CSC4005 Project1 Report

118010045 Cui Yuncong

October 2020

Contents

1 Introduction

This program are required to write a parallel odd-even transposition sort by using MPI. A parallel odd-even transposition sort is performed as follows:

Initially, m numbers are distributed to n processes, respectively.

1. Insides each process, compare the odd element with the posterior even element in odd iteration, or the even element with the posterior odd element in even iteration respectively. Swap the elements if the posterior element is smaller.

2. If the current process rank is P, and there some elements that are left over for comparison in step 1, Compare the boundary elements with process with rank P-1 and P+1. If the posterior element is smaller, swaps them.

3. Repeat 1-2 until the numbers are sorted.

2 Method

2.1 Program Flow

As shown in the Figure 2, from left to right, there are mainly three steps designed in the program.

1. First part: the program goes into the main function, it initializes the variables including random array generation function's configuration, the storage used to contain global and local array, their lengths and other information. Then, it does the MPI initialization, including space allocation, specifying communicator, and get the rank processor number. At the end of this part, the root process called the Scatter function in order to distribute the unsorted array to sub-processes.

2. Second part: p processes are going to do the odd even sort together, in the way demonstrated in the Introdution part. The OddEvenSort() function should not only implement the comparison algorithm, but also the MPI_Send() and MPI_Recv() so that the processes can communicate with each other.

3. Third part: the root process call the MPI_Gather(), all the sub-processes submit sorted array to the root. Then print the result.

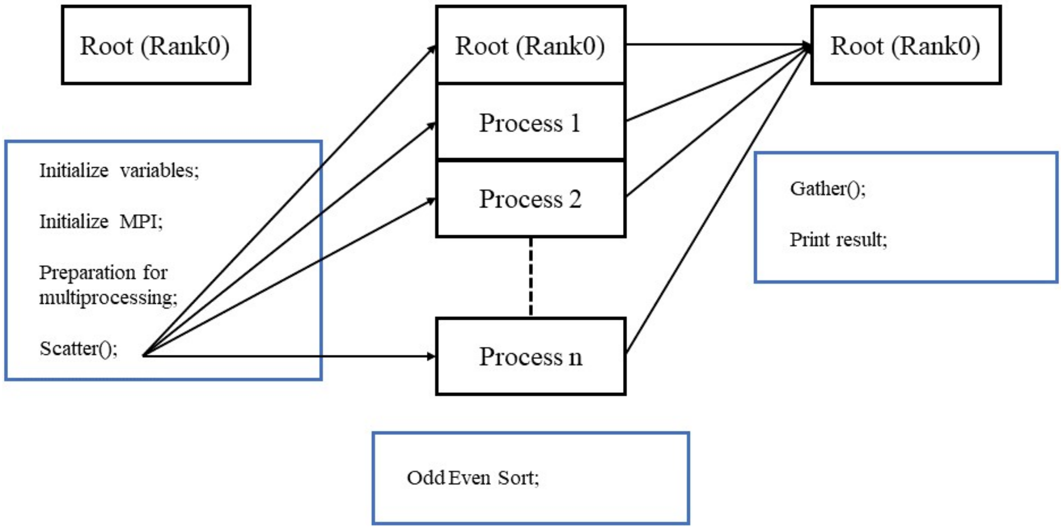

Figure 1: Main Flow Chart

2.2 Data Structure and Algorithm

An array is used to contain the numbers. After scatter, there's a mechanism that test if the current process is holding the sub-array on both ends or in the middle. The program will allocate different number of buffer spaces depends on the sub-array's position: sub-arrays on the ends get one; sub-arrays in the middle get two.

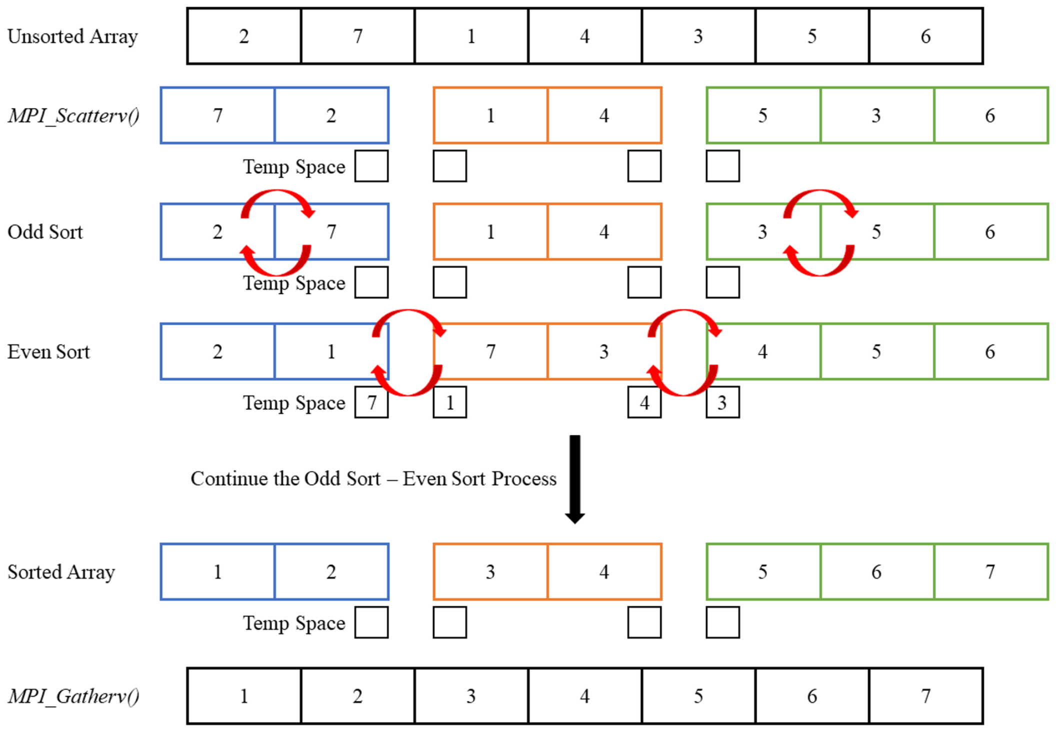

Figure 2: Odd Evenn Sort Steps

2.3 Time Complexity

Odd-even transposition sort works by iterating through the list, comparing adjacent elements and swapping them if they're in the wrong order.

Worst Case: The complexity analysis in the worst case is $O(n^2)$. The array need n times of iteration to be sorted.

Best Case: The complexity analysis in the best case is $O(n)$. The array is sorted at first.

Average Case: The complexity analysis in the average case is $O(n^2)$. Suppose we have n numbers array and p process, in MPI version of odd even sort, each process sorts its $\frac{n}{p}$ numbers; performs p passes of odd-even interchanges; finally, each process has its sorted array. For the calculation of the parallel time, each process can sort in O($\frac{n}{p}\log\frac{n}{p}$). At each phase(odd/even), the communication is $O(\frac{n}{p})$, and merging of two sequence is $O(\frac{n}{p})$. Since we have p phases, the total work of communication and merging should be $O(n)$. Hence, the parallel time should be $O(\frac{n}{p}\log\frac{n}{p}) + O(n)$.

3 Result

3.1 Execution

3.1.1 Execution of mpi.cpp

1. Enter into the file foledr of source file mpi.cpp.
2. Type "mpic++ -o mpi mpi.cpp" to compile.
3. Type "./mpi" to execute and show the result of a sample with 20 numbers.

3.1.2 Execution of sequential.cpp

1. Enter into the file foledr of source file sequential.cpp.
2. Type "mpic++ -o sequential sequential.cpp" to compile. However, sequential.cpp does not use the library of MPI, "g++ -o sequential sequential.cpp" can also compile successfully.
3. Type "./sequential" to execute and show the result of a sample with 20 numbers.

3.2 Different Number of Cores Used in the MPI Program

X-axis is the the size of array. Y-axis is the time (unit: second). In this chart, each line shows the running time of different number of cores.

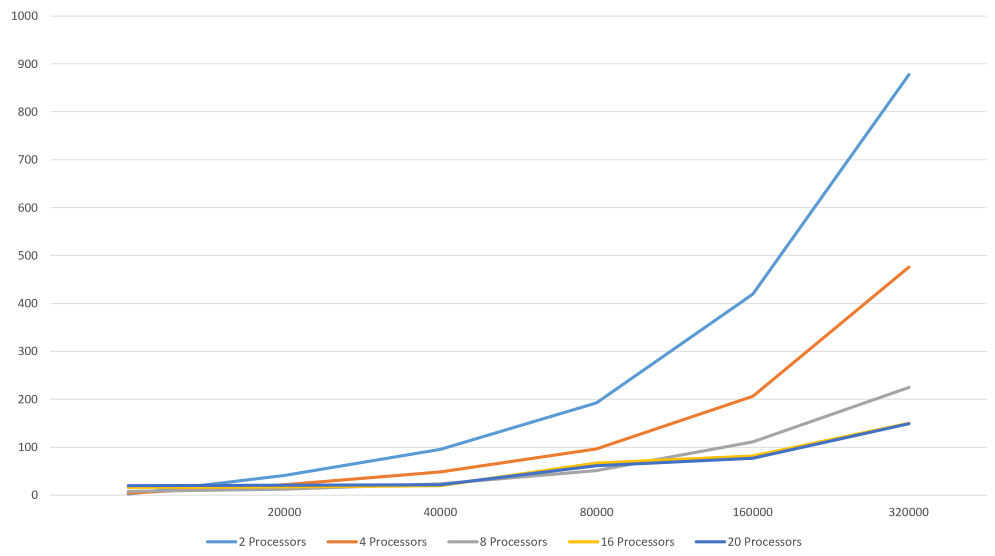

Figure 3: Different Number of Cores Used in the MPI Program

3.3 Different Sizes of Array

X-axis is the the size of array. Y-axis is the time (unit: second). This chart is as same as the last one. Each column in x-axis shows the running time of different sizes of array.

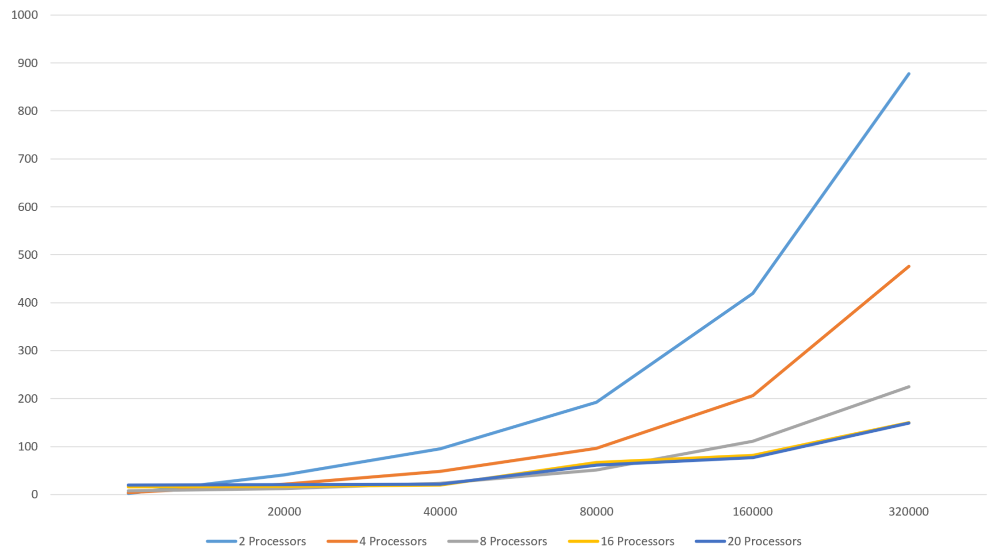

Figure 4: Different Sizes of Array

3.4 MPI vs Sequential

X-axis is the the size of array. Y-axis is the time (unit: second). Here, we use “1 processor” to represent the sequential method. ”2-20 processors” represent the MPI method

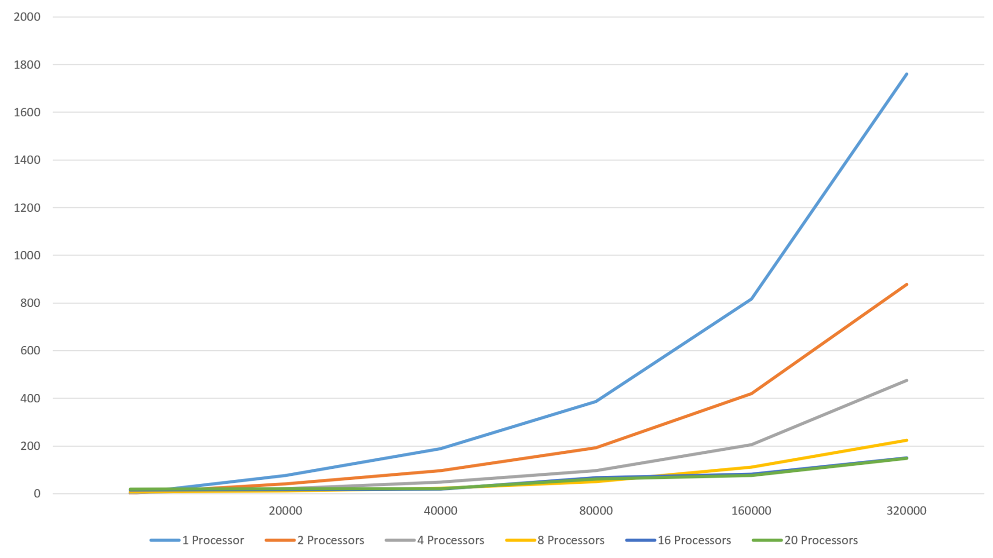

Figure 5: MPI vs Sequential

4 Conclusion

Figure 3, 4, 5 illustrate the time spent to sort the array.

Firstly, regardless of the number of threads, increasing the number of array size can increase running time. This is a fact because of the time complexity (Figure 3 or 4).

Secondly, the sequential method is much slower than the MPI method (Figure 5).

Lastly, regardless of the length of the array, increasing the number of threads can reduce running time. This is because we use several processors to work out the part of the solution together; therefore, the time will be reduced obviously. However, when the number of process becomes larger (16-20), the running time increase in an extremely slow extent. The reason behind this should be the communication overhead (Figure 3 or 4).

To sum up, this program can support the relationship between the parallel computation process and the operating rate mentioned in the lecture. For the potential improvement, when the number of array and the process number are not divisible, the last process would handle more numbers, this could be improved by share the extra number to other processes. The performance should be better than handling by itself.