



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

PARALLEL PROGRAMMING

Assignment1 Parallel Odd-Even Transposition Sort

Author:
Shi Wenlan

Student Number:
119010265

October 6, 2022

Contents

1	Introduction	2
2	Method	2
2.1	Flow charts	3
2.2	Extra explanations	5
3	Result	6
3.1	Output screenshots	6
3.2	How to run my code	9
3.3	Raw data	10
3.4	Data analysis	11
4	Conclusion	14

1 Introduction

In this assignment, I programmed two versions of odd-even transposition sort: a sequential version and a parallel version which was implemented using MPI. And I test their performances using different numbers of processors.

Introduction of odd-even transposition sort: Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array. It consists of 2 phases - the odd phase and even phase.

Odd phase: Every odd indexed element (1, 3, 5...) is compared with the next even indexed element (considering 1-based indexing) and exchanged positions if needed.

Even phase: Every even indexed (2, 4, 6...) element is compared with the next odd indexed element and exchanged positions if needed.

```

Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10
Step 1(odd):  2  1  4  9  5  3  6  10
Step 2(even): 1  2  4  9  3  5  6  10
Step 3(odd):  1  2  4  3  9  5  6  10
Step 4(even): 1  2  3  4  5  9  6  10
Step 5(odd):  1  2  3  4  5  6  9  10
Step 6(even): 1  2  3  4  5  6  9  10
Step 7(odd):  1  2  3  4  5  6  9  10
Step 8(even): 1  2  3  4  5  6  9  10
Sorted array: 1, 2, 3, 4, 5, 6, 9, 10
  
```

Figure 1: Example of Odd-even Transposition Sort

2 Method

In this part, I would focus on the implementation of parallel version.

2.1 Flow charts

I used MPI to implement this and the general idea flow chart of my program is as following:

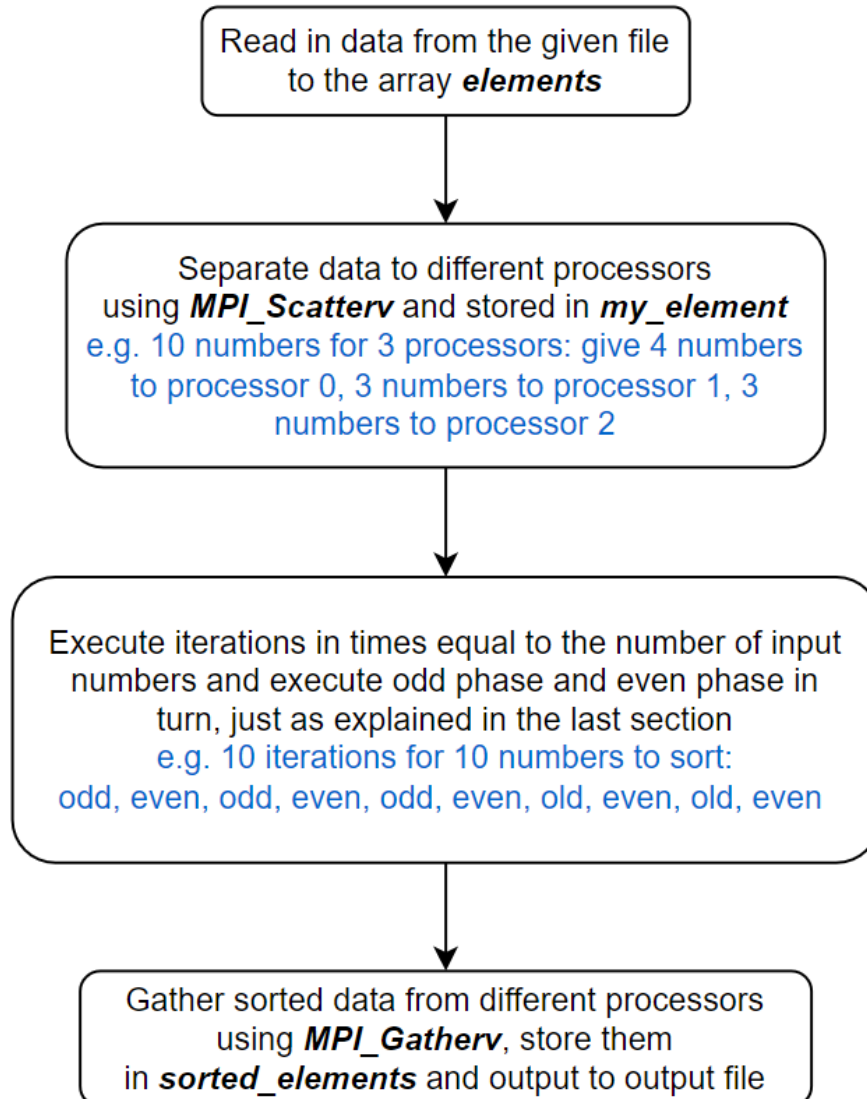


Figure 2: General Flow chart of Parallel Odd-even Transposition Sort

And the detailed flow chart about the third part (comparation for loop) of last figure is as following:

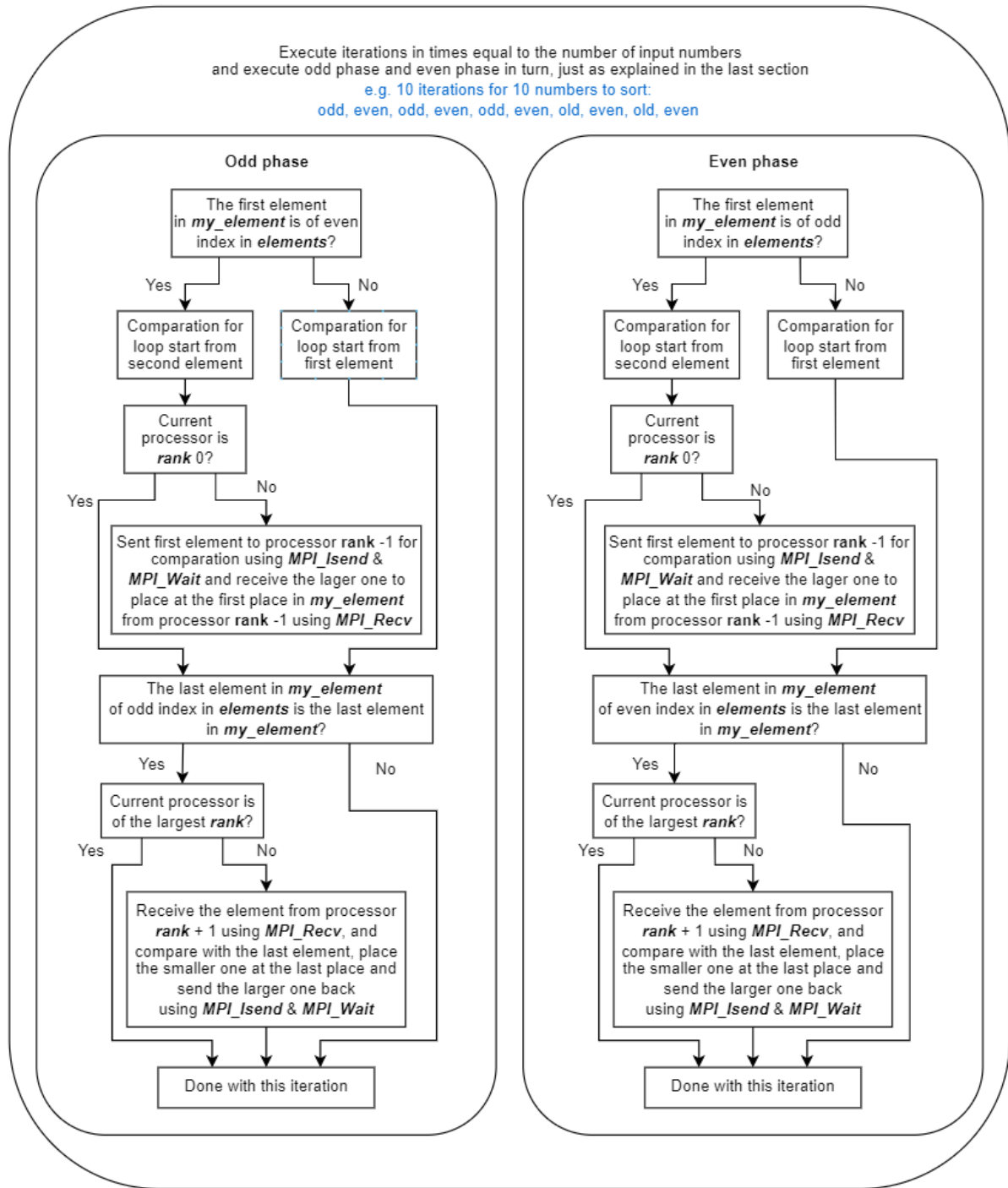


Figure 3: Detailed Flow Chart about Comparison For Loop

2.2 Extra explanations

The following are some extra explanations:

Why use *MPI_Scatterv* and *MPI_Gatherv*: During the experiment, I found that some input numbers were replaced by irrelevant numbers. It is because that the number of input numbers cannot be divided by the number of processors with no remainder. It turns out that the program should send different processors different blocks of data with different lengths and receive in the same way. And this should be a blocking operation since all operations after this depend on the allocation result of this operation. Thus only *MPI_Scatterv* and *MPI_Gatherv* can do this job.

How to determine the tag in the send and receive: During each iteration, the processor would have at most two send-receive pairs:

1. Get data for comparison: send first element to processor rank-1 (form comparison pair 1), and receive another element from processor rank+1 (form comparison pair 2).
2. Return comparison result: receive the result of comparison pair 1 from processor rank-1 to fill the first blank, and sent the result of comparison pair 2 back to processor rank+1.

To distinguish these two pairs, in iteration n , the tag of the first pair would be $2*n$, and the tag of the second pair would be $2*n+1$.

How to avoid deadlock using *MPI_Isend*, *MPI_Wait* and *MPI_Recv*: In each send-receive pair, the code of *MPI_Isend* is always ahead of the code of *MPI_Recv*, thus all the processors would send messages without blocking first, and then wait for receiving messages if needed.

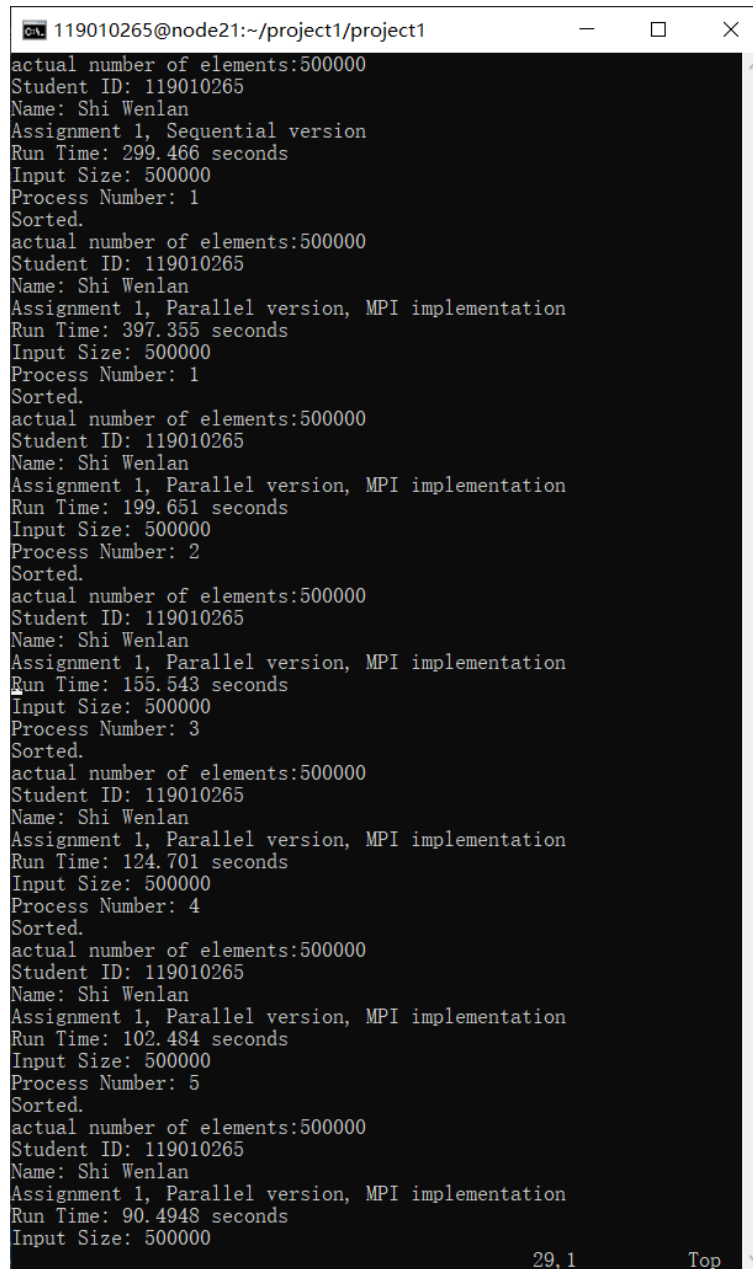
In addition, using unblocking *MPI_Isend* can improve utilization of computation resources since the time waiting for the send operation to complete can be used to do remaining comparisons. *MPI_Wait* is used to make sure the data has been successfully sent so as to enable the buffer to be reused in the next iteration, and should be used before the second *MPI_Recv* since the data to be received is based on the successful send beforehand.

Since the message received from other processors would be put into use immediately, the receive operation should be a blocking version, which is *MPI_Recv*. And the second *MPI_Recv* in the iteration should be behind of all of other *MPI_Isend* and *MPI_Wait* to avoid dead lock.

3 Result

3.1 Output screenshots

Here are some screenshots of one of my experiment:

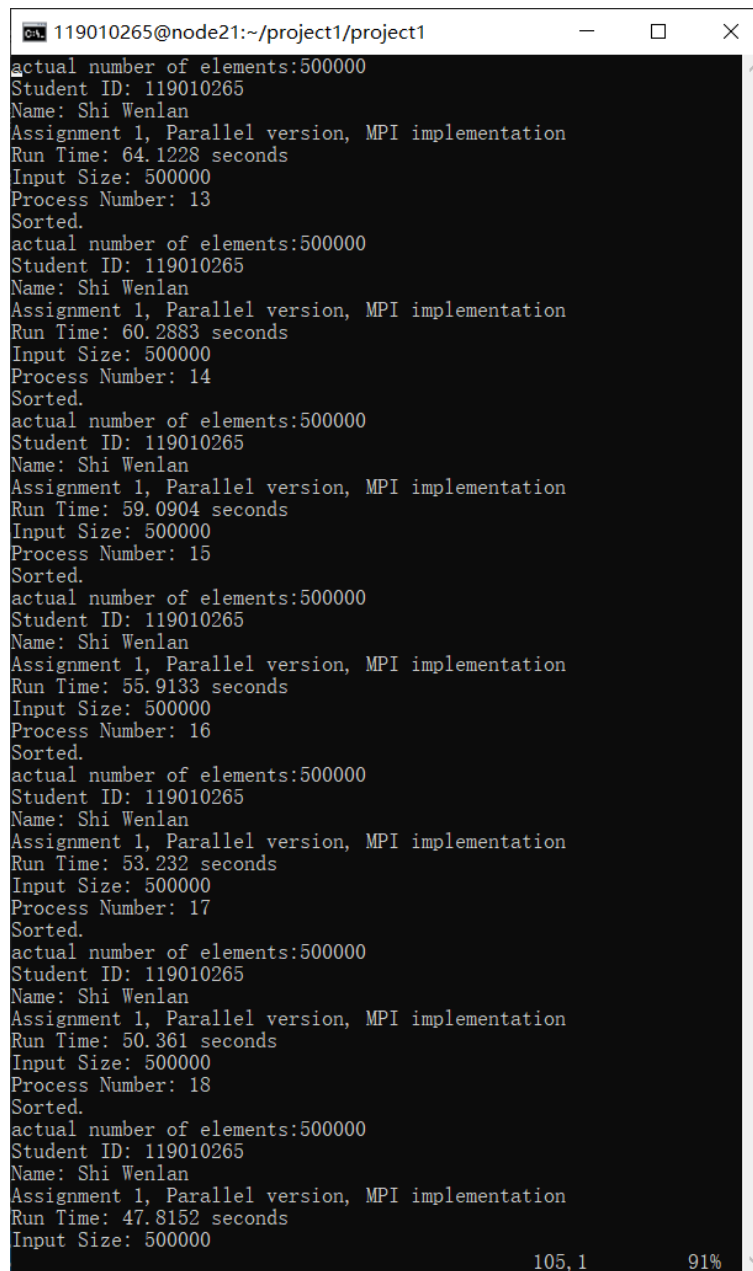


```
119010265@node21:~/project1/project1
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Sequential version
Run Time: 299.466 seconds
Input Size: 500000
Process Number: 1
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 397.355 seconds
Input Size: 500000
Process Number: 1
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 199.651 seconds
Input Size: 500000
Process Number: 2
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 155.543 seconds
Input Size: 500000
Process Number: 3
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 124.701 seconds
Input Size: 500000
Process Number: 4
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 102.484 seconds
Input Size: 500000
Process Number: 5
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 90.4948 seconds
Input Size: 500000
29, 1 Top
```

Figure 4: Screenshot Part 1

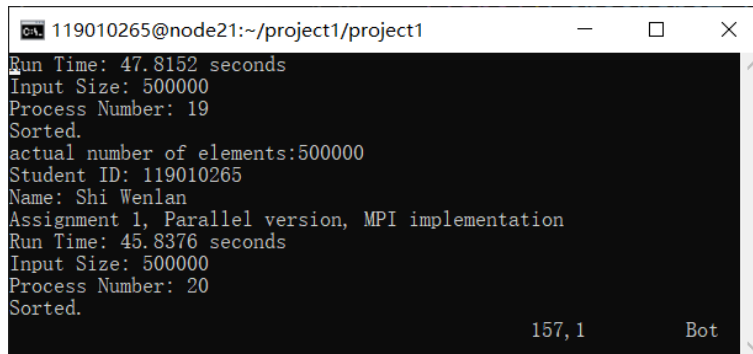
```
119010265@node21:~/project1/project1
Run Time: 90.4948 seconds
Input Size: 500000
Process Number: 6
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 81.9771 seconds
Input Size: 500000
Process Number: 7
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 74.2174 seconds
Input Size: 500000
Process Number: 8
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 67.6436 seconds
Input Size: 500000
Process Number: 9
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 61.8228 seconds
Input Size: 500000
Process Number: 10
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 69.6461 seconds
Input Size: 500000
Process Number: 11
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 72.2332 seconds
Input Size: 500000
Process Number: 12
Sorted.
actual number of elements:500000
Student ID: 119010265
53, 1 45%
```

Figure 5: Screenshot Part 2



```
119010265@node21:~/project1/project1
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 64.1228 seconds
Input Size: 500000
Process Number: 13
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 60.2883 seconds
Input Size: 500000
Process Number: 14
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 59.0904 seconds
Input Size: 500000
Process Number: 15
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 55.9133 seconds
Input Size: 500000
Process Number: 16
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 53.232 seconds
Input Size: 500000
Process Number: 17
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 50.361 seconds
Input Size: 500000
Process Number: 18
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 47.8152 seconds
Input Size: 500000
105, 1 91%
```

Figure 6: Screenshot Part 3



```

119010265@node21:~/project1/project1
Run Time: 47.8152 seconds
Input Size: 500000
Process Number: 19
Sorted.
actual number of elements:500000
Student ID: 119010265
Name: Shi Wenlan
Assignment 1, Parallel version, MPI implementation
Run Time: 45.8376 seconds
Input Size: 500000
Process Number: 20
Sorted.
157, 1 Bot

```

Figure 7: Screenshot Part 4

3.2 How to run my code

If you want to reproduce the experiment above: Open source code folder in the terminal, guarantee that there are enough processors available, then type in the following instruction to run the script.

```
1 /bin/bash compile_run.sh
```

If you want to run the code more flexibly: Open source code folder in the terminal, guarantee that there are enough processors available, then select useful instructions from following manual. Here I take datasize 100 for example, and you can change the number to other datasize, so does the number of processors.

```

1 # Compile the test data generator program
2 g++ -std=c++11 test_data_generator.cpp -o gen
3
4 # Compile the sequential version of odd-even
5 # transposition sort program
6 g++ -O2 -std=c++11 odd_even_sequential_sort.cpp -o ssort
7
8 # Compile the parallel version of odd-even
9 # transposition sort program
10 mpic++ -std=c++11 odd_even_parallel_sort.cpp -o psort
11
12 # Compile the check sorted program
13 g++ -std=c++11 check_sorted.cpp -o check
14
15 # generate 100 test numbers and store in /my_test/100a.in

```

```

16 ./gen 100 ./my_test/100a.in
17
18 # Run sequential sort on ./my_test/100a.in
19 ./ssort 100 ./my_test/100a.in
20
21 # Check the correctness of sequential sort result
22 ./check 100 ./my_test/100a.in.seq.out
23
24 # Run parallel sort on ./my_test/100a.in with 4 processors
25 mpirun -np 4 ./psort 100 ./my_test/100a.in
26
27 # Check the correctness of parallel sort result
28 ./check 100 ./my_test/100a.in.parallel.out

```

3.3 Raw data

Processors \ data size	20000	100000	300000	500000
1	0.607685	15.9266	143.493	397.355
2	0.281416	8.03951	72.1359	199.651
3	0.211088	6.31412	56.284	155.543
4	0.156901	4.96321	45.0547	124.701
5	0.121293	3.9899	37.264	102.484
6	0.196843	3.60571	32.9872	90.4948
7	0.178508	3.29963	29.8806	81.9771
8	0.140797	2.96436	26.8988	74.2174
9	0.128493	2.68091	24.7185	67.6436
10	0.104312	2.70678	22.4267	61.8228
11	0.107034	2.93016	25.8171	69.6461
12	0.105974	2.75103	25.4431	72.2332
13	0.104586	2.57083	21.2811	64.1228
14	0.0953545	2.21643	22.844	60.2883
15	0.171229	2.27931	21.4868	59.0904
16	0.193698	2.08002	20.3927	55.9133
17	0.139354	2.00199	19.3877	53.232
18	0.329843	1.85725	18.254	50.361
19	0.458225	1.80219	17.5685	47.8152
20	0.451025	1.91923	16.6907	45.8376

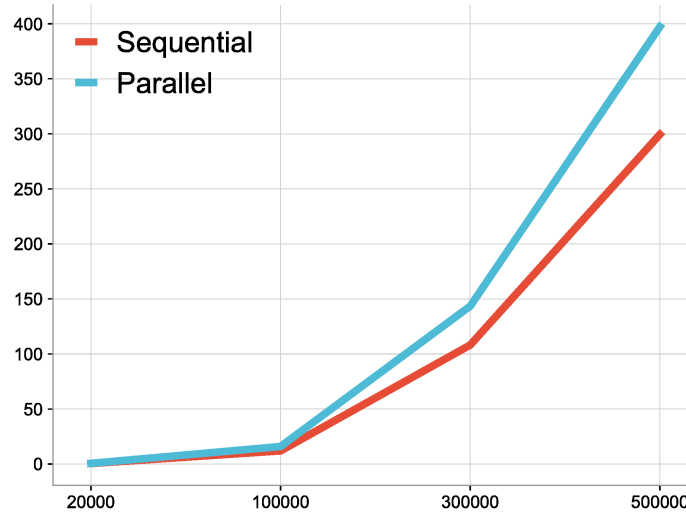
Table 1: Parallel Execution time (seconds)

data size	20000	100000	300000	500000
Execution time (seconds)	0.45495	11.9432	108.151	299.466

Table 2: Sequential Execution time (seconds)

3.4 Data analysis

The following figure compares the execution time of sequential program and parallel program on the same dataset. The horizontal axis is the datasize and the vertical axis is the execution time.

**Figure 8:** Sequential vs. Parallel

Sequential vs. Parallel: It can be seen that even both programs use only one processor, parallel version takes more time than sequential one, and the gap increases as the datasize increases. This is because in each iteration, the parallel version has more branches to check. For example, when skip the first element, the sequential version just go ahead to do other comparisons, while the parallel version need to check whether the first element should be sent to other processors, so does the last element. The larger the datasize, the more such extra checks.

The following figure shows the execution time under different configurations (data-size and number of cores). The horizontal axis is the number of cores, the vertical axis is the execution time, and each line represents a series of data of specific data-size.

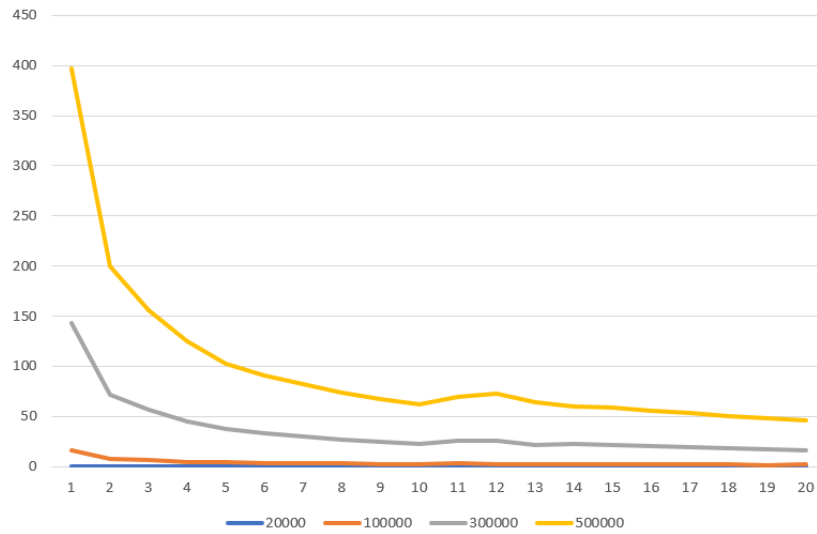


Figure 9: Execution time vs. Different Datasize and number of Processors

The following figure shows the speedup (sequential execution time (parallel versing using 1 processor) / parallel execution time) under different configurations (datasize and number of cores). The horizontal axis is the number of cores, the vertical axis is the speedup, and each line represents a series of data of specific datasize.

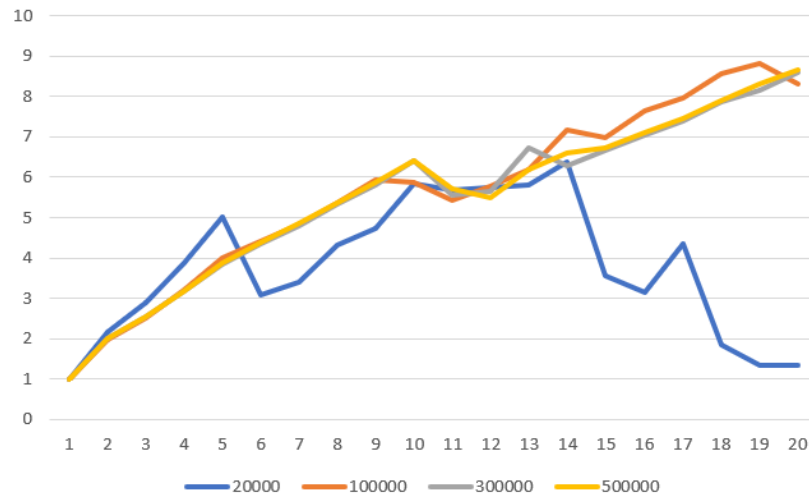


Figure 10: Speedup vs. Different Datasize and number of Processors

The number of cores vs. Execution time: It can be seen that in general cases, execution time decreases as the number of cores being used increases, and this effect

is apparent when there are only a few cores being used. It shows the diminishing marginal utility. This is because when the number of cores being used increases, the data scale each processor processes decreases, and the extent of reduction also decreases. On the other hand, the total overhead of message communication between processors increases, which further reduces the marginal benefit and even makes it negative.

Datasize vs. Execution time It can be seen that when datasize is relatively large, the reduction in execution time is significant, and the speedup increases steadily when the number of cores increases. However when datasize is small, the reduction in execution time is trivial, and the speedup is also unstable and finally decreasing when the number of cores increases. This is because the overhead of message communication is only related to the number of processors, so it is a relatively fixed cost. While the reduction of data scale each processor processes increases as the total datasize increases, which means the benefit is flexible. Therefore, when the datasize is small, the benefit can hardly cover the overhead, while when the datasize is large, the net benefit is more apparent.

Unusual surge in execution time An unusual surge in execution time when the number of cores being used increases from 10 to 11 is observable. It is probably because the the cores are allocated in different nodes, the overhead of message passing increases a lot due to this and even cover the benefit.

The following figure shows the Cost-efficiency (speedup / number of cores being used) under different configurations (datasize and number of cores). The horizontal axis is the number of cores, the vertical axis is the Cost-efficiency, and each line represents a series of data of specific datasize.

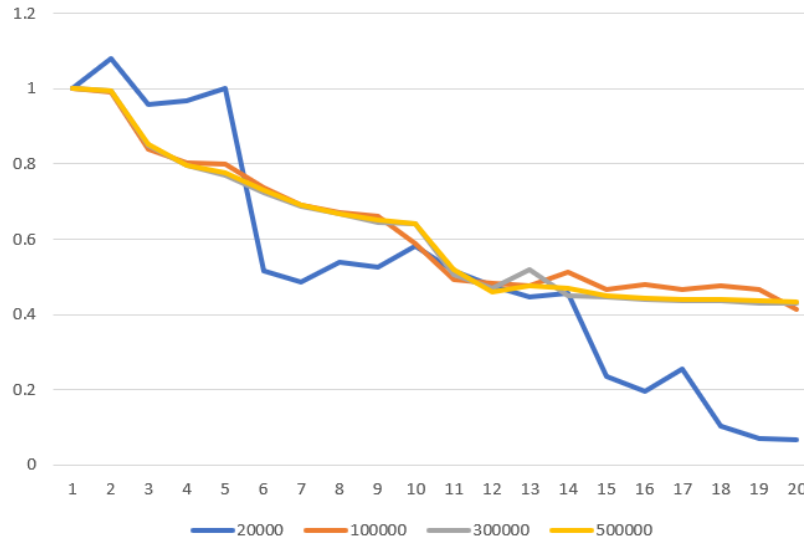


Figure 11: Cost-efficiency vs. Different Datasize and number of Processors

The number of cores vs. Cost-efficiency It can be seen that when datasize is relatively large, the Cost-efficiency keeps decreasing as the number of cores being used increases, and gradually converges to about 0.4.

4 Conclusion

My experiment results show that parallel can greatly speedup the sort task when datasize is large, and the number of cores being used are not so large given the existance of diminishing marginal utility. In this environment, I recommend use 10 processors at once when datasize is large (greater than 300000), since both the speedup and the cost-efficiency decrease obviously when more than 10 cores are used. Besides, the sequential odd-even transposition sort is of $O(n^2)$, but with enough cores (such as sort n numbers using n cores), the parallel version can approach $O(n)$.