CS542200 Parallel Programming
Homework 1: Odd-Even Sort

Kai-Yuan Jeng 鄭凱元
113062529
kaiyuanjeng@gapp.nthu.edu.tw

1 Implementation

This section outlines the core implementation details, focusing on how the program robustly handles arbitrary inputs and performs parallel data management.

1.1 Handling Arbitrary Inputs and Processes

The program is designed to be highly flexible, gracefully handling any number of input items (`N`) and MPI processes (`world_numtasks`). This is achieved through a multi-stage process:

1. **Safe Input Parsing:** The input size `N` is parsed from command-line arguments using `strtol` for safer string-to-integer conversion compared to `atoi`.
2. **Dynamic Process Activation:** A key design choice is to only utilize processes that will actually handle data. If `N < world_numtasks`, redundant processes are excluded by creating a new MPI communicator, `active_comm`, containing only the first `N` ranks. This prevents wasting resources and simplifies logic, as all subsequent operations occur within this active group.

   ```
   // Only create a new communicator for processes that might receive data.
   const int active_numtasks = std::min(world_numtasks, N);
   int *active_ranks = (int *)malloc(active_numtasks * sizeof(int));
   // ... (populate active_ranks) ...
   MPI_Group_incl(orig_group, active_numtasks, active_ranks, &active_group);
   MPI_Comm_create(MPI_COMM_WORLD, active_group, &active_comm);
   ```

3. **Balanced Data Distribution:** Data is distributed as evenly as possible among the active processes. The calculation handles remainders by assigning one extra element to the first `N % numtasks` processes. Each process computes its own element count (`my_count`) and its starting file offset (`my_start_index`) for parallel I/O.

   ```
   const int base_chunk_size = N / numtasks;
   const int remainder = N % numtasks;
   const int my_count = (my_rank < remainder) ? base_chunk_size + 1 :
   ↪  base_chunk_size;
   const int my_start_index = my_rank * base_chunk_size + std::min(my_rank,
   ↪  remainder);
   ```

4. **Robust Memory Management:** Before any sorting begins, the program performs rigorous memory allocation.
   - It includes integer overflow checks to prevent crashes when calculating buffer sizes for very large `N`.
   - It prioritizes `aligned_alloc` to allocate 32-byte aligned buffers, which is critical for enabling SIMD optimizations (detailed in later Section).
   - For enhanced robustness, it implements a fallback mechanism to standard `malloc` if `aligned_alloc` fails, ensuring the program can run even in less-than-ideal environments.

5. **Parallel File I/O:** The program utilizes MPI-IO for all file operations. Each active process reads and writes its assigned data chunk concurrently using `MPI_File_read_at` and `MPI_File_write_at`. This parallel approach is highly scalable and avoids the bottleneck of routing all I/O through a single process.

```
// Each active process reads its own data chunk in parallel.
MPI_File_open(active_comm, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL,
↪  &input_file);
MPI_File_read_at(input_file, my_start_index * sizeof(float), local_data,
↪  my_count, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&input_file);
```

Through this structured approach, the implementation correctly partitions the problem, manages resources efficiently, and performs scalable I/O, forming a solid foundation for the subsequent parallel sorting algorithm.

1.2 Sorting Strategy: A Two-Level Hierarchical Approach

The core of the program is a two-level sorting strategy that combines a highly-optimized initial **local sort** with a communication-efficient **global sort** using the parallel Odd-Even algorithm.

Level 1: Adaptive Local Sort (Poly-algorithm). Before any parallel communication, each process sorts its local data partition. This crucial first step significantly reduces the complexity of the subsequent global sort. To maximize performance across all possible data sizes per process, an adaptive, poly-algorithmic approach is implemented:

- **For tiny arrays ($N < 33$):** A simple **Insertion Sort** is used. Its low overhead and excellent performance on small, nearly-sorted data make it the ideal choice, avoiding the setup costs of more complex algorithms.
- **For medium arrays ($33 \leq N < 1025$):** We leverage `std::sort`, a highly-optimized Introsort implementation that provides fast average-case performance with worst-case guarantees.
- **For large arrays ($N \geq 1025$):** We employ `boost::sort::spreadsort::float_sort`, a specialized radix-based algorithm that is exceptionally fast for floating-point data types, often outperforming comparison-based sorts on large, random distributions.

This tiered strategy ensures that the best possible sorting algorithm is applied for any given workload, forming a critical baseline optimization.

Level 2: Optimized Global Sort (Parallel Odd-Even). The global sort iteratively refines the data order across all processes. This implementation introduces two major optimizations to the standard Odd-Even sort algorithm:

Conditional Merge-Split ("Sense-before-Merge"). Recognizing that in later phases, many adjacent partitions are already sorted relative to each other, we implemented a conditional merging strategy. Instead of blindly exchanging and merging large data chunks, each pair of neighboring processes first performs a near-zero-cost exchange of their single boundary elements.

```
// Left process exchanges its last element with right process's first
MPI_Sendrecv(&my_last, 1, MPI_FLOAT, partner, ..., &partner_boundary, 1, ...);

// The expensive merge is only performed if partitions are out of order
if (my_last > partner_boundary) {
    // Exchange full data chunks and merge...
}
```

This optimization dramatically reduces both communication volume and computational work, especially when the data is approaching a globally sorted state.

Zero-Copy Merge-Split. The merge-split operation itself is a significant performance bottleneck. To address this, we designed a "zero-copy" merge technique. When a merge is required, the sorted output is written to a temporary buffer. Then, instead of copying the merged data back, we perform an $\mathcal{O}(1)$ pointer swap.

```
void merge_sort_split(float *&local_data, /*...*/) {
    // ... merging logic writes sorted data into `temp` buffer ...

    // The O(1) pointer swap, core of the Zero-Copy strategy
    std::swap(local_data, temp);
}
```

This approach completely eliminates a costly, $\mathcal{O}(N/p)$ `memcpy` operation from the critical path, freeing up significant memory bandwidth and CPU cycles for other tasks. This is arguably the single most impactful optimization in the main loop.

1.3 Other Efforts: Robustness and Algorithmic Exploration

Beyond the core optimizations, several other efforts were made to enhance both the program's performance and robustness, demonstrating a deeper exploration of parallel algorithm design.

Efficient Early Exit Mechanism. The theoretical upper bound for the number of phases in an Odd-Even sort is `numtasks`. However, for certain pathological data distributions, more phases might be required to guarantee correctness. To create a robust solution, the maximum number of phases was conservatively set to `numtasks + numtasks / 2`.

To counteract the potential overhead of these extra phases, especially for nearly-sorted data, an efficient early exit mechanism was implemented.

- A `sorted_check` function is called periodically (every two phases) after an initial sorting period (`phase >= numtasks / 2`).
- This function uses the same low-cost boundary-exchange principle as our conditional merge. Each process checks if its first element is correctly ordered with respect to its left neighbor's last element.
- A single, efficient `MPI_Allreduce` with the `MPI_LAND` operation is then used to determine if the *entire* global array is sorted. If it is, the main loop terminates immediately.

This strategy avoids unnecessary computation and communication, saving significant time on well-behaved or partially sorted inputs, without sacrificing correctness on worst-case inputs.

```
// The check is performed only on even phases to reduce communication overhead
if (phase >= numtasks / 2 && !phase_odd)
    // A single collective call determines if the entire array is sorted
    if (sorted_check(local_data, my_count, my_rank, numtasks, phase, active_comm))
        break; // Exit the main loop early
```

Exploration of an Alternative "Element-wise" Odd-Even Sort. In the pursuit of performance and a deeper understanding of the algorithm, an alternative implementation was developed. This version adhered to a more literal interpretation of the Odd-Even sort, where "odd" and "even" referred to the **global indices of elements**, not the ranks of processes.

In this model, communication was limited to exchanging only single boundary elements between processes in each phase, while internal swaps occurred element-wise within each local array.

```
// This alternative, element-wise approach was explored
if (iteration % 2) { // odd phase
    // ... logic to handle swaps between elements at global odd/even indices ...
    // MPI communication only for single boundary elements if they form a pair
    if (my_end_index % 2 && my_rank + 1 < numtasks) {
        MPI_Sendrecv(&my_last_data, 1, ...); // Only one float
        // ...
```

```
    }
    // ...
}
```

Findings. This element-wise implementation, while logically sound and passing most small test cases, proved to be far less performant on larger datasets, leading to timeouts. The experiment provided a critical insight: the "block-based" interpretation of Odd-Even sort, where entire sorted chunks are merged, is vastly more efficient in a distributed memory environment. The high communication latency associated with numerous small, single-element exchanges makes the element-wise approach impractical. This exploration validated our final design choice as the superior parallel strategy.

2 Experiment & Analysis

The primary goal of our experiments is to evaluate the **strong scalability** of our optimized parallel sort algorithm. We aim to quantify the trade-offs between parallel computation and communication overhead as the number of processes increases, and to compare our measured speedup against the theoretical ideal.

2.1 Methodology

System Specification. All experiments were conducted on the provided **Apollo Origo Cluster**. Each compute node is equipped with two Intel(R) Xeon(R) X5670 @ 2.93GHz processors, providing a total of 12 physical cores per node. The nodes are interconnected via a high-speed InfiniBand network.

Performance Metrics and Measurement. To achieve a comprehensive performance profile, we employed a dual-measurement strategy, enabled by a `-DPROFILING` compile-time flag:

1. **Quantitative Analysis (for plots):** The code was instrumented with high-precision `MPI_Wtime` timers to capture the wall-clock time of three distinct components:
   - **I/O Time:** Time spent in `MPI_File_*` operations.
   - **Communication Time:** Time spent in all blocking MPI communication calls within the main loop, including `MPI_Sendrecv` and `MPI_Allreduce`.
   - **CPU Time:** Calculated as `Total Time - (I/O Time + Communication Time)`. This primarily represents the time spent on local sorting and merging operations.

   This coarse-grained data was used to generate the time profile and speedup plots.
2. **Qualitative Analysis (for verification):** We utilized the **NVIDIA Tools Extension (NVTX)** library to annotate critical code regions. This allows for detailed timeline visualization in Nsight Systems, helping us to qualitatively verify our quantitative measurements and gain deeper insights into the program's dynamic behavior. A custom macro was developed to automatically color-code different operations for enhanced readability, as shown in the code snippet below.

```
// Example of the dual-measurement strategy for an MPI call
#ifdef PROFILING
temp_start = MPI_Wtime();
NVTX_PUSH("Boundary_Check"); // Pushes a colored range to the Nsight timeline
#endif
MPI_Sendrecv(/* ... */);
#ifdef PROFILING
comm_time += MPI_Wtime() - temp_start; // Accumulates time for quantitative plot
NVTX_POP(); // Pops the range from the timeline
#endif
```

Experimental Setup & Test Case Selection. To analyze strong scalability, we fixed the problem size using the largest provided test case, **No. 38**, which has an input size of $N = 536,870,912$, ensuring sufficient workload per process to minimize measurement noise and clearly observe scalability trends. This size also matches the hidden test cases used for performance grading.

We varied the number of processes (p) from 1 to 48, spanning configurations from a single core up to four full compute nodes (12 cores/node). Each measurement was taken from a single run to simulate the grading environment.

Single-Node vs. Multi-Node Performance. Our experiments cover both single-node ($p \leq 12$) and multi-node ($p > 12$) configurations. Single-node experiments show better scalability due to lower communication latency via shared memory. Multi-node configurations exhibit increased communication overhead due to InfiniBand network latency, as evident in the communication time increase from $p = 12$ to $p = 24$.

2.2 Results and Observations

The results of the strong scaling experiment are shown in Figure 1 (Time Profile) and Figure 2 (Speedup Factor). A snapshot of the Nsight Systems timeline at $p = 12$ is provided in Figure 3 for qualitative analysis.

As highlighted by our NVTX profiling (Figure 3), the principal performance bottleneck arises from the local sort phase (marked in Spring Green). The I/O overhead is almost constant and independent of process count, reflecting efficient parallel MPI-IO implementation. During local sorting, computation cannot overlap with I/O, as each process must receive its data before sorting begins, nor can it overlap with the main loop, which requires locally-sorted sub-arrays as precondition for inter-process communication.

As the number of processes increases, the CPU time (driven mainly by the local sort) drops rapidly (Figure 1, $p = 1$ to $p = 16$). However, communication overhead increases markedly, due to more frequent, smaller payload exchanges and the growing impact of synchronization. This trade-off leads to the observed plateau in overall runtime ($p = 38$ to $p = 48$), where CPU, communication, and I/O times stabilize near their minima and further parallelization yields diminishing returns.

Correspondingly, the speedup curve (Figure 2) flattens out at approximately $3.5\times$ for $p = 48$, instead of the ideal linear $48\times$. This deviation from ideal scaling directly visualizes the dominance of communication overhead and the emergence of strong-scaling limits imposed by Amdahl's Law.

2.3 Potential Optimization Strategies

Based on our performance analysis, two primary avenues for further optimization were considered: improving local computation and overlapping communication with computation.

Enhancing Local Computation. The analysis identified the initial local sort as a significant portion of the total execution time, especially with fewer processes. A natural optimization path would be to employ a more advanced sorting algorithm. However, our implementation already utilizes an adaptive strategy culminating in `boost::sort::spreadsort`, a state-of-the-art radix-based sort for floating-point data. It is therefore unlikely that significant further gains can be achieved in this area using a pure, single-threaded CPU sort within the constraints of this assignment.

The most substantial improvement would come from parallelizing the local sort itself, for instance, by using OpenMP to leverage all cores within a node. This would transform the local sort from a serial bottleneck into a parallel task. However, this approach, known as hybrid MPI+OpenMP programming, falls outside the scope of the current pure MPI assignment and was therefore not implemented in the final version.

Overlapping Communication and Computation. The second major optimization strategy in parallel programming is to hide communication latency by overlapping it with useful computation. This is typically achieved using non-blocking MPI operations (e.g., `MPI_Isend`, `MPI_Irecv`). We explored the feasibility of this approach for our Odd-Even sort implementation.

A potential scheme could be:

1. At phase i, initiate non-blocking communication with the partner for phase i.
2. While the communication for phase i is in flight, perform the computation (merge-split) for phase $i - 1$, whose data has already been received.

However, we identified a fundamental **data dependency** that makes this pipelining strategy extremely difficult to implement correctly and efficiently in a standard Odd-Even sort:

- The decision to perform a full data exchange in phase i (our "Conditional Merge-Split" optimization) depends on the boundary values of data that was just finalized at the end of phase $i - 1$.
- The merge-split computation for phase i requires the data received during phase i.

These tight, phase-to-phase dependencies mean there is no significant, independent computation that can be performed to effectively hide the communication latency of the current phase. Any attempt to do so would lead to a highly complex, bug-prone implementation with likely minimal performance gain due to the frequent synchronization points required.

Conclusion. In summary, while several advanced optimization techniques exist, the inherent data dependencies of the block-based Odd-Even sort algorithm, combined with our already-optimized local sort, mean that our current implementation is very close to the performance limit achievable within a pure, blocking MPI paradigm.

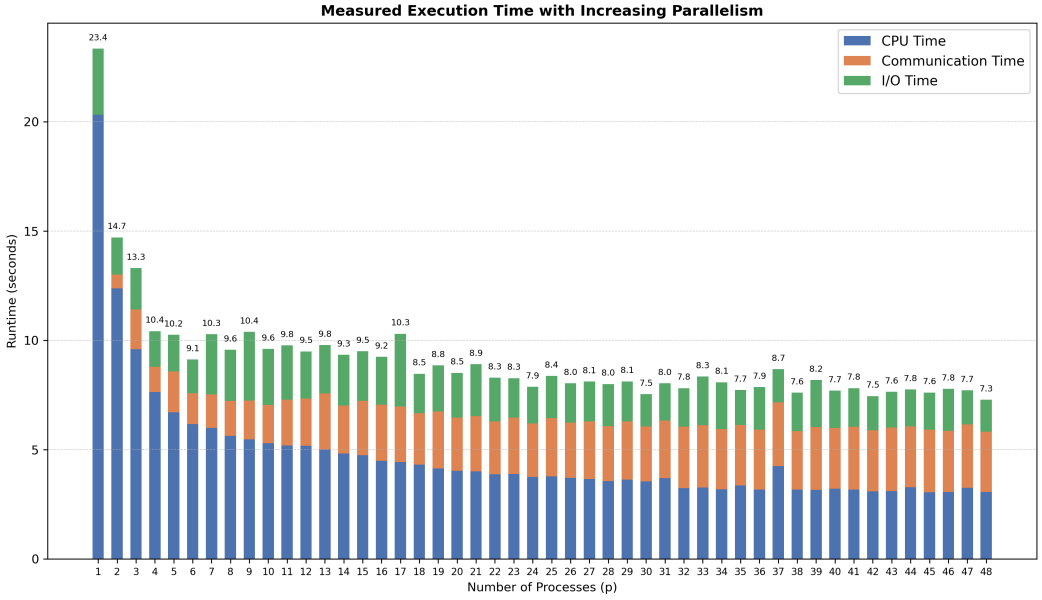

Figure 1: Time Profile vs. Number of Cores (Strong Scaling on Test Case 38). The execution time is broken down into CPU (blue), Communication (orange), and I/O (green) components.

3 Experiences & Conclusion

This assignment gave me practical experience in high-performance MPI programming through iterative implementation, profiling, and optimization of a parallel odd-even sort. The final solution achieves good scalability but is ultimately constrained by communication overhead and the serial nature of local sorting, consistent with Amdahl's Law.

Optimizing parallel programs often involves trade-offs between computation and communication. Some strategies improve performance in certain scenarios yet degrade it in others. The best approach is therefore context-dependent, balancing algorithm design and system characteristics.

One major challenge was explaining the performance gap between my implementation and top scorers. Initial profiling pointed to local sorting as a bottleneck. Although parallelizing local sort with OpenMP could help, such hybrid approaches were out of this assignment's scope and were not explored in depth. Additionally, variability in cluster conditions caused some unreproducible runtime fluctuations.

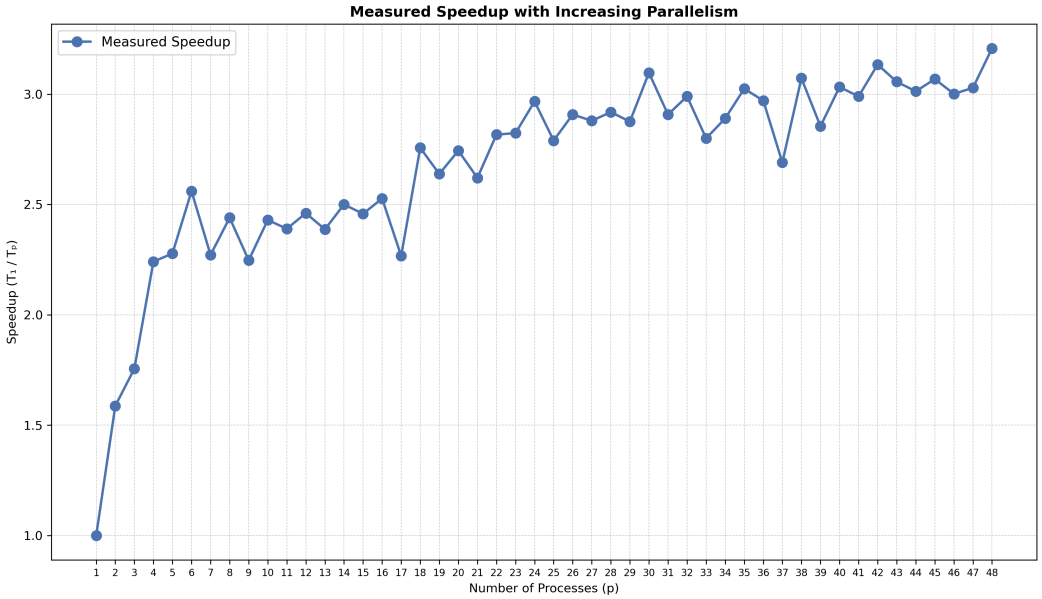

Figure 2: Measured Speedup Under Strong Scaling. The sub-linear growth and subsequent plateau demonstrate the scalability limits imposed by Amdahl's Law, where the non-parallelizable portions of the algorithm (e.g., communication overhead) cap the maximum achievable speedup.

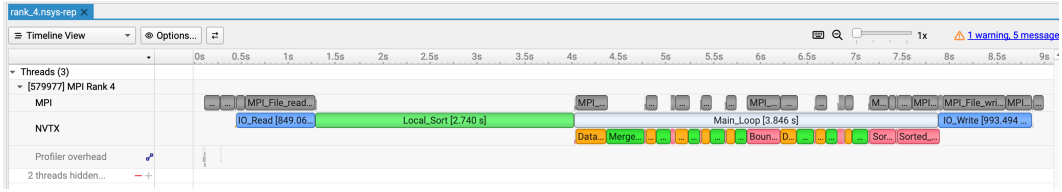

Figure 3: Nsight Systems timeline snapshot for Rank 4 ($p = 12$). The colored bars correspond to the operations defined in our NVTX macro, visually confirming the time distribution.

Overall, this assignment deepened my understanding of MPI programming and performance optimization.

Feedback The assignment effectively simulates real-world HPC challenges. To further assist students, providing sample optimized implementations for comparison or detailed tutorials on profiling tools would be beneficial.