香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

CSC4005
Parallel Computing

Project 1

Author	**Student ID**
Luo Haoyan	119010221

October 10, 2022

Contents

1 Introduction

Odd-even sort transposition sort is a parallel sorting algorithm derived from the Bubble sort algorithm, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array. However, the cost of Bubble sort will be expansive if the problem size is quite large. In the worst case, the time complexity could be theoretically $O(n^2)$. Therefore, we try to investigate the effect of parallel computing in improving the performance of odd-even transposition sort.

In the project two versions of odd-even transposition sort are implemented: a sequential version and a parallel version. Comprehensive experiments are conducted and performance under different configurations in three dimensions are analyzed: performance of MPI program vs. Sequential program; different size of random generated array (from 20 up to 100000); different number of cores used in the MPI program.

1.1 Sequential Odd-Even Transposition Sort

The sequential version of odd-even transposition sort is very similar to the original bubble sort. The odd element is compared with the posterior even element in odd iteration and the even element is compared with the posterior odd element in even iteration. The total number of iterations can be up to the length of the input array, where in the worst case (the array was originally reversely sorted) the array will be sorted in the last iteration.

1.2 Parallel Odd-Even Transposition Sort

The parallel version is also very similar to the above mentioned two algorithms, but with a slightly modification in the way of distributing data. In parallel programming such as MPI program, there are a number of processors that can be executed at the same time to improve the performance. Assume there are m numbers distributed to n processes respectively. In the parallel odd-even transposition sort, insides each process, compare the odd element with the posterior even element in odd iteration, or the even element with the posterior odd element in even iteration respectively. Then the algorithm will swap the elements if the posterior element is smaller.

The next step is the main difference of the parallel odd-even transposition sort. Since data are distributed separately in each process and the number of data in each process vary (it can be odd or even), there may be "left over" in each process i.e. the number that is not compared in one iteration. The elements that are left over need to be compared between processes and switched the order if necessary. Thus, if the current process rank is P, and there some elements that are left over for comparison in the above mentioned iteration, the boundary elements will be compared with process with rank P-1 and P+1. If the posterior element is smaller, the algorithm will swap them. The below figure shows a graphical illustration of the algorithm.

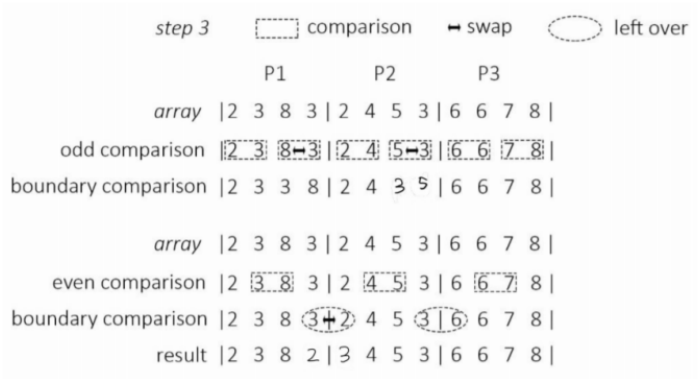

Figure 1: *Graphical illustration of parallel odd-even transposition sort*

2 Method

2.1 Overview

In the implementation of two versions of odd-even transposition sort, the program can be decomposed into the sorting algorithm implementation and MPI structure implementation (mainly for the parallel version). For the sequential version, the overall data flow of the program is identical to the algorithm. For the parallel version, the overall data flow is described as many algorithms encapsulated inside each process under a large MPI structure. Thus, the overview of two programs can be interpreted by the following two figures. Details will be explained in the following sessions.

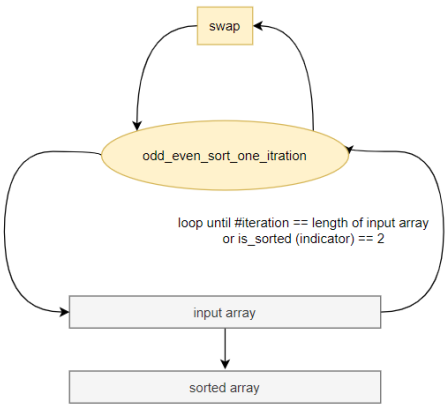

Figure 2: *Graphical illustration of sorting algorithm*

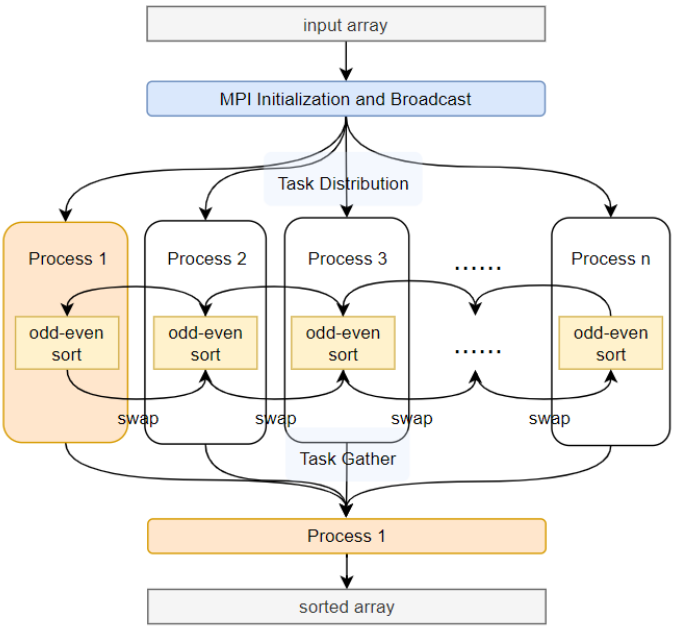

Figure 3: *Graphical illustration of overall flow chart of parallel program*

2.2 Sequential Version

As described in the above overview session, the overall design of sequential odd-even transposition sort is just the algorithm it self. As shown in Figure 2, the implemented algorithm is composed of two parts: *odd_even_sort_one_iteration* and *swap*. The loop of the algorithm will terminate until exceeding the maximum number of iteration or the *is_sorted* indicator equals 2. More information is presented in the following sessions.

2.2.1 Odd Even Transposition Sort in One Iteration

Following the basic idea of bubble sort, the implementation of the sequential version is simple. Following the description in the introduction session, the sample code of the algorithm when in the odd iteration is shown below.

```
void odd_even_sort_one_iter(int *arr, int start, int end, int is_odd, int *is_sorted) {
  if (is_odd) {
    for (int i = start; i < end; i += 2) {
        if (arr[i] > arr[i + 1]) {
            swap(arr, i, i + 1);
            is_sorted = 0;
        } else {
```

```
                is_sorted++;
            }
        }
    }
```

As shown in the above code, *start* indicates the starting position of the input array and *end* indicates the ending position. *is_odd* indicator represents if the loop is in odd iteration. When in the odd iteration and the posterior number is smaller than the odd number, the *swap* function will be called to switch the position. There is also a *is_sorted* indicator which records after each swapping operation. This is in the use of checking whether the array is already sorted and helping quit the loop in advanced instead of extra computation. This will be illustrated in the next session.

2.2.2 Early Stopping Indicator

The following code can help to describe the idea in a clearer way:

```
int is_sorted = 0;  // 2 indicates unchanged for two consecutive iterations
// odd even sort
for (int i = 1; i <= num_elements; i++) {
    odd_even_sort_one_iter(elements, 0, num_elements - 1, i % 2, &is_sorted);
    if (is_sorted == 2) break;
}
```

In the original bubble sort, if there is no swapping in one iteration, the sorting ends (this indicates the array is already sorted). In the odd-even parallel sort, if there are two consecutive iterations that perform no swapping (one odd iteration and one even iteration), then the array is sorted. When there are two consecutive runs without swapping, the *is_sorted* indicator will add up to two and quit the loop. Thus, the *is_sorted* indicator helps to perform this idea of early stopping to improve performance.

2.3 Parallel Version

In the parallel version of odd-even transposition sort, the sorting part is basically the same as sequential version. Thus in the following sessions, the details of the sorting algorithm will not be mentioned. You can refer to the above session if needed. Here, the task and data distribution, message passing design, and data gathering mechanism will be elaborated.

2.3.1 Initialization and Broadcasting

The initialization of the MPI parallel programming is straightforward. As described in Figure 3, process 1 (i.e. the process with rank 0) is used as the root process, which executes tasks like reading in input array,

timing, and outputting the sorted array. For the MPI part, we have to perform functions like $MPI\_Init()$ to initialize and $MPI\_Comm\_Rank()$ and $MPI\_Comm\_Size()$ to record the rank of the current process and the size of all processes, respectively. Note that here we also need to explicitly call the broadcast function in MPI ($MPI_Bcast()$) since every process also needs to know the length of the array to record the odd and even positions.

2.3.2 Task Distribution

Task distribution is the key part of parallel sort. According to Figure 3, the input data will be distributed into different processes, and each process will perform the sorting respectively. Two important aspects should be considered:

- How many numbers of elements should be allocated in each process?
- What MPI function should be used to distributed data to each process?

The following paragraphs answer these questions one by one.

Data allocation Intuitively, the data should be allocated evenly to each process so that we can better utilize the computation power. However, the number of programs may not be able to evenly divide the number of elements in the array. Thus, a simple strategy is applied: if there are element remained after evenly divided the number of processes, the remaining element will be distributed one by one from the first process (i.e. rank==0) to the next. In this way, we can best utilize the computation power among all processes. The following figure describe this idea with an example of 83 input size:

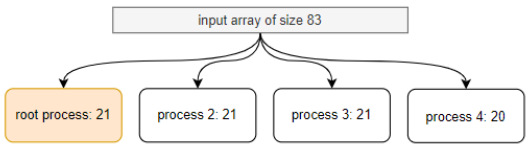

Figure 4: *Example of data allocation when 83 numbers are sorted by 4 processes*

Since the number of elements allocated to different processes may be different, the MPI function chosen here is $MPI\_Scatterv()$, which is able to distributed different number of data to different processes according to a number-recorded list. To facilitate this function, an *element_count_list* and *displacements* are stored to record the number of elements and the starting point of the array in each process. The code snippet of this function is shown below:

```
MPI_Scatterv(elements, element_count, displacements, MPI_INT,
                my_element, num_my_element, MPI_INT, 0, MPI_COMM_WORLD);
```

2.3.3 Sorting and Swapping

Recall that the key difference between the sequential odd-even sort and parallel odd-even sort is that there exists element comparison and swapping between processes in the parallel version. Thus, the message passing is an essential part. Similarly, an example of sorting 13 numbers with 4 processes in odd iteration is provided to better illustrate the idea:

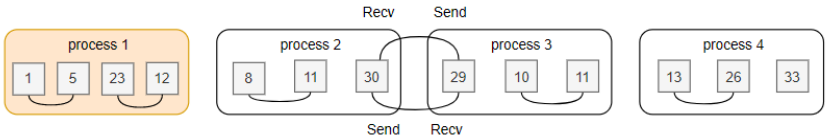

Figure 5: *Odd iteration of sorting 13 numbers with 4 processes*

As shown in the figure, the swapping occurs when there are elements left over in some processes. The swapping process will perform in two directions at the same time: checking and comparing with the left process (lower rank) and right process (higher rank). Note that we have to ensure that the boundary conditions, where the root process (i.e. process rank==0) cannot be compared with left values, and process n behaves similarly. At the same time, the arrangement of sending message and receiving message also matters for preventing the deadlock situation.

2.3.4 Task Gathering

After allocating the data and perform odd-even sort in each process (including the swapping operations), the next step is gathering all the sorted numbers from other processes to root process. Similar to the distribution strategy above where the MPI scatter function is used, here, $MPI_Allgatherv()$ is used, where different number of elements in different processes can be gathered to the root process according to the previously recorded *displacement*. The code snippet of $MPI_Allgatherv()$ is shown below:

```
MPI_Allgatherv(my_element, num_my_element, MPI_INT, elements,
            element_count, displacements, MPI_INT, MPI_COMM_WORLD);
```

3 Experiments

Comprehensive experiments are conducted to test and analyze the performance of parallel computing. Various input data are generated (range from size 20 to size 100000) to test my program. Here, small size (20 - 100), medium size (1000 - 20000), and large size (50000 - 100000) are respectively defined. The test data generator file is the one provided by the template. The generated test data are all under /test_data folder.

3.1 Evaluation Metrics

In addition to the running time of the program, another evaluation metric: Speedup Factor, is introduced in the evaluatioin part. Parallel speedup is defined as the ratio of the time required to compute some function using a single processor (t_s) divided by the time required to compute it using P processors (t_p), the formula can be explicitly written as:

$$S(n) = \frac{Execution\ time\ using\ one\ processor(single\ processor\ system)}{Execution\ time\ using\ amultiprocessor\ with\ n\ processors} = \frac{t_s}{t_p}$$

3.2 Results

3.2.1 Main results

The Speedup results (based on the equation above) of both sequential odd-even sort and parallel odd-even sort are all presented in Table 1. The running time results for different configurations are also provided in Table 2 (See appendix) To better understand the results in a clearer way, the visualization of the data is

Table 1: *The Speedup results for both sequential version and parallel version from input size 20 to input size 100000.* ***Note:*** *Speedup for sequential sorting algorithm constantly equals to 1.*

#	1	2	3	4	5	6	7	8	9	10
20	1	0.030423	0.021926	0.022336	0.022334	0.022555	0.024635	0.01972	0.024499	0.006479
100	1	0.295949	0.214081	0.256495	0.075941	0.173404	0.114185	0.148704	0.1941	0.06079
1000	1	2.767406	1.700075	2.947915	2.66249	1.799516	2.02832	2.292322	2.072531	1.824228
2000	1	2.999692	2.898176	3.700184	3.28135	3.363637	3.702729	3.506679	3.514345	3.218115
5000	1	3.986608	3.690728	4.247158	3.142566	4.879077	5.200067	4.95881	5.518289	5.86871
8000	1	5.232497	5.970609	7.20497	7.711366	7.344589	8.160577	8.785031	8.09115	10.26264
10000	1	4.944912	6.069888	7.017341	8.02507	9.144501	9.079068	9.824038	9.253177	7.669317
20000	1	4.723436	6.223277	6.566871	9.491821	7.514645	7.588726	7.994326	8.520591	9.534681
50000	1	2.085225	2.757487	3.794515	4.673486	5.320915	3.840421	6.85895	7.209122	8.717345
80000	1	4.962144	6.366289	8.602006	10.57335	12.6298	13.74709	15.66142	17.83165	19.84825
100000	1	4.880743	6.583339	8.100599	8.963468	8.160474	11.17017	11.89285	12.35491	11.57344
#	11	12	13	14	15	16	17	18	19	20
20	0.006544	0.00634	0.004465	0.00899	0.007251	0.005661	0.005757	0.010233	0.003289	0.003287
100	0.060073	0.051587	0.055741	0.061839	0.043094	0.07818	0.071145	0.063662	0.079605	0.037147
1000	2.024215	2.13621	1.882406	1.877804	1.905333	1.945741	1.743	1.810832	1.86907	2.633386
2000	3.493131	3.569968	3.218434	3.293873	3.731643	3.223544	2.843292	2.949087	2.539479	3.091587
5000	4.624744	5.181031	4.504892	4.445223	4.629828	5.462744	4.888343	6.13224	6.623101	5.212878
8000	9.450882	8.985242	9.803437	9.497869	8.778218	9.50997	9.278693	9.12063	8.541275	10.41749
10000	8.764176	8.720979	8.380349	8.179204	8.9503	9.081449	9.879736	9.769498	11.36931	9.630372
20000	10.2875	8.775588	10.2505	9.173415	8.858075	9.160317	9.426917	10.59202	9.021149	10.63124
50000	6.00287	6.994785	6.960111	7.084272	7.497125	7.995915	8.742622	9.048127	9.441437	9.590966
80000	15.68708	15.11724	18.79845	17.77948	18.31692	19.27492	19.82722	21.68253	22.1167	23.10159
100000	12.74018	14.90403	14.79492	16.06933	17.35588	18.69145	19.48078	20.46433	21.73099	20.9735

presented in the following two figures:

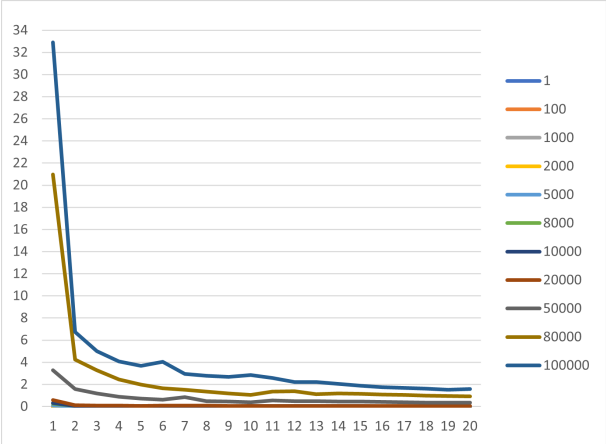

Figure 6: *Visualization for running time results*

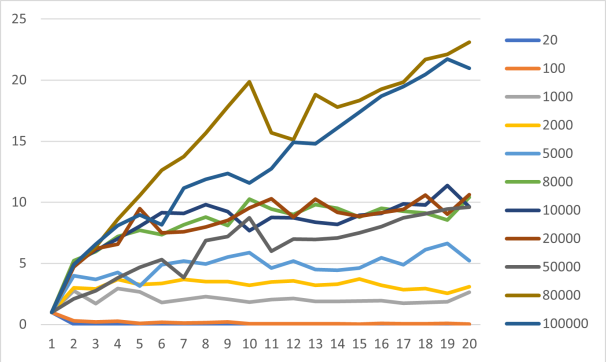

Figure 7: *Visualization for Speedup results*

3.2.2 20-dims Input Size Demo

The program outputs when the input is a 20-dims array are shown below. The parallel program uses 4 cores as a demo example.

```
12:02:23 119010221@node21 project1 → ./ssort 20 ./test_data/20a.in
actual number of elements:20
The input 20-dim array:
1999978 41402839 18517414 42433117 22320624 28653630 47709512 66225354 97485258 13245389
94772824 4088563 2727287 86461980 9883645 55024865 63623508 8686919 37529053 77337238
Student ID: 119010221
Name: Haoyan Luo
Assignment 1
Run Time: 7.845e-06 seconds
Input Size: 20
Process Number: 1
The sorted 20-dim array:
1999978 2727287 4088563 8686919 9883645 13245389 18517414 22320624 28653630 37529053 4140
2839 42433117 47709512 55024865 63623508 66225354 77337238 86461980 94772824 97485258
```

Figure 8: *Result for sequential sort*

```
12:05:12 119010221@node21 project1 → mpirun -np 4 ./psort 20 ./test_data/20a.in
actual number of elements:20
The input 20-dim array:
1999978 41402839 18517414 42433117 22320624 28653630 47709512 66225354 97485258 13245389
 94772824 4088563 2727287 86461980 9883645 55024865 63623508 8686919 37529053 77337238
rank 0: num_my_element = 5, displacements = 0
rank 1: num_my_element = 5, displacements = 5
rank 2: num_my_element = 5, displacements = 10
rank 3: num_my_element = 5, displacements = 15
Student ID: 119010221
Name: Haoyan Luo
Assignment 1
Run Time: 0.000463313 seconds
Input Size: 20
Process Number: 4
The sorted 20-dim array:
1999978 2727287 4088563 8686919 9883645 13245389 18517414 22320624 28653630 37529053 414
02839 42433117 47709512 55024865 63623508 66225354 77337238 86461980 94772824 97485258
```

Figure 9: *Result for parallel sort*

3.3 Performance Evaluation: Sequential vs. Parallel

The direct performance comparison between sequential sorting and parallel sorting is explicitly represented in the speedup results. From Figure 7 we can have the following observations:

1. With a small input size (20 $\sim$ 100), the parallel approaches of the odd-even transposition obtain worse performances (speedup ¡ 1). According to the first two rows in Table 1, the performance is actually getting worse when the number of processes increases.

2. With a medium input size (1000 $\sim$ 20000) and large input size (50000 $\sim$ 100000), the parallel approaches are all better than the sequential approach. The trend of the performance lines of different number of cores will be analyzed in the following sessions.

3. According to Figure 6, the performance will increase dramatically as the number of cores increases at the beginning, i.e. when the number of processes is not that large. This is significantly obvious for

input data with a large size. It may be worth noticing that the critical point (the x-axis point where the following lines decrease much slower) in the experiments are mostly 2 or 3, which also supports the view mentioned above.

Analysis Parallel computing (i.e. adding the number of cores) will enhance the performance of the sorting algorithm when the input size is a relatively large number. We can make a good inference from this observation: when the problem size is small, it is not worthy to compute in parallel, because it may even harm the performance. This phenomenon can be explained by the fact that there exists time-consuming context-switch cost between processes (if parallel programming is performed), which is overhead to the program when the input size is relatively small. As the number of processes increases, the context-switch time gradually increases to a significant level, and finally even diminish the the performance improvement that parallelization brings.

Thus, in order to maximize the performance, it is a good choice to balance the mentioned factors and find an optimal number of processes to execute the program based on the input size. Above all, parallelization provides notable performance improvement that the running time of the program can be 20+ times faster in a proper settings (in this experiment).

3.4 Performance Evaluation: Different Size of Input Array

Similar to the above discussion, the effect of different size of input data is also concluded in the above figures. Here, the performance evaluation objective is to study the running time / speedup in relation to the input size. To better facilitate the analysis, the relationship between speedup and problem size is also visualize in the following figure.

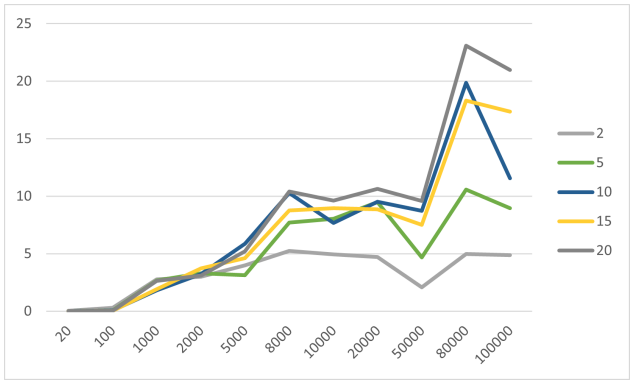

Figure 10: *The relationship between speedup and different input size for selected number of cores*

The observations of different generated input sizes are stated below:

1. Generally speaking, the performance increases as the problem size increases (when the input size is medium or large, as discussed in the previous session). When the size is less than 8000 (in this

experiment), the performance improves steadily.

2. However, the increase of performance slows down when the input size is larger than 8000. The performance also fluctuate in some settings. Interestingly, we notice that when the input size is nearly 50000, there is a surge on the performance.

3. After the surge on the input size of 50000, the performance keeps increasing when the size increases to 80000. Nevertheless, the performance slows down (and drops in some settings) after input size of 100000. It is a reasonable guess that the performance will not i,prove dramatically as the input size increases. The corresponding analysis is given below.

Analysis The observation above can be explained by the Gaustafson's law, which claims that: as the problem size increases, parallel computing will have a higher performance which can be interpreted by the formulation:

$$speedup\ factor = S(n) = n + (1 - n)s$$

where s is the number of cores available and n is the proportion of sequential computing. As the problem size increases, the sequential proportion n will decrease, which makes the speedup more close to the number of cores. However, there also exists a reasonable boundary for this improvement as the problem size getting larger and larger. Thus, the performance improvement will slow down and cease when the problem size is too large (relative to the number of cores).

3.5 Performance Evaluation: Different Number of Cores

The following discussion is based on the speedup results in the above figures and tables. To have an insight into the problem, the following figures draw the speedup results for small+medium input and medium+large input respectively.

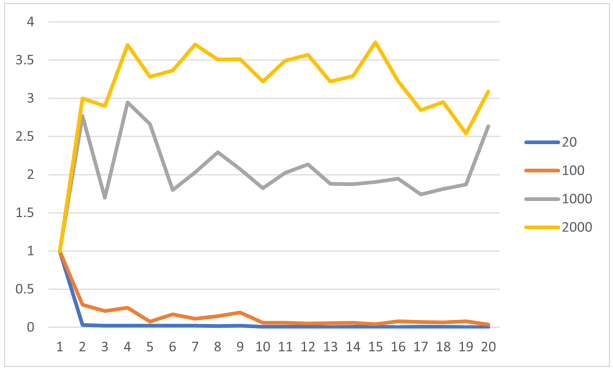

Figure 11: *Visualization of small and medium input sizes on different number of cores*

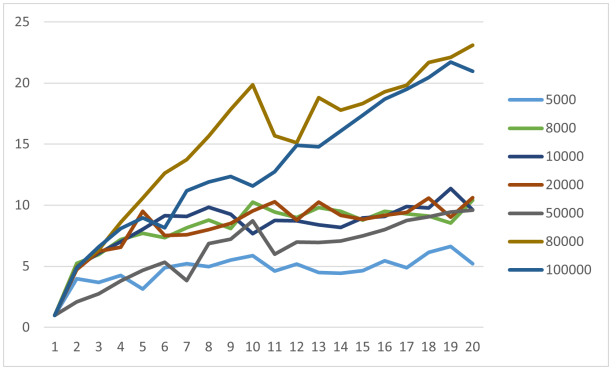

Figure 12: *Visualization of medium and large input sizes on different number of cores*

The observations are given below:

1. According to Figure 7, in general, as the number of cores increases, the overall performance of the parallel program increases (as long as it is not under a relatively small size).

2. From the upper left figure we can observe that the parallel approach will first enhance the performance as the number of cores increases (in the experiment, it is around 4 cores) in the small-medium input settings, and then the performance will fluctuate but basically remain the same.

3. From the upper right figure we observe that with a relatively large problem size, the performance keeps increasing as the number of cores increases. There are some surge in some configuration settings but the trend is generally upward.

Analysis The observations above enlighten that the optimal number of processes for a given configuration depends on many factors such as problem size. Particularly, parallel computing can improve performance notably when the number of cores fits the problem size (i.e. the number of input elements can be fully divided by the number of cores). Moreover, it can be found that when the problem size is large (greater than 20000), the performance ceases increasing at certain point. This can be explained by Amdahl's law, which suggests that the number of cores will contribute little improvement to the performance when the number of processes is large enough.

4 Discussion and Conclusion

The performance of parallel computing can be affected by many factors, such as the input problem size and the number of cores available. When the input size is very small, the sequential version of program (in this experiment is the odd-even transposition sort) often performs better than the parallel version. However, when the problem size becomes larger, the parallel computing gradually outperforms the sequential one. The performance will be enhanced by more cores with reasonable size of input data. At the same time, there may be a notable boundary that limits the performance of parallel computing, which can be interpreted by some parallel computing principles. Further experiments can focus on generating larger input size while fixing to number number of cores to a reasonable number, or fixing a large number of input while increasing the number of cores.

Appendix

A Steps to Run My Program

The codes of sequential sorting and parallel sorting are in *odd_even_sequential_sort.cpp* and *odd_even_parallel_sort.cpp*, respectively. Note that since it is meaning less to have a parrallel program with n number of cores run on a input with m number of elements, where $n < m$, the program did not implemented to handle the situation that the number of input elements is less than the number of cores. Thus, please do not test the program in this way. Following are the steps to run my code (First please ensure that you are under the project folder). All the test data are generated in advanced in the *test_data* folder. You can also regenerate the data by typing: (here, size of 10000 is used as an example)

```
g++ test_data_generator.cpp -o gen
./gen 10000 ./test_data/10000a.in
```

or simply typing the following command to generate all the input data in my experiment:

```
g++ test_data_generator.cpp -o gen
bash generate_test_data.sh
```

To run the sequential sorting code, first you need to compile it:

```
g++ odd_even_sequential_sort.cpp -o ssort
```

Then run it by: (here, input size of 10000 is used as an example)

```
./ssort 10000 ./test_data/10000a.in
```

You can test the accuracy by typing:

```
g++ check_sorted.cpp -o check
./check 10000 ./test_data/10000a.in.seq.out
```

For the parallel sorting code, first you need to compile it with mpi command: (here, the number of processes is 4 as an example)

```
mpic++ odd_even_parallel_sort.cpp -o psort
```

Then you can either test the program in an iterative way:

```
salloc -n4 -p Debug # allocate cpu for your task
mpirun -np 4 ./psort 10000 ./test_data/10000a.in
./check 10000 ./test_data/10000a.in.parallel.out
```

or in a batch style. Note that the sbatch script for my experiment is available at *sbatch_submission.sh*, thus you can also run the experiment by simply submitting it to the cluster:

```
sbatch sbatch_submission.sh
```

You can check the results in the corresponding output file. My sample output is in $slurm - 5248.out$.

B Running Time Results

Table 2: *The running time results for both sequential version (process number equals to 1) and parallel version from input size 20 to input size 100000.* ***Note:*** *The unit is second in scientific notation.*

#	1	2	3	4	5	6	7	8	9	10
20	8.81E-06	2.89E-04	4.02E-04	3.94E-04	3.94E-04	3.90E-04	3.58E-04	4.47E-04	3.59E-04	1.36E-03
100	1.07E-04	3.62E-04	5.01E-04	4.18E-04	1.41E-03	6.18E-04	9.39E-04	7.21E-04	5.52E-04	1.76E-03
1000	8.34E-03	3.01E-03	4.91E-03	2.83E-03	3.13E-03	4.64E-03	4.11E-03	3.64E-03	4.03E-03	4.57E-03
2000	2.34E-02	7.79E-03	8.06E-03	6.32E-03	7.12E-03	6.95E-03	6.31E-03	6.67E-03	6.65E-03	7.26E-03
5000	8.80E-02	2.21E-02	2.39E-02	2.07E-02	2.80E-02	1.80E-02	1.69E-02	1.78E-02	1.60E-02	1.50E-02
8000	2.41E-01	4.61E-02	4.04E-02	3.35E-02	3.13E-02	3.28E-02	2.95E-02	2.74E-02	2.98E-02	2.35E-02
10000	3.22E-01	6.51E-02	5.31E-02	4.59E-02	4.01E-02	3.52E-02	3.55E-02	3.28E-02	3.48E-02	4.20E-02
20000	5.70E-01	1.21E-01	9.16E-02	8.69E-02	6.01E-02	7.59E-02	7.52E-02	7.13E-02	6.69E-02	5.98E-02
50000	3.29E+00	1.58E+00	1.19E+00	8.68E-01	7.05E-01	6.19E-01	8.57E-01	4.80E-01	4.57E-01	3.78E-01
80000	2.10E+01	4.23E+00	3.29E+00	2.44E+00	1.98E+00	1.66E+00	1.53E+00	1.34E+00	1.18E+00	1.06E+00
100000	3.29E+01	6.74E+00	5.00E+00	4.06E+00	3.67E+00	4.03E+00	2.95E+00	2.77E+00	2.66E+00	2.84E+00

#	11	12	13	14	15	16	17	18	19	20
20	1.35E-03	1.39E-03	1.97E-03	9.80E-04	1.21E-03	1.56E-03	1.53E-03	8.61E-04	2.68E-03	2.68E-03
100	1.78E-03	2.08E-03	1.92E-03	1.73E-03	2.49E-03	1.37E-03	1.51E-03	1.68E-03	1.35E-03	2.89E-03
1000	4.12E-03	3.91E-03	4.43E-03	4.44E-03	4.38E-03	4.29E-03	4.79E-03	4.61E-03	4.46E-03	3.17E-03
2000	6.69E-03	6.55E-03	7.26E-03	7.10E-03	6.26E-03	7.25E-03	8.22E-03	7.93E-03	9.20E-03	7.56E-03
5000	1.90E-02	1.70E-02	1.95E-02	1.98E-02	1.90E-02	1.61E-02	1.80E-02	1.44E-02	1.33E-02	1.69E-02
8000	2.55E-02	2.68E-02	2.46E-02	2.54E-02	2.75E-02	2.54E-02	2.60E-02	2.64E-02	2.82E-02	2.31E-02
10000	3.67E-02	3.69E-02	3.84E-02	3.94E-02	3.60E-02	3.55E-02	3.26E-02	3.30E-02	2.83E-02	3.34E-02
20000	5.54E-02	6.50E-02	5.56E-02	6.22E-02	6.44E-02	6.23E-02	6.05E-02	5.38E-02	6.32E-02	5.36E-02
50000	5.48E-01	4.71E-01	4.73E-01	4.65E-01	4.39E-01	4.12E-01	3.77E-01	3.64E-01	3.49E-01	3.43E-01
80000	1.34E+00	1.39E+00	1.12E+00	1.18E+00	1.14E+00	1.09E+00	1.06E+00	9.67E-01	9.48E-01	9.08E-01
100000	2.58E+00	2.21E+00	2.22E+00	2.05E+00	1.90E+00	1.76E+00	1.69E+00	1.61E+00	1.51E+00	1.57E+00