Parallel Programming

HW1: Odd-Even Sort

111065510 黃韋慈

A. Implementation

A.a Overall

This assignment needs to implement the Odd-even-sort algorithm by using MPI. The overall process can be as follows:
Allocate the amount of input data -> Read input data -> Local sort -> Communicates with adjacent nodes through MPI -> Merge the received data with local data and sort them -> Confirms that each node is sorted with adjacent nodes -> Write output into a file

The following will describe each stage in detail

A.b Allocate the amount of input data

Writing parallel programs requires consideration of all possibilities, so we need to consider whether our program can handle any number of input data and any number of processes in the beginning. Therefore, the possible situations:

A.b.1 Number of processes (P) > Input data(N)

In order to reduce the complexity of judgment in the MPI stage, I choose to directly re-establish a new communication group in the existing communication group. That is, use MPI_Comm_group() to create a group of MPI_COMM_WORLD, then use MPI_Group_range_excl() to discard the useless process to create a new group, and finally use MPI_Comm_create() to create a new communication. So that when the processes whose rank is larger than the processes size or processes that are not in the new communication will display MPI_COMM_NULL, they can be terminated by calling MPI_Finalize().

After discarding the redundant processes, the number of data will be equal to the number of processes, so one process can get one data(A.b.2).

A.b.2 Number of processes (P) = Input data(N)

Each process gets one piece of data.

A.b.3 Number of processes (P) < Input data(N)

At first, I did the original method (A.b.3.1) intuitively, but after careful consideration, I found that this method not only influences the overall performance but also doesn't conform to the concept of parallelization programs. Therefore, I subsequently implemented the improved method (A.b.3.2)

A.b.3.1 Original method

In the original method, each process will be allocated N/P pieces of data, and the remaining data will be processed by the last process. However, this method easily causes a load imbalance between processes. If the remaining data = P-1, it means that the last node needs to process twice as much data as the other nodes.

A.b.3.2 Improved method

In the improved method, I also allocate the amount of data of N/P to each node, and the remaining data is processed by the process whose rank number is less than the remaining data amount (R). They only need to process one more datum to allocate the remaining data. In this way, the processes whose rank number is larger than or equal to R only need to process the amount of data of N/P, so that the amount of data processed by each process can be relatively average (the amount of data is at most one different), and it can also have better performance in parallel programming.

The following is an example: (This example is also explained in the code comments)

Ex. 9(N) / 6 (P) =1(N/P) …3(R)

then

proc0~ proc2 will get 2 nuns.

proc3~ proc5 will get 1 nun.

A.c Read input data

I read the file through MPI_File_read_at(), so I only need to calculate the position where each process should start reading. The starting position of the data is calculated according to the number of data to be processed by each process mentioned above. Note that the remaining data of the note are allocated by the process whose rank is lower than R, so I add the remaining data amount to the starting position of the data for the process whose rank is larger than R.

A.d Local sort

This part is that each process sorts the read data. I originally used qsort of <algorithm> to sort, but later found out that using sort in <algorithm> can be applied to a wider range of elements, and the time complexity is n* log2(n) is also faster than qsort, so use this function to speed up the program.

A.e Communicates with adjacent nodes through MPI

I will use an example to explain, each phase can be distinguished by a different number of processes: (This part is also explained in the comments in the code)

odd phase:

oddsize:*0 12 34 56 evensize:*0 12 34 56 *7...

even phase: 0 1 2 3

oddsize:01 23 45 67 *8 evensize: 01 23 45 67...

A.e.1.1 even phase

We observe the above example, in the even phase, all processes with an odd number of ranks transmit data to the previous process (rank-1), while processes with an even number of ranks transmit data to the following process (rank+1). Note that in the even phase, when the rank size is odd, the last process will not send data (with *). I use a variable evenPhaseRank to keep track of which process to send to in the even phase.

A.e.1.2 odd phase

Next, let's look at the odd phase. Similar to the even phase, all processes with an odd number of ranks transmit data to the next node (rank+1), while processes with an even number of ranks transmit data to the previous process(rank-1). Note that at this phase, the first process will not send data, and when the number of ranks is even, the last process will not send and receive data.

A.e.1.3 Send and receive data from another process

I set each process to send and receive data and sort it locally to get the right data, instead of a process sending data to its adjacent process and sorting it by the adjacent process and then sending half of the data back to this process. This can make the performance better because no process is idle. I use MPI_Sendrecv for data transmission and reception.

A.f Merge the received data with local data and sort them

I merge two sorted arrays into one sorted array by creating a function (MergeTwo). First, I will judge whether the process will keep a smaller array or a larger array. If a smaller array is left, it will first compare whether

the last number in the local array is smaller than the first number of the received array. If so, return it directly, because local data is the smaller data. Then through the while loop compare the local data and the received data which is smaller, and then put the smaller value into the temporary array. If the data of any array is run out, then use the while loop to put all the values in the other array until the amount of data in the temporary array is the same as the number of local data, and finally give the value of the temporary array to local data.

If the larger array is to be retained, the last value of the two arrays is compared, and the larger one is placed in the temporary array. Others are similar to the above. If the data of one array runs out first, all the values in the other array are put in through the while loop until the number of data in the temporary array is the same as the number of local data and give the value of the temporary array to local data.

In this part, I exchanged and gave data through the original way of the array at the beginning, but later found that this seems to influence the performance. Many test cases will cause runtime errors, so I use the pointer instead of the original way to exchange data.

A.g Confirms that each node is sorted with adjacent nodes

When sending data, I will first use MPI_Sendrecv to send and receive a piece of data to determine whether this process and the adjacent process are in a sorted state, and then set the variable isSorted to true. After each process has been confirmed with the adjacent processes, use MPI_Allreduce to collect the results of each node, and use the logical operation AND to see whether all processes have been sorted.

A.h Write output into a file

Similar to reading data, use MPI_File_write_at() to write data into the file, and write through the data start position set above.

B. Experiment & Analysis

B.a Methodology

B.a.1 System Spec

The program is tested on the cluster provided in this course. The quota for each person is a maximum of 4 nodes, and each node has 12 processes.

B.a.2 Performance Metrics

B.a.2.1 Computing time

Add MPI_Wtime() after MPI_Init() and before MPI_Finalize() to measure. Calculate the overall time and deduct Communication time and IO time, and the rest is Computing time.

B.a.2.2 Communication time

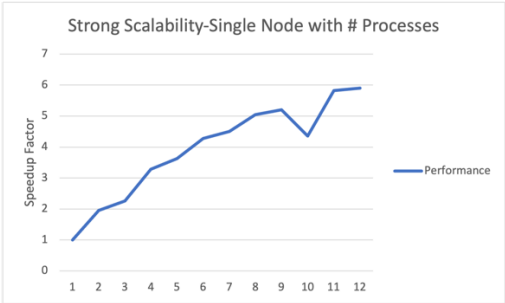
Add MPI_Wtime() before and after MPI_Sendrecv(), get the difference, and then add all the times to get the time spent by the node in Communication.

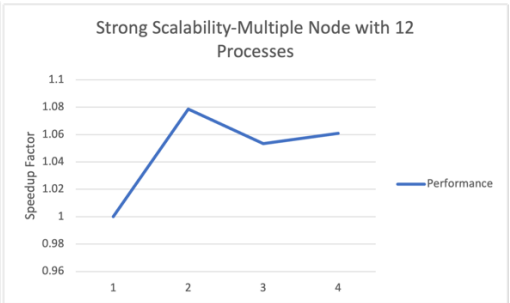
B.a.2.3 IO time

Add MPI_Wtime() before and after MPI_File_read_at() and MPI_File_write_at(), get the difference, and add it up to get the IO time of the process for reading and writing files.

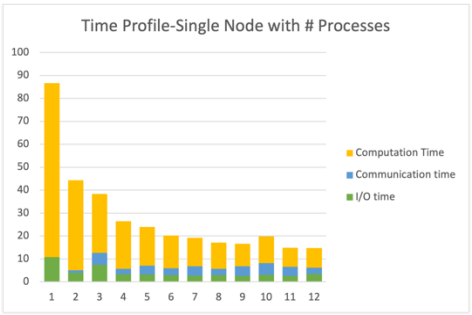
B.b Plots: Speedup Factor & Time Profile

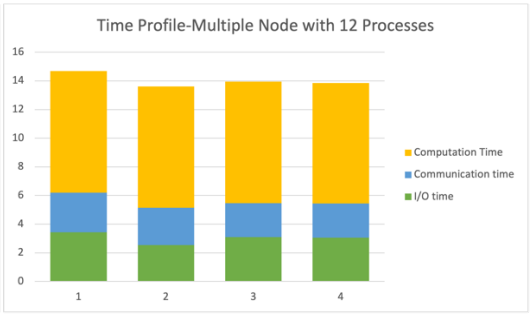
B.b.1 Speedup Factor





B.b.2 Time Profile





B.c Discussion

Experiments are all implemented with test case 38, and there are 536831999 data. I chose test case 38 because it obviously takes more time after test case 35, so I chose test case 38 to observe the performance of the program.

B.c.1 Strong Scalability

We first observe the Speedup Factor, and we can see that both single node and multiple nodes have a certain improvement in performance over time, although the effect is not obvious, and even in some specific processes, the performance will be a bit abnormal. We can find that the process10 of the single node will suddenly drop. With the Time Profile, it is found that the Communication time is obviously more than other processes. It may be because I have an extra MPI_Sendrecv() in the programming to confirm whether there is a transmission. If the numbers are messed up or allocated messed, it may take an extra MPI_Sendrecv() to confirm.

B.c.2 Time Profile,

Single node shows that with the increase of processes, more processes can process data together, and indeed the time required for program execution will decrease significantly. In particular, the single process has no communication time because there are no other processes. As time goes on, IO_Time is almost fixed, CPU_Time also decreases significantly, and the cost of Comm_Time becomes higher, affecting a key part of the entire time.

C. Experiences / Conclusion

This is my first time writing a parallel program through MPI, which is very different from the programs written in the past, and requires a lot of energy to focus on time complexity. I need to think about how to optimize my programs, such as using pointers instead of the original arrays as much as possible, which allows me to think from the perspective of computer memory.

Overall, the main difficulty I encountered in this program was figuring out how to collect the results of each round and use it as a condition for sending and receiving to continue running. I spent a lot of time thinking about how to set the loop conditions and tried many methods, which often led to the program be crash. In the end, this assignment made me understand that in the past, I often sacrificed time and space for the convenience of programming logic, and it also made me think about how to make changes.