Parallel Sort Investigation

Lab assignment

April 2025

1 Introduction

Student Name: David Andreea-Raluca
Program: Informatics
Group: 10LF221
System Configuration: 11th Gen Intel(R) Core(TM) i5-11400H

2 Task 1 - Implementation

In this project, an MPI-based program written in C/C++ was developed to sort an array of 1,000,000 elements using the following sorting algorithms, each implemented with parallel support:

- Direct Sort
- Bucket Sort
- Odd-Even Sort
- Ranking Sort
- Shell Sort

The algorithms were executed and evaluated on systems using 1, 2, 4, 6, 8, and 12 processing cores. The implementation successfully managed the full dataset of 1 million elements across all tested core configurations, demonstrating scalability and parallel efficiency.

3 Task 2.1 - Execution times, communication times, and processing times

3.1 Measure Execution Time

Execution times were measured using `MPI_Wtime()` across varying MPI processes (1, 2, 4, 6, 8, 12). Key observations:

- **Direct Sort** and **Odd-Even Sort** show near-linear speedup, indicating strong scalability.
- **Bucket Sort** and **Shell Sort** exhibit diminishing returns beyond 4–6 processes due to communication overhead.
- **Ranking Sort** has the worst absolute performance but achieves reasonable speedup (8.4× at 12 processes).

3.2 Measure Computation Time

The actual processing time was measured by isolating local operations while excluding all MPI communication.

- **Direct Sort:** 72-98% of time spent on local $O(n^2)$ selection sorts
- **Bucket Sort:** 49-92% on efficient std::sort (O(n log n)), but sensitive to data distribution
- **Odd-Even Sort:** 60-65% on computations, the rest on communication
- **Ranking Sort:** 98% consumed by $O(n^2)$ pairwise comparisons - fundamentally unscalable
- **Shell Sort:** 60-98% wasted on sequential gap insertion passes that resist parallelization

3.3 Measure Communication Overhead

Communication costs were derived by subtracting computation time from total runtime.

- **Direct Sort:** 2-28% overhead (efficient Scatterv/Gatherv)
- **Bucket Sort:** 8-51% (costly Gatherv redistribution)
- **Odd-Even Sort:** 35-60% (frequent Sendrecv between neighbors)
- **Ranking Sort:** 2% (minimal, but computation-bound)
- **Shell Sort:** 2-40% (moderate, but poor speedup overall)

3.4 Scalability Analysis

To assess scalability, experiments were conducted using different numbers of MPI processes (1, 2, 4, 6, 8, 12 cores). The scalability of each sorting algorithm was analyzed by monitoring the changes in execution times as the number of processes increased. Speed-up was calculated as the ratio of the sequential execution time to the parallel execution time, as follows:

$$\text{Speed-up} = \frac{\text{Execution Time (1 process)}}{\text{Execution Time (n processes)}}$$

A table summarizing the execution times and speed-up achieved for each sorting algorithm across different numbers of MPI processes is provided below:

Algorithm	Number of Processes	Execution Time in seconds	Speed-up
Direct Sort	1	1133.719	1
Direct Sort	2	558.757	2.0290018738
Direct Sort	4	265.317	4.2730733425
Direct Sort	6	131.55	8.6181603953
Direct Sort	8	90.7394	12.4942307311
Direct Sort	12	54.6494	20.7453146787
Bucket Sort	1	0.38623	1
Bucket Sort	2	0.205911	1.8757132936
Bucket Sort	4	0.118	3.2731355932
Bucket Sort	6	0.0938614	4.1148970716
Bucket Sort	8	0.0996603	3.8754649545
Bucket Sort	12	0.0900231	4.2903432563
Odd-Even Sort	1	1722.128	1
Odd-Even Sort	2	766.167	2.2477188394
Odd-Even Sort	4	383.083	4.4954435462
Odd-Even Sort	6	255.389	6.7431565181
Odd-Even Sort	8	169.658	10.1505852951
Odd-Even Sort	12	95.8618	17.9646950089
Ranking Sort	1	22523.630	1
Ranking Sort	2	10690.92	2.1067999761
Ranking Sort	4	8018.19	2.8090666347
Ranking Sort	6	5345.46	4.2135999521
Ranking Sort	8	4009.095	5.6181332695
Ranking Sort	12	2672.73	8.4271999042
Shell Sort	1	0.901811	1
Shell Sort	2	0.707856	1.2740034696
Shell Sort	4	0.494514	1.823630878
Shell Sort	6	0.45251	1.992908444
Shell Sort	8	0.58021	1.5542837938
Shell Sort	12	0.390441	2.3097241325

4 Task 2.2 - Discussion and analysis of results

4.1 Comparative Analysis of Sorting Methods

- **Fastest Execution:** Bucket Sort (0.09s at 12 processes) benefits from O(n log n) local sorting
- **Best Scalability:** Direct Sort achieves 20.7× speedup due to minimal communication

- **Worst Performer:** Ranking Sort's $O(n^2)$ complexity makes it impractical despite low communication

4.2 Discussion of Communication Overhead

Communication impacts vary dramatically:

- **Most Sensitive:** Odd-Even Sort (60% overhead at 12 processes) due to:
  - Synchronous `MPI_Sendrecv` in every phase
  - No computation-communication overlap
- **Optimization Potential:**
  - Replace with non-blocking `MPI_Isend/Irecv`
  - Implement message bundling for boundary elements
  - Use topology-aware communicators to reduce latency

4.3 Analysis of Computational Time and Bottlenecks

Critical computation inefficiencies:

- **Direct Sort:** $O(n^2)$ local selection sort → Replace with `std::sort`
- **Bucket Sort:** Load imbalance during redistribution → Dynamic bucket ranges
- **Odd-Even Sort:** Idle cycles during neighbor synchronization → Overlap computation/communication with non-blocking MPI
- **Ranking Sort:** Fundamental $O(n^2)$ limit → Switch to Radix Sort
- **Shell Sort:** Sequential gap passes → Hybrid MPI+OpenMP for local parallelism

4.4 Speedup and Efficiency Evaluation

Table 1: Speedup and Efficiency Comparison at 12 Processes

Algorithm	Theoretical Speedup	Actual Speedup	Efficiency
Direct Sort	12×	20.7×	1.73
Bucket Sort	12×	4.3×	0.36
Odd-Even Sort	12×	17.9×	1.49
Ranking Sort	12×	8.4×	0.70
Shell Sort	12×	2.3×	0.19

Key observations:

- **Superlinear speedup** in Direct Sort (20.7×) occurs due to:
  - Cache effects from distributed memory partitioning
  - Efficient memory access patterns during local sorting
- **Sublinear performance** in Bucket Sort (4.3×) results from:
  - MPI_Gatherv overhead during bucket redistribution
  - Load imbalance when n/p is small
- Efficiency drops below 1 when communication exceeds 50% of runtime (e.g., Bucket Sort at 51% overhead)
- Odd-Even Sort maintains good efficiency (1.49) despite high communication due to:
  - Balanced computation phases between communications
  - Regular communication patterns

4.5 Possible Inefficiencies and Suggestions for Improvement

Cross-Algorithm Improvements:

- **Load Balancing:** Implement work-stealing for uneven distributions
- **Communication:** Use `MPI_Neighbor_alltoallv` for topology-aware exchanges
- **Hybrid Model:** Combine MPI with OpenMP for intra-node parallelism
- **Memory Access:** Optimize data layouts for cache locality (blocking/strided patterns)

Implementation Roadmap:

1. Profile to identify dominant bottlenecks (compute vs. communication)
2. Implement hybrid parallelization for compute-bound algorithms
3. Optimize data redistribution patterns for communication-bound cases
4. Validate improvements with strong/weak scaling tests

Table 2: Key inefficiencies and optimization strategies

Algorithm	**Inefficiency**	**Optimization Strategy**
Direct Sort	• O(n^2) local selection sort • Single-threaded final merge	• Replace with `std::sort` (O(n log n)) • Parallel merge with task stealing
Bucket Sort	• Static bucket ranges cause imbalance • All-to-all communication during redistribution	• Dynamic bucket sizing via histogram • Staged redistribution pipeline
Odd-Even Sort	• Synchronous neighbor exchanges • Process idle time during communication	• Non-blocking `MPI_Isend/Irecv` • Communication-computation overlap
Ranking Sort	• Fundamental O(n^2) complexity • Minimal parallelization benefit	• Replace with Radix Sort (O(n)) • Hybrid MPI+OpenMP implementation
Shell Sort	• Sequential gap passes • Poor cache utilization	• Block-based gap sorting • Intra-node OpenMP parallelization