The Chinese University of Hong Kong, Shenzhen

CSC 4005

Parallel Programming

CSC 4005 Project 1: Parallel Odd-Even Transposition Sort

Author:
Zijuan Lin

Student Number:
121090327

October 11, 2022

Contents

1 Problem statement

In this homework, you are required to write a parallel odd-even transposition sort by using MPI. A a parallel odd-even transposition sort is performed as follows:

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

1. For each process with odd rank P, send its number to the process with rank $P-1$.

2. For each process with rank $P-1$, compare its number with the number sent by the process with rank P and send the larger one back to the process with rank P.

3. For each process with even rank Q, send its number to the process with rank $Q-1$.

4. For each process with rank $Q-1$, compare its number with the number sent by the process with rank Q and send the larger one back to the process with rank Q.

Repeat 1-4 until the numbers are sorted.

2 Compile and run the provided programs

The source files of sequential and parallel program are contained in the **project1** folder.
To compile the sequential program **seq.cpp**, use **gcc -O2 seq.cpp**.
For parallel program **para.cpp**, use **mpic++ para.cpp -o para˙project1**.
To test the programs, you can do as the **README.md** in the template folder.
The programs have been compiled and tested on the HPC 10.26.200.21.

Execution Results:

```
[121090327@node21 executable]$ mpirun -np 10 ./para_project1 10000 ../test_data/10000a.in
actual number of elements:10000
Student ID: 121090327
Name: Zijuan Lin
Assignment 1
Run Time: 0.0305898 seconds
Input Size: 10000
Process Number: 10
```

Figure 1: The output of running the parallel program

```
[121090327@node21 executable]$ ./seq_project1 10000 ../test_data/10000a.in
actual number of elements:10000
Student ID: 121090327
Name: Zijuan Lin
Assignment 1
Run Time: 0.0931678 seconds
Input Size: 10000
Process Number: 1
```

Figure 2: The output of running the sequential program

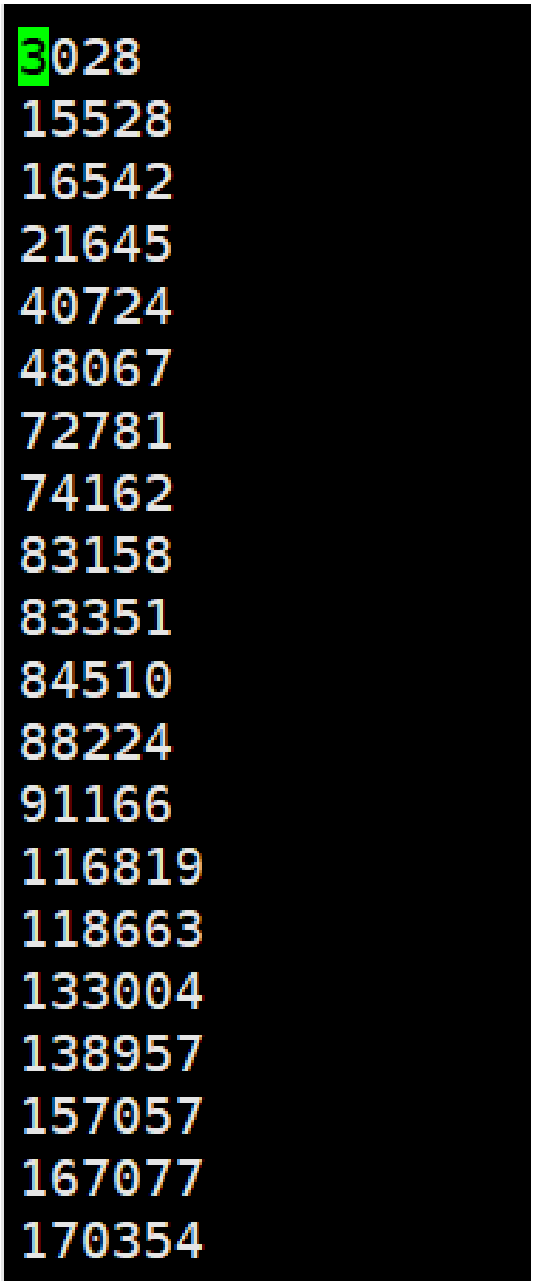

Figure 3: The result of running the programs

3 Introduction to the sorting algorithm

This part contains the basic concept of the algorithm and its construction.

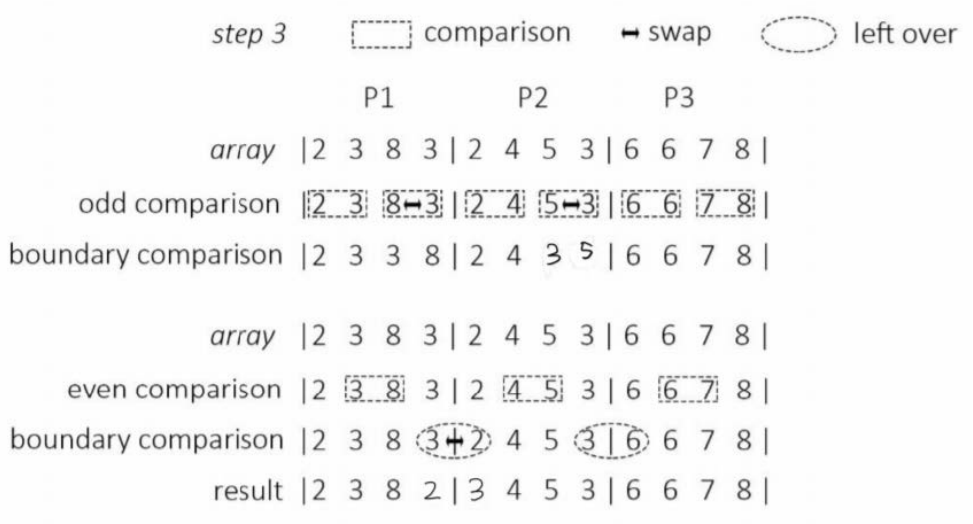

Figure 4: Parallel structure of Odd-Even transposition sort

Basic concepts Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array.

From the above figure, we can see that the core operation is the comperison and value exchange of the elements with their neighbours. In sequential programs, this can be done directly with **std::swap()**, but in the parallel program, we have to use **MPI_Recv()** and **MPI_Send()** in order to perform the comparison and value exchange in different processes.

In each odd/even sort process, the elements will be sorted with their posterior neighbours, then the elements that are not in volved in the process (The first or the last element, or both) will be compared with the "neighbour" in preceding process or posterior process respectively.

Algorithm analysis Let's analyze the complexity of the sequential implementation first.

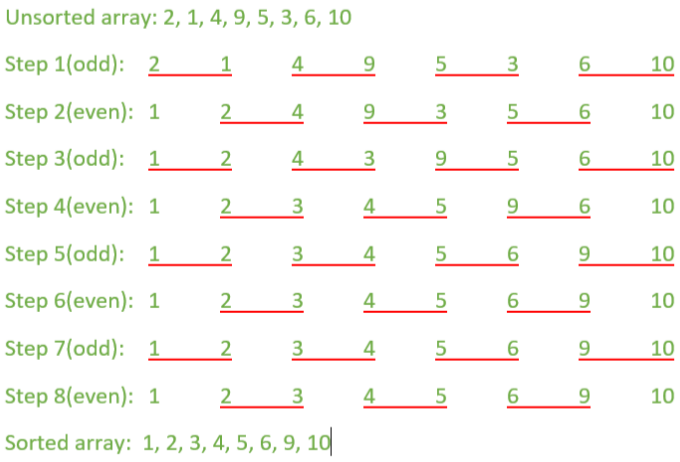

Figure 5: Odd/even transposition sort in sequential form

We can view the sequential implementation as some sort of bubble sort.

- In the worst case, every element are swaped, the time complexity is $O(n^2)$, just as bubble sort.

- In the best cases, we only need to conduct one odd sort and one even sort, which means the time complexity is $O(n)$.

- For average cases, the time complexity is $O(n^2)$

The C++ implementation is as follows.

```
bool is_sorted= false;
while (!is_sorted){
    is_sorted= true;
    for (int i=0;i<num_elements-1;i+=2){

        if (sorted_elements[i]>sorted_elements[i+1]){
            std::swap(sorted_elements[i],sorted_elements[i+1]);
            is_sorted= false;
        }
    }
    for (int i=1;i<num_elements-2;i+=2){

        if (sorted_elements[i]>sorted_elements[i+1]){
            std::swap(sorted_elements[i],sorted_elements[i+1]);
            is_sorted= false;
        }
    }
}
```

Figure 6: C++ implementation of sequential odd/even transposition sort

Then let's analyze the complexity of parallel implementation.

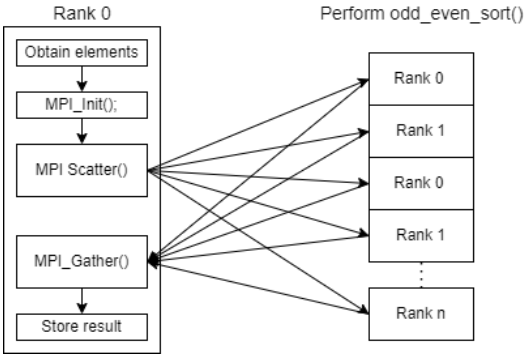

Figure 7: structure of the parallel program

The figure above shows the structure of the parallel program. Rank 0 is used to obtain and distribute elements to processes, and it also gathers the sorted elements and store in a file for checking.

Assume the elements are in the worst case, which means that the elements form a decending array.

- We can see that $\frac{n}{p}$ elements are distributed to each process. Sequential odd/even sort operations are conducted in each process, and the computation complexity is $O((\frac{n}{p})^2)$. - As for data communication, its complexity is $O(n)$ since each process will perform the operation.

According to the formula

$$T(n) = t_{compute}(n) + t_{comm}(n)$$

We can obtain the time complexity of the parallel implementation $O(n^2)$.

The C++ implementation is as follows, this part contains the important part of value exchange in each process and the inter-process exchange.

```
if (i%2==0){
    for (local_indx=num_my_element-1;local_indx>0;local_indx-=2){
        if (my_array[local_indx] < my_array[local_indx - 1]) {
            std::swap(my_array[local_indx - 1], my_array[local_indx]);
        }
    }
}else{
    for (local_indx=num_my_element-2;local_indx>0;local_indx-=2){
        if (my_array[local_indx] < my_array[local_indx - 1]) {
            std::swap(my_array[local_indx - 1], my_array[local_indx]);
        }
    }
    if(rank!=0){
        tmp_tx=my_array[0];
        MPI_Send(&tmp_tx,1,MPI_INT,rank-1,0,comm);
        MPI_Recv(&tmp_rx,1,MPI_INT,rank-1,0,comm,MPI_STATUS_IGNORE);
        if (tmp_rx>my_array[0])
            my_array[0]=tmp_rx;
    }
    if (rank!=world_size-1){
        buf_tx=my_array[num_my_element-1];
        MPI_Recv(&tmp_rx,1,MPI_INT,rank+1,0,comm,MPI_STATUS_IGNORE);
        MPI_Send(&buf_tx,1,MPI_INT,rank+1,0,comm);
        if (tmp_rx<my_array[num_my_element-1])
            my_array[num_my_element-1]=tmp_rx;
    }
}
```

Figure 8: C++ implementation of parallel odd/even transposition sort

Processor-time optimality We can evaluate the efficiency of the algorithm with the processor-time optimality. A parallel algorithm is cost-optimal if

$$(Parallel\ time\ complexity) \cdot (number\ of\ processors) = (Sequential\ time\ complexity)$$

For the odd/even transposition algorithm, $O(n^2) \cdot p = O(n^2)$, it's effective if $p << n$. Thus we can say that the algorithm is cost optimal.

4 Performance analysis

The following tests are done on the HPC 10.26.200.21. Raw data are provided in the Appendix.

In the experiment, we've tried to run the parallel program on [1,2,4,6,8,10,20] processes, tested sorting [100,1000,10000,100000,200000,300000,500000] elements.

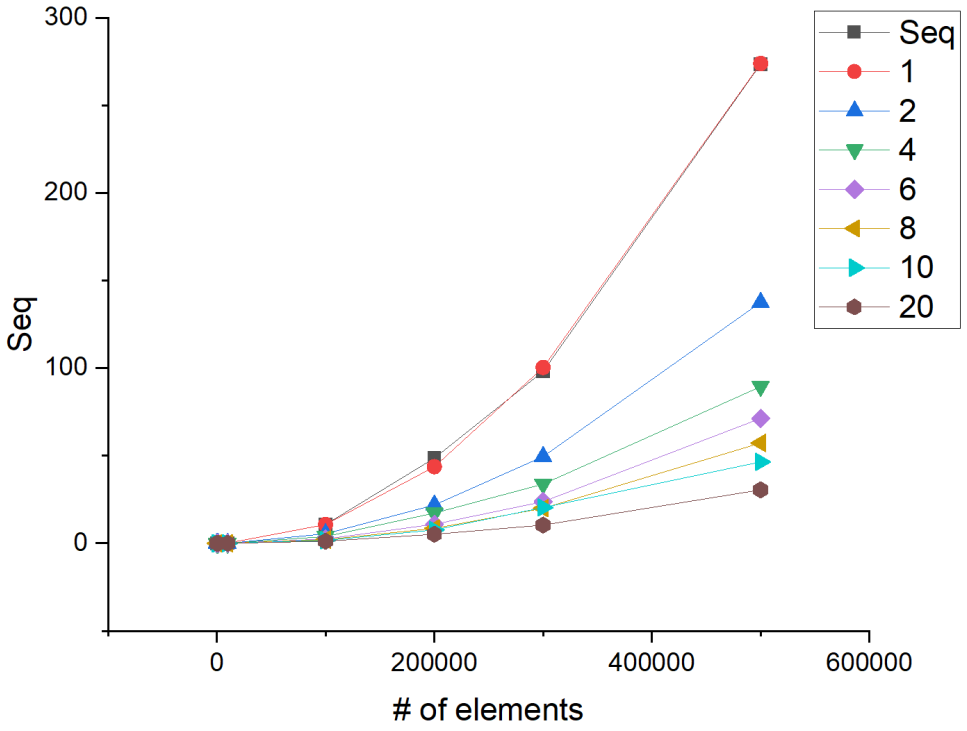

Figure 9: Execution time of the algorithm

The sequential implementation has nearly the same execution time as parallel implementation running in single process.

From the line chart we can get that the execution time is growing quadraticly.

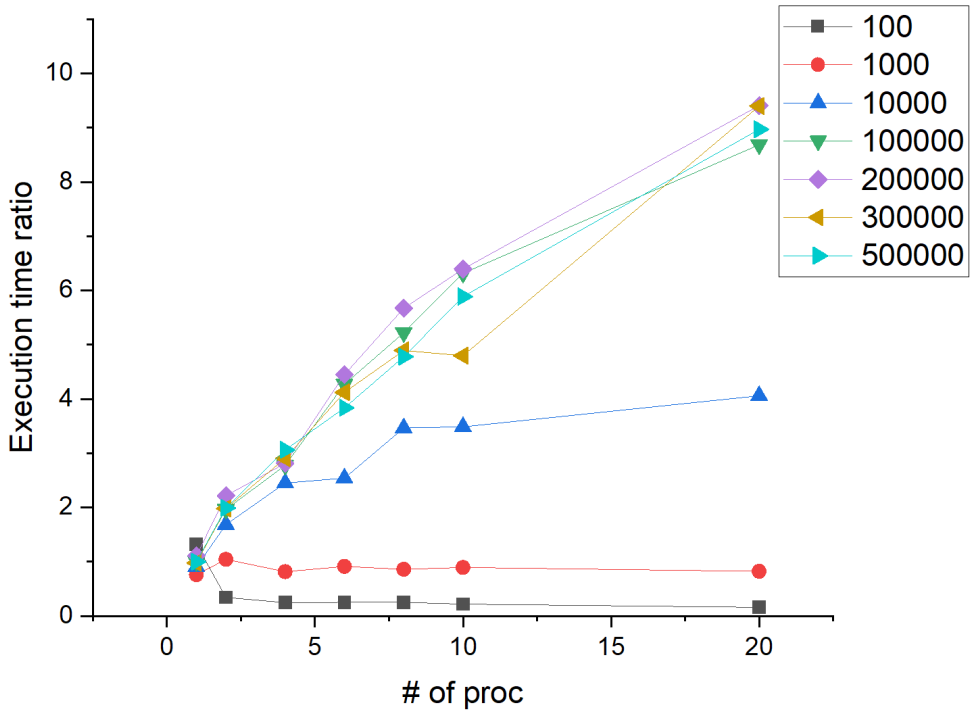

Figure 10: Execution time ratio $t_{seq}/t_{n\ proc}$

Here we can see influence of the sequential parts of the parallel program, the parts include operations like data distribution and data collection which are performed in Rank 0.
Moreover, the inter-process operations are usually slower than same operations performed within a single process, which can be infered from the situation when the number of process is 1.

Result analysis We can infer that the execution time roughly satisfies the following equation from the above experiment:

$$(Parallel\ time\ complexity) \cdot (number\ of\ processors) = (Sequential\ time\ complexity)$$

Thus the algorithm is quite cost-optimal when we view the sequential form as a permutation of bubble sort.

As the number of element rises from n_1 to n_2, the increase of execution time can be approximated to $(\frac{n_2}{n_1})^2$, which proves that the time complexity is $O(n^2)$ as mentioned above. What's more, as the number of processes continuingly grows, we can see

that $t_{seq}/t_{n\ proc}$ has dropped because the sequential operations is occupying a bigger fraction of execution time.

5 Conclusion

From the implementation of odd/even transposition sort algorithm in both sequential and parallel approach, we can have an understanding of the high efficiency of parallel algorithms. Even with only 20 cores in the HPC, we can see the imporvement in execution time when performing the same algorithm parallelly.

However, since this algorithm is a permutation of bubble sort, it's time complexity is still $O(n^2)$, which means it may perform poorly when the data is significantly large.

6 Appendix

This part includes the original data of the experiment, whose visualization has been mentioned above.

Table 1: This table records the execution time of the algorithm.

elements	Seq	1	2	4	6	8	10	20
100	8.75E-05	6.61E-05	0.000255	0.000356	0.000349	0.000345	0.000403	0.000556
1000	0.001729	0.002274	0.001656	0.002124	0.001898	0.00201	0.001939	0.002105
10000	0.094159	0.103947	0.055906	0.038444	0.037091	0.027172	0.027014	0.023206
100000	10.7251	10.7499	5.46485	3.86334	2.51185	2.05519	1.69826	1.23496
200000	48.6073	43.7126	21.9459	17.2719	10.9307	8.57004	7.60383	5.16769
300000	98.1184	100.358	49.5356	33.7579	23.8153	20.0514	20.4614	10.442
500000	273.556	273.852	137.535	89.426	71.3163	57.2441	46.4769	30.5102

Table 2: This table records the Execution time ratio $t_{seq}/t_{n\ proc}$

# of proc	100	1000	10000	100000	200000	300000	500000
1	1.323346	0.760344	0.905833	0.997693	1.111975	0.977684	0.998919
2	0.342591	1.04411	1.68424	1.962561	2.214869	1.980765	1.988992
4	0.245994	0.813968	2.449228	2.776121	2.814242	2.906532	3.059021
6	0.250651	0.910811	2.538597	4.269801	4.446861	4.119973	3.835813
8	0.253361	0.860349	3.465229	5.218544	5.67177	4.893344	4.778763
10	0.217329	0.891933	3.485574	6.315346	6.392476	4.795293	5.885849
20	0.157363	0.82145	4.057528	8.684573	9.406002	9.396514	8.966051