CS542200 Parallel Programming
Homework 1: Odd-Even Sort
Due: Sun October 5, 2025, 23:59

1 Goal

This assignment helps you get familiar with MPI by implementing a parallel odd-even sort algorithm. Experimental evaluations on your implementation are required to guide you analyze the performance and scalability of a parallel program. You are also encouraged to explore any performance optimization and parallel computing skills in order to receive a higher score.

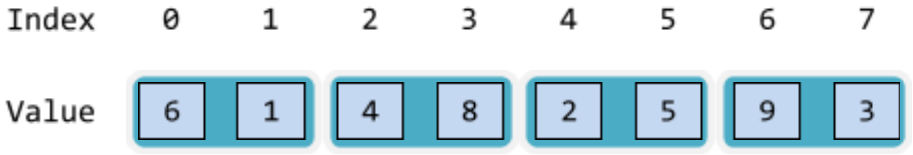
2 Problem Description

In this assignment, you are required to implement the odd-even sort algorithm using MPI. Odd-even sort is a comparison sort that consists of two main phases: *even-phase* and *odd-phase*. In each phase, processes perform compare-and-swap operations repeatedly as follows until the input array is sorted.

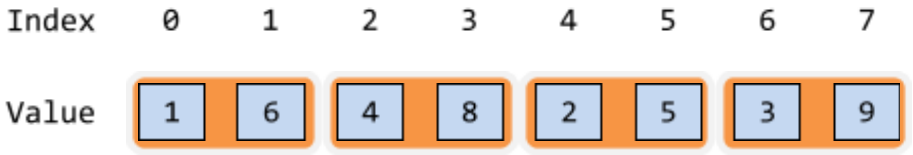
In the even-phase, all even/odd indexed pairs of adjacent elements are compared. If the two elements of a pair are not sorted in the correct order, the two elements are swapped. Similarly, the same compare-and-swap operation is repeated for odd/even indexed pairs in odd-phase. The odd-even sort algorithm works by alternating these two phases until the input array is completely sorted.

For you to understand this algorithm better, the execution flow of odd-even sort is illustrated step by step as below: (The array is sorted in ascending order)

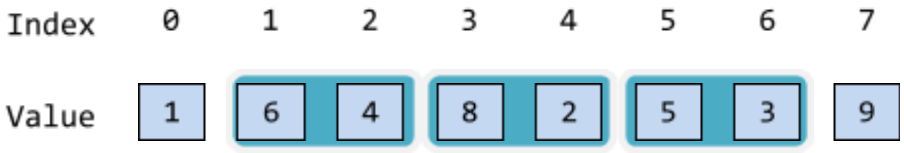
1. [Even-phase] even/odd indexed adjacent elements are grouped into pairs.



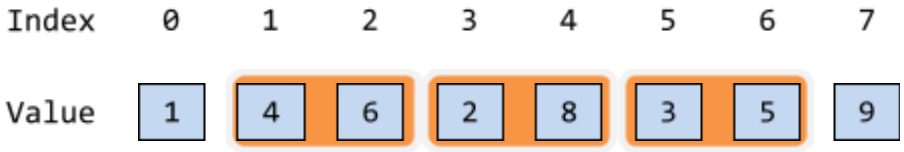
2. [Even-phase] elements in a pair are swapped if they are in the wrong order.



3. [Odd-phase] odd/even indexed adjacent elements are grouped into pairs.



4. [Odd-phase] elements in a pair are swapped if they are in the wrong order.



5. Run even-phase and odd-phase alternatively until **no swap happens** in both ∑even-phase and odd-phase.

The above example is a simple case where the number of MPI tasks(processes) and the size of the array is the same. But your implementation for the homework must be able to handle an arbitrary number of MPI tasks and array elements by letting each ∑array element. More details on the implementation restrictions are given in Section 4.

3 Input / Output Format

1. Your program is required to read an unsorted array from an input file, and generate the sorted results to another output file.
2. Your program accepts 3 input parameters separated by space. They are:

   i、 (Integer) the size of the array ***N*** ($1 \leq N \leq 536870911$)
   ii、 (String) the input file name
   iii、 (String) the output file name

   Your program will be tested by our judge system with the command below:

   ```
   $ srun -N$NNODE -n$NPROC ./hw1 $N $INF_FILE $OUT_FILE
   ```

   Note, `$NPROC` is the number of processes, `$NNODE` is the number of nodes and hw1 is the name of your binary code.
3. The input file contains ***n*** 32-bit floats in binary format. The first 4 bytes represent the first floating point number, the fifth to eighth byte represents the second one, and so on. (Sample input files will be given by our judge system.)
4. The output file must be generated by following the same format as the input file. (Sample output files will also be given by our judge system.)
5. We also provide the `hw1-floats` command to confirm the correctness of the output. For example, if you run test case "01" then, the ground truth is in pp25 share dir, and you can use the file to compare to your program output.

   ```
   $ hw1-floats /home/pp25/share/hw1/testcases/01.out
   out_file
   ```

Note:

The float here refers to **IEEE754 binary32**, known as a **single-precision floating-point**.

You can use the `float` type in C/C++.

Any valid float values are possible to show up in the input, except the following values:

- `-INF`
- `+INF`
- `NAN`

4 WORKING ITEMS

1. **Problem assignments:** You are required to implement a parallel version of odd-even sort under the given restrictions. Your goal is to optimize the performance and reduce the total execution time.

   - A process can sort or perform any computations on its own local elements.
   - For the odd-even sorting phases, a process can only exchange its local elements with its neighbor processes according to the communication pattern described in Section 2. For instance, MPI task with rank 5 can only exchange *elements* with ranks 4 and 6, but not the ones with ranks 3, and 7. **Note that the neighbor relationship is not circular**. For example, if your MPI program creates 10 processes, the MPI task with rank 0 cannot send any message to rank 9 during the sorting, and vice versa.
   - However, any communication method, including collective communications (i.e., broadcast, gather, scatter, etc.), is allowed for **initialization** before the sorting begins, or **termination checking**.

2. **Requirements & Reminders**:

   - **Must use MPI-IO** (`MPI_File*` functions) to do file input and output. The example code of MPI-IO can be found in the `/home/pp25/share/hw1/sample/mpiio.cc`
   - **Must follow the input/output file format and execution command line arguments specifications described in Section 3.**
   - The implemented algorithm **must follow the odd-even sort principle, and comply with the restrictions described in Section 4.1**. If you are not sure whether your implementation follows the rules, **please discuss with TA for approval**.
   - It is encouraged to experiment with different combinations of compilers & MPI implementations.

5 REPORT

1. **Title, name, student ID**

2. **Implementation**

   Briefly describe your implementation in diagrams, figures, sentences, especially in the following aspects:

   - ✔ How do you handle an arbitrary number of input items and processes?
   - ✔ How do you sort in your program?
   - ✔ Other efforts you've made in your program.

3. **Experiment & Analysis**
   Explain how and why you do these experiments? Explain how you collect those measurements? Show the results of your experiments in plots, and explain your observations.

   i、 Methodology

   (a). **System Spec** (If you run your experiments on your own cluster) Please specify the system spec by providing the CPU, RAM, disk and network (Ethernet / InfiniBand) information of the system.

   (b). **Performance Metrics**: How do you measure the computing time, communication time and IO time? How do you compute the values in the plots?

   ii、 Plots: Speedup Factor & Profile

   - Experimental Method:
     - Test Case Description: Explain the test data and its sizes you've chosen.
     - Parallel Configurations: Describe the number of processes used, and how nodes and cores are distributed.
   - Performance Measurement:
     - Use a profiler (such as perf, nsys, mpiP) for performance analysis.
     - Provide basic metrics like execution time, communication time, IO time, etc.
   - Analysis of Results:
     - Display results generated by the profiler; this could be in the form of tables, charts, or other visualization tools.

- Identify Performance Bottlenecks: Use the profiler's data to point out which parts of the code or which communication patterns are creating performance issues.
- Conduct strong scalability experiments, and plot the speedup and time profile results as shown in figure 1 and figure 2.

- Optimization Strategies:
  - Based on the analysis results, propose potential optimization strategies.
  - If optimizations have been implemented, provide a comparison of performance before and after the enhancements.

- Your plots must contain at least 4 settings (e.g., scales) for both single node and multi-node environments.
- **Make sure your plots are properly labeled and formatted.**
- You are encouraged to generate your own test case with proper problem size to ensure the experimental results are accurate and meaningful. (e.g. Make sure the execution time is long enough to have meaningful difference for comparison)

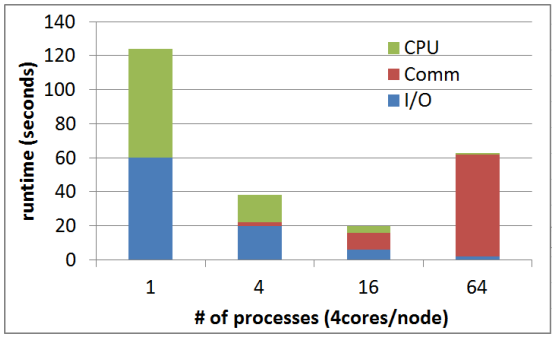


Figure 1: Time profile

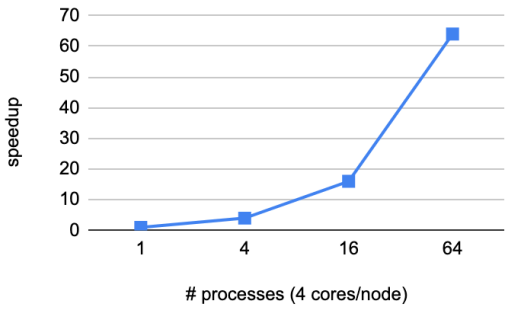


Figure 2: Speedup

iii、Discussion (Must base on the results in your plots)

- Compare I/O, CPU, Network performance. Which is/are the bottleneck(s)? Why? How could it be improved?
- Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability? You may discuss the two implementations separately or together.

iv、 Others

You are strongly encouraged to conduct more experiments and analysis of your implementations.

4. **Experiences / Conclusion**

It can include these following aspects:

- ✔ Your conclusion of this assignment.
- ✔ What have you learned from this assignment?
- ✔ What difficulties did you encounter in this assignment?
- ✔ If you have any feedback, please write it here. Such as comments for improving the spec of this assignment, etc.

6 GRADING

1. **[35%] Correctness**
   - There are 40 test cases.
   - You can see all 40 of them at `/home/pp25/share/hw1/testcases`
   - You lose 1 point for each failed case.
2. **[15%] Performance**

   It is graded by the execution time of your program on 6 hidden test cases (similar to public testcase 35-40). Incorrect results will not get any points.
3. **[30%] Report**

   It is graded based on the evaluations, discussions, and writing in your report. Higher scores will be rewarded if a more detailed performance analysis of your implementation can be shown or explained by experiments.
4. **[20%] Demo**

   The demo will mainly focus on the following aspect:

   - ✔ Explain your implementation.
   - ✔ Explain the key results and findings from your report.
   - ✔ **Your extra efforts. (Why do you deserve more bonus points?)**
5. **Policy**

- ✔ Do not copy others' code. Zero **point will be given to the cheater** (even copying code from the Internet).
- ✔ **No late submission after the deadline will be accepted.**

7 Homework Submission & Reminder

- **Deadline: October 5, 2025 23:59**.
- **Upload your report** in **PDF format** named **hw1_<student-id>.pdf (e.g. hw1_109062500.pdf)** on eeclass before the deadline.
- **Upload the following files** to eeclass before the deadline.
  - **i、 hw1.cc**
  - **ii、 Makefile (optional, if you change any compile flags)**
  - **iii、module.list (optional)**
    - **By default, will use GCC & Intel MPI to judge your code**
    - **If you are using other modules (such as openmpi), please generate this file by running `module save ./module.list`.**
- A **self-checking script** is provided to verify the correctness of your code. To test, simply type the command **`hw1-judge`** under your source code directory.
- **A scoreboard system will be available for you to check out the current ranking of your implementation**.
  - https://pp.lsalab.cs.nthu.edu.tw/hw1/
- Since we have limited resources for everyone to share, please **start your work ASAP to avoid long queuing delays**. Do not leave it until the last day!
- You are welcome to ask questions through eeclass!