CS542200 Parallel Programming
Homework 1: Odd-Even Sort
Due: Mon October 26, 2020 11:59

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.

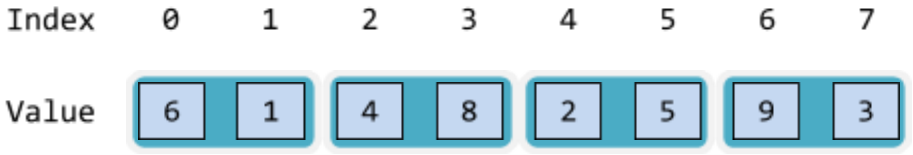
2 Problem Description

In this assignment, you are required to implement the odd-even sort algorithm using MPI. Odd-even sort is a comparison sort which 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.

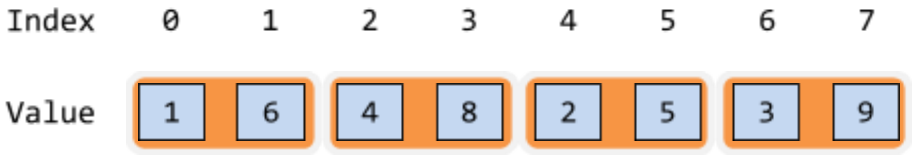
In even-phase, all even/odd indexed pairs of adjacent elements are compared. If the two elements of a pair are not sorted in the correct order, the two elements are swapped. Similarly, the same compare-and-swap operation is repeated for odd/even indexed pairs in odd-phase. The odd-even sort algorithm 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)

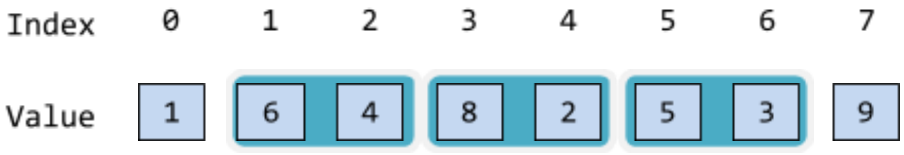
1. [Even-phase] even/odd indexed adjacent elements are grouped into pairs.



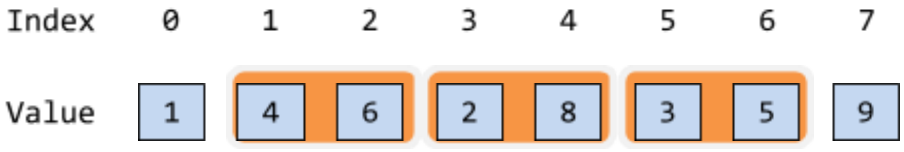
2. [Even-phase] elements in a pair are swapped if they are in the wrong order.



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 array are 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 process manage a sub-array partition instead of a single 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 ≤ n ≤ 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 -Nnodes -nNPROC ./hw1 n in_file out_file
   ```

   Noted, `NPROC` is the number of processes, `nodes` 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 represents 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 of the input file. (Sample output files will also be given by our judge system.)

Note:

The float here refers to **IEEE754 binary32**, as known as **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 rank 3, 7. **Note that the neighbor relationship is not circular**. For example, if your MPI program creates 10 processes, 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.), are 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/pp20/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**.

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.

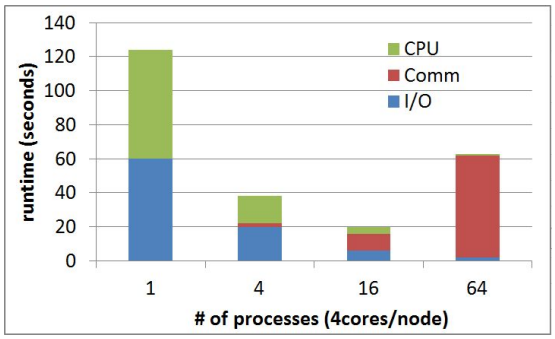
   i、 Methodology

   (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 & Time Profile

   - Conduct strong scalability experiments, and plot the speedup and time profile results as shown in figure 1 and figure 2.
     (Please note that these plots are just examples of how you can layout your figures, and you are unlikely to get the same results)
   - 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)



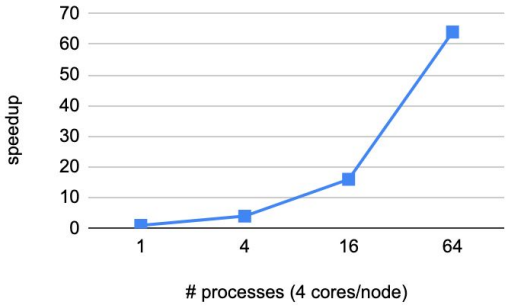


Figure 1: Time profile

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 for 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. **[40%] Correctness**
   - There are 40 test cases.
   - You can see all 40 of them at `/home/pp20/share/hw1/testcases`
   - You lose 1 point for each failed case.
2. **[20%] 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 point.
3. **[20%] Report**

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

   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

1. **Upload your report** in **pdf format** named **hw1_student_id.pdf (e.g. hw1_108062100.pdf)** on iLMS before the deadline.
2. **Upload your source code and Makefile** to iLMS before the deadline.
   - **i、 hw1.cc**
   - **ii、 Makefile**
3. 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.
4. Since we have limited resources for everyone to share, please **start your work ASAP to avoid long queuing delay**. Do not leave it until the last day!
5. **A scoreboard system will be available for you to check out the current ranking of your implementation**.
   Go to http://apollo.cs.nthu.edu.tw/pp20/scoreboard/hw1 to check the scoreboard.
6. You are welcome to ask questions through iLMS!