

CSC 4005
Distributed and PC

Message Passing Computing for Odd-even Transposition Sort

the Chinese University of Hong Kong, shenzhen

Author: DING, Xinyi
Student ID: 116010035

October 25, 2018

Content

1. Intruduction

2. Instruction

2.1 Environmmnet

2.2 Manipulate

3. Design

3.1 Ideas

3.2 Pseudo-code

4. Experience & analysis

4.1 How to analyze the performance

4.2 experiments and results

4.3 Anlysis

5. Experience

5.1 Main implementation issues and solutions.

5.2 Problems remain to solve

5.3 Refletion

- 5.3.1 What went well and what I have learned

- 5.3.2 What you would do differently next time

- 5.3.3 What the instructors and Tas might have done differently to promote learning

6. Acknowledges,

7. Reference

8. Source code

1. Introduction

This is the report for assignment 1 of course CSC4005 in the Chinese University of Hong Kong(shenzhen). The assignment requires us to write a parallel odd-even transposition sort by using MPI. The report aims to instruct you how to compile and execute the code source, discuss the ideas about designing the program, show some experiment results and performance analysis and share experience of this assignment in the end. Here is the content:

2. Instruction

2.1 Environment

- i. Platform: Windows 10 x64
- ii. Language: C++
- iii. Compiler: Microsoft Visual C++
- iv. MPI: MS-MPI v9.0.1

2.2 Manipulate

Step 1

Firstly, you should configure the MPI environment and have a C++ compiler. Here we use Visual Studio 2017 and MS-MPI v9.0.1 to execute the program.

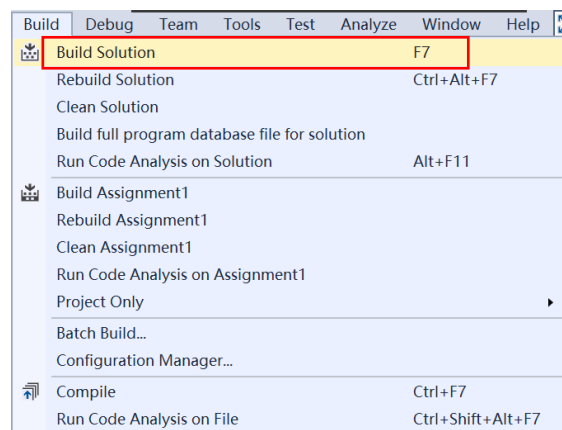
Step 2

Construct a C++ project and paste the code to your main file. Go to the code file to alternate the number of the data which needs to sort according to your need:

```
int const N = 200;           // the number of data
int sequence[N];            // elements need to be scattered
int rank, size;             // id of rank and the number of processes
int reminder;               // reminder if it is not divisible
```

Step 3

To build Solution. You can use the key F7 or click the option “Build Solution” as following:



Step 4

Open the terminal on your computer (use windows + R if your computer system is windows) and go to the file where contains the .exe file of your program like the following.

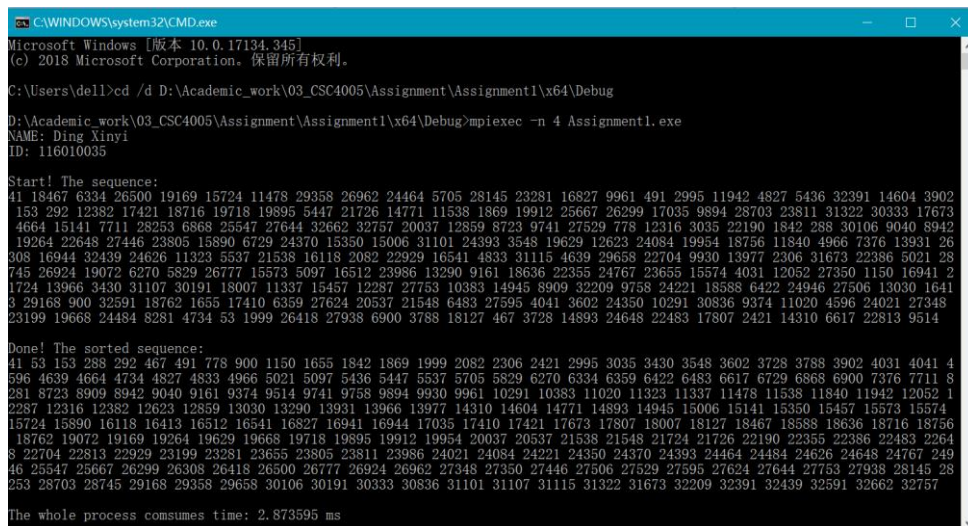
```
C:\Users\de11>cd /d D:\Academic_work\03_CSC4005\Assignment\Assignment1\x64\Debug
```

Execute the program in your terminal. The number after 'n' is the number of the processes you open and the backmost file name is the exe file name. Here is an example:

```
D:\Academic_work\03_CSC4005\Assignment\Assignment1\x64\Debug>mpiexec -n 4 Assignment1.exe
```

Step 5

Then you can get a result like:



```
Microsoft Windows [版本 10.0.17134.345]
(c) 2018 Microsoft Corporation. 保留所有权利。

C:\Users\de11>cd /d D:\Academic_work\03_CSC4005\Assignment\Assignment1\x64\Debug
D:\Academic_work\03_CSC4005\Assignment\Assignment1\x64\Debug>mpiexec -n 4 Assignment1.exe
NAME: Ding Xinyi
ID: 116010035

Start! The sequence:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464 5705 28145 23281 16827 9961 491 2995 11942 4827 5436 32391 14604 3902
153 292 12382 17421 18716 19718 19895 5447 21726 14771 11538 1869 19912 25667 26299 17035 9894 28703 23811 31322 30333 17673
4664 15141 7711 28253 6868 25547 27644 32662 32757 20037 12859 8723 9741 27529 778 12316 3035 22190 1842 288 30106 9040 8942
19264 22648 27446 23805 15890 6729 24370 15350 15006 31101 24393 3548 19629 12623 24084 19954 18756 11840 4966 7376 13931 26
308 16944 32439 24626 11323 5537 21538 16118 2082 22929 16541 4833 31115 4639 29658 22704 9930 13977 2306 31673 22386 5021 28
745 26924 19072 6270 5829 26777 15573 5097 16512 23986 13290 9161 18636 22355 24767 23655 15574 4031 12052 27350 1150 16941 2
1724 13966 3430 31107 30191 18007 11337 15457 12287 27753 10383 14945 8909 32209 9758 24221 18588 6422 24946 27506 13030 1641
3 29168 900 32591 18762 1655 17410 6359 27624 20537 21548 6483 27595 4041 3602 24350 10291 30836 9374 11020 4596 24021 27348
23199 19668 24484 8281 4734 53 1999 26418 27938 6900 3788 18127 467 3728 14893 24648 22483 17807 2421 14310 6617 22813 9514

Done! The sorted sequence:
41 53 153 288 292 467 491 778 900 1150 1655 1842 1869 1999 2082 2306 2421 2995 3035 3430 3548 3602 3728 3788 3902 4031 4041 4
596 4639 4664 4734 4827 4833 4966 5021 5097 5436 5447 5537 5705 5829 6270 6334 6359 6422 6483 6617 6729 6868 6900 7376 7711 8
281 8723 8909 8942 9040 9161 9374 9514 9741 9758 9894 9930 9961 10291 10383 11020 11323 11337 11478 11538 11840 11942 12052 1
2287 12316 12382 12623 12859 13030 13290 13931 13966 13977 14310 14604 14771 14893 14945 15006 15141 15350 15457 15573 15574
15724 15890 16118 16413 16512 16541 16827 16941 16944 17035 17410 17421 17673 17807 18007 18127 18467 18588 18636 18716 18756
18762 19072 19169 19264 19629 19668 19718 19895 19912 19954 20037 20537 21538 21548 21724 21726 22190 22355 22386 22483 2264
8 22704 22813 22929 23199 23281 23655 23805 23811 23986 24021 24084 24221 24350 24370 24393 24464 24484 24626 24648 24767 249
46 25547 25667 26299 26308 26418 26500 26777 26924 26962 27348 27350 27446 27506 27529 27595 27624 27644 27753 27938 28145 28
253 28703 28745 29168 29358 29658 30106 30191 30333 30836 31101 31107 31115 31322 31673 32209 32391 32439 32591 32662 32757

The whole process consumes time: 2.873595 ms
```

3. Design

3.1 Ideas (sentences first, use pictures & diagrams)

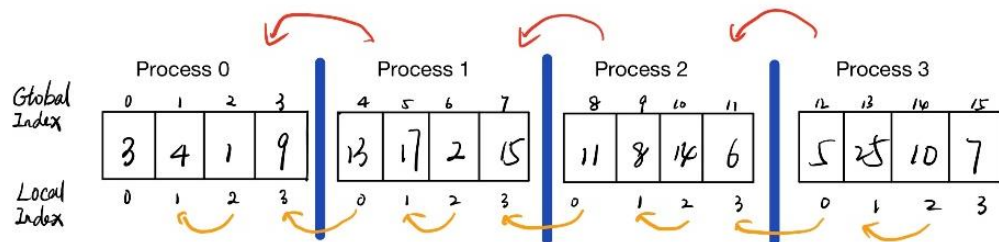
- 1) Generate a random sequence of integer numbers and stored in an array. Use rand() function to get the random sequence. If no other specification, the default seed is 1.
- 2) Distribute all numbers in the array to every process by using MPI_Scatterv() function. Generally, if the number of the data can be divided exactly by the size of processes, we have equal distribution. If not, we add the reminder to the last process.
 - a) Define three dynamic arrays, sendCounts, displs and revseq. sendCounts allocates memory for array storing number of elements. Displs calculate corresponding displacement. revSeq() is the receive buffer.

The code:

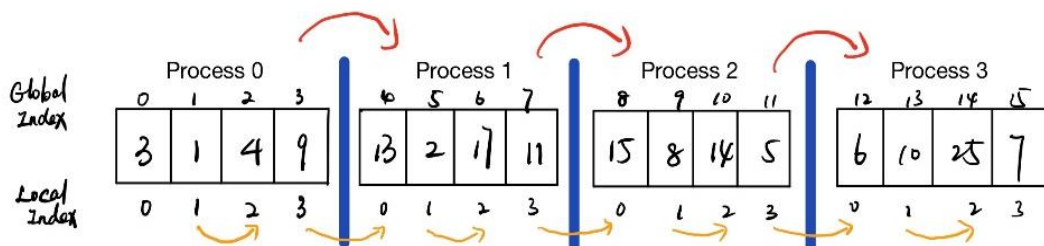
```
MPI_Scatterv(sequence, sendCounts, displs, MPI_INT, revSeq, sendCounts[rant], MPI_INT, 0, MPI_COMM_WORLD);
```

b) Then we do the odd-even transposition sort. We need to discuss seven different situations:

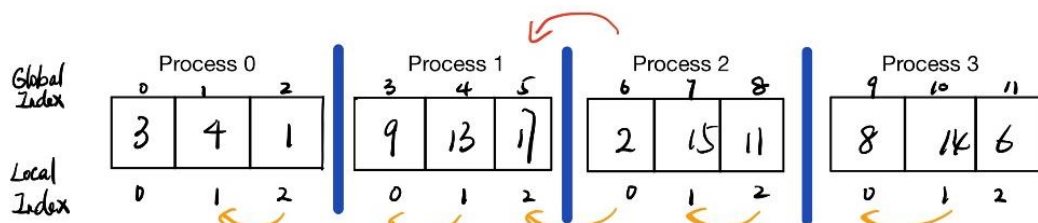
- **Even sort, the first size – 1 processes have even numbers, rank is not equal to 0.** The processes except for rank 0 process send head number to the previous process and compare tail number of the previous process. Numbers in other places within one process do even sort sequentially. The program in all processes are parallel.



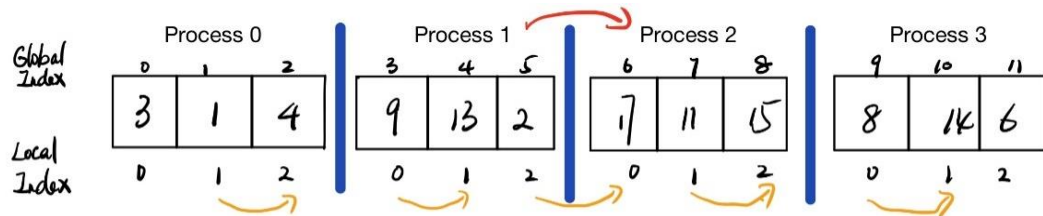
- **Even sort, the first size – 1 processes have even numbers, rank is not equal to size – 1.** The processes except for rank size - 1 processes send back a larger number to the next process. Numbers in other places within one process do even sort sequentially. The program in all processes are parallel.



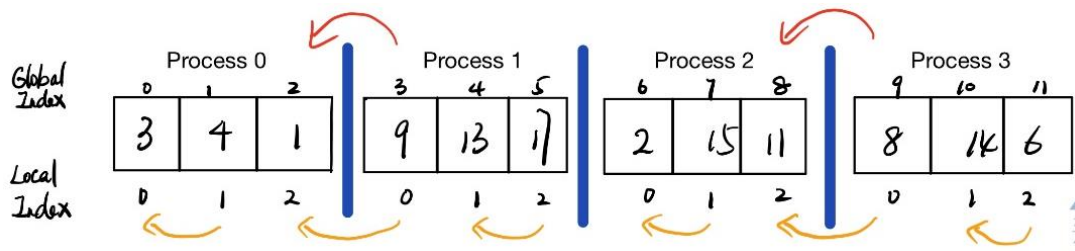
- **Even sort, the first size – 1 processes have odd numbers, rank is even and not equal to 0.** The even rank processes except for rank 0 send head number to the previous process and compare tail number of the previous process. Numbers in other places within one process do even sort sequentially. The program in all processes is parallel.



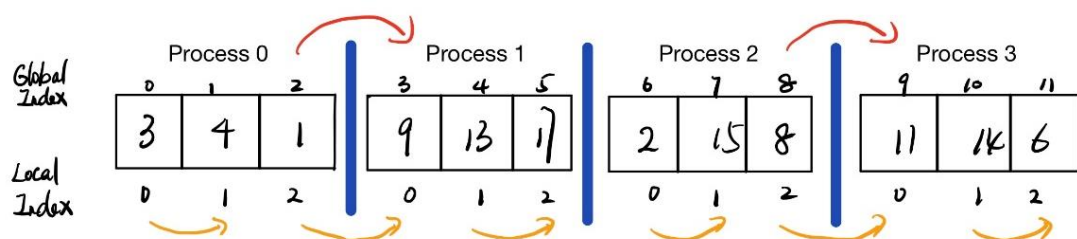
- **Even sort, the first size – 1 processes have odd numbers, rank is odd and not equal to size -1.** The odd rank processes except for rank size - 1 send back a larger number to the next process. Numbers in other places within one process do even sort sequentially. The program in all processes are parallel.



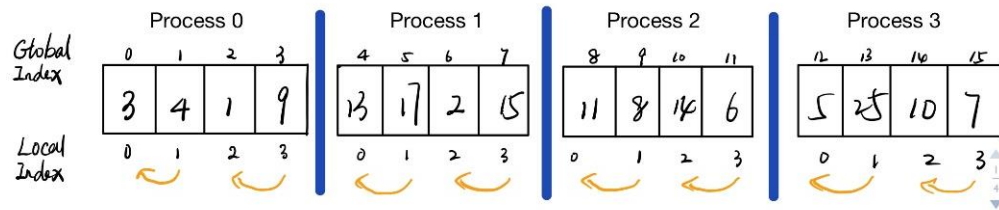
- **Odd sort, the first size – 1 processes have odd numbers, rank is odd.** The odd rank processes send head number to the previous processes and compare with tail number of the previous process. Numbers in other places within one process do odd sort sequentially. The program in all processes are parallel.



- **Odd sort, the first size – 1 processes have even numbers, rank is even and not equal to size-1.** The even rank processes send back a larger numbers to the next process. Numbers in other places within one process do odd sort sequentially. The program in all processes is parallel.



- **Odd sort, the first size – 1 processes have even numbers.** In this situation, we only need to do the sequential odd sort in each process without communicating between processes.



c) Gather sorted numbers to form one array by using MPI_Gatherv.

The code:

```
MPI_Gatherv(revSeq, sendCounts[rank], MPI_INT, sequence, sendCounts, displs, MPI_INT, 0, MPI_COMM_WORLD);
```

3.2 Pseudo-code

```
1. MPI_Scatterv(scatter N numbers to m processes);
2.
3. for (int i = 0; i < N; i++) {
4.     if (even-sort) {
5.         /* even sort in each process */
6.         evenSort(&arrayInProc, numberOfData);
7.
8.         /* even sort (change numbers) between proecess (head and tail) */
9.         if (/* the number of data in one process is even */) {
10.            if (rank != 0) {
11.                /* send head number to the previous process */
12.                /* receive the number from the previous process */
13.            }
14.            if (rank != m - 1) {
15.                /* receive the number from the next process */
16.                if (receiveNum < tailNum) {
17.                    int a = receiveNum;
18.                    receiveNum = tailNum;
19.                    tailNum = a;
20.                }
21.                /* send receiveNum to the next process */
22.            }
23.        }
24.        if (/* the number of data in one process is odd */) {
25.            if ((rank % 2 == 0) and (rank != 0)) {
26.                /* send head number to the previous process */
27.                /* receive the number from the previous process */
28.            }
29.            if ((rank % 2 == 1) and (rank != size - 1)) {
30.                /* receive the number from the next process */
31.                if (receiveNum < tailNum) {
```

```

32.         int a = receiveNum;
33.         receiveNum = tailNum;
34.         tailNum = a;
35.     }
36.     /* send receiveNum to the next process */
37. }
38. }
39. }
40. if (odd-sort) {
41.     /* odd sort in each process */
42.     oddSort(&arrayInProc, numberOfData);
43.     if (/* the number of data in one process is odd */) {
44.         if (rank % 2 == 1) {
45.             /* send head number to the previous process */
46.             /* receive the number from the previous process */
47.         }
48.         if ((rank % 2 == 0) and (rank != size - 1)) {
49.             /* receive the number from the next process */
50.             if (receiveNum < tailNum) {
51.                 int a = receiveNum;
52.                 receiveNum = tailNum;
53.                 tailNum = a;
54.             }
55.             /* send receiveNum to the next process */
56.         }
57.     }
58. }
59. }
60.
61. MPI_Gather(gather numbers from m processes);

```

4 Experiment & Analysis

4.1 Introduction

At the beginning, I calculate the time compexity to have an overall view about the amount of time it takes to run the algorithm. Then, I test the program by choosing different amount of processes and numbers and get the time it cost. After that, I can respectively calculate the speedup, efficiency, cost and discuss the computer scalbity. Besides, I compare odd-even transposition sort and odd-even mergr sort. In the end, I discuss the results and give some analysis about my algorithm analysis.

4.2 Experiments and Results

- Time complexity

At first glance of parallelizing the bubble sort algorithm it seems that the performance will increase a factor of p (the number of processors). However, careful analysis of the complexity reveals that it is actually much more than the stated value. Below is the analysis of the time complexity for the odd-even transposition sorting algorithm:

The performance of the sequential bubble sort algorithm is:

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2), \text{ where } a = \frac{1}{2}$$

The performance of the odd-even transposition algorithm is:

$$\sum_{i=1}^{\frac{n}{p}} i = 1 + 2 + 3 \dots + \frac{n}{p} = \frac{1}{2} \times \frac{n}{p} \times \frac{n}{p-1} = \frac{n^2}{2p^2} - \frac{n}{2p} = O(bn^2), \text{ where } b = \frac{1}{2}p^2$$

This means that theoretically speaking the time will reduce by $\frac{1}{p^2}$.

- Experiments

In order to test whether the performance of the algorithm is good, we need to calculate the speedup, efficiency and cost of the algorithm. So, I choose 1, 2, 4, 8, 12 processes and 10, 100, 1000, 10000 and 100000 numbers to test my program. And I use MPI_Wtime() to calculate the executing time which doesn't include the time of generating random numbers and output time. The time unit is millisecond (ms). Since the limitation of my laptop, the maximum test number of the data is 200000.

- Results

	10	100	1000	10000	100000
1	0.0405	0.0785	3.0489	253.5146	27781.0277
2	0.5294	0.6229	2.6798	158.9198	18532.8959
4	1.1298	2.2216	3.9116	119.7062	16241.5198
8	2.8514	64.7636	1377.2162	20280.4129	227199.1348
12	2.2993	188.4519	3136.7784	38600.4341	364412.3291

Table 1: odd-even transposition sort

	10	100	1000	10000	100000
1	0.0375	0.1028	2.467	12.5478	120.6103
2	0.9071	0.6178	1.7173	5.6362	59.3424
4	1.1951	1.2488	2.2037	6.8454	43.0194
8	3.1646	20.3869	3.1714	7.3484	46.2006
12	4.6335	12.9271	19.5080	22.7395	57.3823

Table 2: odd-even merge sort

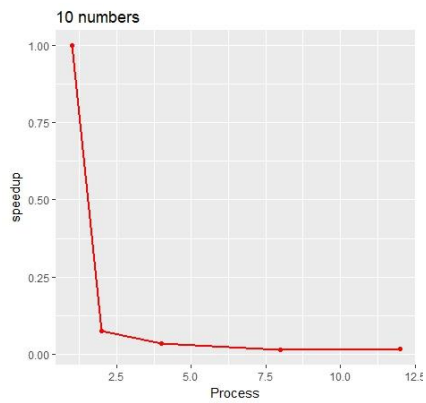


Figure speedup 1

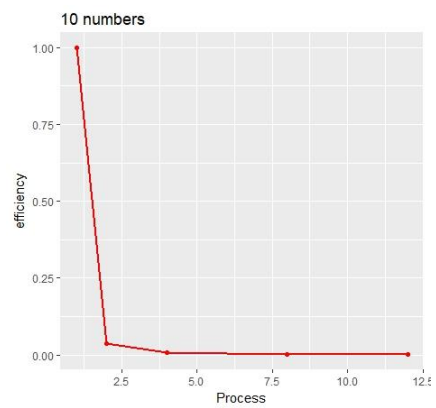


Figure efficiency 1

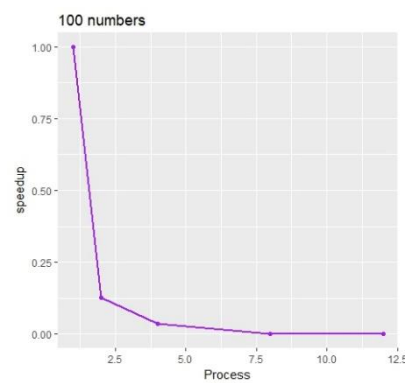


Figure speedup 2

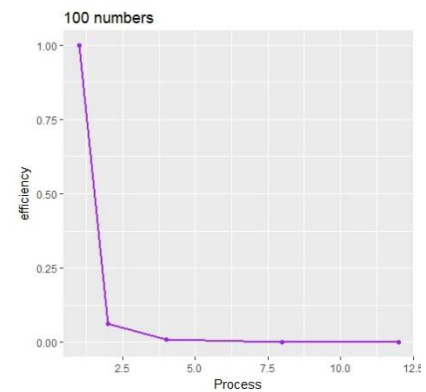


Figure efficiency 2

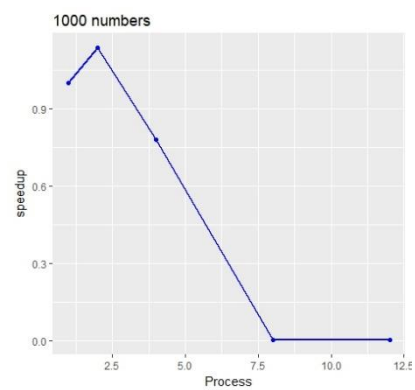


Figure speedup 3

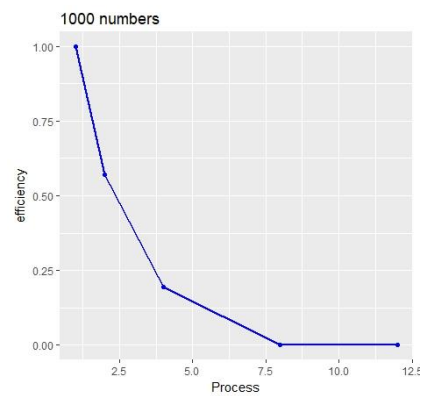


Figure efficiency 3

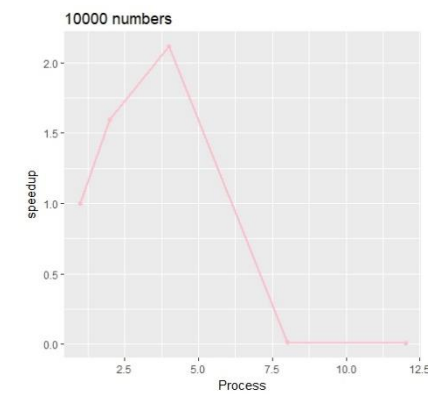


Figure speedup 4

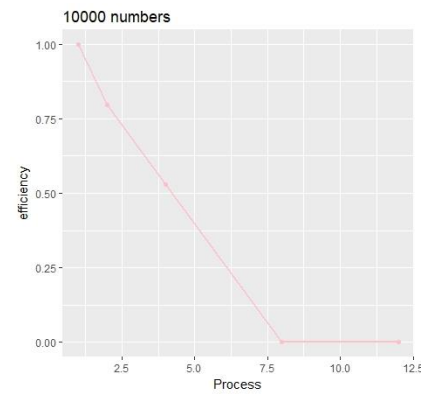


Figure efficiency 4

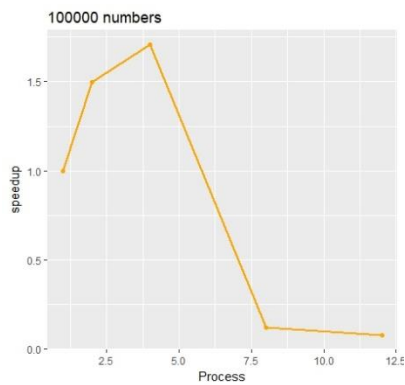


Figure speedup 5

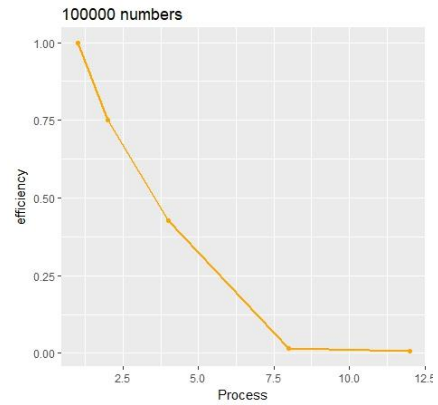


Figure efficiency 5

4.3 Performance analysis

- Explain the results of speed up and efficiency and analyze causes of those results

- Total time

The table 1 shows the total execution time for the odd-even transposition sorting algorithm. It can easily be seen that parallel algorithm is by far faster than than sequential bubble sort algorithm when the number of data is large.

- Speed up

$$\text{SpeedUp} = \frac{\text{Execution time using one processor}}{\text{Execution time using } p \text{ processpr}}$$

When the length of array is too short (10 and 100), as the number of processes increases, speedup decreases drastically. When the length of array is larger than 100 (1000, 10000, 100000), as the number of processes increases, speedup first increaes and then decreases. With small amount of data, the program takes more time to communicate between processors, so the speedup always goes down. With large amount of data, since the program can execute simutaneously, the speed up increases. However, if the amount of data is too large, the time complexity of program also become large. So, the speedup decreases again.

- Efficiency

$$\text{Efficiency} = \frac{\text{Execution time using } p \text{ processor}}{\text{Total number of processor}}$$

From the figures, we can know that as the number of processor increases, the efficiency decreases. That means, as the number of processes goes up, they take more time to wait other. However, larger of data can help to raise the efficiency. All in all, the efficiency of my algorithm is not enough and it needs to be optimized.

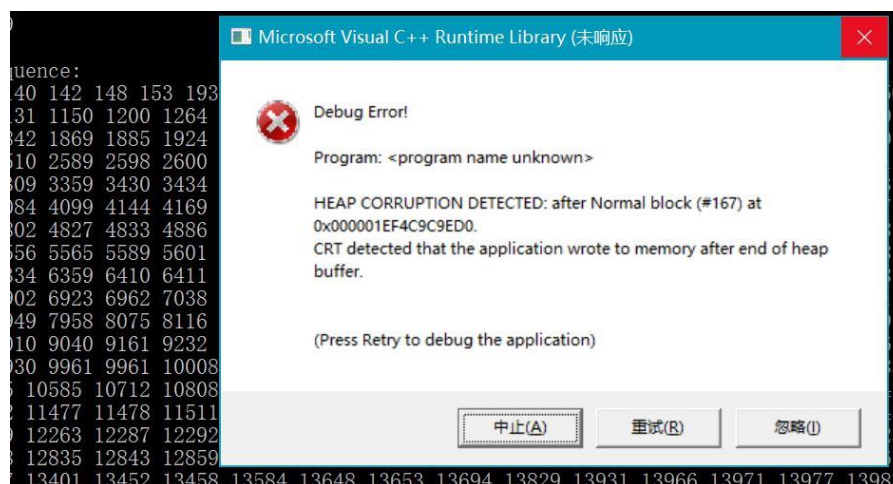
- Compare odd-even transposition sort with odd-even merge sort

As we can see the results from Table 1 and table 2, the odd-even merge sort does much better performance than odd-even transposition sort. The reason is that the time complexity of odd-even merge sort is less than odd-even transposition sort. Besides, odd-even transposition sort takes to initially sort its elements locally in each processor using sequential bubble sort which has a performance of $O(n^2)$. The odd-even transposition sorting algorithm can be improved by adapting a faster sequential sorting algorithm to sort the elements locally for each processor in the order of $O(n \log n)$ (e.g. sequential merge sort or quick sort).

5 Experience

5.1 Main implementation issues and solutions. (What were the most difficult parts of your program to implement?)

- Heap corruption



In my case, the error happened with the case of delete[] part. Firstly, I delete the delete[] NAME, the program can run without error for several times. However, it will end prematurely and may have crashed. exit code 0xc0000374 sometimes. By searching on the Internet, I found the reason. The heap corruption happened when releasing the heap memory. When it exceeds the bound of the address, it changes values on other addresses which do not belong to itself. If these addresses are vital regions in the system, the program will crashed. If these addresses are not important, the error cannot be detected immediately. Then the results will be normal. So when I distribute numbers to each process, the capacity of the last process is less than the number of distributed data since I forgot to give capacity to this process for the reminder. I solved it by add the moerate capacity to the process.

For me, heap corruption is one of the most complex problems in C++. I took so long time to figure out what the problem is, why I had this problem and how to solve it. And

this process helped me to have a profound comprehension about heap corruption and dynamic array.

- Figure out how MPI does the parallel behavior

Although Professor Yeh explained message passing clearly in the class, it's still an abstract concept idea for me. When I set about to write code I found I was unfamiliar with message passing. I have no ideas that how these functions realize the parallel behavior. Therefore, I searched for many tutorial manuals to study MPI by myself. Although starting learning MPI was kind of difficult, I was pleasant that I grew up.

5.2 Problems remain to solve

- while loop

I once tried to use `while` loop to control the program. However, it's hard to find the break point to stop all process. And I haven't found an adaptable method to realize it. But we can use `for` loop because the most times of completing the task is the number of processes, it will not spend too much time on finishing it.

5.3 Reflection

5.3.1 What went well and what I have learned

The algorithm of odd-even transposition sort is easy to understand and to realize. However, since I didn't listen to Professor's explanation for the homework, I mixed up odd-even merge sort and odd-even transposition sort. So, I used a wrong algorithm at the beginning. After discussing in the WeChat group, I learned it and rewrote my code. Through coding twice, I have a more profound understanding about MPI and a proficient application of C++.

5.3.2 What you would do differently next time

Firstly, I'll figure out and understand entirely the problem I need to solve. I should learn to ask questions rather than learned just by myself. If I have problems in coding, I should communicate with classmates or teachers. With different perspective, solving problems is faster and I can learn more from others. Programming need patience and carefulness, so I should start the homework early and leave more time to think and optimize.

5.3.3 What the instructors and Tas might have done differently to promote learning

🌈 Still the comprehension about the problem. I hope TA can explain the problems in tutorial and communicate with Professor Yeh more. More communication, more efficiency. I still hope we can use the same server to execute our program, since the limitation of laptop causes trouble of testing our programs.

- ✚ I hope TA can teach more practical knowledge and give more examples of code in tutorials. Examples and practical exercise can help us to learn and understand the textbook content.

6 Acknowledgements

- 6.1.1 Thank to Profess Yeh. When I used wrong algorithm to do the homework and encounter problems in coding, he gave me much advice and talked to me with patience. And with his encouragement, I have more confidence in this course and raise my enthusiasm.
- 6.1.2 Thank to TA. The explanation for assignment gave me the chance to reconsider my code and fix the mistakes in time.
- 6.1.3 Thank to my classmates, Tu Yuxiao and Qin Yuze. They helped me to solve some bugs and the discussion with them help to understand some pinpoints.

7 Reference

- <http://mpitutorial.com/tutorials/mpi-hello-world/>
- https://www.researchgate.net/figure/Parallel-sort-algorithm-sorting-12-elements-using-4-processors_fig2_221133294
- https://www.researchgate.net/figure/Efficiency-of-the-odd-even-transposition-sort-algorithm_fig5_221133294

8 Source code

```
1. #include "pch.h"
2. #include "mpi.h"
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <iostream>
6. using namespace std;
7.
8. /* Function Prototype */
9. void oddSort(int *A, int n);
10. void evenSort(int *A, int n);
11.
12. int main(int argc, char **argv) {
13.     int const N = 200;           // the number of data
14.     int sequence[N];             // elements need to be scattered
15.     int rank, size;              // id of rank and the number of processes
16.     int reminder;                // reminder if it is not divisible
17.     int revNum;                  // receive number
```

```

18.     double Start, End;                // timing
19.     int *sendCounts = NULL;           // allocate memory for array storing number of elements
20.     int *displs = NULL;               // Claculate corresponding displacement
21.     int *revSeq = NULL;               // receive buffer
22.
23.     /* Initialize */
24.     MPI_Init(&argc, &argv);
25.     MPI_Comm_size(MPI_COMM_WORLD, &size);
26.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
27.
28.     /* Generate random integers */
29.     if (rank == 0) {
30.         cout << "\nNAME: Ding Xinyi" << endl;
31.         cout << "ID: 116010035" << endl;
32.         cout << "\nStart! The sequence: " << endl;
33.         for (int i = 0; i < N; i++) {
34.             sequence[i] = rand();
35.             cout << sequence[i] << " ";           // sequence initialize only in rank 0
36.         }
37.     }
38.
39.     Start = MPI_Wtime();                // start timing
40.
41.     /* Initialize array */
42.     sendCounts = new int[size];         // allocate memory for array storing number of elements
43.     reminder = N % size;                // reminder
44.
45.     /* Calculate number of elements need to be scattered in each process */
46.     for (int i = 0; i < size; i++) {
47.         sendCounts[i] = int(N / size);
48.     }
49.     // Number of elements in the last process
50.     sendCounts[size - 1] = sendCounts[size - 1] + reminder;
51.
52.     displs = new int[size];            // Claculate corresponding displacement
53.     for (int i = 0; i < size; i++) {
54.         displs[i] = 0;
55.         for (int j = 0; j < i; j++) {
56.             displs[i] = displs[i] + sendCounts[j];
57.         }
58.     }

```

```

59.     revSeq = new int[sendCounts[rank]];           // allocate the receive buffer
60.
61.     /* Scatterv Operation */
62.     MPI_Scatterv(sequence, sendCounts, displs, MPI_INT, revSeq, sendCounts[rank], MPI_INT,
        0, MPI_COMM_WORLD);
63.     /* Odd-even sort between ranks */
64.     for (int i = 0; i < N; i++) {
65.         // Even sort
66.         if (i % 2 == 0) {
67.             evenSort(revSeq, sendCounts[rank] - 1);
68.             // even number of data in first process
69.             if (sendCounts[0] % 2 == 0) {
70.                 // send and receive
71.                 if (rank != 0) {
72.                     MPI_Send(&revSeq[0], 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
73.                     MPI_Recv(&revSeq[0], 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD, MPI_STATU
        SES_IGNORE); // put recv after send
74.                 }
75.                 // receive, compare and send
76.                 if (rank != size - 1) {
77.                     MPI_Recv(&revNum, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUSES
        _IGNORE);
78.                     if (revSeq[sendCounts[rank] - 1] > revNum) {
79.                         int a = revNum;
80.                         revNum = revSeq[sendCounts[rank] - 1];
81.                         revSeq[sendCounts[rank] - 1] = a;
82.                     }
83.                     MPI_Send(&revNum, 1, MPI_INT, rank + 1, 1, MPI_COMM_WORLD);
84.                 }
85.             }
86.             //odd number of data in first process
87.             else {
88.                 // send and receive
89.                 if ((rank % 2 == 0) and (rank != 0)) {
90.                     MPI_Send(&revSeq[0], 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
91.                     MPI_Recv(&revSeq[0], 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD, MPI_STATU
        SES_IGNORE);
92.                 }
93.                 // receive compare and send
94.                 if ((rank % 2 == 1) and (rank != size - 1)) {
95.                     MPI_Recv(&revNum, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUSES
        _IGNORE);
96.                     if (revSeq[sendCounts[rank] - 1] > revNum) {
97.                         int a = revNum;

```



```

98.             revNum = revSeq[sendCounts[rank] - 1];
99.             revSeq[sendCounts[rank] - 1] = a;
100.        }
101.        MPI_Send(&revNum, 1, MPI_INT, rank + 1, 1, MPI_COMM_WORLD);
102.    }
103.    }
104.    }
105.    // Odd sort
106.    else {
107.        oddSort(revSeq, sendCounts[rank] - 1);
108.        // odd number of data in first process
109.        if (sendCounts[0] % 2 == 1) {
110.            // send and receive
111.            if (rank % 2 == 1) {
112.                MPI_Send(&revSeq[0], 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
113.                MPI_Recv(&revSeq[0], 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD, MPI_STA
TUSES_IGNORE);
114.            }
115.            // receive, compare and send
116.            if ((rank % 2 == 0) and (rank != size - 1)) {
117.                MPI_Recv(&revNum, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS
ES_IGNORE);
118.                if (revSeq[sendCounts[rank] - 1] > revNum) {
119.                    int a = revNum;
120.                    revNum = revSeq[sendCounts[rank] - 1];
121.                    revSeq[sendCounts[rank] - 1] = a;
122.                }
123.                MPI_Send(&revNum, 1, MPI_INT, rank + 1, 1, MPI_COMM_WORLD);
124.            }
125.        }
126.    }
127.    }
128.
129.    MPI_Gatherv(revSeq, sendCounts[rank], MPI_INT, sequence, sendCounts, displs, MPI_INT,
0, MPI_COMM_WORLD);
130.
131.    End = MPI_Wtime();           // End Time
132.
133.    if (rank == 0) {
134.        printf("\n\nDone! The sorted sequence: \n");
135.        for (int i = 0; i < N; i++) {
136.            cout << sequence[i] << " ";
137.        }
138.    }

```

```
139.
140.     MPI_Barrier(MPI_COMM_WORLD);
141.     double time = End - Start;
142.     if (rank == 0) {
143.         printf("\n\nThe whole process consumes time: %f ms\n", time * 1000);
144.     }
145.
146.     delete[] sendCounts;
147.     delete[] displs;
148.     delete[] revSeq;
149.     MPI_Finalize();
150.     return 0;
151. }
152.
153. void oddSort(int *A, int n) {
154.     for (int i = 1; i <= n; i += 2) {
155.         if (A[i] < A[i - 1]) {
156.             int a = A[i];
157.             A[i] = A[i - 1];
158.             A[i - 1] = a;
159.         }
160.     }
161. }
162.
163. void evenSort(int *A, int n) {
164.     for (int i = 2; i <= n; i += 2) {
165.         if (A[i] < A[i - 1]) {
166.             int a = A[i];
167.             A[i] = A[i - 1];
168.             A[i - 1] = a;
169.         }
170.     }
171. }
```