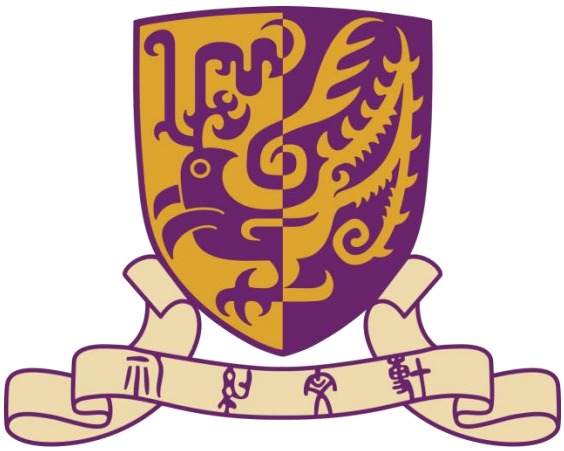


THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

Distributed and Parallel Computing

Report for Assignment 1
Parallel Odd-Even Transposition Sort

Author:
Li Jingyu 李璟瑜

Student Number:
118010141

Oct 13, 2021

Contents

1 Introduction

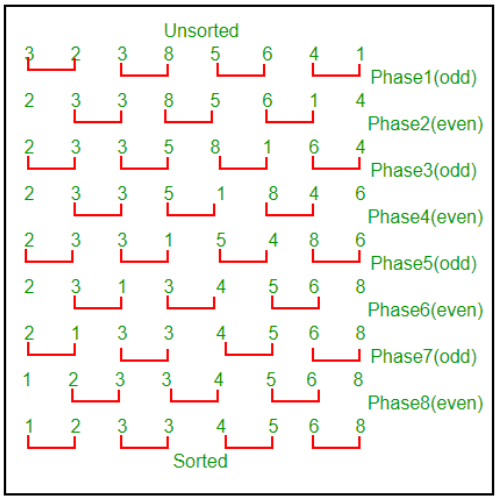
Assignment 1 requires us to design a program that simulates the process of odd-even transposition sort. Both the sequential and parallel versions should be implemented. For parallel versions, we use MPI (Massage Passing Interface) to finish the multi-process work. MPI is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between processes. A sample MPI interface is shown as follows. MPI_Send is one of the most popular functions in MPI.

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator);
```

Odd-even transition sort is basically a comparison sort that is similar with bubble sort. There are two phases called odd phase and even phase. In odd phase, the comparison starts from the first element. Each pair of elements will exchange their order into the increasing form. That is, if the previous element is larger than the next one, then exchange their position. No coverage will occur between two pairs. Similarly, in even phase the sorting starts from the second element, leaving the first element out and continues by pair comparison. The idea of the process can be peeped at the figure below.

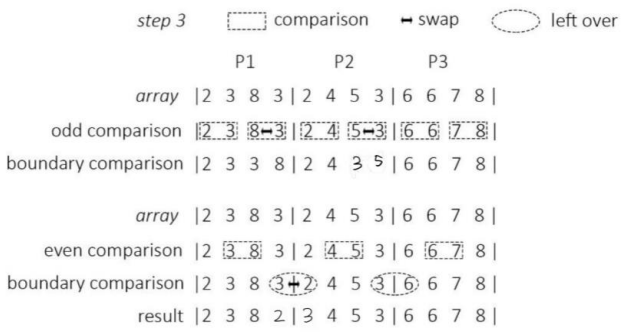
If no check is required to verify the completion of the sort, the sorting process will go through N phases, where N is the total number of elements in the input array.

Each phase at most (N/2 + 1) exchanges will happen, so the time complexity is **O(N^2)**. The space complexity equals to O(1) since it is an in-place sort algorithm.



The algorithm is correct since after N phases, the number will finally move to its correct position in increasing (decreasing) order of the array, given that there are at most N positions.

The parallel version of odd-even transposition sort is similar with the sequential version, differing in the distribution of the numbers. Initially, **m** numbers are distributed to **n** processes respectively, and the pair comparison in each process goes as mentioned before. On the boundary, the process needs to decide whether to pass or receive the element from the neighboring processes. The pair comparison occurred on the boundary requires message passing of processes. The figure shows as follows.

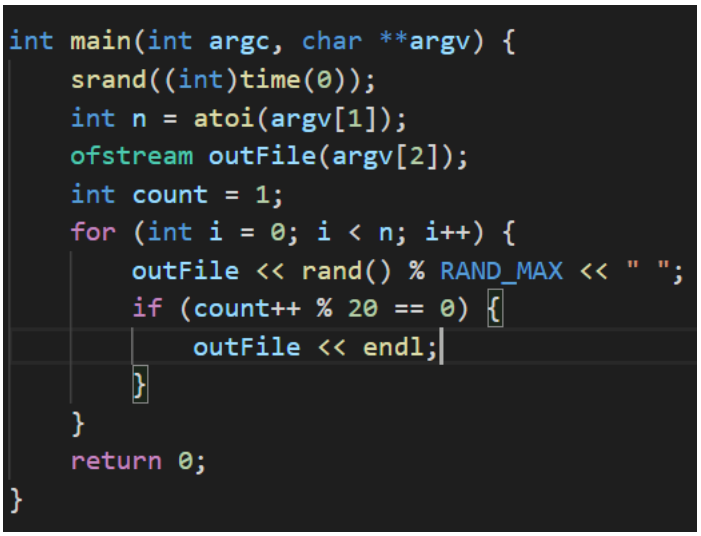


2 Design

In this part the logic of the codes and some details will be elaborated.

- Random number generator

To generate the random numbers for sorting, a simple generation file is written. The user can specify the output file name as well as the data size. The main function is shown below. The range of numbers is from 0 to RAND_MAX, whose value will vary according to the operating system. The numbers are in the format of 20-dim array as the requirement states.



```
int main(int argc, char **argv) {
    srand((int)time(0));
    int n = atoi(argv[1]);
    ofstream outFile(argv[2]);
    int count = 1;
    for (int i = 0; i < n; i++) {
        outFile << rand() % RAND_MAX << " ";
        if (count++ % 20 == 0) {
            outFile << endl;
        }
    }
    return 0;
}
```

- Sequential Version odd-even sort design

For sequential version, it goes as the algorithm illustrates. To maintain the similarity between the sequential and parallel versions, the code is written like the template.

In the main function, random numbers from the file will be read to a vector. The start and end pointers of the vector will serve as two arguments of odd_even_sort()

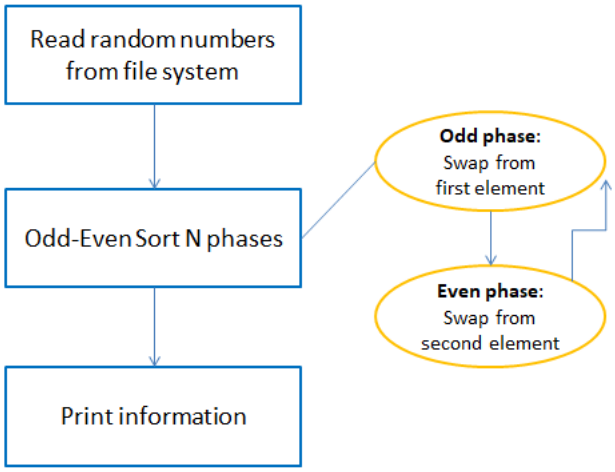
function. In the sorting function, a for loop executes the odd phase and even phase in total *m* times (m is the data size). In odd phase, compare neighboring pair and exchange the position if necessary, and in even phase, perform similar operations. A swap function assists the position exchange procedure. At last, personal information like name and student id as well we runtime profiling like process numbers, sort duration will print out. All the sorted data will be written to a new file.

Core function part and a structural flow figure are displayed here.

```
for (int i = 0; i < size / 2 + 1; i++) {  // Fixed times sorting

    // Odd phase
    for (int j = 0; j < size - 1; j += 2) {
        if (*(begin + j) > *(begin + j + 1)) {
            swapE(begin + j, begin + j + 1);
        }
    }

    // Even phase
    for (int j = 1; j < size - 1; j += 2) {
        if (*(begin + j) > *(begin + j + 1)) {
            swapE(begin + j, begin + j + 1);
        }
    }
}
```



- Parallel Version odd-even sort design

For parallel version, it is more complex since it corresponds to multiple processes while different process shares different memory. The code fits the template and makes some changes.

In the main function, the program reads the input file name as well as the output file name from command line. Then root process (rank 0) reads all the random numbers into a vector. Root process 0 then calls the function context.mpi_sort(), passing the start and end pointers of the vector as the arguments.

In the MPI sort function, some variables are initialized. A buffer number is used to receive element passed from neighboring processes. Totalcount and locaCount record the number of elements totally and locally. LocalArray is used to store the elements belonging to the current process. At first, root process 0 stores some information like array size and process number into a metadata structure. Then it broadcasts the total numbers in array to all other processes (useful information). By using totalCount, each process can calculate the local elements number and allocates relative memory to store them. Then root process 0 scatters the numbers **evenly** to other process while keeping the remaining elements itself (Make sure totalCount can be divided by localCount). Then the algorithm starts.

The algorithm has a for loop to execute totalCount times. Each time, every process will perform oddSort() or EvenSort() by determining whether the previous elements are odd or even (also with the odd or even phase). If the previous elements are even in odd phase, it means no process communication will occur between current

process and the last process. So the pair comparison will just start from the first element. Here oddSort() is encapsulated to finish pair exchange starting from the first element, similar as the evenSort(). After local sort, the process then determines whether it should send a number to next process according to parity. Process communication occurs if one element is left at the end of the local array, MPI_Send and MPI_Recv are executed to deal with the message passing logic. In summary, two decisions are needed. First, decide whether to implement odd sort or even sort by checking the parity of previous elements (If odd, it also needs to receive one element from previous process). Second, decide whether to pass an element to the next process after local sort.

Root process 0 and the last process (size-1) need to be classified in special cases since they have only one neighboring process.

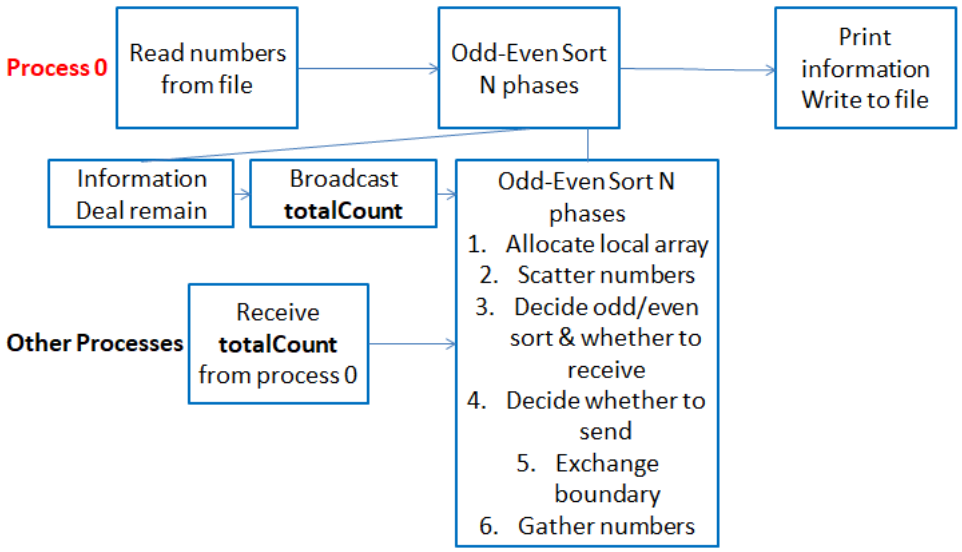
Odd phase and even phase can be combined in this way by checking the parity of the remains.

After local sorting, root process 0 will deal with the remaining elements and gather all the numbers from all other processes.

At last, relevant information (personal & running time) will print out in process 0 and sorted result will be written to an output file.

Core parts and program flow show as follows.

```
int previous = localCount * rank + remain;
if (previous % 2 == i % 2) {  // The number of previous elements is even
    oddSort(localArray, localCount);
    if (localCount % 2 == 1) {  // need to send a number to next process
        MPI_Send(localArray + localCount - 1, 1, MPI_LONG, rank + 1, MASTER, MPI_COMM_WORLD);
        MPI_Recv(&buffer, 1, MPI_LONG, rank + 1, MASTER, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        *(localArray + localCount - 1) = buffer;
    }
} else {  // The number of previous elements is odd, receive one from last process
    evenSort(localArray, localCount);
    MPI_Recv(&buffer, 1, MPI_LONG, rank - 1, MASTER, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (buffer > localArray[0]) {
        swapE(&buffer, localArray);
    }
    MPI_Send(&buffer, 1, MPI_LONG, rank - 1, MASTER, MPI_COMM_WORLD);
    if (localCount % 2 == 0 && localCount > 1) { // even element, need to send one to the next
        MPI_Send(localArray + localCount - 1, 1, MPI_LONG, rank + 1, MASTER, MPI_COMM_WORLD);
        MPI_Recv(&buffer, 1, MPI_LONG, rank + 1, MASTER, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        *(localArray + localCount - 1) = buffer;
    }
}
```



- Some details

The number of phases is the same for sequential and parallel versions so as to make them more comparable. If for some array the sorting stops at some phase in the middle, then the comparison result is not so accurate.

The remaining numbers that cannot be divided by the process number are all stored in process 0 at the beginning of *begin and the local array. Some offset operations will handle these remaining numbers in order to divide all the elements evenly into slave processes.

The method I use to handle the situation when the number of processes is larger than the size of elements is by adding RAND_MAX to the data array to make it divisible by the process number. At last, just write the original size numbers into the output file, leaving the added RAND_MAX numbers out. It is feasible since RAND_MAX is always placed at the end of the array.

3 Execution

The execution for my code performs on the remote cluster. There are 8 internal nodes each with one **NVIDIA 2080Ti GPU card** with 32 cores each. The maximum CPU time limit is 640 (4 nodes, 128 cores for 5 minutes).

Environment: Linux operated with MPI library

Way to compile and execute:

1. Unzip 118010141.zip
2. cd 118010141 & cd csc4005-assignment-1-sequential/
3. g++ generateNum.cpp -o generateNum
4. generateNum 10000 in.txt
5. g++ -std=c++11 odd-even-sort_sequential.cpp -o odd-even-sort_sequential
6. odd-even-sort_sequential in.txt out.txt [cat out.txt to check results]
7. (cd ..) cd csc4005-assignment-1
8. mkdir build && cd build
9. cmake .. -DCMAKE_BUILD_TYPE=Release
10. cmake --build . -j4

```
cd /path/to/project
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug # please modify this to `Release` if you
want to benchmark your program
cmake --build . -j4
```

11. ./gtest_sort [Check correctness]
12. Move in.txt to build folder
13. cp in.txt /home/118010141/118010141/csc4005-assignment-1/build/
14. **mpirun -np 4 main in.txt out.txt** [Can modify the process number here]

Some notice:

If no library information: Export LD_LIBRARY_PATH=<build_dir>

I run in **/pvfsmnt/118010141**, the share file system.

For experiment with process number from 1 to 20, I use salloc to run interactively. **Salloc –N1 –n32 –t10**

For experiment with process number larger than 32, I use sbatch to run as a script.

A sample script:

```
#!/bin/bash
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --nodes=1
#SBATCH --ntasks=32
#SBATCH -J core32
#SBATCH -o core32.out

mpirun -np 32 main in.txt out.txt
```

4 Result & Analysis

- Correctness

By executing **./gtest_sort**, the code can generate the following result.

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from OddEvenSort
[ RUN      ] OddEvenSort.Basic
[       OK ] OddEvenSort.Basic (0 ms)
[ RUN      ] OddEvenSort.Random
[       OK ] OddEvenSort.Random (2591 ms)
[----------] 2 tests from OddEvenSort (2591 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (2592 ms total)
[  PASSED  ] 2 tests.
```

A general result shows as follows:

mpirun -np 4 main in.txt out.txt

```
Name: Li Jingyu
Student ID: 118010141
Assignment 1, odd-even sort, MPI implementation
input size: 10000
proc number: 4
duration (ns): 33402964
```

A demo output for 10 number are shown (random → sorted):

```
bash-4.2$ cat 10.txt
261503569 208036025 1012735387 751569422 1069596751 787212424 711210633 1431765744
428039239 1072344857 bash-4.2$ cat out.txt
208036025 261503569 428039239 711210633 751569422 787212424 1012735387 1069596751 1
072344857 1431765744 bash-4.2$
```

- Robustness

By executing **mpirun -np 10 main 0.txt out.txt**, the code can deal with the empty data.

```
Name: Li Jingyu
Student ID: 118010141
Assignment 1, odd-even sort, MPI implementation
input size: 0
proc number: 10
duration (ns): 228750
```

By executing **mpirun -np 15 main 10.txt out.txt**, the code can deal with the case where data size is smaller than the number of processes (10.txt means a file with 10 random numbers).

```
Name: Li Jingyu
Student ID: 118010141
Assignment 1, odd-even sort, MPI implementation
input size: 10
proc number: 15
duration (ns): 334258
```

- Relationship between running time & process number

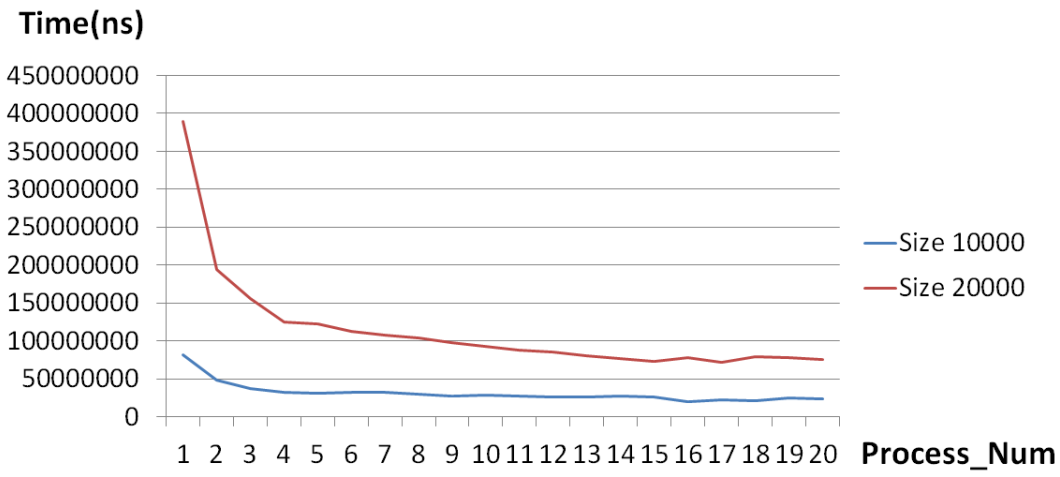
To explore the relationship between running time in nanoseconds (y-axis) and the process number (x-axis), following experiment are presented. Using the control-variable method, the array size should be fixed among different process numbers. So I set two groups of array size. One has **10000** elements; the other has double size, which is **20000** elements. In each group, process number from 1 (sequential) to 20 will be tested, whose duration time will be recorded and graphed in a line chart to make the comparison.

The detailed data and the line chart are shown as follows.

Process number	Size 10000 time (ns)	Size 20000 time (ns)
1	81745242	388983871
2	48145663	194672731
3	37249942	156042272
4	31814981	125412916
5	31273321	122993544
6	32586501	112472110
7	32246456	107311175
8	29205116	103607050
9	26735879	98203612

10	27895071	92998687
11	26781532	88181154
12	25990610	84887595
13	26342242	80319921
14	26710181	76679566
15	25833601	73013239
16	20028008	77930177
17	21741043	71873421
18	21121758	79738405
19	24319923	77405813
20	23431157	75248752

Relationship between running time(ns) & process number(1-20)



As can be seen from the data and the graph, the program runs the longest time sequentially, and then the running time drops sharply from 1 process to 2

processes, the speed nearly doubles. From the data we can see for array size 20000, the speed doubles, which benefits from the surplus of one process. For array size 10000, the speedup is slightly smaller than 2, since there may be some costs for process creation and synchronization. The smaller the array size, the bigger the effect for extra cost for creating a process.

As the number of processes continues to increase, the running time reduce. However, the reduction speed is becoming slower and slower. For example, the time to finish the sorting task sequentially both for array size 10000 and 20000 is smaller than N multiplies # of processes. And as N increases, the time difference becomes even larger, which means the speedup effect is more and more tiny! The reason of this may be the cost of process communication, as the number of processes increases, the communication and synchronization cost is surging, given the idea of the algorithm is to exchange at the boundary number and wait for sending and receiving numbers and then put to the local array. In sequential case, no extra exchange cost occurs. So the cost of message passing increases as processes number increases. The more the processes, the larger the influence of the process communication has. And at last, the time becomes nearly constant when # of processes > 10. The slight fluctuation may be caused by the network stability. The change tendency for both array size is similar since they have the same data size level (10^4). More array size data can be conducted.

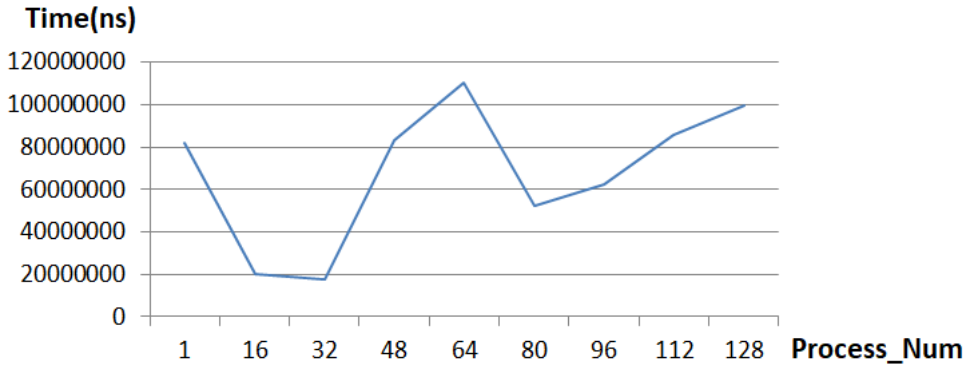
Comparing the time between this two groups, the former time is nearly a quarter of the latter one in ratio, proving that this algorithm is O(N^2) in complexity.

I made another experiment, setting the number of processes the multiple of 16. Since every node has 32 cores, I would like to explore the efficacy when the cores are distributed in different GPU nodes. Here shows the result.

Process – Running time (ns)

1	8.2E+07
16	2E+07
32	1.8E+07
48	8.3E+07
64	1.1E+08
80	5.3E+07
96	6.3E+07
112	8.6E+07
128	1E+08

Relationship between running time(ns) & process number(1-128)



From the graph, the running time reduces greatly from 1 process to 16 processes and reduces smoothly from 16 processes to 32 processes, which fits our discussion preciously. Because the cost for process creation/communication/ synchronization increases as the number of processes adds up. However, strange

things happen, the running time increases greatly from 32-48-64 nodes!!! From this phenomenon and the fact that every node contains 32 cores running 32 processes I can conclude that the cost for process communication between different nodes is far greater than the that in the same node. In the same GPU, the cost is acceptable, but in the different GPU, the cost cannot be ignored! For small size array like this experiment (10000), the lowering effect is obvious. For larger size array like 10w+, the effect may reduce due to the increasing of total running time. The running time then has a reduction due to the process adding and then again increases, which can be explained by the structure of the cluster. There are 32 cores in a node, and every one node addition will increase the running time. Since the partition of internal node is unknown, it is hard to gain a accurate conclusion, here it is just an insight.

The analysis between sequential and parallel processes is also included ahead (when # of processes = 1).

- Relationship between running time & array size

Similar to the previous experiment, to explore the relationship between the array size and running time, the number of processes should be fixed. Here I choose two kinds of processes: sequential (n = 1) and parallel (n = 10) and fix the process numbers. The array size varies from 10 to 100000, with 10X each time. The results are shown as follows.

Array size	Process = 1 (ns)	Process = 10 time (ns)
10	13421	230968
100	28833	506908
1000	911073	1709320
10000	81385243	28032928
100000	12074108420	2174675178

Due to the huge level difference, the graph has not been plotted here. We can have a glance and considering that the time complexity is O(N^2), the time ration should be X100 between two neighboring groups. However, for sequential case, the ration is smaller than X100 for 10-10000 array size definitely, showing the influence of the part not belong to the algorithm but occupying some time. When the array size increases, the ratio is closer and closer to 100, showing the correctness of the time complexity. When the array size continues to grow, the influence of the constant term for the time complexity has shown.

For parallel version program, initialization of the multiple processes costs time, so the running time is even larger than sequential case in relatively small array size. Then it grows with the same tendency with the sequential case. When array size comes to 10000, the power of parallel computing finally shows off and the time is smaller. The fluctuation is caused by the networking.

The analysis between sequential and parallel processes is also included ahead (when # of processes = 1).

5 Summary

Here I will summarize what I have learned when writing assignment 1.

- Understanding of parallel computing. For example, different process shares different memory, so the same pointer cannot make effect.
- The problems to be noticed when writing parallel programs (process synchronization, process communication).
- MPI library usage
- Coding ability improvement
- Cluster experience. Queuing to obtain the resources! Crowded queue before ddl.
- Some optimization: the remaining numbers can also be dealt with by adding RAND_MAX to make the size divisible by process numbers and then write original data size to the file. The logic is simple. Besides, more experiments can be conducted for different array size exploring the relationship between process number and running time. Also, more process numbers can be set for the second experiment. The analysis can be more fine-grained.
- By the way, TUT is quite practical for writing the project and TA & USTF are helpful in answering the questions in the wechat Group, Kudos!

That’s all.