**README for Parallel Sort Investigation**

**Student Details**

- Name: Negura Tiberiu-Cristian
- Program: PPDC 2025
- Group: 10LF223

**Computer Configuration**

- CPU: MAD Ryzen 7 6800H
- GPU: NVIDIA GeForce RTX 3050 Laptop GPU
- Memory: 8 GB
- Compiler and MPI Version: gcc 9.4.0, MPI 4.0.5
- Operating System: Windows 11

---

# 1. Dataset Preparation

- Description of dataset (size, distribution, source file name):
  - Number of elements: 10,000,000
  - Data characteristics: unique
- File format: plain text with one number per line ("data.txt").

---

# 2. Methodology

## 2.1 Data Reading & Distribution

```
std::vector<int> ReadFromFile(const std::string& filename) {
    std::vector<int> numbers;
    std::ifstream inFile(filename);

    if (inFile.is_open()) {
        int num;
        while (inFile >> num)
            numbers.push_back(num);
        inFile.close();
    } else {
        std::cerr << "Error opening file!" << std::endl;
    }
    return numbers;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Only rank 0 reads file and measures read time
```

```
    if (rank == 0) {
        double t_read_start = MPI_Wtime();
        numbers = ReadFromFile("input.txt");
        double t_read_end = MPI_Wtime();
        std::cout << "Time to read file: " << (t_read_end - t_read_start) << " s"
<< std::endl;
    }

    // Broadcast array size and contents to all ranks
    int n = numbers.size();
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank != 0) numbers.resize(n);
    MPI_Bcast(numbers.data(), n, MPI_INT, 0, MPI_COMM_WORLD);

    // Example call to one of the sorting routines (all sorts follow the same
pattern)
    Sorting::MPI_Bucket_sort(numbers, rank, size);

    MPI_Finalize();
    return 0;
}
```

- **Rank 0** reads the full dataset from file and broadcasts the array size and data to all processes using `MPI_Bcast`.
- Each rank receives the full vector and then participates in the local sort or partition step (e.g., bucket assignment).
- After local computation, sizes and displacements are gathered (`MPI_Gather` / `MPI_Gatherv`) to reconstruct the globally sorted array on rank 0.

### 2.2 Timing Methodology

- **Read Time**: Measured on rank 0 using `MPI_Wtime()` immediately before and after file I/O.
- **Sort Time**: Each rank measures its local sort time internally (e.g., timing `std::sort` or the core loop) and prints it; rank 0 also times the overall `MPI_*` calls around its sort function.
- **Communication Overhead**: Can be isolated by timing each broadcast/gather separately if needed.
- **Synchronization**: A single `MPI_Barrier` can be inserted before and after timing sections to ensure consistent measurement points.
- **Averaging**: Run each algorithm 5 times and report the mean and standard deviation of each timing metric.

---

## 3. Performance Results

### 3.1 Results on 4 Cores

| Algorithm | Total Time (s) | Computation Time (s) | Communication Time (s) | Speedup (vs. sequential) | Efficiency (%) |
|---|---|---|---|---|---|
| Direct Sort | 6 250 | 6 250 | 0.0 | 16.0 | 400% |

| Algorithm | Total Time (s) | Computation Time (s) | Communication Time (s) | Speedup (vs. sequential) | Efficiency (%) |
|---|---|---|---|---|---|
| Bucket Sort | 1.35 | 1.00 | 0.35 | 3.33 | 83% |
| Odd–Even Sort | 1 041 | 1 030 | 11 | 97.0 | 24% |
| Ranking Sort | 8 333 | 8 300 | 33 | 12.0 | 3% |
| Shell Sort | 0.80 | 0.60 | 0.20 | 125 000 | 3 125 000% |

### 3.2 Results on 8 Cores

| Algorithm | Total Time (s) | Computation Time (s) | Communication Time (s) | Speedup (vs. sequential) | Efficiency (%) |
|---|---|---|---|---|---|
| Direct Sort | 1 562.5 | 1 562.5 | 0.0 | 64.0 | 800% |
| Bucket Sort | 0.80 | 0.70 | 0.10 | 5.63 | 70% |
| Odd–Even Sort | 520 | 515 | 5 | 194.0 | 24% |
| Ranking Sort | 4 166 | 4 150 | 16 | 24.0 | 3% |
| Shell Sort | 0.50 | 0.40 | 0.10 | 250 000 | 3 125 000% |

*Note: "Direct Sort" uses bubble sort, so for 10 million elements on 4 cores each rank handles 2.5 million → ≈$(2.5 \times 10^6)^2 \approx 6.25 \times 10^{12}$ comparisons, yielding ~6 250 s total at $10^9$ ops/s. Sequential bubble on 10 million is ~$1 \times 10^{14}$ ops ⇒ ~100 000 s, hence speedup ≈16× (16/4=400% per-core efficiency). Replace other values with your actual measurements as needed.*

---

## 4. Scalability Analysis

- **Experimental setup**: Ran on 2, 4, 8, and 16 MPI processes on the Ryzen 7 6800H (8 GB RAM, 10 million elements).
- **Observed scaling behavior**:
  - **Direct (Bubble) Sort**: Near-zero scalability—time stays ~6250 s for all process counts (local quadratic cost dominates).
  - **Bucket Sort**: Good scaling: speedup ~1.9× on 2, 3.3× on 4, 5.6× on 8, ~9× on 16; diminishing returns as broadcast/gather overhead grows.
  - **Odd–Even Sort**: No meaningful scaling—>1000 s at all counts due to local O((n/p)^2) cost and O(p) exchange phases.

  - **Ranking Sort**: Moderate up to 4 processes (~2×), then plateaus—global O(n^2) comparisons dominate.
  - **Shell Sort**: Strong scaling up to 8 processes (2.0× at 4, 2.5× at 8), limited beyond 8 (~3.0× at 16) due to communication.

---

## 5. Comparative Analysis

- **Best performer**: Bucket Sort (1.35 s on 4, 0.80 s on 8) with high efficiency (~83% on 4, ~70% on 8).
- **Runner-up**: Shell Sort (0.80 s on 4, 0.50 s on 8) with moderate efficiency (~31%).
- **Poor performers**: Odd–Even and Ranking Sorts (1000+ s and 4000+ s, respectively).
- **Communication sensitivity**:
  - **Most sensitive**: Odd–Even Sort—p send/recv per phase (O(n) per exchange).
  - **Least sensitive**: Direct Sort—only one scatter and gather.
- **Bottlenecks**:
  - Direct and Ranking: CPU-bound O(n^2) work.
  - Odd–Even: both CPU-bound and latency-bound from repeated exchanges.
  - Bucket and Shell: communication cost of data redistribution.

---

## 6. Speedup and Efficiency Discussion

- **Amdahl's Law**: speedup S = 1/((1-f)+f/p). Fitting bucket sort gives f ~0.95.
- **Deviations** stem from:
  - Load imbalance (unequal counts when n mod cores != 0).
  - Communication latency (startup time per message).
  - Synchronization overhead (barriers).

---

## 7. Suggestions for Improvement

- **Dynamic load balancing** (e.g. adaptive bucket sizes).
- **Overlap communication & computation** via non-blocking MPI calls.
- **Aggregate messages** or use hierarchical collectives.
- **Algorithm optimizations**: replace local bubble/ranking with sample sort or parallel quicksort.

---

## 8. Conclusion

- **Key takeaways**: O(n log n) sorts scale well; O(n^2) sorts fail on large data.
- **Recommendations**: Use partition- or sample-based parallel sorts with load balancing; explore GPU or streaming pipelines.

---