Parallel Programming HW1

Odd-even Sort

ssh: pa128427359、程祥恩

1. Implementation

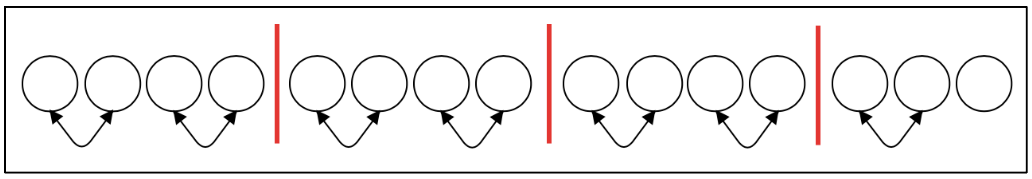
i. Basic

First, the program calls MPI_File_open() to open a input file, then calls MPI_File_set_view() and MPI_File_read_all() for every processor to allocate memory for input elements, next, calls MPI_Get_count() to get the exact number of element in each processor. For instance, if the input size is 21 and the number of processor is 4, elements will be allocated in (6, 6, 6, 3).

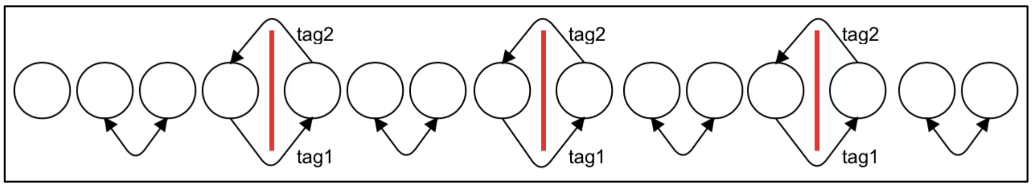
Second, in the odd-even sort part, for the even phase, take 4 processors, 15 and 11 input size as examples, **the program can handle the condition of the number of input item and the arbitrary number of processor by checking the number of elements in most processors.**

Case 1: The number of elements in most processor is even.

- Even-phase: There is no cross-processor comparison.

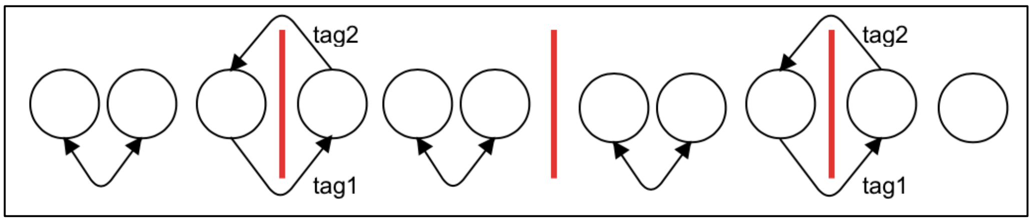


- Odd-phase: There are cross-processor comparisons, the program will create to tags for every two-way comparison and swap.

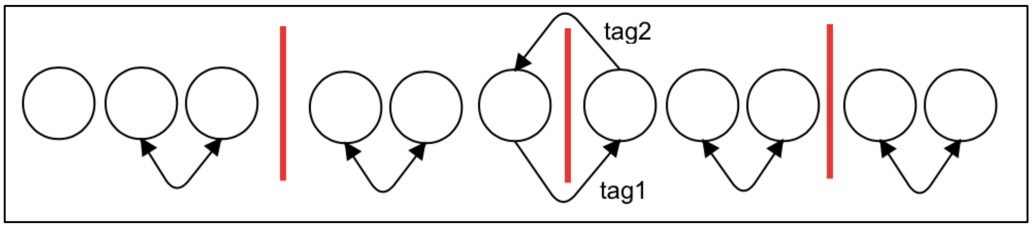


Case 2: The number of elements in most processor is odd.

- Even-phase: There are cross-processor comparisons for all **last element of even ranks** between **first element of odd ranks**.



- Odd-phase: There are cross-processor comparisons for all **last element of odd ranks** between **first element of even ranks**.

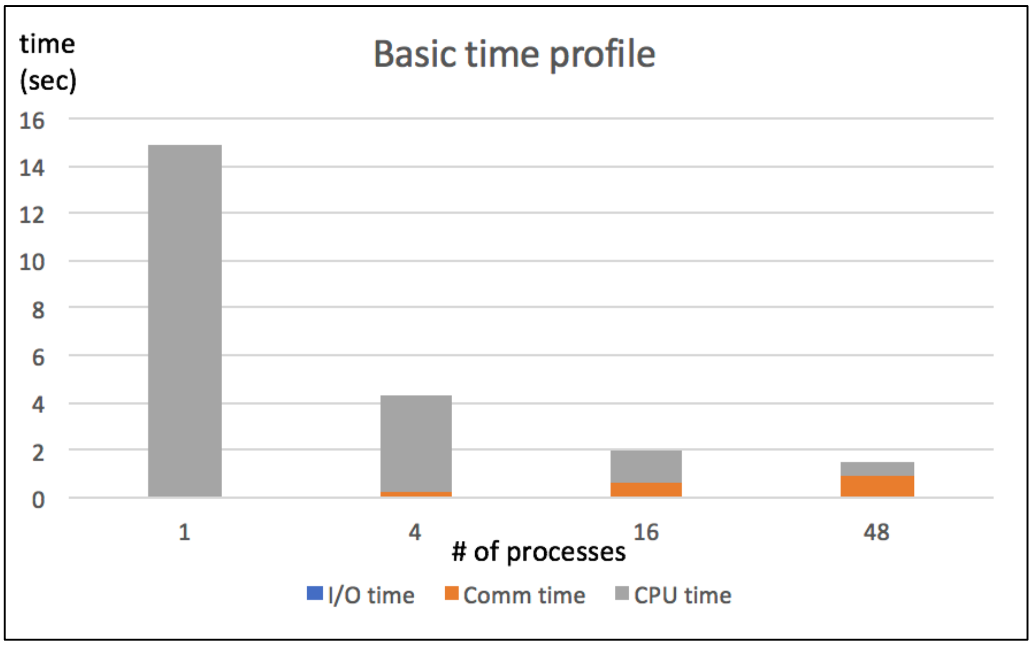


ii. Advanced

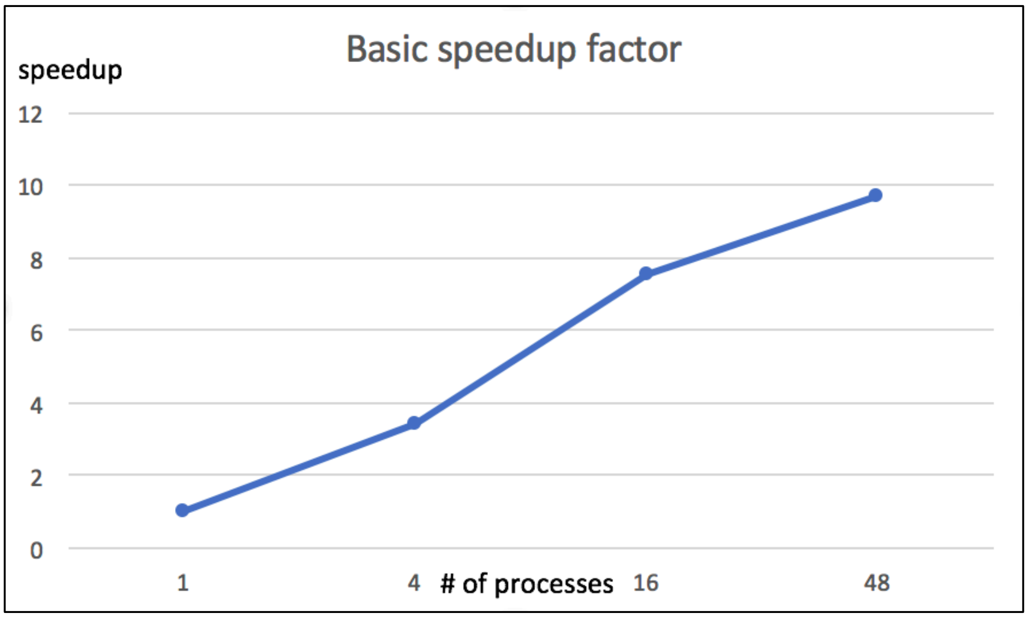
First, the program calls quick sort for elements inside every processor. Next, using MPI to do cross-processor comparisons same as the basic version. After that, the program calls insertion sort for all processors since insertion sort is the most effective soring algorithm for nearly sorted list. Loop the above process until no swap in cross-processor comparisons.

2. Experiment & Analysis

i. Basic

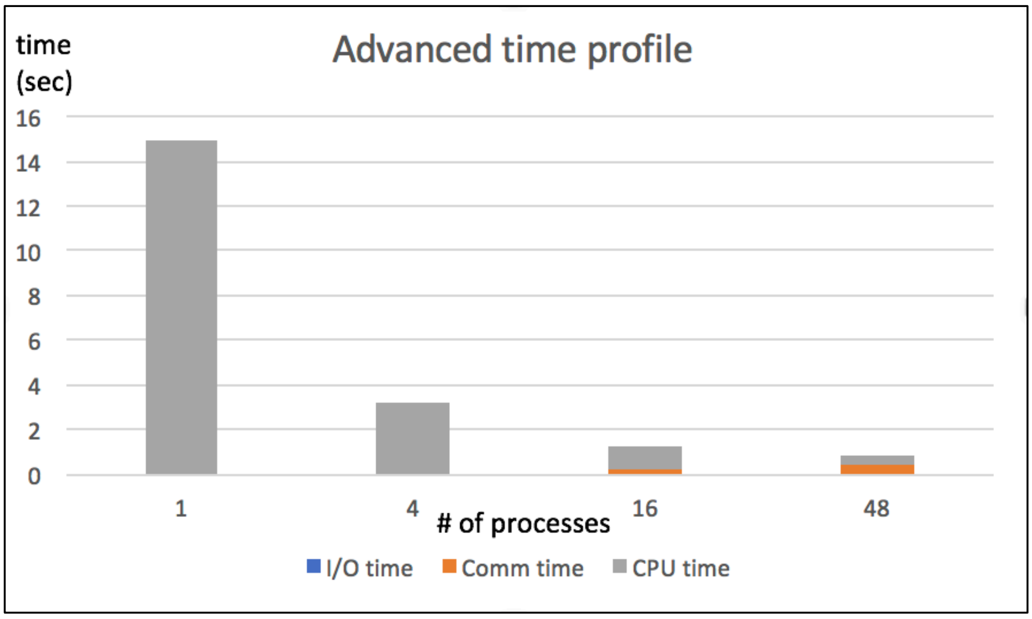


PROC	1	4	16	48
I/O TIME (SEC)	0.01	0.01	0.01	0.01
COMM TIME (SEC)	0	0.23	0.66	0.9
CPU TIME (SEC)	14.91	4.1	1.3	0.63

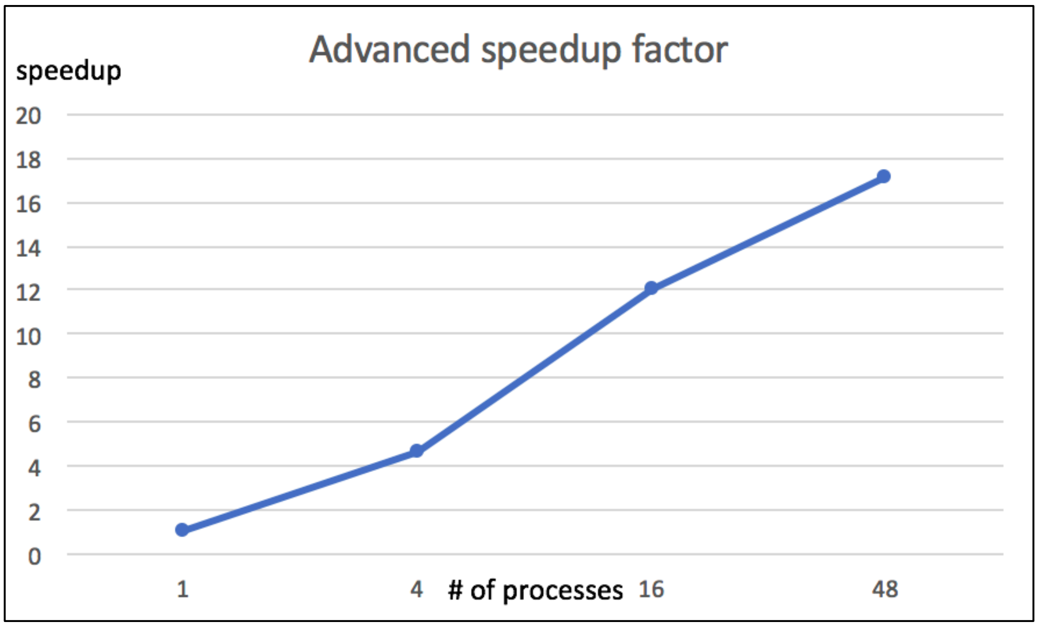


PROC	1	4	16	48
SPEEDUP	1	3.44	7.57	9.69

ii. Advanced



PROC	1	4	16	48
I/O TIME (SEC)	0.01	0.01	0.01	0.01
COMM TIME (SEC)	0	0.01	0.24	0.42
CPU TIME (SEC)	14.91	3.17	0.99	0.44



PROC	1	4	16	48
SPEEDUP	1	4.68	12.03	17.15

(Note: Run time of processes = 1 is from basic version)

iii. Discuss

In advanced version, the program runs quick-sort for all processes first, so that elements in all processes are nearly-sorted. In this case, compare to pure odd-even-sort, the program in advanced version can reduce the redundant cross-process comparisons and message passing time.

I think the bottleneck would be communication time. For the basic version as well as the advanced version, they run "one-element-passing" for many times instead of "many-elements-passing" for less time. Also, I think the CPU time of the advanced version could be improved by speeding up the insertion-sort part.

I think the scalability of the "inner process sorting" part is not bad since programmers just need to modify the sorting algorithm outside the core section. For other parts, I think the scalability is not very good because programmers need to modify all codes inside the core section to modify the "one-element-passing" part.

3. Conclusion

This is the first time I finish an assignment of parallel programing, different from simple sequential programming, parallel programming requires the ability of "parallel thinking", which means I need to change the way of solving problems and consider all possible cases during message passing. Although the speedup is not very good, it's a big step for me to learn parallel programming.