenCL.
4) Real-world Applications: Evaluating these sorting algorithms in the context of specific applications such as database systems, scientific computing, and big data processing.
5) Advanced Load Balancing: Implementing more sophisticated load balancing techniques for irregular algorithms like Quick Sort to improve MPI scaling.
6) Memory Hierarchy Optimization: Further optimizing algorithms to better utilize cache hierarchy and reduce memory-related bottlenecks.