Homework 1 / Odd-Even Sort

李豪韋 (HW-Lee) ID 103061527

Overview

There is a comparison sorting algorithm called odd-even sort, which is also related to bubble sort. It consists of two phases to sort a sequence, odd phase and even phase. At each iteration, the sequence will be separated into a couple of pairs, where the processor should arrange its to keep its in descending/ascending order. The process will keep running until there is no swapping in both even and odd phase. Due to independence among pairs, this algorithm can be accellerated with parallel implementations. In this project, the advantages of parallelism can be observed with different strategies.

Implementation

1. Basic

   The concept of the algorighm has been illustrated below, see Fig.1. After each processor has got its own data, only numbers at sides, i.e. at first or at end, have to be sent/received. Every processor swaps a pair of data if needed, and repeats the operation until there is no need to swap data.

2. Advanced

   The schema is similar to the basic version, the only difference is at the stage where each processor commences to re-arrange the sequence. After got the data, each processor will sort its sequence first. Then processors with rank k will send the first number, which is the minimum (in ascending) or the maximum (in descending), to those with rank $k-1$, and then processors with rank $k-1$ will insert the received number and return back the last number, which is the maximum (in ascending) or the minimum (in descending). The operation will be executed until there is no change of all processors. In this project, there are two options chosen to be the pre-sorting algorithm, namely merge and bubble sort.

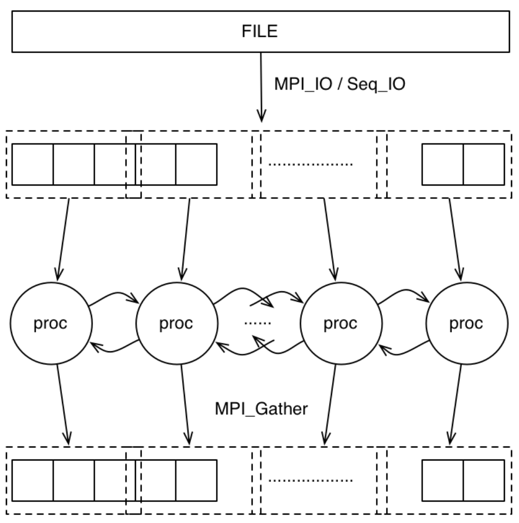


Figure 1: Diagram of odd-even sort.
It mainly consists of 5 stages:
1) Input, to load data into registers.
2) Scattering, to choose the fragment for which each processor should take responsibility. Each fragment will be sorted in advanced version. (with bubble/merge sort)
3) Swapping at odd-even phase, core of the algorithm, iterates operation until there is no swapping occurred.
4) Gathering, to merge all sorted data from processors.
5) Output, to write the sorted sequence into the disk.

Analysis & Results

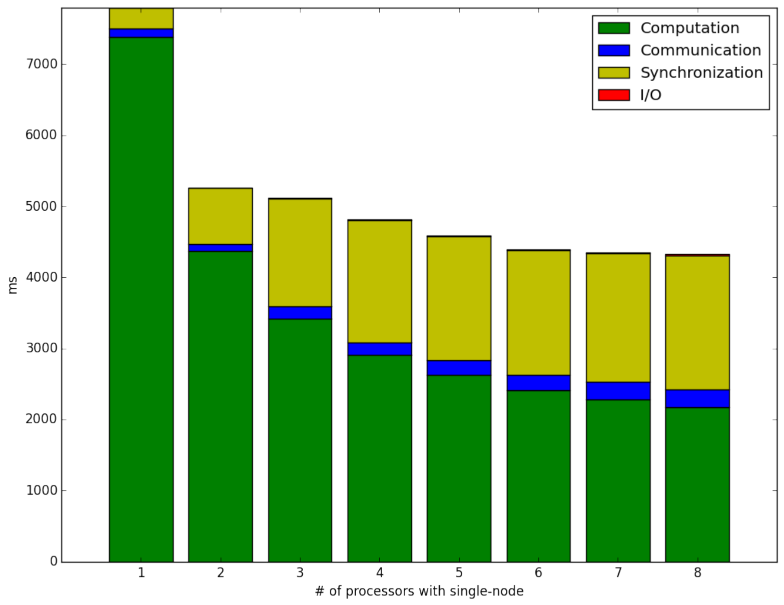
- Complexity

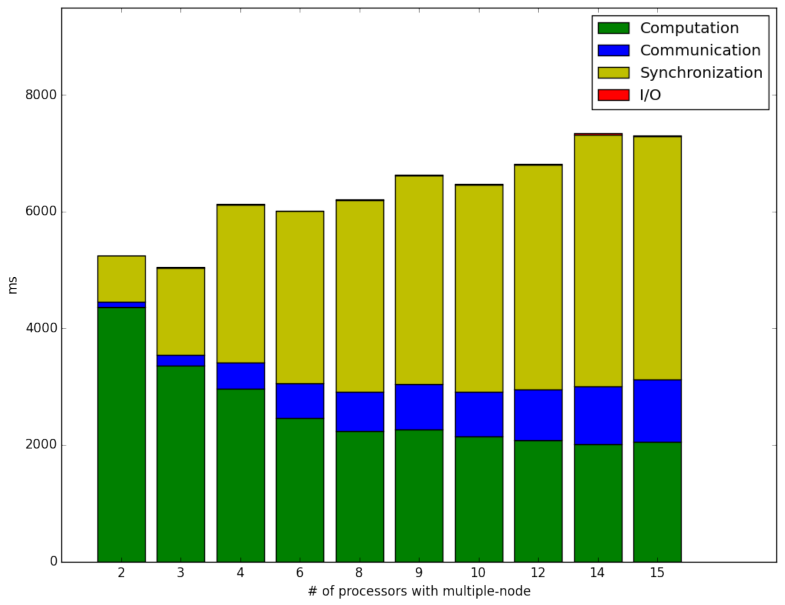
In advanced sort, we can simply use binary search to insert the received number because the sequence in each processor is sorted. Therefore, the complexity of insertion is $O(\frac{n}{p} + \log(\frac{n}{p}))$. If there are p processors, sorting an n-length sequence, with k iterations, the complexity can be written as:

Algorithm \ Type	Comm.	Comp.	Overall with large n	Prop. to
Basic	$O(kp)$	$O(\frac{n}{p}k)$	$O(\frac{n}{p}k)$	p^{-1}
Advanced-bubble	$O(kp)$	$O((\frac{n}{p})^2) + O((\frac{n}{p} + \log(\frac{n}{p}))k)$	$O((\frac{n}{p})^2 + \frac{n}{p}k)$	p^{-2}
Advanced-merge	$O(kp)$	$O(\frac{n}{p}\log(\frac{n}{p})) + O((\frac{n}{p} + \log(\frac{n}{p}))k)$	$O(\frac{n}{p}k)$	p^{-1}

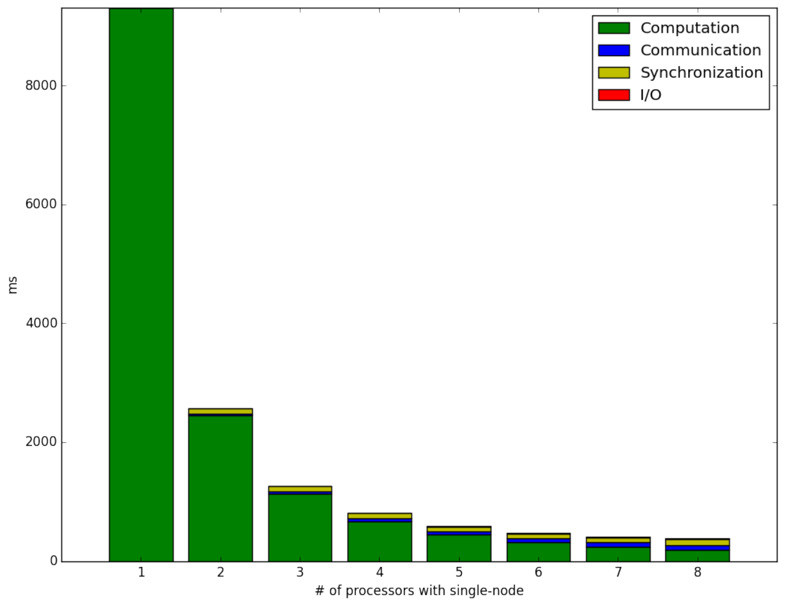
Note that the parallelism will be easier to be observed when the sequence is long, i.e. large n. In addition, I redefine the speedup factor as $(\frac{T_s}{T_p})^{\frac{1}{\alpha}}$ if the overall time complexity is propotional to $p^{-\alpha}$ rather than the original definition $\frac{T_s}{T_p}$ because the value α's are not the same in three algorithms. If I want to keep the ideal curve linear, the speedup factor should be redefined.

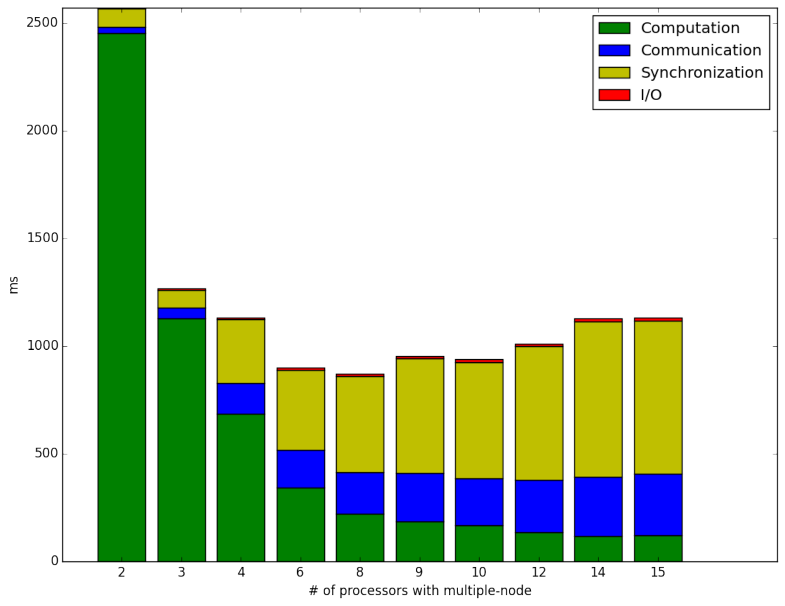
- Basic



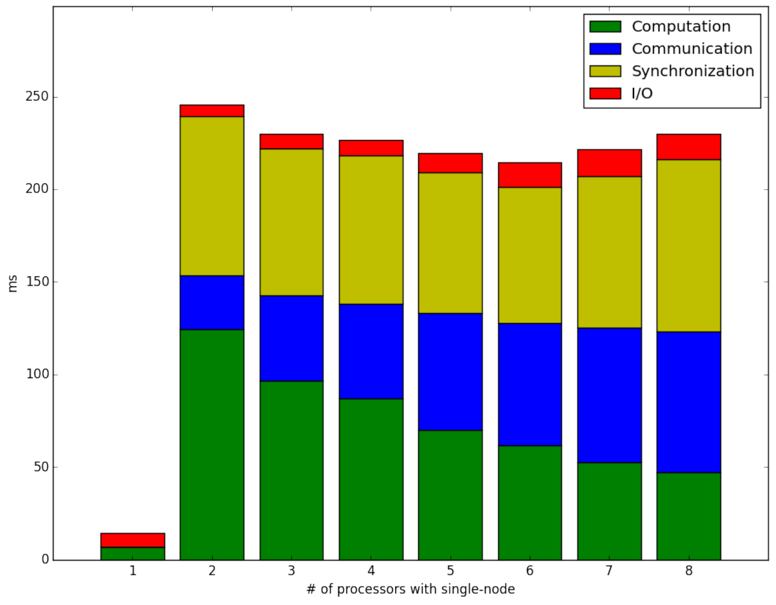


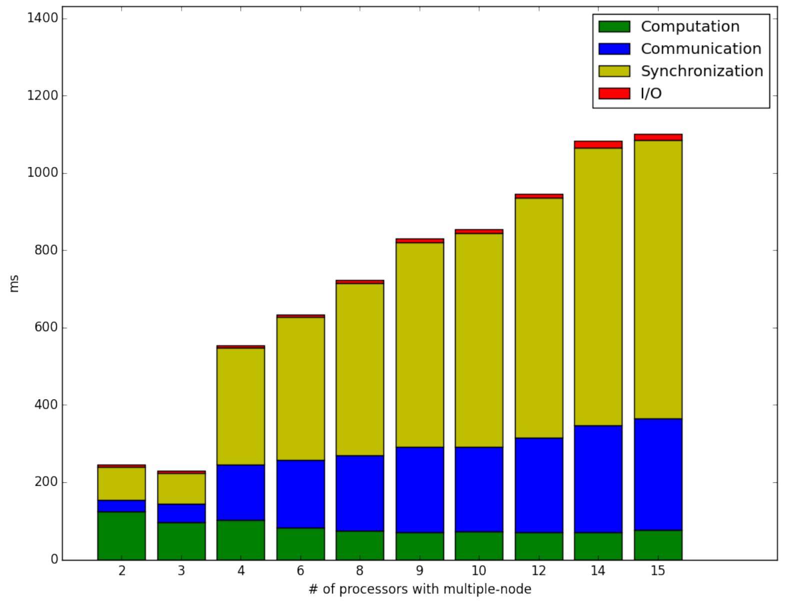
- Advanced-bubble





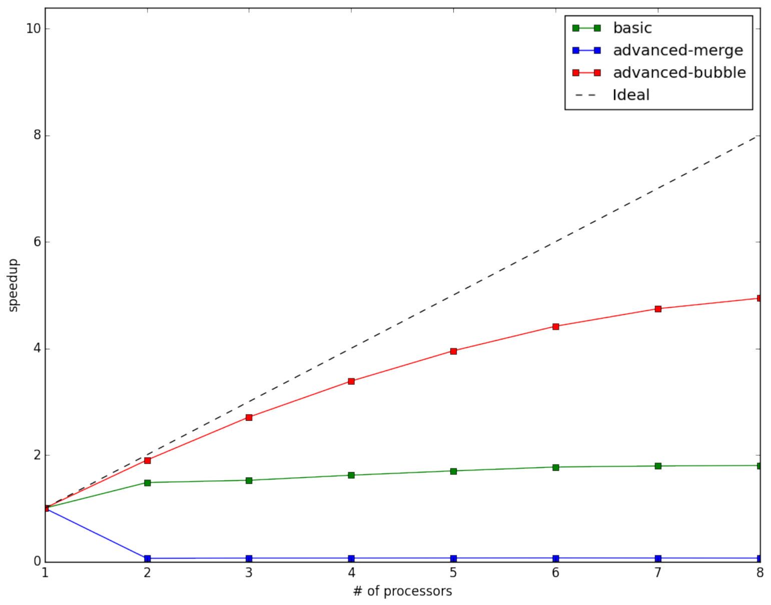
- Advanced-merge

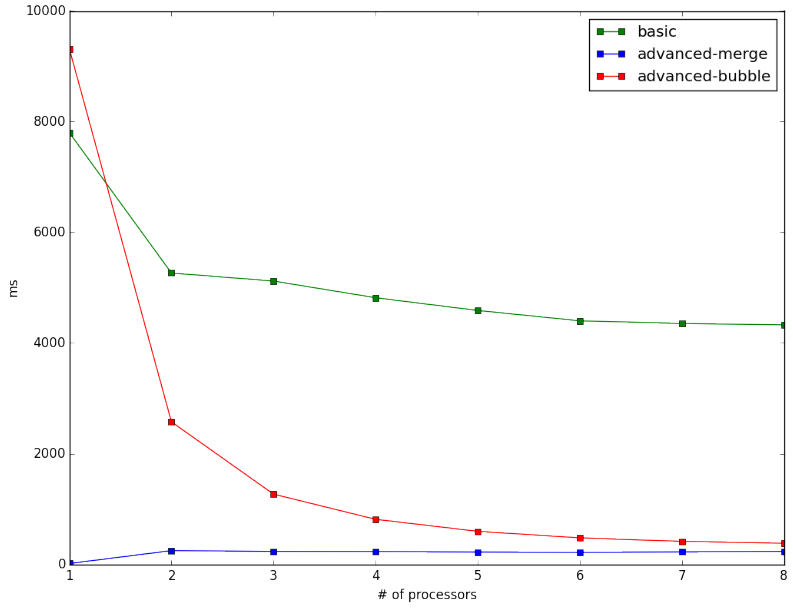




- Speedup & Overall execution

Execution time figure has been shown at right-hand side, it shows the performance of advanced version is superior to that of basic version.





- I/O Strategies

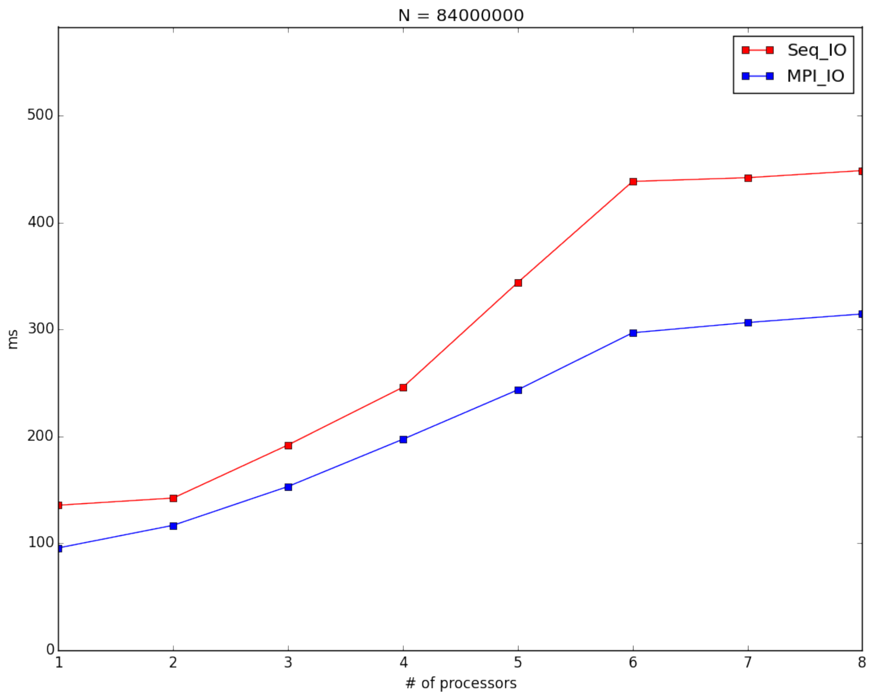


Figure 2: Execution time between different I/O.
Consumption time rises as the number of processors increases, and MPI_IO is always faster than Seq_IO

Discussion

1. Work Balance

   It requires less communication time with single node than with multiple node, and then it requires less synchronization time also. With different strategies:

   - Basic

   According to the consumption of synchronization, it shows the loading scheduling is not balanced. In each iteration, each processor should check all pairs it possesses and determine if the pair should be swapped. I think it is the main reason why the loading scheduling is not balanced because it is hard to presume each processor has same numbers of instructions at each iteration.

   - Advanced

   In advanced version, the data in each processor has been sorted before communication. The insertion operation consists of searching ($O(\log(N))$) and array moving ($O(N)$), it guarantees the task will be well-balanced because each processor keeps same size of numbers. In addition, merge sort is not suitable in this kind of strategy because the complexity will be significantly smaller when $p = 1$, and required execution time rises as p increases.

2. Speedup Efficiency

   According to the result, the performance of advanced-bubble is the best, and that of advanced-merge is the worst. The bad result of advanced-merge is because the time consumed in swapping stage is highly larger than pre-sorting and will dominate the overall execution time. After all, the result has verified that the efficiency is subject to how well the work is balanced.