CS542200 Parallel Programming

Homework 1: Odd-Even Sort

109065517 林佳縈

1. Implementation

In this section, I'd like to briefly introduce how I implement the odd-even sort algorithm by elaborating the algorithm's working flow, accompanied with a diagram which might help for clearer understanding.

Read Data & Local Sort

First, the program reads data from *.in testcases and distributes those data into corresponding ranks. After each rank acquires its data, every rank then starts sorting its data locally.

About how I handle an arbitrary number of input items and processes is as following:

- Separate all data evenly by how many ranks I have (array size n/rank size)
- If elements can't be divided evenly, assign the remainders (array size n % rank size) to the last rank.

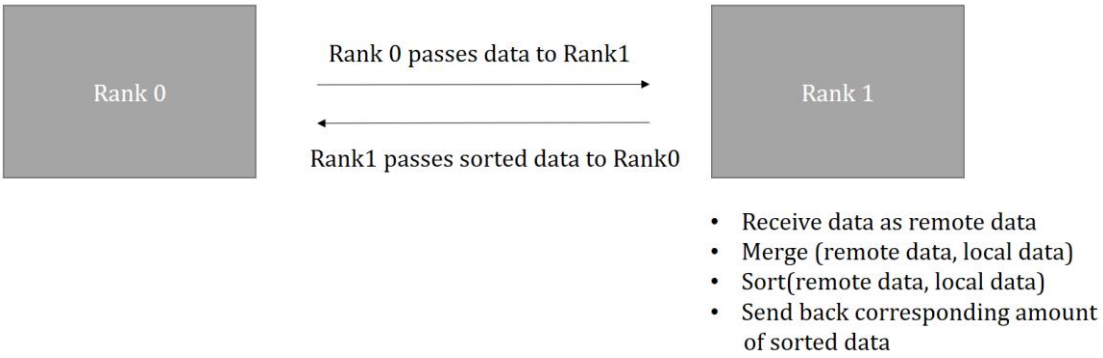
Ex. If array size n is 10 and rank size is 4, then ranks and their data will be:
(suppose array be like: array a[10]={0,1,2,3,4,5,6,7,8,9})
How many elements one rank will get: n/size(10/4)
Remainder that will be assigned to the last rank: n%size(10%4)

Rank	0	1	2	3
Elements	0,1	2,3	4,5	6,7, 8,9

Odd-Even Sort

After data is distributed and the aforementioned local sort is done, the program then enters the odd-even sort part.

In this section, the swaps basically work like this:



Let's take rank0 and rank 1 in even phase for example:
As illustrated above, in this phase, which is even, rank0 sends its data to rank1. Thus, rank1 does merging, sorting and sends a certain amount of sorted data back to rank0, depends on how many remote data elements it has received.
In a nutshell, for each phase, first we recognize whether it is an even or odd phase; thus, ranks' number which is homogeneous to phases' number will pass its data to its right neighbor (rank+1). Once the right neighbor(rank+1) receives the sent data, annotated as "remote data", it then merges two data, remote and its own local data, together and then perform the sorting process. Once everything is done, it sends back the data to its left neighbor and completes its job.

Write file

Once all the swaps in odd-even sort are completed, the program starts writing data into output *.out files.

2. Experiment & Analysis

i. Methodology

(a) System Spec

During the entire work, I implement and test on Apollo.

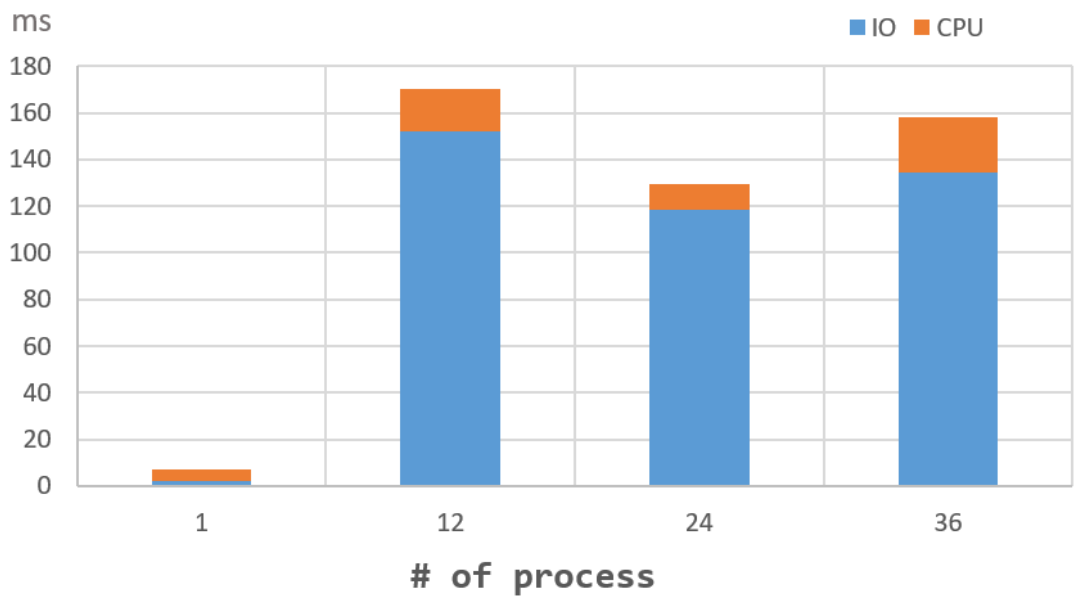
(b) Performance Metrics

As for how I measure the computing time so on and so forth, I chose the a MPI function called **clock_gettime()**.

The structure is as follow:

```
struct timespec tt1, tt2;
clock_gettime(CLOCK_REALTIME, &tt1);
clock_gettime(CLOCK_REALTIME, &tt2);
printf("consumes %ld ns!\n", (tt2.tv_nsec - t1.tv_nsec));
```

ii. Plots: Speedup Factor & Time Profile



iii. Discussion

On the above analysis chart, it's discovered system I/O occupies a great deal of time, which makes it probably a bottleneck in this case.

Yet the weird thing is that, at first I thought the more processes are used, the less time CPU would have spent, but it seems to be not as how I've expected through the chart above.

3. Experiences/Conclusion

Speaking of what I've learned during my implementation, I divided them into the followings:

1. **There must be some libraries providing faster sort!** (Try as many as

possible)

At the beginning of my implementation, I didn't take different sorting algorithm into account. I simply used qsort. But then, heard from TA and some friends, they suggested me to change my sorting algorithm. AND std::sort() IS WAY FASTER THAN THE ORIGINAL ONE!

2. **Numerical problems**

   As I was still implementing the odd-even sort with locally sorting algorithm using qsort, there's a problem that I found no clues about why it exists. Since qsort requires a compare function, and it normally looks like this:

```
int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}
```

   But since the testing data type is float, so I simply changed the above data type to float and here comes the problem. I guess it is because the function cmpfunc is comparing two numbers by subtracting one from another, which seems not going to work for floating-point types as it seems to then produce imprecise results, might overflow or somewhat.

3. **Dynamic memory allocation**

   This seems to be basic but I didn't even bear in mind. This cause fatal runtime error when allocating spaces for array, like in this case:

```
//float remote[localn] = {0};   //this is not going to work!
float *remote = (float*) malloc(sizeof(float)*localn);
```

4. **System might allocate more memory for the arrays**

   This problem was found because when I was testing my program by hw1-judge, sometimes for testcase 5, there might be a runtime error, and the weird thing is this error came unexpectedly, not every single trail. At the end, I found that the problem was caused by allocating too few index for an array. The magical thing is that the computer system in probably memory section or somewhere (heard from Prof. Jerry) will allocate for the program more spaces than when it is declared, so sometimes it failed while sometimes not.

This is the first time I use MPI to implement parallel program. Instead of the aforementioned solved problem, the main difficulty I face during this work was still when I was trying to do the optimization part. Except from those small but crucial changes, such as which sorting algorithm to choose from or cleverer use of variables and so on, I found that optimization that could do significant impacts on speeding up still lies in the changes on the whole structure, like the entire parallel part. During implementation, I've came up one, but eventually still failed to implement it successfully with full correct answers. Yet still, perhaps can do some further fixes and trials afterwards.