香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Parallel Odd-Even Transposition Sort

Assignment I

CSC4005: Parallel Programming

Name: *Derong Jin*
Student ID: *120090562*

Date: October 12, 2022

1 Introduction

The topic of assignment 1 is to implement a parallel odd-even transposition sort using MPI. To better understand the characteristics of the parallel Odd-Even Transposition Sort algorithm, I have also implemented a serial odd-even sort algorithm in this assignment for comparison. In this report, I will elaborate on the implementation of each part and analyze their results in details.

1.1 Odd-Even Transposition Sort

An odd-even transposition sort assume there an array initially with n elements unsorted, the algorithm first checks every element with odd index, compares it with its posterior element. If an element with odd index is greater than its successor, the algorithm swaps their order. A process that checks and swaps every odd-index element is called an odd comparison. Similarly, there is even comparison, which checks on even-index elements. It is ensured that, if odd comparisons and even comparisons are conducted repeatedly for at most n times, the array will become sorted then.

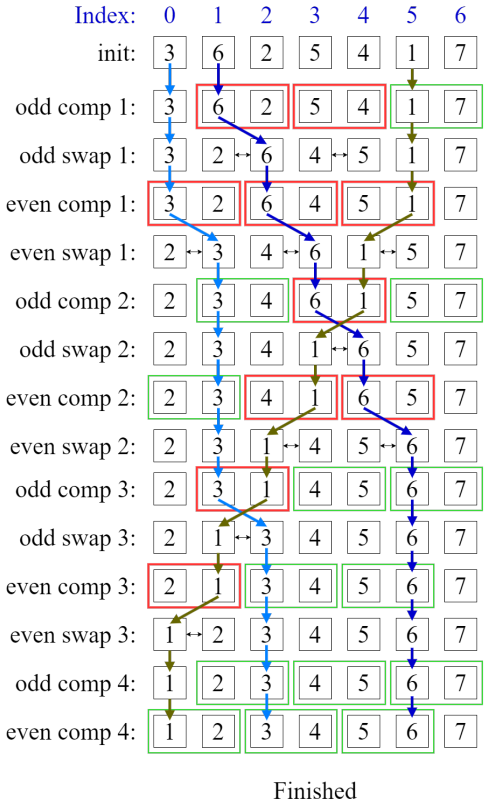


Figure 1: The diagram shows an example of sequential Odd-Even Transposition Sort on an array of length 7. The algorithm terminates in 4 rounds of comparison (in the last round, no element is moved). The arrows show traces of some elements.

1.2 Basic Idea About Parallelizing

It can be observed that, in one round of comparison, each number will be comapred at most once. This induces that, in one round of comparison, the order of each comparison has not effect to the result, and thus, this algorithm can be processed concurrently. And following diagram shows a possible way in parallelizing the algorithm (detailed discussion in "Method" section):

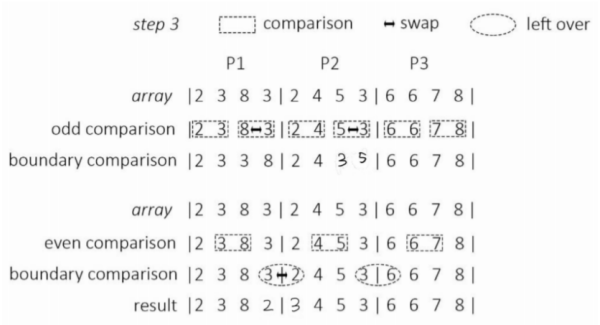


Figure 2: An example of parallel odd-even sort. **NOTE** that in my Implementaion, the array starts from index 0, which is different to the figure

According to the assignment guideline, the procedure of parallel odd-even transposition sort is declared as following procedure:

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.
5. Repeat 1-4 until the numbers are sorted.

1.3 Message Passing Interface (MPI)

All parallel programs in this assignment are based on MPICH, which provides efficient and easy-to-use MPIs. These interfaces are used to communicate and transfer information between processors.

2 Method

2.1 Sequential Implementation

Implementing serial odevensort is very simple, just repeatedly compare and swap over odd and even positions until they are ordered.

```
template<typename RandomAccessIt>
void inplace_odd_even_sort_serial(RandomAccessIt first,
  RandomAccessIt last)
{
  int length = std::distance(first, last);
  int sorted = (length == 0), __even = length & 1, __odd = 1 ^ (length & 1);
  while( ! sorted ) // not sorted
  {
    sorted = 1;
    // Odd sort
    for(RandomAccessIt __i = first + 1; __i + __odd != last; __i += 2)
    {
      if( *(__i) > *(__i + 1) )
      {
        std::iter_swap(__i, __i + 1);
        sorted = 0;
      }
    }

    // Even sort, the same process with different start point
    for(RandomAccessIt __i = first; __i + __even != last; __i += 2)
    {
      if( *(__i) > *(__i + 1) )
      {
        std::iter_swap(__i, __i + 1);
        sorted = 0;
      }
    }
  }
}
```

The program does 3 things:

1. Read the array stored in given file, say, the path is `<path-to-file>`.
2. Perform Sequential Odd-Even Sort on the array and times the algorithm.
3. Print the array out to `<path-to-file>.sequential.out`.

The program is writen in file `odd_even_serial.cpp`

The output of sequential program:

```
[120090562@node21 src]$ ./s_sort data.in
Name: Derong Jin
Student ID: 120090562
Assignment 1, Odd-Even Transposition Sort, sequenential version
runTime is 0.000002 sec
188773419 309591154 685407625 687162515 929843300 1058342303 1200501950 1550269022 1723918352 1928129635
[120090562@node21 src]$
```

2.2 Parallel Implementation

From introduction section, the parallel scheme of Odd-Even Sort is clearly derived, and a buffer is added in each processor. The computational steps should be as the following figure.

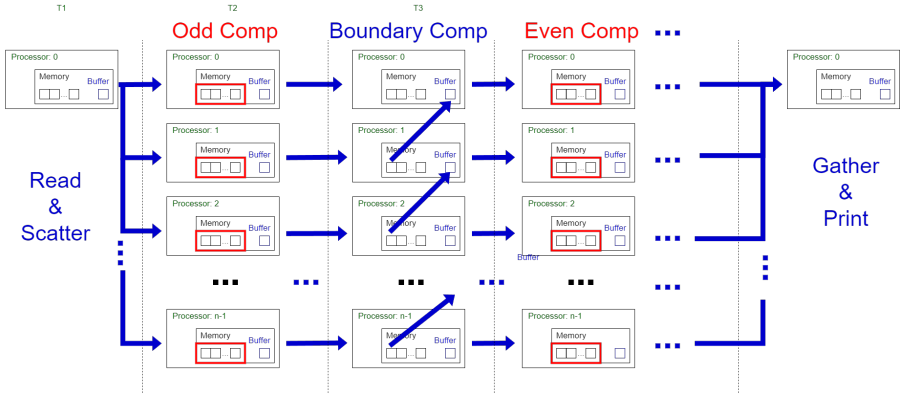


Figure 3: The diagram shows the algorithm of Parallel Odd-Even Sort

To make it more clear with MPI, the flowchart of the MPI design should be as follow:

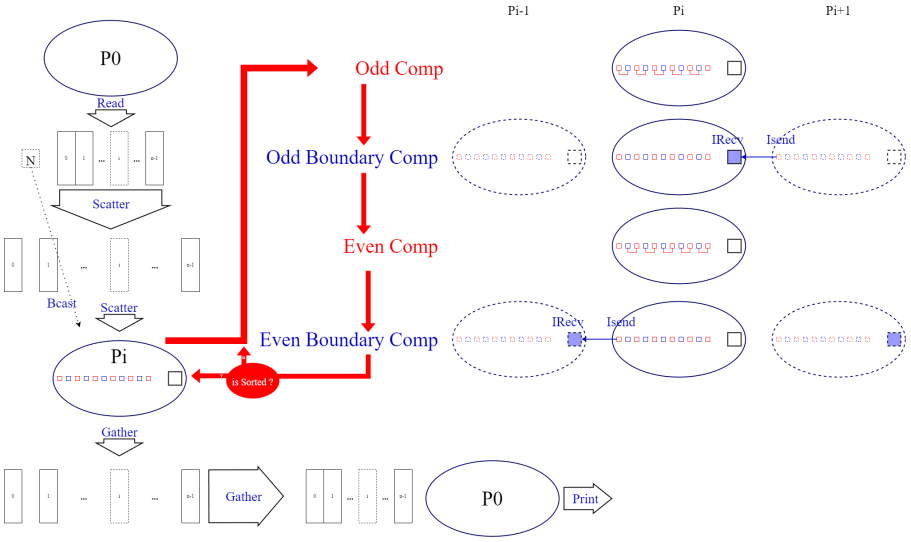


Figure 4: The flowchart above shows the flowchart of the how the MPI calls are used in the program

To achieve this implementaion with message passing interface provided by MPICH, we shaw use several interfaces:

Name	Description
`MPI_Init`	initialize MPI program
`MPI_Comm_rank`	getter of the rank of current processor
`MPI_Comm_size`	getter of the number of processors in the group
`MPI_Bcast`	broadcast the message to the whole communication group
`MPI_Scatter`	chunk data into pieces and evenly spread to processors in the group
`MPI_Gather`	combine the data from each processors
`MPI_Finalize`	Fialize MPI program
`MPI_Isend`	non-blocking send procedure
`MPI_Isend`	non-blocking send procedure
`MPI_Wait`	wait until send/recv procedure finshed

In order to avoid deadlock, the send & recv procedure in communication between processors are non-blocking, waiting operation is after message are sent/to be recieved.

The program is writen in file `odd_even_serial.cpp`

The output of parallel program:

```
[120090562@node21 src]$ mpiexec -np 4 ./p_sort data.in
Name: Derong Jin
Student ID: 120090562
Assignment 1, Odd-Even Transposition Sort, parallel version, MPI implementation
runTime is 0.661157 sec
Input Size: 10
Proc Num: 4
188773419 309591154 685407625 687162515 929843300 1058342303 1200501950 1550269022 1723918352 1928129635
[120090562@node21 src]$
```

2.3 Data Generator

A data generator is implemented and is to generate arrays randomly.

2.4 Theoretical Analysis on Parallelizing

From the above description, it is clear that in the parallel version of the odd-even sort algorithm, the arrays should be evenly divided among the processors for sorting. Since each comparison in the odd-even sort algorithm is a comparison of adjacent numbers, there will exist only two possibilities for each comparison in the parallel version of odd-even sort: the first is a comparison between two adjacent numbers in the same processor; the second is a comparison between two numbers that are adjacent to each other in the original array at the same time on the array boundary of both processors. According to previous observations, changing the order of comparison in the comparison process will not affect the sequential odd-even sort algorithm, so performing the comparison at the boundary of the separated arrays after the comparison

of the arrays within the processors will not affect the efficiency of the parallel odd-even sort algorithm either, i.e., if the overall odd comparison is split into *partial odd comparison* and *boundary comparison* does not increase the number of comparisons in the sorting process.

For now it is analyzed that the parallel program has the same number of comparisons as the serial version of odd-even sort algorithm, while partitioning the comparison operations in the algorithm with multiple processors: while the sequential version makes one comparison, the parallel algorithm can theoretically perform p comparisons (p is the number of processors), which is the theoretical basis for improving the odd-even sort algorithm via parallel methods. If we consider only the time overhead caused by the comparisons and assume that all comparison operations are performed in constant time, the formal parallel version of the odd-even sort algorithm has a time complexity of

$$T_p(N) = O(\frac{N^2}{p} + (C + \delta p)N) = O(\frac{N^2}{p} + Np) \tag{1}$$

where N, p and C stand for the length of the array, the number of processors, and an upper-bound of the ratio of the boundary comparison time to partial odd/even comparison time, and δ is a constent that represents the sequential part passing message between processors. In common cases (p can be treated as constant and N is relatively large), equation (1) is simplified as $T(n) = O(N^2/p) = O(N^2)$, and we shaw notice that this simplified term does not holds when N is small or p is large.

3 Result

The theoretical efficiency of parallel programs depends on the number of processors and the scale of the problem (i.e., the length of the array). This section will focus on the effect of these two factors on the MPI's version of Odd-Even Sort performance. Finally, this section also compares the parallel program implemented by MPI with a normal sequential implementation

To find out the effect of these two factors on the time cost by MPI program, I use the data generator to generate data of following size. And for each data scale, I have measured the performance (time) when using different number of processors (1-20). Full experiment data can be found in **Appendx B**.

Scale	Tiny				Small		Medium		Large	
Array Length	10	30	100	300	1000	3000	10000	30000	100000	300000

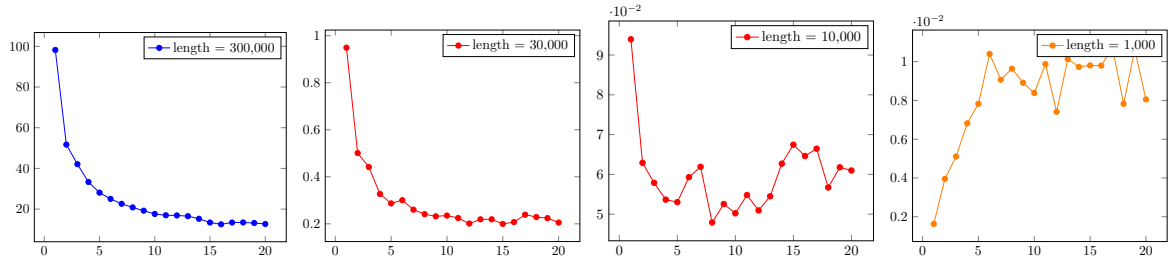
3.1 Performance Under Different Number of Cores

When analyzing the number of processors versus the performance of parallel algorithms, we need to fix the size of the test data first. In the following I have chosen four cases with data sizes of 1000, 10000, 30000 and 300000 to analyze the efficiency of parallel algorithms versus the number of processors.

The following table shows the respecting data:

time (s) / np / len	1	2	3	4	5	6	7	8	9	10
1000	0.00163	0.00395	0.00511	0.00682	0.00783	0.01040	0.00907	0.00964	0.00891	0.00838
10000	0.09395	0.06291	0.05787	0.05366	0.05301	0.05928	0.06191	0.04791	0.05255	0.05022
30000	0.94900	0.50095	0.44172	0.32715	0.28674	0.30035	0.26011	0.24099	0.23185	0.23516
300000	98.22718	51.70174	42.07319	33.33142	28.09684	25.02844	22.60227	20.83337	19.22777	17.61633
time (s) / np / len	11	12	13	14	15	16	17	18	19	20
1000	0.00987	0.00742	0.01013	0.00973	0.00980	0.00980	0.01072	0.00782	0.01062	0.00805
10000	0.05480	0.05094	0.05449	0.06270	0.06746	0.06462	0.06646	0.05671	0.06178	0.06097
30000	0.22461	0.20132	0.21898	0.21910	0.19960	0.20742	0.23885	0.22890	0.22417	0.20547
300000	16.95813	16.88801	16.53838	15.24188	13.41075	12.52612	13.40751	13.44086	13.17393	12.67915

Table 1: This table exhibits the time spent by MPI program to finish the sort. The same row represents the execution is on the same data scale, while the same column indicates the same number of processors used



(a) Large Scale (b) Medium Scale 1 (c) Medium Scale 2 (d) Small Scale

Figure 5: These figures show the change in runtime with respect to the growth of number of processors. It can be observed that, parallelization has its significance when the scale of experiment is large. However the efficiency might reduce with the increasing number of processors, when data scale is small enough

Figure 3.1 illustrates the four pieces of data as line graphs. It is very clear to see that when the problem size is large, the additional processors used for computing have a significant improvement on the overall performance. However, when the problem size is small, the cost of information transfer greatly depletes the advantage of multiple processors because the time it takes for the processors to transfer information is much greater than the time it takes to exchange elements within the processors, and this transfer time grows with the increase in processors.

It is interesting that Fig 5(c) shows a surg when the processor number becomes 6. In my oppinion, it might be related to the pattern of organizarion. For in this implementation, processors only communicate with those whose have adjacant `rank` value. However, the communicate pattern might not be a structure of a list. So exceeding a number of processors might have a worse embedding in the communication network.

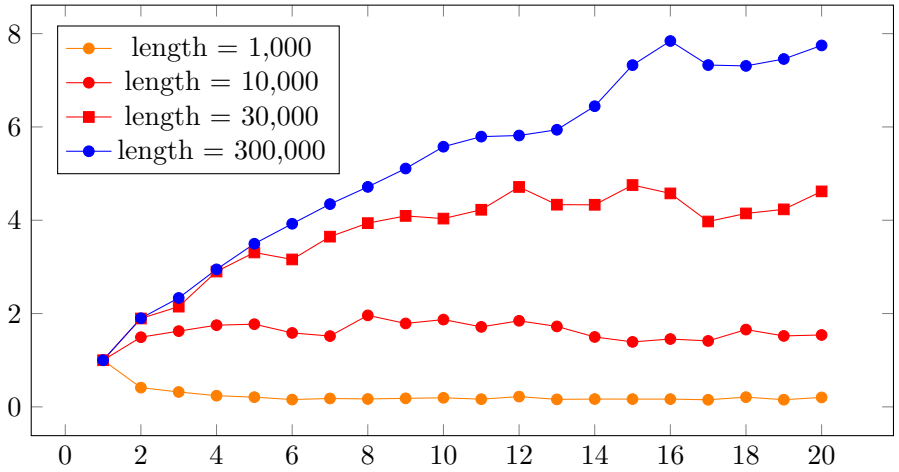


Figure 6: Speed-up rate of parallel program with respect to the number of processors in various problem size, where $speedup = \frac{T_{sequential}}{T_{parallel}}$. It can be observed that increasing number of processors increases the speedup of a parallel program in large-scale problem. While in small-scale problem, additional processors may improve the efficiency by small factor, or even worsen the performance (e.g., when length = 1000, multi-core performance is worse than that of program running on only 1 core.)

It is clear from the speedup rate that the time taken to transfer information between processors not only consumes the parallelism advantage brought by multiple processors, but sometimes when the ratio of the frequency of swapping to the frequency of internal operations is too large (i.e., in the cases of small data) it makes parallel programs inferior to ordinary ones.

In my opinion, the disadvantage of parallel computing with small data is that the serial rate is relatively too high, i.e., the parallel program no longer becomes better with more processors when the length of the interval to which each processor is assigned is less than **a certain threshold**. It also shows that when analyzing the runtime, regardless of the problem size, the performance of the parallel program will continuously degrade with more processors when the number of processors increases, and the performance of the whole parallel program will degrade with more processors when the performance degradation caused by more processors cannot compensate for the advantage of chunking the data. This can be seen on the index of efficiency ($efficiency = \frac{T_{sequential}}{N_{core} \times T_{parallel}}$).

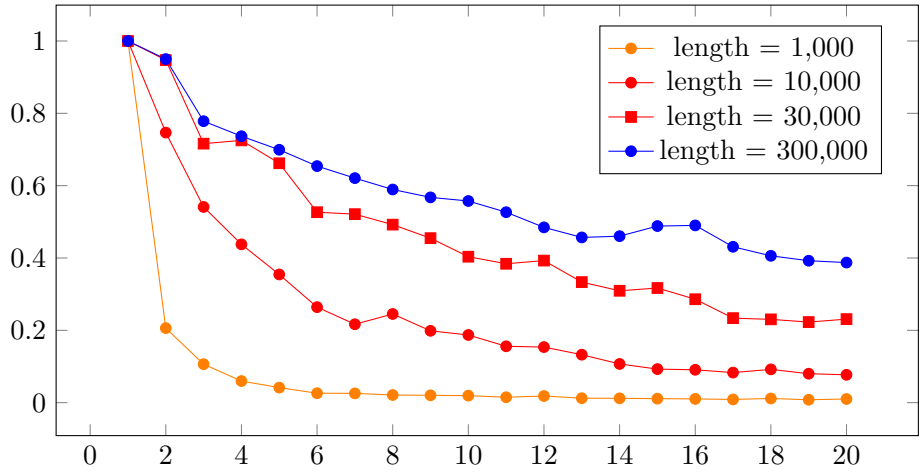


Figure 7: The figure illustrates the Efficiency of pallel program with respect to the number of processors in various problem size. It is shown that the efficiency reduces with adding new processors no matter with the problem size.

3.2 Performance Under Different Problem Scale

As the problem size increases, the time consumption of the program inevitably increases. In exploring the effect of problem size on runtime, it is necessary to test the time consumption of the program at different orders of magnitude while ensuring that the number of processors called is the same. The problem size increases roughly linearly on the logarithm, so scales of 30, 300, 3000, etc. were added to the initial range of scales of 10, 100, 1000, etc. To gain insight into how problem size affects runtime, I compared the data under four processor number choices of 1, 4, and 16

time (s) \ len / np	10	30	100	300	1000	3000	10000	30000	100000	300000
1	0.00008	0.00005	0.00008	0.00020	0.00163	0.00862	0.09395	0.94900	10.79241	98.22718
4	0.00109	0.00277	0.00405	0.00462	0.00682	0.01597	0.05366	0.32715	3.80517	33.33142
16	0.00058	0.00283	0.00522	0.00604	0.00980	0.01841	0.06462	0.20742	1.37082	12.52612

Table 2: This table exhibits the time spent by MPI program to finish the sort. The same row means the execution is using the same number of processors, while the same column indicates the same data is recieved

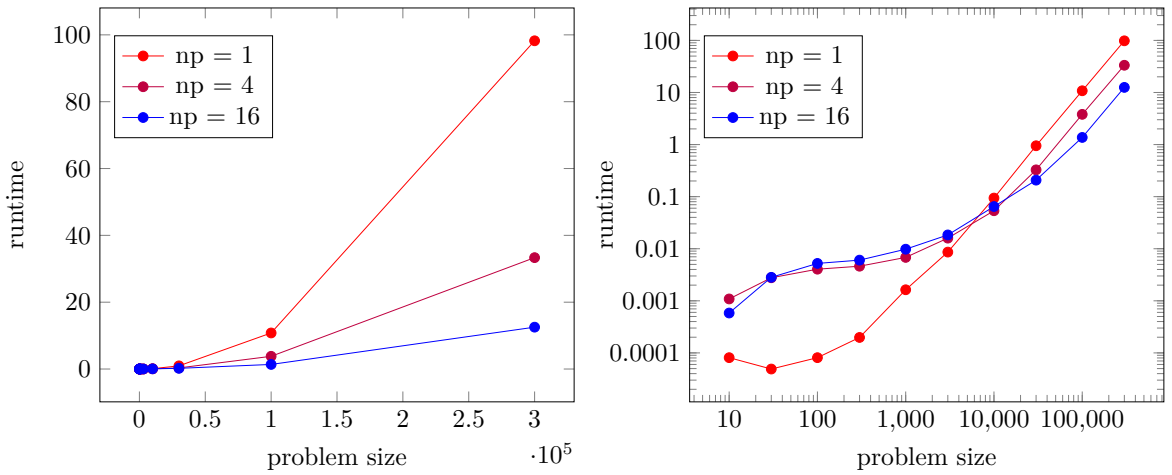


Figure 8: Figure 8(a) and 8(b) represent the **same** data. 8(a) plots the original data in normal scale, while 8(b) plots the data in logarithm scale. The first figure clearly shows that the runtime increase sharpely when problem size keeps increasing, while 5(b) shows that the growing pattern of sequential program is quiet different with that of parallel programs when program size is small, parallel program is slower and intends to grow faster in runtime.

The observations of 3.2 support the equation (1) a lot. When problem size is large, and the number of processors is relatively small, the equation (1) indeed can be simplified as $O(N^2)$, so we found the runtime increase faster as problem size increases in Fig8(a), and their growth seems to be at the same scale in logarithm space. What is more, when problem size N is small but not too small (e.g, $N \geq 100$), sequential programming grows as a normal quarduatic function, whle parallel programs seems to maintain an unchanged status. According to equation (1), when N is small or p is large, Np should be the domaint term of the expression, so it grows faster at the beginng, and slowly change into a slower growth, in the changing process, the advantage of chunking the array is approximately the same as the disadvantage of passing message in between. And the phenomina are well explained.

3.3 MPI v.s. Naive Sequential

In this section, we are going to compare the relation between MPI Sequential (i.e., 1 core) and Naive Sequential implementaion. Because further comparison loses it meaning.

	10	30	100	300	1000	3000	10000	30000	100000	300000
1	0.00008	0.00005	0.00008	0.00020	0.00163	0.00862	0.09395	0.94900	10.79241	98.22718
seq	0.00000	0.00002	0.00016	0.00120	0.00766	0.04255	0.49137	4.64593	51.61703	467.06487

And it is surprising to see that MPI sequential program is running faster than regular sequential Implementaion. The reason behind might be because of the different implementaion in compilation and calling procedures.

4 Conclusion

In this assignment, a MPI-based parallel odd-even transposition sort is implemented. From the analysis on the performance under different configuration, it is shown that parallel computing is suitable in solving large-scale problems. For other cases, it is not ensured that parallel program is better. These conclusion is based on the observation of several limitations in parallel computing, e.g., the communication between workers are much less efficient than the operations in memory. The decision of parallelization is a trade off between the slow parallel communication and the large problem scale in sequential ones. Finally, theoretical analysis is useful in performance of parallel program, however not as solid as the analysis from experimental data.

As the first assignment of parallel computing, it guides me a lot in understanding the world of parallel programs.

Appendix A. How to Compile & Run the Program

Compilation

There are two main ways to compile my programs: **1. make**

1. `$cd` into `src` folder
2. `$make`

And compilation will be done.

```
[120090562@node21 src]$ make
mpic++ odd_even_parallel.cpp -o p_sort -std=c++11 -w
g++ odd_even_serial.cpp -o s_sort -std=c++11 -w
g++ data.cpp -o data
g++ check_sorted.cpp -o checker
[120090562@node21 src]$
```

2. using directly compiling commands There are 4 files:

1. `odd_even_parallel.cpp`:
   - `mpic++ odd_even_parallel.cpp -o p_sort -std=c++11 -w`
2. `odd_even_serial.cpp`: `g++ odd_even_serial.cpp -o s_sort -std=c++11 -w`
3. `data.cpp`: `g++ data.cpp -o data`
4. `check_sorted.cpp`: `g++ check_sorted.cpp -o checker`

Execution

1. Data `data.cpp`:
   - command: `./data N <data-file-name>`
   - N is the number of random elements, `<data-file-name>` will store the random element
   - NOTE: there are $N+1$ in the generated `<data-file-name>`, the first line is N, from the second line, each line will store a number, the number at $(i+1)^{th}$ line represents the i^{th} element in the array
2. Sequential `odd_even_serial.cpp`:
   - command: `./s_sort <data-file-name>`
   - `<data-file-path>` represents the path of file where data is stored
   - NOTE: the first line in the data file should contains an integer n, and in the following n lines, the integer of the $(i+1)^{th}$ line represents the i^{th} integer in the array

- After the sort is complete, it will write n lines into `<data-file-name>.sequential.out` represents the n elements after sorting.
- If the array size is smaller or equal than 20, the program will **also** print out the sorted elementes

3. Parallel `odd_even_parallel.cpp`
   - command(optional): `salloc -N1 -n20 -t100 -p Project` (or other commands to allocate resources)
   - command: `mpirun -np N ./p_sort <data-file-path>`
   - `N` represents the number of processors allocated
   - `<data-file-path>` represents the path of file where data is stored
   - NOTE: the first line in the data file should contains an integer n, and in the following n lines, the integer of the $(i+1)^{th}$ line represents the i^{th} integer in the array
   - After the sort is complete, it will write n lines into `<data-file-name>.parallel.out` represents the n elements after sorting.
   - If the array size is smaller or equal than 20, the program will **also** print out the sorted elementes
4. Checker `check_sorted.cpp`
   - command: `./check_sorted <file-unsorted> <file-sorted>`
   - use to check the result is corret.

Sample Compile & Run

```
[120090562@node21 src]$ ls
check_sorted.cpp  data.cpp  Makefile  odd_even_parallel.cpp  odd_even_serial.cpp
[120090562@node21 src]$ make
mpic++ odd_even_parallel.cpp -o p_sort -std=c++11 -w
g++ odd_even_serial.cpp -o s_sort -std=c++11 -w
g++ data.cpp -o data
g++ check_sorted.cpp -o checker
[120090562@node21 src]$ ./data 5 data.in
[120090562@node21 src]$ ./s_sort data.in
Name: Derong Jin
Student ID: 120090562
Assignment 1, Odd-Even Transposition Sort, sequenential version
runTime is 0.000002 sec
512114634 523985219 564420326 1290318423 1368773577
[120090562@node21 src]$ ./checker data.in data.in.sequential.out
Sort result checked, no problem found
```

(a) sequential

Appendix B. Full Table of Experiment Data

time (s) len / np	10	30	100	300	1000	3000	10000	30000	100000	300000
1	0.00008	0.00005	0.00008	0.00020	0.00163	0.00862	0.09395	0.94900	10.79241	98.22718
2	0.00131	0.00143	0.00234	0.00313	0.00395	0.01816	0.06291	0.50095	5.71360	51.70174
3	0.00062	0.00257	0.00392	0.00510	0.00511	0.01506	0.05787	0.44172	4.69044	42.07319
4	0.00109	0.00277	0.00405	0.00462	0.00682	0.01597	0.05366	0.32715	3.80517	33.33142
5	0.00052	0.00280	0.00366	0.00451	0.00783	0.01671	0.05301	0.28674	3.13541	28.09684
6	0.00050	0.00300	0.00439	0.00449	0.01040	0.01743	0.05928	0.30035	2.90714	25.02844
7	0.00041	0.00458	0.00501	0.00599	0.00907	0.01959	0.06191	0.26011	2.52225	22.60227
8	0.00047	0.00195	0.00515	0.00349	0.00964	0.01992	0.04791	0.24099	2.31891	20.83337
9	0.00053	0.00326	0.00376	0.00560	0.00891	0.01795	0.05255	0.23185	2.12994	19.22777
10	0.00087	0.00431	0.00399	0.00383	0.00838	0.01602	0.05022	0.23516	2.01005	17.61633
11	0.00060	0.00458	0.00375	0.00504	0.00987	0.02158	0.05480	0.22461	1.81714	16.95813
12	0.00126	0.00507	0.00506	0.00719	0.00742	0.01791	0.05094	0.20132	1.68088	16.88801
13	0.00194	0.00424	0.00349	0.00477	0.01013	0.01817	0.05449	0.21898	1.65907	16.53838
14	0.00167	0.00465	0.00511	0.00639	0.00973	0.02244	0.06270	0.21910	1.52486	15.24188
15	0.00062	0.00284	0.00575	0.00633	0.00980	0.01864	0.06746	0.19960	1.53393	13.41075
16	0.00058	0.00283	0.00522	0.00604	0.00980	0.01841	0.06462	0.20742	1.37082	12.52612
17	0.00238	0.00352	0.00347	0.00562	0.01072	0.01996	0.06646	0.23885	1.46301	13.40751
18	0.00055	0.00505	0.00360	0.00725	0.00782	0.02282	0.05671	0.22890	1.23665	13.44086
19	0.00134	0.00285	0.00412	0.00504	0.01062	0.01985	0.06178	0.22417	1.24012	13.17393
20	0.00388	0.00539	0.00469	0.00554	0.00805	0.01936	0.06097	0.20547	1.25443	12.67915

	10	30	100	300	1000	3000	10000	30000	100000	300000
seq	0.00000	0.00002	0.00016	0.00120	0.00766	0.04255	0.49137	4.64593	51.61703	467.06487

Appendix C. Source Code

Listings

Listing 1: **Sequential Implementation of Odd-Even Transposition Sort**

```
1  /**
2   * @file odd_even_serial.cpp
3   * @author Derong Jin (120090562@link.cuhk.edu.cn)
4   * @brief Sequential implementation of Odd-Even Transportation Sort
5   *        compile:  g++ odd_even_serial.cpp -o s_sort -std=c++11 -w
6   *          usage:  ./s_sort <data-file-path>
7   * @date 2022-10-11
8   *
9   */
10 #include <cstdio>
11 #include <chrono>
12 #include <cstring>
13 #include <algorithm>
14
15
16 /**
17  * @brief Sort the elements of a sequence, using sequential
18  *        odd-even sort
19  *
20  * @param first   An iterator
21  * @param last    Another iterator
22  * @return Nothing.
23  *
24  * This is an implementation of sequential Odd-Even Sort algo.
25  * that sorts elements in [first, last) in acsending order,
26  * i.e., *(i) <= *(i + 1) holds for all i in the given range.
27  * When executing, this algo. checks if elements and their
28  * posterior neighbours are in ascending order, and do
29  * correction if they are not. The iterative checking operation
30  * is conducted twice, on every element with odd index and
31  * with even index. And the order of the check-and-swap
32  * operation of elements in each checking operation has no
33  * effect on efficiency and correctness.
34 */
35 template<typename RandomAccessIt>
36 void inplace_odd_even_sort_serial(RandomAccessIt first,
37   RandomAccessIt last)
38 {
39   if(first == last)
40     return ;
41   int length = std::distance(first, last);
```

```
42    int sorted = 0, __even = length & 1,\
43      __odd = 1 ^ (length & 1);
44    while( ! sorted ) // not sorted
45    {
46      sorted = 1;
47      // Odd sort
48      for(RandomAccessIt __i = first + 1; __i + __odd != last; __i += 2)
49      {
50        if( *(__i) > *(__i + 1) )
51        {
52          std::iter_swap(__i, __i + 1);
53          sorted = 0;
54        }
55      }
56
57      // Even sort, the same process with different start point
58      for(RandomAccessIt __i = first; __i + __even != last; __i += 2)
59      {
60        if( *(__i) > *(__i + 1) )
61        {
62          std::iter_swap(__i, __i + 1);
63          sorted = 0;
64        }
65      }
66    }
67    return ;
68  }
69
70  // print my information
71  void my_info()
72  {
73    fputs(
74      "Name: Derong Jin\nStudent ID: 120090562\n",
75      stdout);
76    fputs(
77      "Assignment 1, Odd-Even Transposition Sort, sequenential version\n",
78      stdout
79    );
80    return ;
81  }
82
83
84  // main function
85  int main(int argc, char **argv)
86  {
87    // read
88    int n, *a;
89    FILE* fp = fopen(argv[1], "r");
90    fscanf(fp, "%d", &n);
91    a = new int[n];
92    for(int i = 0; i < n; i++) fscanf(fp, "%d", a + i);
```

```
  fclose(fp);

  // sort
  auto time1 = std::chrono::high_resolution_clock::now();
  inplace_odd_even_sort_serial(a, a + n);
  auto time2 = std::chrono::high_resolution_clock::now();
  my_info();
  fprintf(stdout, "runTime is %.6lf sec\n",
   std::chrono::duration<double, std::ratio<1, 1>>(time2 - time1).count()
  );

  // if n is small, also print to stdout
  if( n <= 20 )
  {
    for(int i = 0; i < n; i++)
      fprintf(stdout, "%d ", *(a + i));
    fprintf(stdout, "\n");
  }


  // write and terminate
  char* output_path = new char[strlen(argv[1]) + 30];
  sprintf(output_path, "%s.sequential.out", argv[1]);
  fp = fopen(output_path, "w");
  for(int i = 0; i < n; i++) fprintf(fp, "%d\n", *(a + i));
  fclose(fp);
  delete[] output_path;
  delete[] a;
  return 0;
}
```

Listing 2: **Parallel Implementaion of Odd-Even Transposition Sort**

```
/**
 * @file odd_even_parallel.cpp
 * @author Derong Jin (120090562@link.cuhk.edu.cn)
 * @brief Parallel implementation of Odd-Even Sort using MPI
 *       compile:  mpic++ odd_even_parallel.cpp -o p_sort -std=c++11 -w
 *         usage:  mpiexec -np <N core> ./p_sort <data-file-path>
 *
 * @date 2022-10-11
 *
 */

#include <mpi.h>
#include <cmath>
#include <cstdio>
#include <chrono>
#include <cstring>
#include <algorithm>
#define ODD_TAG 1
```

```
19 #define ODD_TAG_ 2
20 #define EVEN_TAG 4
21 #define EVEN_TAG_ 8
22
23 /**
24  * @brief Sort the element of a partial sequence,
25  *        processor are co-working in this function
26  *
27  * @tparam RandomIt: Random Access Iterator
28  * @param first An iterator
29  * @param last Another iterator
30  * @param rank the processor's rank
31  * @param numprocs the total number of processor
32  * @param datatype use to indicate the datatype,
33  *                 should be kept compatible with
34  *                 typeof(*RandomIt)
35  *
36  * This is an MPI implementaion of parallel Odd-Even Sort
37  * algo., every processor is corresponding to its first and
38  * last iterator, and is responsible to sort elements in
39  * [first, last) in acsending order. When executing,
40  * processors are conducting odd-even sort in its own range,
41  * as well as in extended range by comparing the boundary element
42  * sent by its adjacant processors. The algorithm is equivalent
43  * to sequential version, however is more efficiency in some
44  * (large) case.
45  */
46 template<typename RandomIt>
47 void inplace_odd_even_sort_parallel(
48   RandomIt first, RandomIt last, int rank, int numprocs,
49   MPI_Datatype datatype)
50 {
51   // Nothing to do when there is no elements.
52   if(first == last) return ;
53
54   // MPI variables
55   MPI_Status status;
56   MPI_Request send_request, recv_request;
57
58   // status
59   int done = 0, sorted = 0;
60
61   // information related to the partial sequence
62   // fill_buffer indicates sending information to privious processor
63   // check_buffer indicates receiving information
64   // odd and even means the duty is on odd/even round
65   int length = std::distance(first, last);
66   int odd_fill_buffer = 1 - rank * length % 2;
67   int even_fill_buffer = rank * length % 2;
68   int odd_check_buffer = 1 - (rank + 1) * length % 2;
69   int even_check_buffer = (rank + 1) * length % 2;
```

```
70  if( rank == numprocs - 1 ) odd_check_buffer = even_check_buffer = 0;
71  if( rank == 0 ) odd_fill_buffer = even_fill_buffer = 0;
72
73  // the first/last element that needs compare in odd/even round
74  RandomIt even_first = first + rank * length % 2;
75  RandomIt even_last = last - (rank + 1) * length % 2;
76  RandomIt odd_first = first + (1 - rank * length % 2);
77  RandomIt odd_last = last - (1 - (rank + 1) * length % 2);
78
79  // allocating buffer, into which storing the message
80  RandomIt recv_buffer = (RandomIt) malloc(sizeof(*first));
81  RandomIt send_buffer = (RandomIt) malloc(sizeof(*first));
82  RandomIt maxv_buffer = (RandomIt) malloc(sizeof(*first));
83  while( ! done )
84  {
85    sorted = 1;
86
87    // odd sort
88    for(RandomIt i = odd_first; i != odd_last; i += 2)
89    {
90      if(*(i) > *(i + 1))
91      {
92        std::iter_swap(i, i + 1);
93        sorted = 0;
94      }
95    }
96
97    // on odd boundary, Isend/Irecv to avoid deadlock
98    *send_buffer = *first;
99    if( odd_fill_buffer ) MPI_Isend(send_buffer, 1, datatype, rank - 1, ODD_TAG,
          MPI_COMM_WORLD, &send_request);
100   if( odd_check_buffer ) MPI_Irecv(recv_buffer, 1, datatype, rank + 1, ODD_TAG
        , MPI_COMM_WORLD, &recv_request);
101   if( odd_fill_buffer ) MPI_Wait(&send_request, &status);
102   if( odd_check_buffer )
103   {
104     MPI_Wait(&recv_request, &status);
105     if( *recv_buffer < *(last - 1) )
106     {
107       std::iter_swap(recv_buffer, last - 1);
108       sorted = 0;
109     }
110     MPI_Isend(recv_buffer, 1, datatype, rank + 1, ODD_TAG_, MPI_COMM_WORLD, &
          send_request);
111   }
112   if( odd_fill_buffer )
113   {
114     MPI_Recv(maxv_buffer, 1, datatype, rank - 1, ODD_TAG_, MPI_COMM_WORLD, &
          status);
115     if( *first < *maxv_buffer )
116     {
```

```
117         *first = *maxv_buffer;
118         sorted = 0;
119       }
120     }
121
122
123     // even sort
124     for(RandomIt i = even_first; i != even_last; i += 2)
125     {
126       if(*(i) > *(i + 1))
127       {
128         std::iter_swap(i, i + 1);
129         sorted = 0;
130       }
131     }
132
133     // on even boundary
134     *send_buffer = *first;
135     if( even_fill_buffer ) MPI_Isend(send_buffer, 1, datatype, rank - 1,
          EVEN_TAG, MPI_COMM_WORLD, &send_request);
136     if( even_check_buffer ) MPI_Irecv(recv_buffer, 1, datatype, rank + 1,
          EVEN_TAG, MPI_COMM_WORLD, &recv_request);
137     if( even_fill_buffer ) MPI_Wait(&send_request, &status);
138     if( even_check_buffer )
139     {
140       MPI_Wait(&recv_request, &status);
141       if( *recv_buffer < *(last - 1) )
142       {
143         std::iter_swap(recv_buffer, last - 1);
144         sorted = 0;
145       }
146       MPI_Isend(recv_buffer, 1, datatype, rank + 1, EVEN_TAG_, MPI_COMM_WORLD, &
            send_request);
147     }
148     if( even_fill_buffer )
149     {
150       MPI_Recv(maxv_buffer, 1, datatype, rank - 1, EVEN_TAG_, MPI_COMM_WORLD, &
            status);
151       if( *first < *maxv_buffer )
152       {
153         *first = *maxv_buffer;
154         sorted = 0;
155       }
156     }
157
158     // the array is sorted if no change is detected in each processor
159     MPI_Allreduce(&sorted, &done, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
160   }
161
162   // free the memory
163   free(send_buffer);
```

```
164   free(recv_buffer);
165   free(maxv_buffer);
166   return ;
167 }
168
169 // print my information
170 void my_info()
171 {
172   fputs(
173     "Name: Derong Jin\nStudent ID: 120090562\nAssignment 1, Odd-Even
          Transposition Sort, parallel version, MPI implementation\n",
174     stdout
175   );
176 }
177
178 // main function
179 int main(int argc, char **argv)
180 {
181   // declaration
182   int *a, *arr;
183   int numprocs, rank, real_n, n;
184
185   // MPI initialization
186   MPI_Init(&argc, &argv);
187   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
188   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
189
190   // load data
191   if(rank == 0 && argc == 2)
192   {
193     n = -1; // to detect error.
194     FILE* fp = fopen(argv[1], "r");
195     if(fp != NULL)
196     {
197       fscanf(fp, "%d", &real_n);
198       n = real_n % numprocs == 0 ? real_n : (real_n / numprocs + 1) * numprocs;
199       arr = new int[n];
200       int inf = 0x7fffffff; // INT_MAX
201       for(int i = 0; i < real_n; i++) fscanf(fp, "%d", arr + i);
202       for(int i = real_n; i < n; i++) arr[i] = inf;
203       fclose(fp);
204     }
205   }
206
207   // sorting & timing
208   std::chrono::high_resolution_clock::time_point time1, time2;
209   time1 = std::chrono::high_resolution_clock::now();
210   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
211   if(n == -1) {
212     if(rank == 0)
213       fprintf(stderr, "Usage: ./psort <input-file-name>\n");
```

```
214     return 1;
215   }
216   a = new int[n / numprocs + 1];
217   MPI_Scatter(arr, n / numprocs, MPI_INT, a, n / numprocs, MPI_INT, 0,
          MPI_COMM_WORLD);
218   inplace_odd_even_sort_parallel(a, a + n / numprocs, rank, numprocs, MPI_INT);
219   MPI_Gather(a, n / numprocs, MPI_INT, arr, n / numprocs, MPI_INT, 0,
          MPI_COMM_WORLD);
220   time2 = std::chrono::high_resolution_clock::now();
221
222   // print out and terminate
223   if(rank == 0)
224   {
225     char* output_path = new char[strlen(argv[1]) + 30];
226     sprintf(output_path, "%s.parallel.out", argv[1]);
227     my_info();
228     fprintf(stdout, "runTime is %.6lf sec\n",
229       std::chrono::duration<double, std::ratio<1, 1>>(time2 - time1).count()
230     );
231     fprintf(stdout, "Input Size: %d\nProc Num: %d\n", real_n, numprocs);
232
233     // if n is small, also print to stdout
234     if( real_n <= 20 )
235     {
236       for(int i = 0; i < real_n; i++)
237         fprintf(stdout, "%d ", *(arr + i));
238       fprintf(stdout, "\n");
239     }
240
241     // print out to result file
242     FILE* fp = fopen(output_path, "w");
243     for(int i = 0; i < real_n; i++) fprintf(fp, "%d\n", *(arr + i));
244     fclose(fp);
245     delete[] arr;
246     delete[] output_path;
247   }
248
249   delete[] a;
250
251   MPI_Finalize();
252   return 0;
253 }
```