CSC4005 Project 1 Report

Odd-Even Transposition Sort

Name: 李沛霖 Li Peilin

ID: 119020026

2022.10.12

# Introduction

This report implements the odd-even transposition sort method both in sequential and parallel manners, and investigates their performances with different inputs.

# How to Use My Program

## Compiling the Sequential Version Program

Run the following command in the terminal......

g++ odd\_even\_sequential\_sort.cpp -o myssort

...... then it will generate a binary executable file called myssort

## Running the Sequential Version Program

Change the template.sh file to the following (fill the {} with your choice of parameters) ......

#!/bin/bash

cd {location\_of\_source\_code}

./myssort {input\_size} {input\_file}

...... and then use the sbatch template.sh command to submit it. Suppose the input file is called 10000.in, then the sorted array will be in he 10000seq.out file.

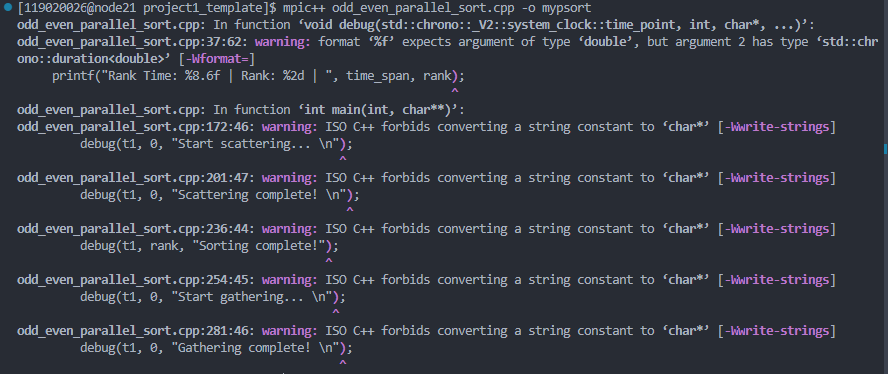
## Compiling the Parallel Version Program

Run the following command in the terminal......

mpic++ odd\_even\_parallel\_sort.cpp -o mypsort

...... then it will generate a binary executable file called mypsort

Note: some warnings may show up when you compile this program. From my experience, they do not affect the execution of the program.



## Running the Parallel Version Program

Change the template.sh file to the following (fill the {} with your choice of parameters) ......

#!/bin/bash

#SBATCH --job-name mypsort

#SBATCH --nodes=1

#SBATCH --ntasks={num\_proc}

#SBATCH --cpus-per-tasks=1

#SBATCH --time={time\_limit}

#SBATCH --partition={partition}

#SBATCH --output=%x.%j.out

cd {location\_of\_source\_code}

mpirun -np {num\_proc} ./mypsort {input\_size} {input\_file}

...... and then use the sbatch template.sh command to submit it. The printing and slurm messages will be in the mypsort.{jobid}.out file. Suppose the input file is called 10000.in, then the sorted array will be in he 10000par.out file.

# Methods

## Algorithm Design

### Standard C++ Sort

This method just use the std::sort function to sort the input array as a control group. This sorting method will be considered as the best sorting sequential algorithm for performance measurement purposes.

### Sequential Odd-Even Transposition Sort

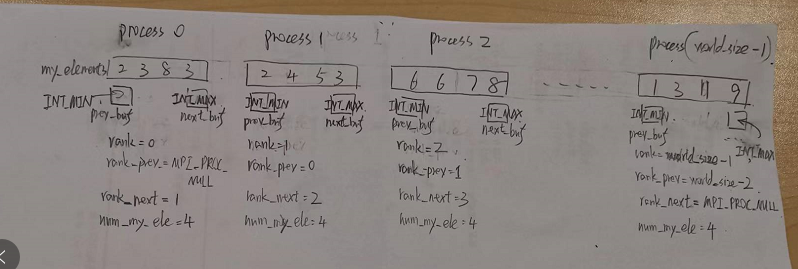
The sequential version is based on the

### Parallel Odd-Even Transposition Sort

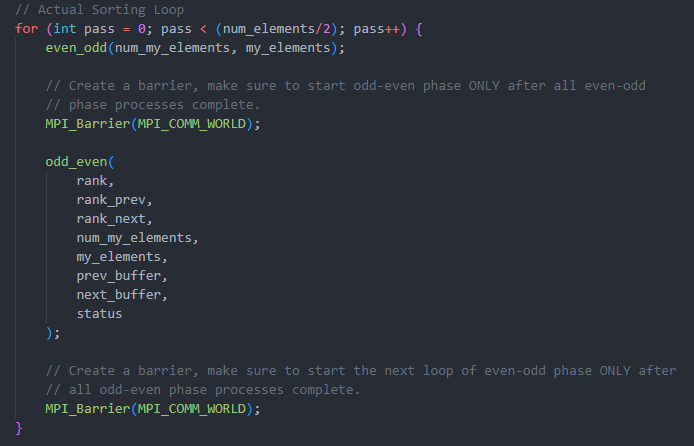
Demo and flow chart:

Suppose the first 12 unsorted elements are 2, 3, 8, 3, 2, 4, 5, 3.

After (1) variable declaration, (2) read input file to elements[], (3) printing information (e.g. student name, ID.....), (4) start timing, and (5) scattering elements[] to each process’s my\_elements, the stack should look like this: (the elements[] variable, sorted\_elements[] in master process are omitted)

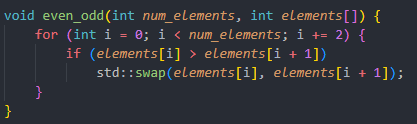


Then the program goes into the actual sorting loop:

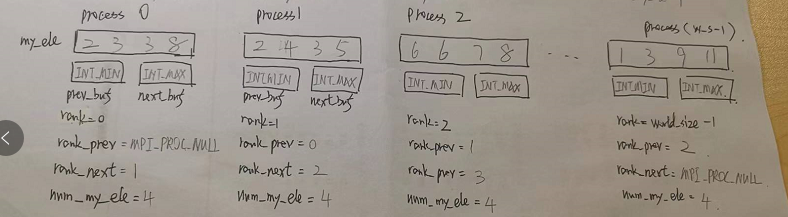


Note that in this loop and many other places, I use MPI\_Barrier() to make sure the relevant operations are synced. About the specific use of each MPI\_Barrier(), please read the corresponding comment in the source code.

The first phase is even-odd phase. This is easy, just sort elements with in each process, not inter-process communication needed:



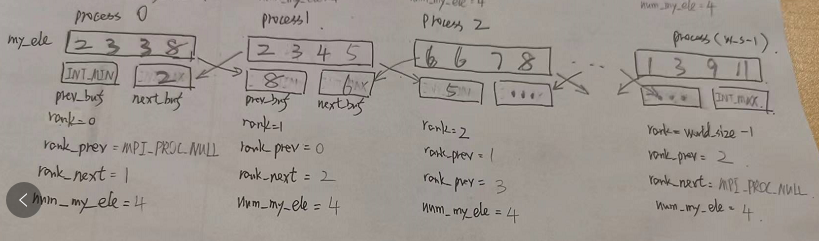
After even-odd phase sort, the stack should look like this:



Then the program goes to odd-even phase, which involves both sorting (1) within the current process elements and (2) between the current process and it neighbouring processes:

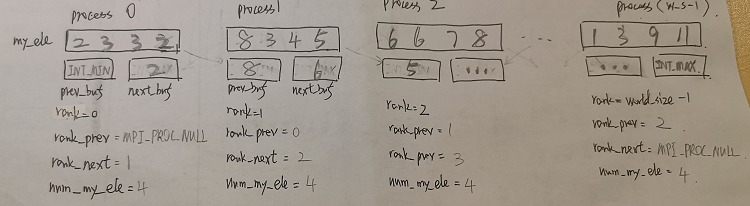


After the sorting within current process and the 2 MPI\_Sendrecv() routines, the stack should look like this:



We can see that using the 2 MPI\_Sendrecv() routines, the program has sent the last element of the previous process to the current processes’ prev\_buffer, and has sent the first element of the next previous process to the current processes’ next\_buffer. **Now, we can compare the head and tail of neighbouring processes!!!**

After the 2 comparison-and-assign codes are executed, the program stack should look like this:



Now, we have completed a even-odd phase and a odd-even phase. Then the loop goes on. In total, the loop executes world\_size / 2 times of even-odd phase and world-size / 2 times of odd-even phase, totaling world-size times phases.

Let me explain why I choose to run even-odd phase N/2 times and odd-even phase N/2 times: let’s define a odd-even phase sorting phase or a even-odd phase as a pass. As A. Nico Habermann had proven in his paper in 1927, with an input array with length of N, the parallel sorting algorithm will guarantee to produce the sorted result in N passes. Therefore, in order to (1) omit the algorithm design of “determining whether the current array is sorted or not” and (2) make sure the algorithm does EQUAL number of comparisons among same settings of input but different number of processes to make the outcome more comparable, and (3) to simulate better for the worse case scenarios, I just do N passes WITHOUT tring to determine whether the final result (or the intermediate result) is sorted or not during the sorting.

After the actual sorting loop, we are ready to use MPI\_Gather() to gather the sorted array from each process to sorted\_elements[] in the master process.

Then we stop timing, calculate elapsed time and print it out.

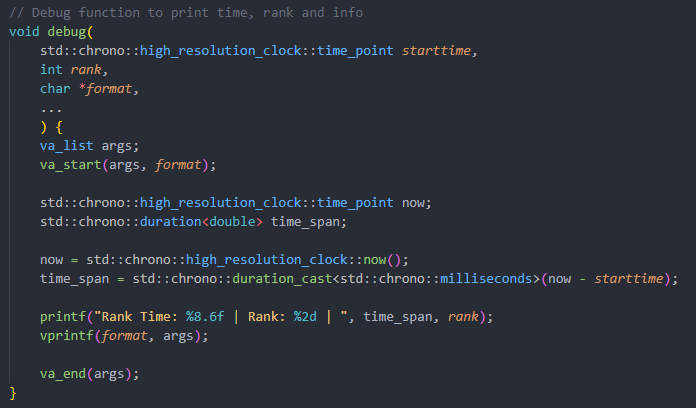
Then we print out the first 20 elements of the whole sorted array as requested in the project.

Then we store the sorted array in an designated output file.

That’s all of the parallel version algorithm.

### Miscellaneous

I created a debug function to print rank time (rank time means the time since the current process / rank has started), rank and other information (inputted on demand) to facilitate debugging. If you want the debug functions to work, follow the instructions specified in the source code comment.



Note: If you use the debug function in ranks other than rank 0, you should realize that the t1 in that rank doesn’t equal to the t1 of rank 0 (which is the starting time of the main timer). That’s why I wrote Rank Time in printf(“Rank Time: ”......) , because that time printed is relative to the current rank’s t1, not rank 0’s t1. The current processes’ t1 should be slightly later than t1 in rank 0. However, that time difference is very small, so I didn’t do anything to make sure the current rank’s t1 = rank 0’s t1.

## Input Generation

The generation of random array of integers is done by a program called test\_data\_generator provided in the project template.

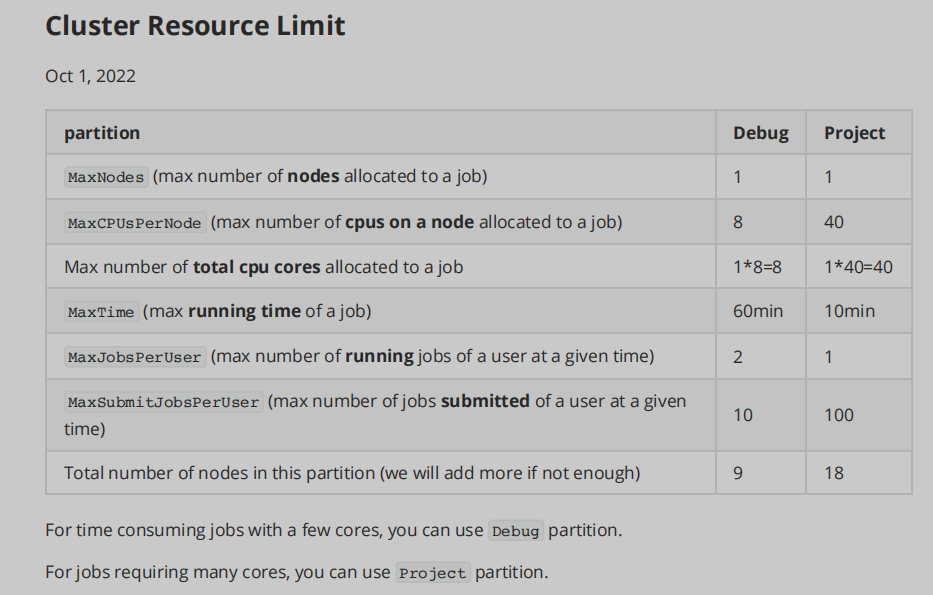
Note that the input sizes are generated in such a way that guarantees num\_my\_elements are always even. This is to simplify the algorithm design of the actual parallel odd-even transposition sort. Otherwise, the parallel version must behave differently with odd length input and even length inputs. Since out focus is on the performance of different implementations, we just always feed the program with even size input.

## Timing

Timing is implemented by the std::chrono::high\_resolution\_clock related functionalities (also inspired by the code from project template). We start timing in the master process after declaring necessary variables and printing out general information (e.g. student name and ID). We stop timing at the end of the sorting program in the master process. Elapsed time is then calculated as the time difference between starting time t1 and ending time t2.

## Experiment Settings

The HPC allows the following configurations:

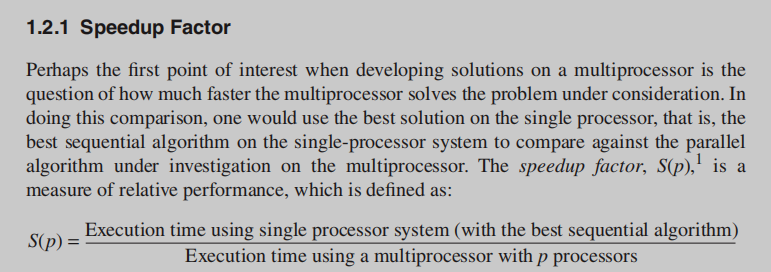


So I decide to use 2, 4, 8, 16 and 32 processes to test each size of input with parallel version algorithm.

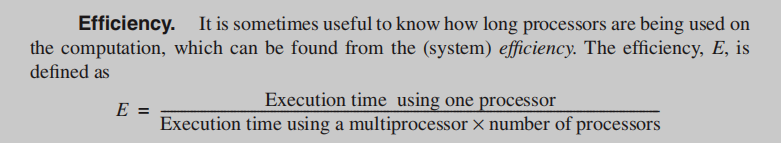
## Performance Analysis

In this project, we use the following metrics to measure the performance of different sorting methods: (1) the elapsed time, (2) the speedup factor

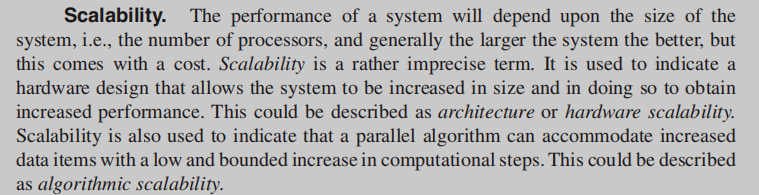
The speedup factor is defined on page 6 in the textbook:



The efficiency is also defined on page 7 in the textbook:

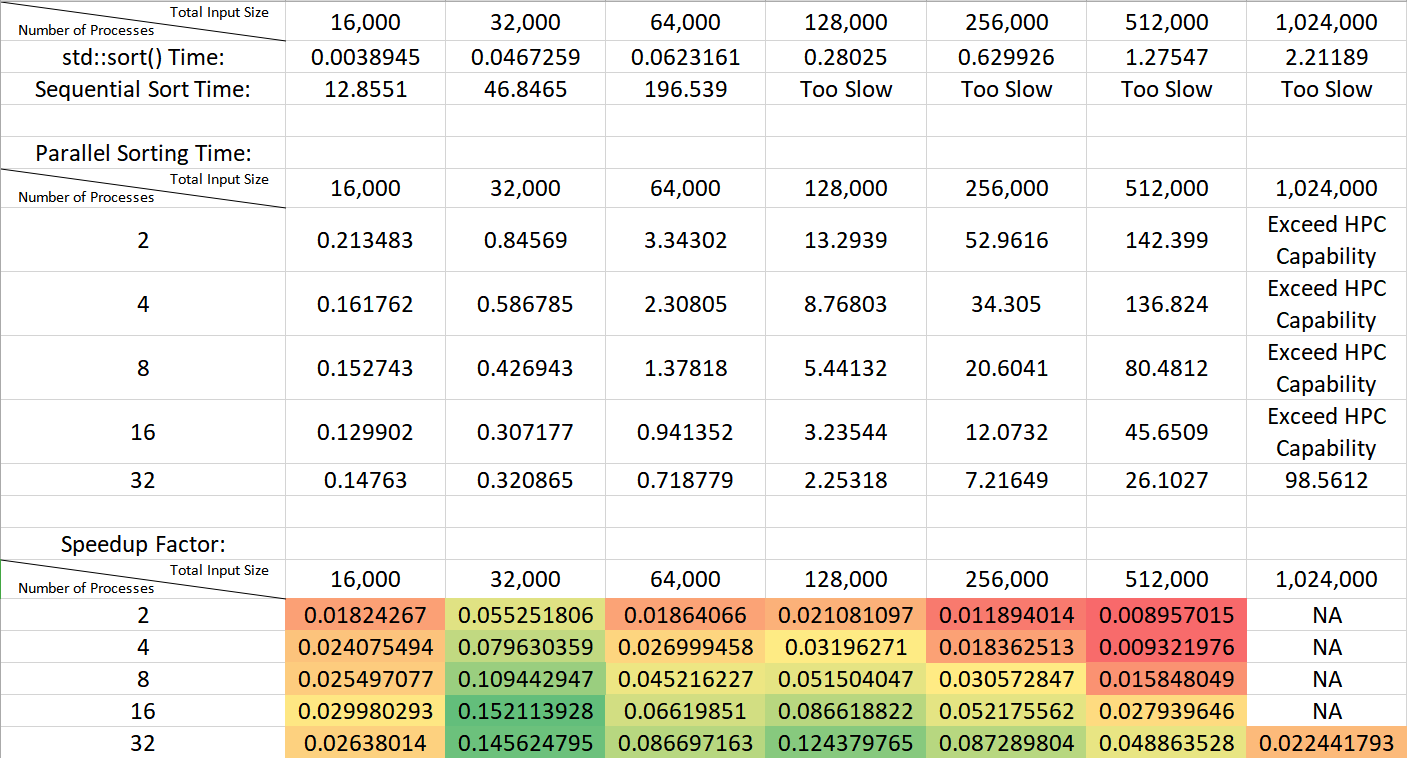


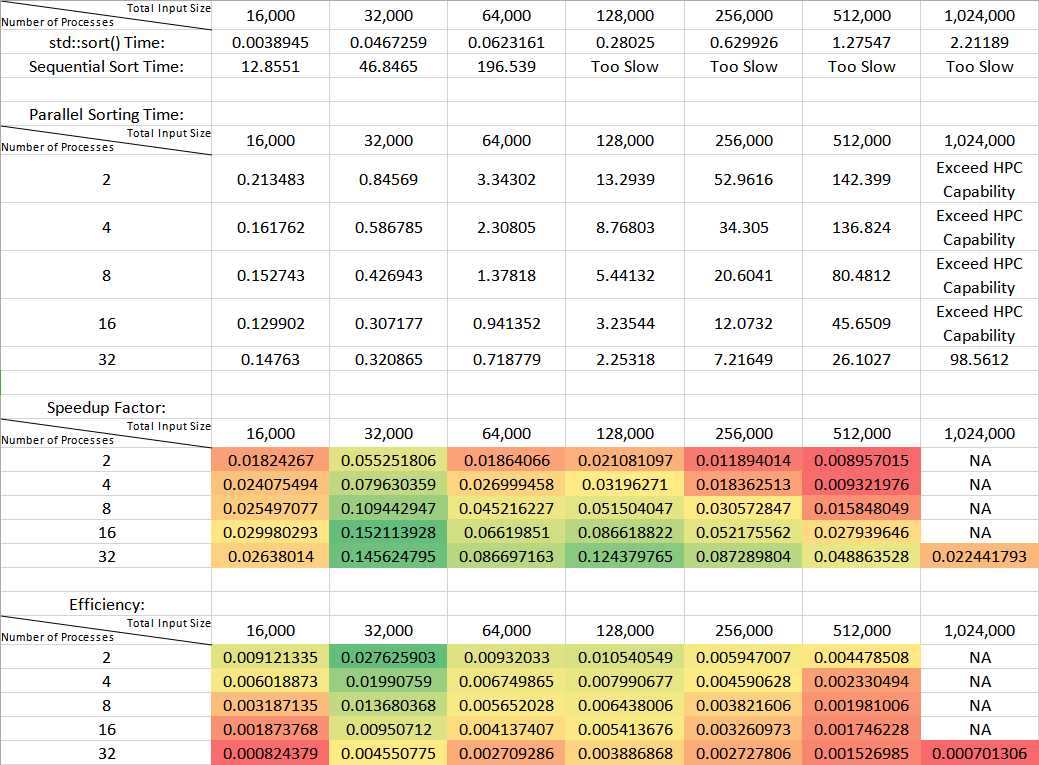
The scalability is loosely defined on page 11:



Here we just use algorithmic scalability as a measure.

# Results and Performance Analysis



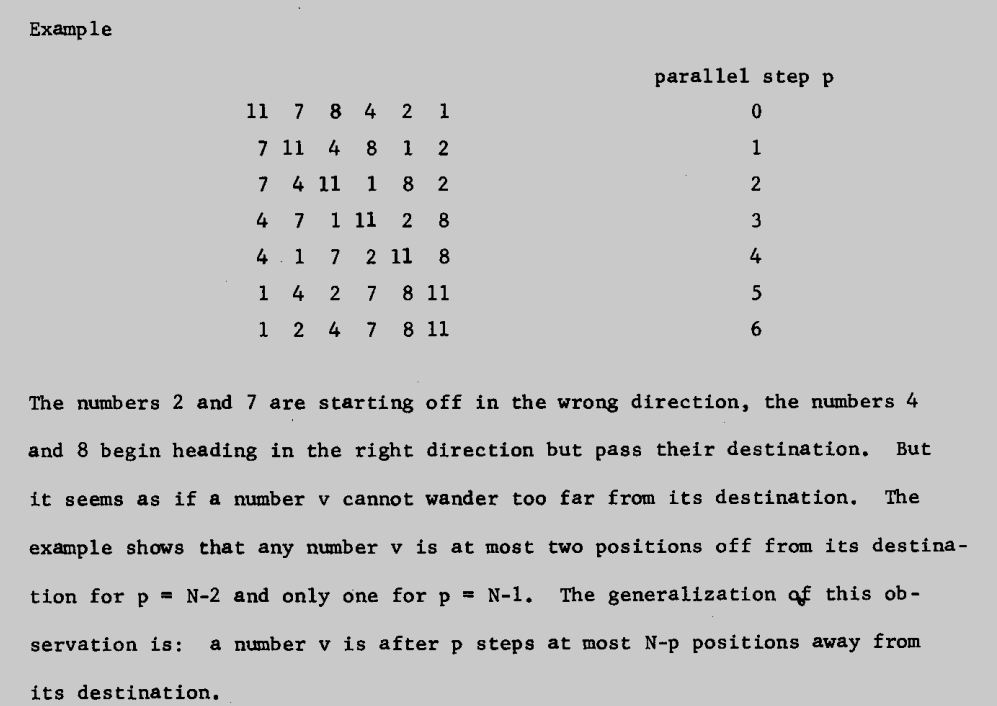


## Using Standard Sorting Algorithm in C++

## Using Sequential Odd-Even Transposition Sort

We can easily observe that the sequential odd-even transposition sort is VERY slow. This is primarily due to the (1) sequential and (2) inefficient nature of this algorithm.

Why is this algorithm naturally inefficient? We can take a look at an example of A. Nico Habermann’s paper:



As described in this paper, during the sorting, some numbers will initially “head for the wrong direction”, then “head for the right direction and reach their final place”. **This feature of the odd-even sorting algorithm is a HUGE drawback on its efficiency. We can observe this pattern in both sequential odd-even transposition sorting algorithms and parallel odd-even transposition sorting algorithms.**

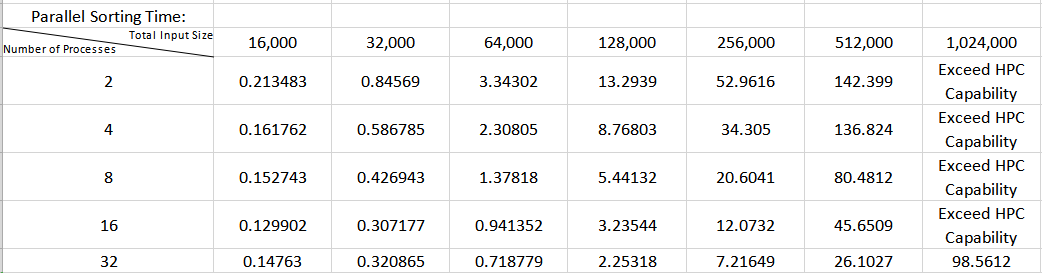
We can also observe that given the same number of processes, as the input size doubles, the sorting time approximately quadruples. This is because the number of operation of a odd-even transposition sorting algorithm is the product of input size and number of passes. And as we discussed, I set the total number of passes equal to the input size. Therefore, as the input size doubles, the sorting time approximately quadruples.

## Using Parallel Odd-Even Transposition Sort

We can observe that given the same number of processes, as the input size doubles, the sorting time approximately quadruples. This is the same as the sequential case.

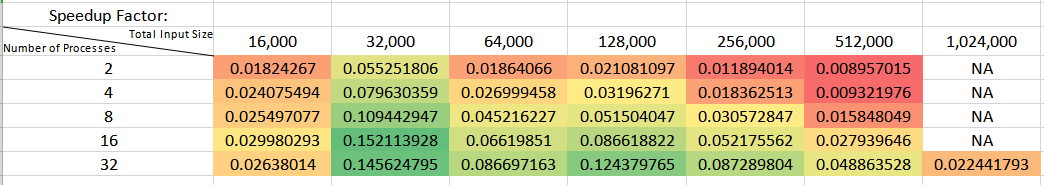
We can observe that give the same total input size, as the number of processes doubles, the decrease in sorting time decreases. In some cases, the elapsed time even increase as the number of processes increase.

This can be explained by the increasing time cost of creating, maintaining processes and inter-processes communications.



From the screenshot below, by columns, we can observe that as the input size increase, the speedup factor of the algorithm increases shortly but eventually decreases greatly.

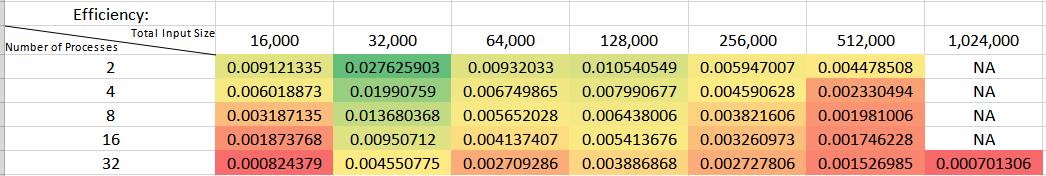
The reason is that: as the input size increases, the “primitive”, “neighbour-exchanging” nature of odd-even transposition sort’s drawback worsens more and more when compared with other sorting algorithms (e.g. quicksort, which is usually used by std::sort()).



4.

From the screenshot below, by columns, we can observe that as the input size increase, the efficienty of the algorithm increases shortly but eventually decreases greatly.

The reason is similar to the reason of speedup factor decrease.



# Conclusion and Discussion

Generally speaking, the parallel odd-even sorting algorithm are inferior to the standard C++ sorting algorithm due to their primitive “exchanging neighbours” sorting mechanism.

A possible way to improve that may be to change the sorting mechanism within each process. For example, we may change that to quicksort.

# References

Wilkinson, P. (2006). Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2/E. Pearson Education India.

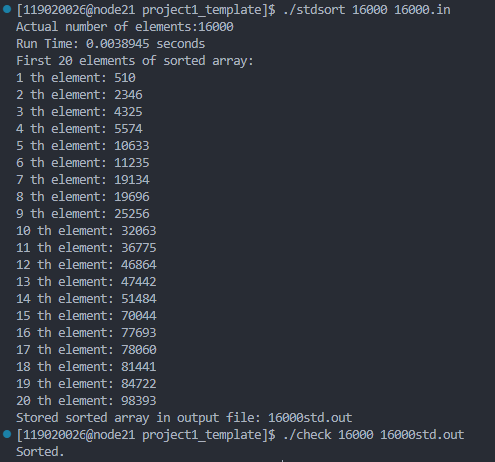
1. Haberman (1972) "Parallel Neighbor Sort (or the Glory of the Induction Principle)," CMU Computer Science Report (available as Technical report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Rd Springfield VA 22151).

Link to the paper: <https://kilthub.cmu.edu/articles/journal_contribution/Parallel_neighbor-sort_or_the_glory_of_the_induction_principle_/6608258>

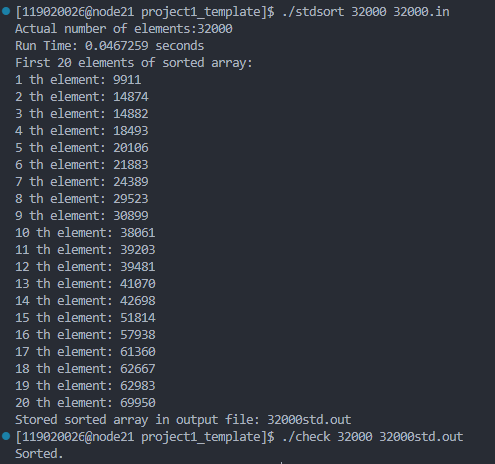
# Screenshots of Outputs

## Using std::sort

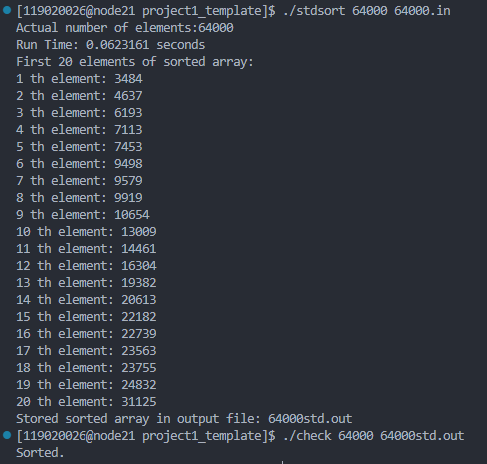
### 16000 inputs



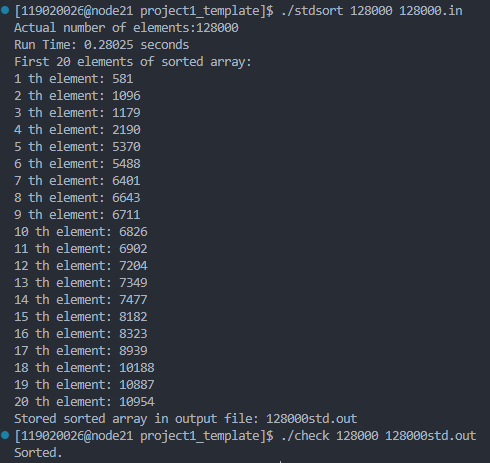
### 32000 inputs



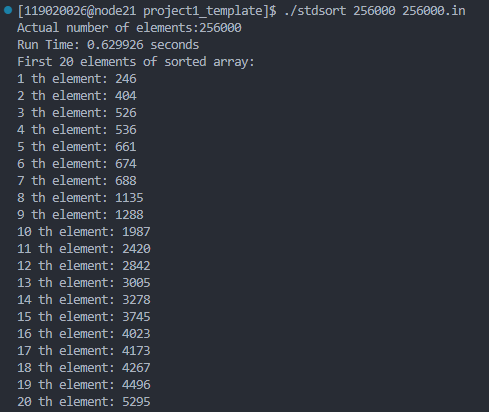
### 64000 inputs



### 128000 inputs

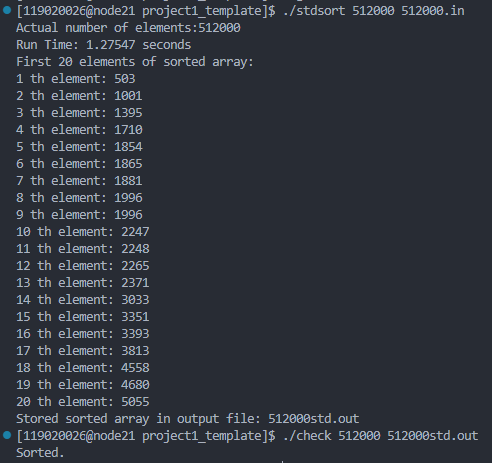


### 256000 inputs

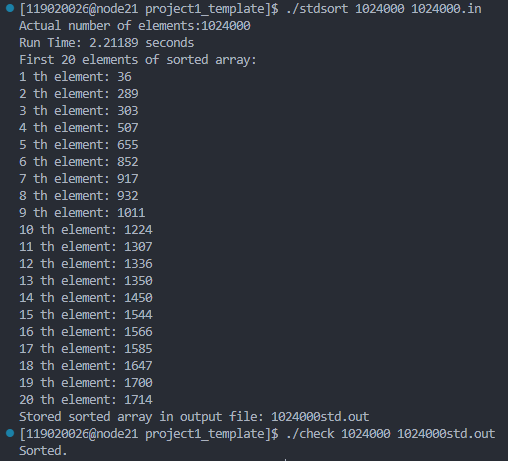




### 512000 inputs

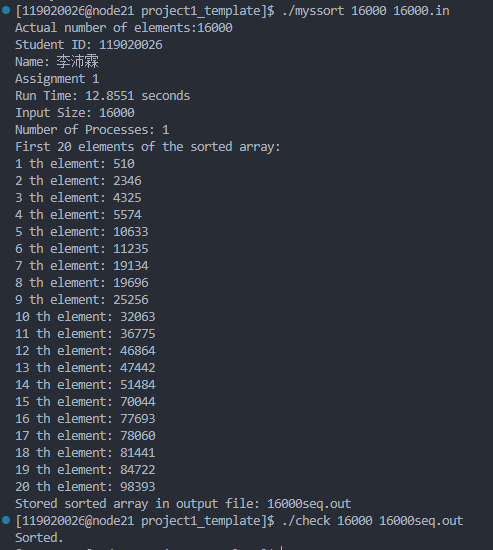


### 1024000 inputs

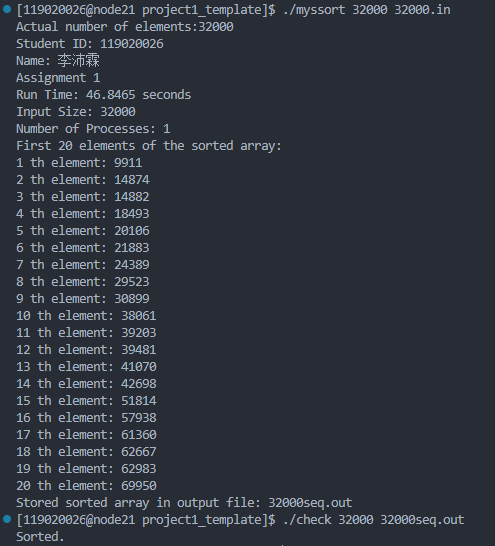


## Using Sequential Version

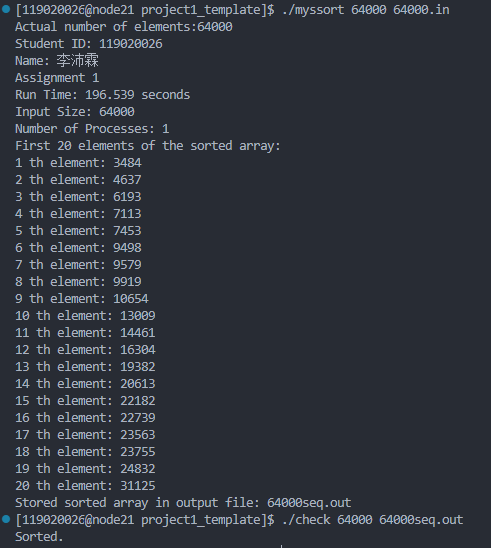
### 16000 input



### 32000 inputs



### 64000 inputs



### 128000 inputs

### 256000 inputs

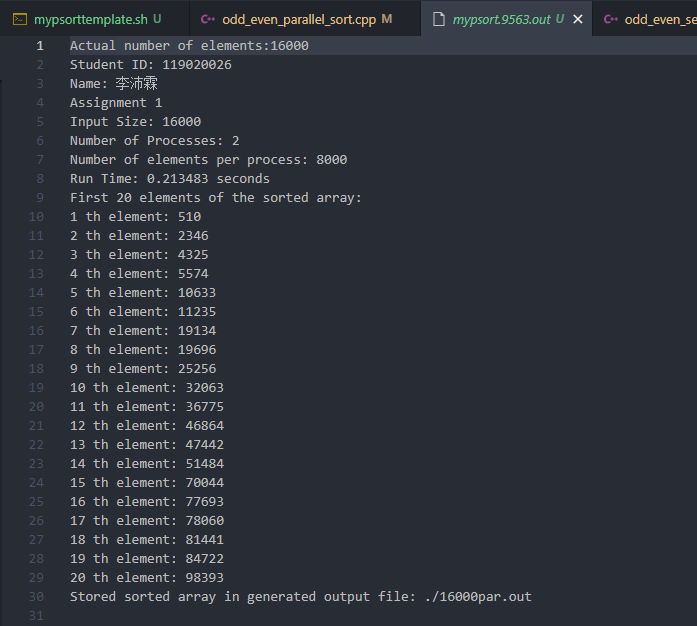
### 512000 inputs

### 1024000 inputs

## Using Parallel Version

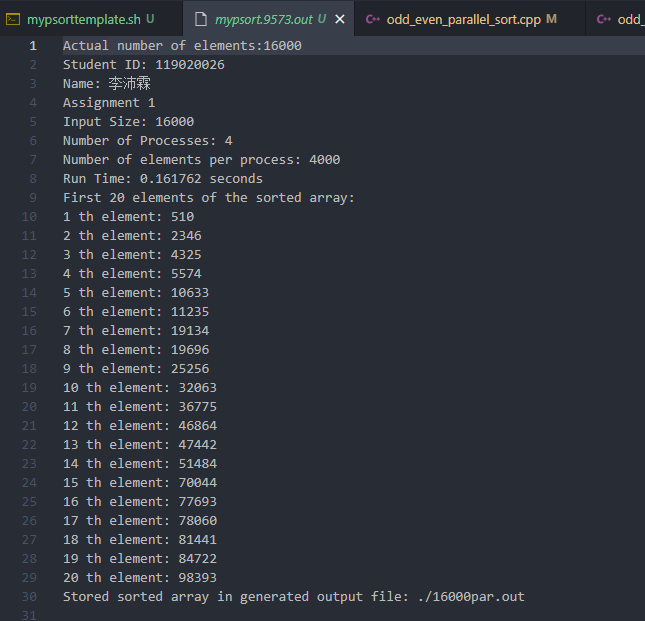
### 16000 inputs

#### 2 processes

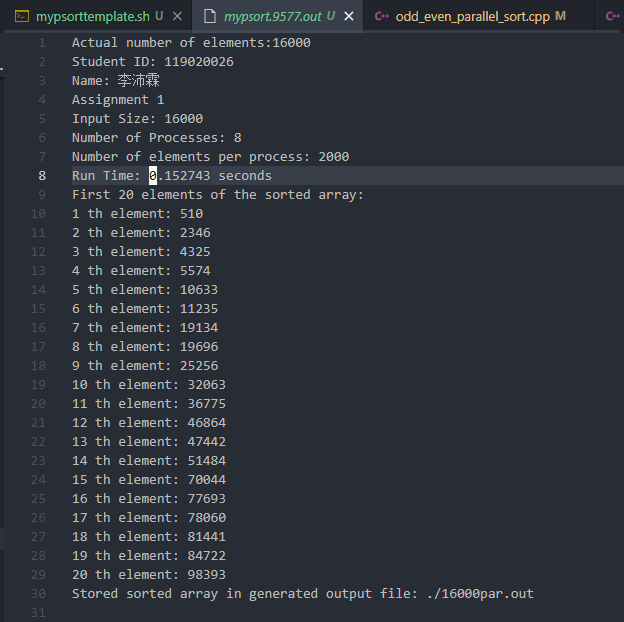




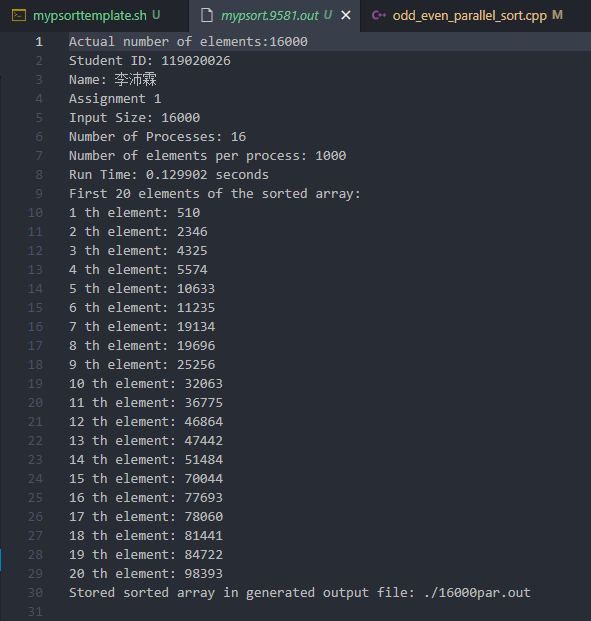
#### 4 processes



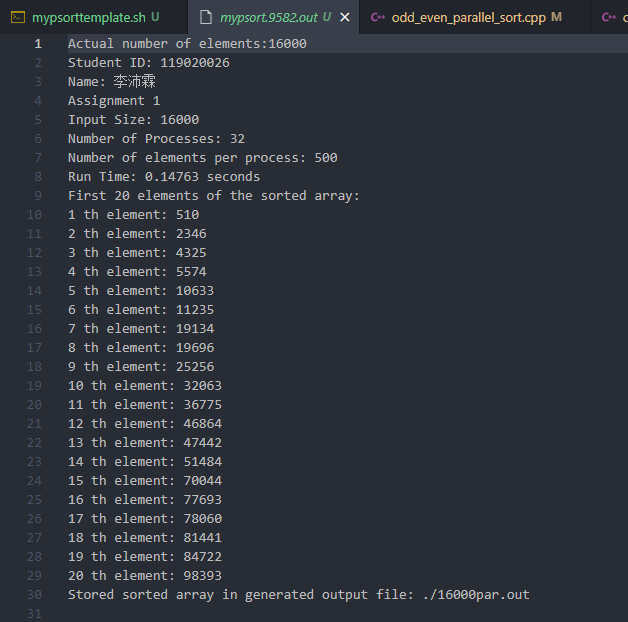
#### 8 processes



#### 16 processes

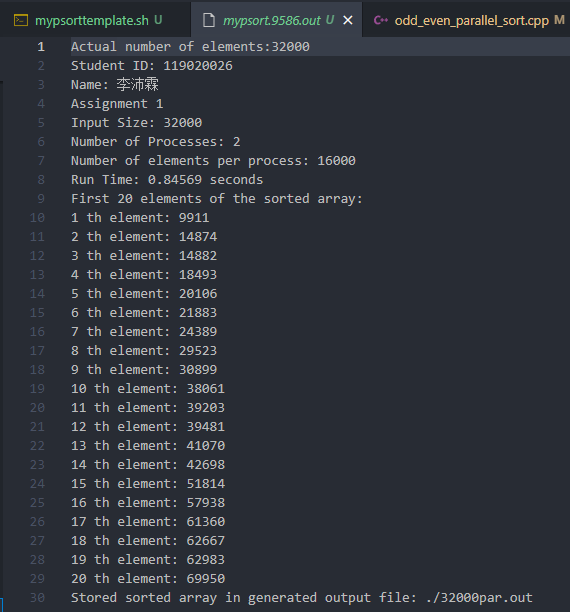


#### 32 processes

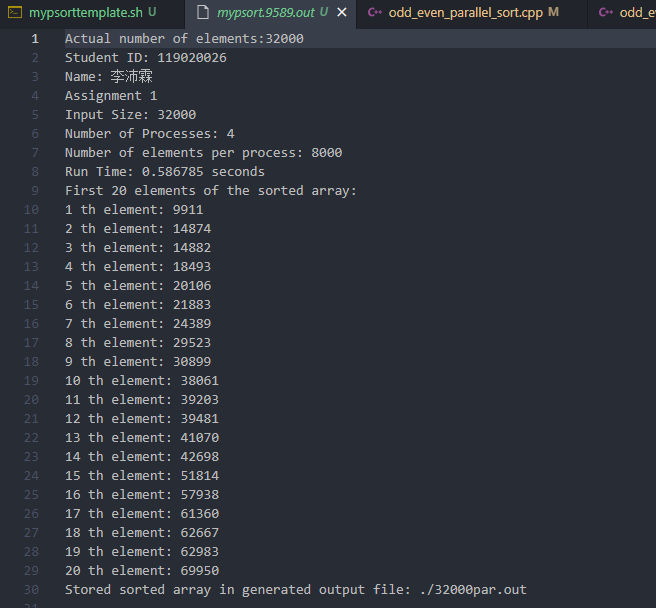


### 32000 inputs

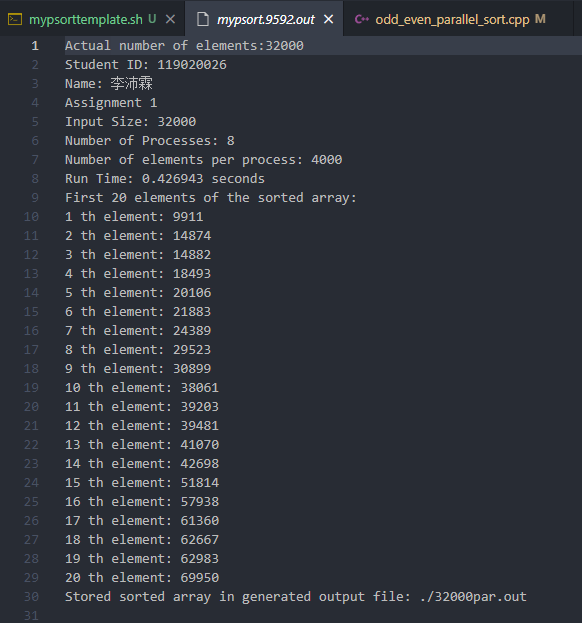
#### 2 processes



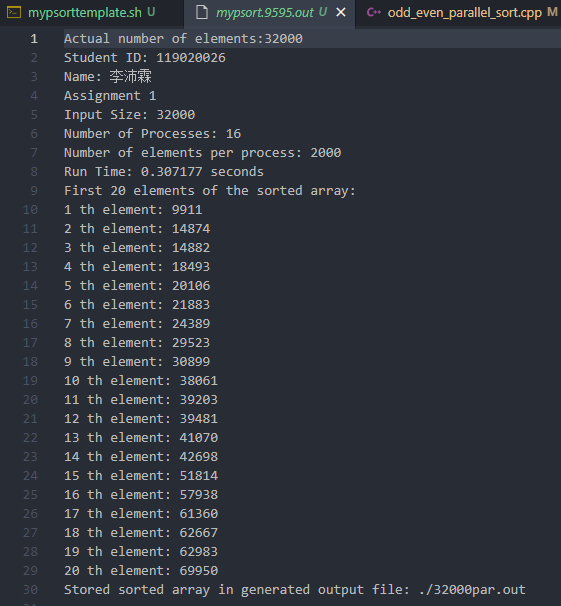
#### 4 processes



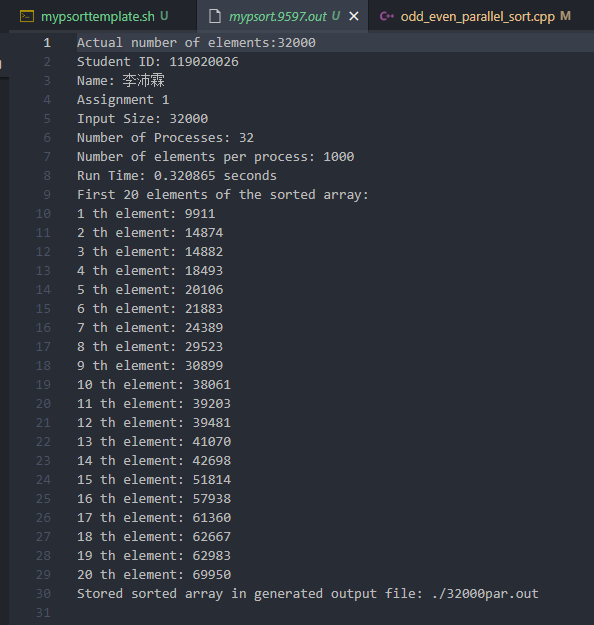
#### 8 processes



#### 16 processes

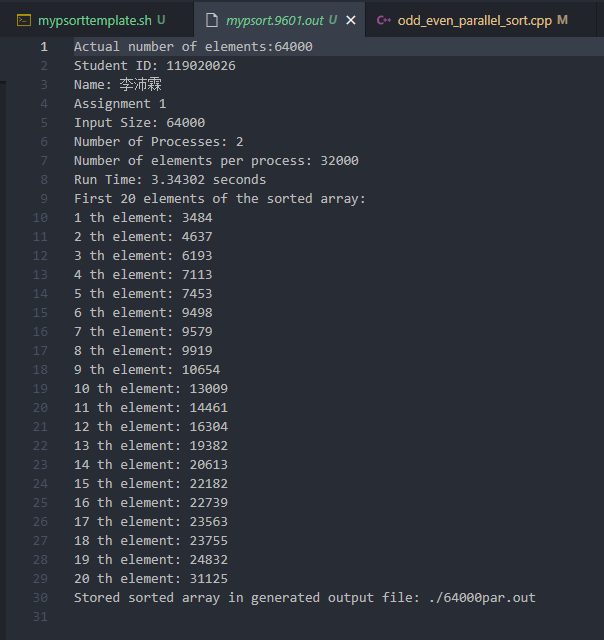


#### 32 processes

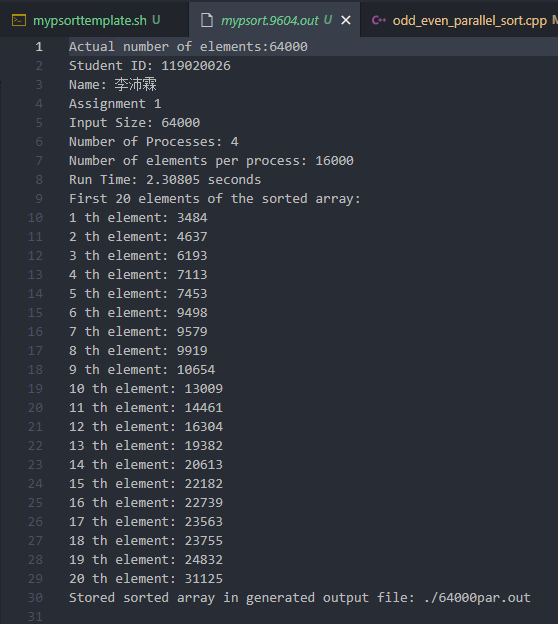


### 64000 inputs

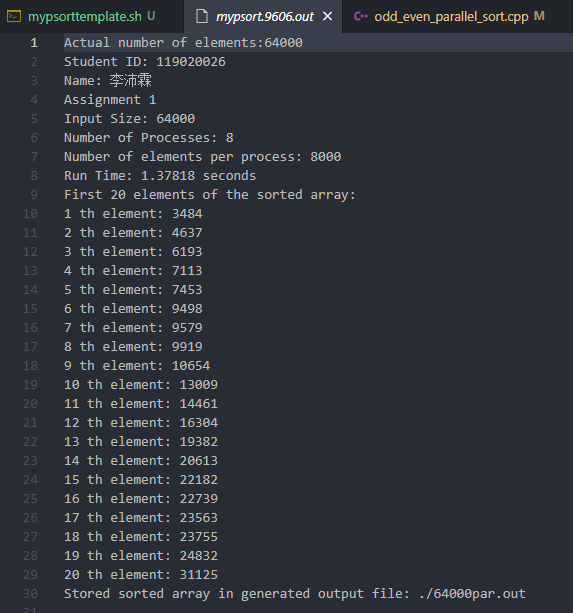
#### 2 processes



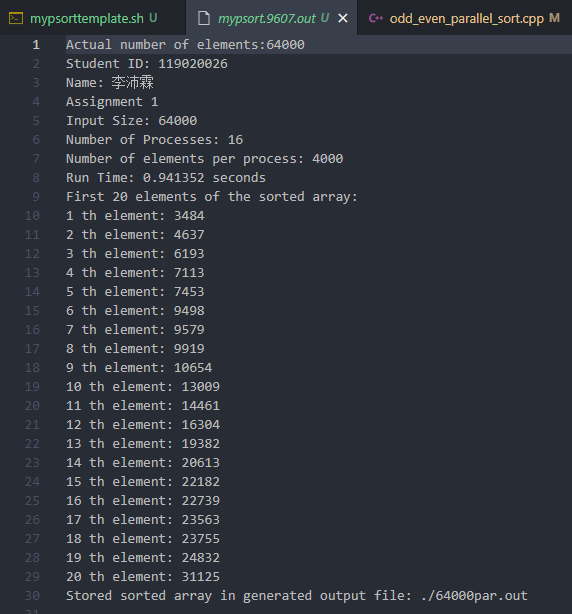
#### 4 processes



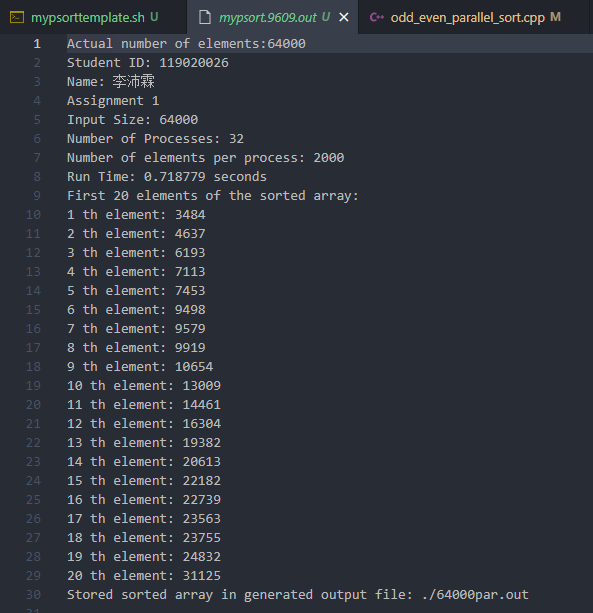
#### 8 processes



#### 16 processes

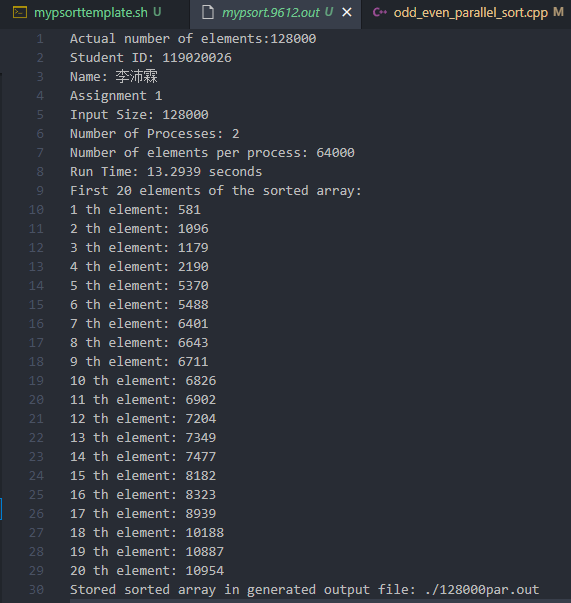


#### 32 processes

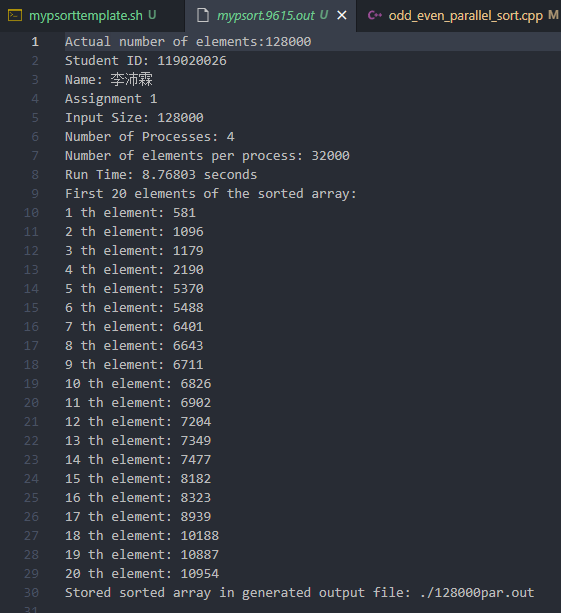


### 128000 inputs

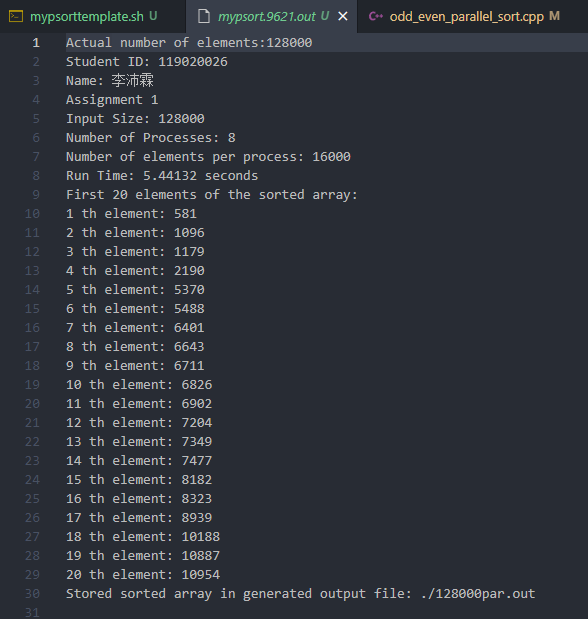
#### 2 processes



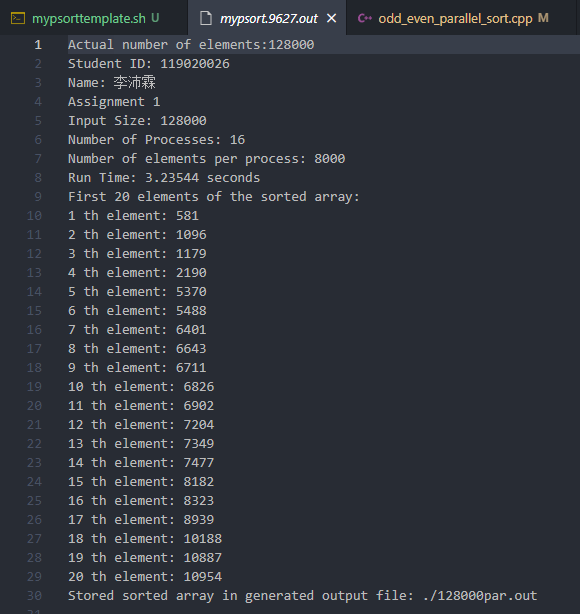
#### 4 processes



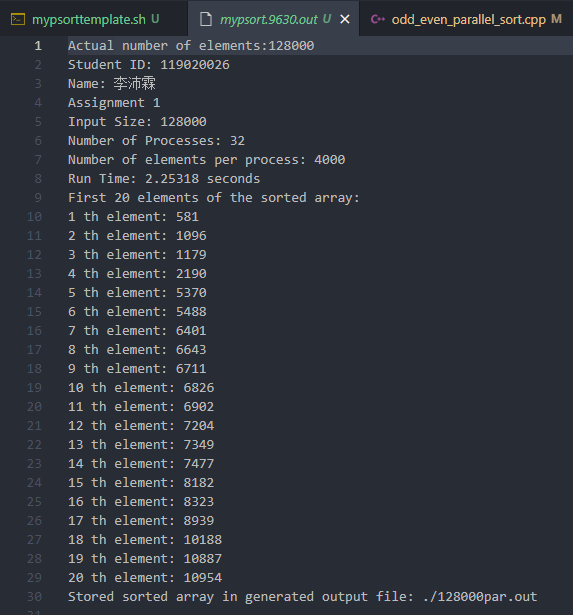
#### 8 processes



#### 16 processes

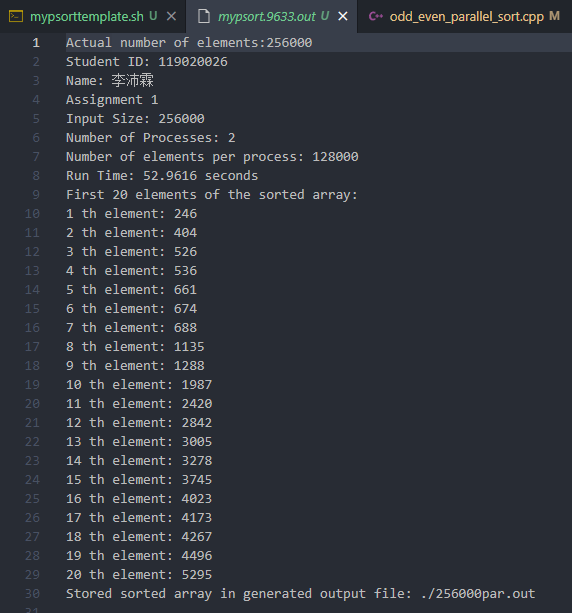


#### 32 processes

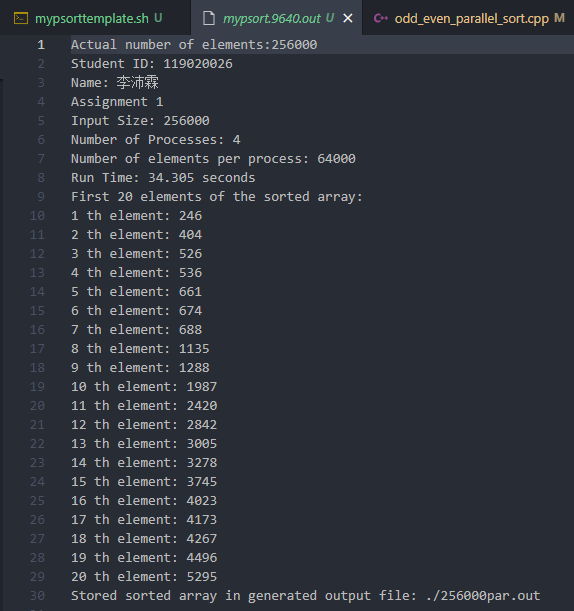


### 256000 inputs

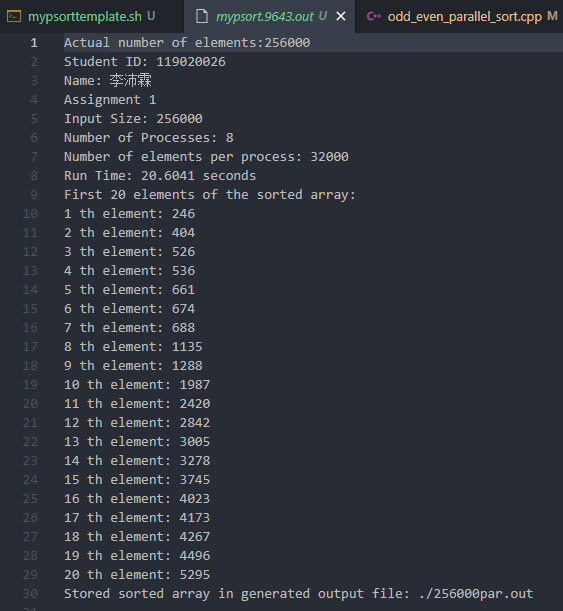
#### 2 processes



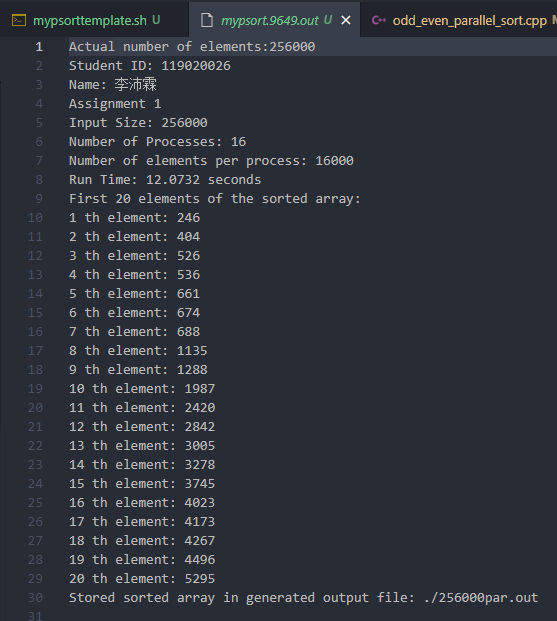
#### 4 processes



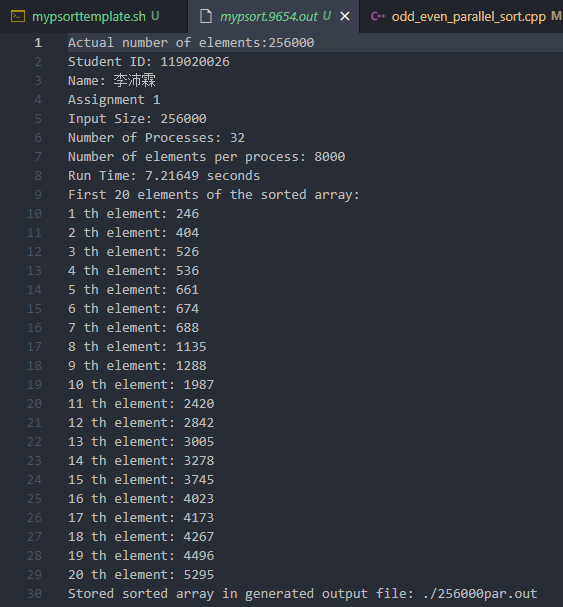
#### 8 processes



#### 16 processes

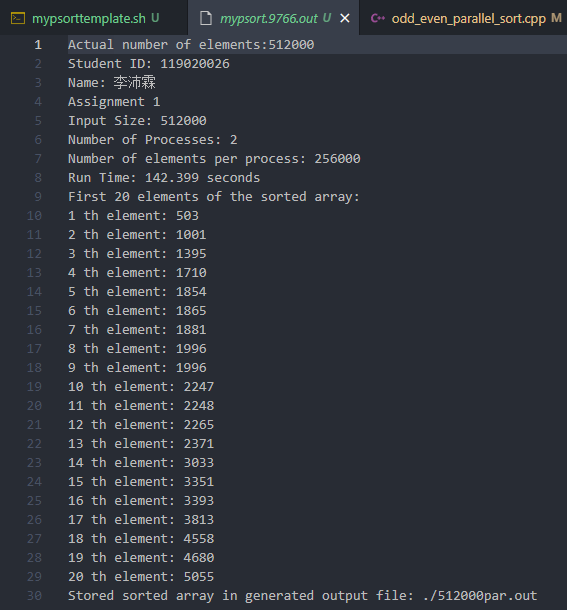


#### 32 processes

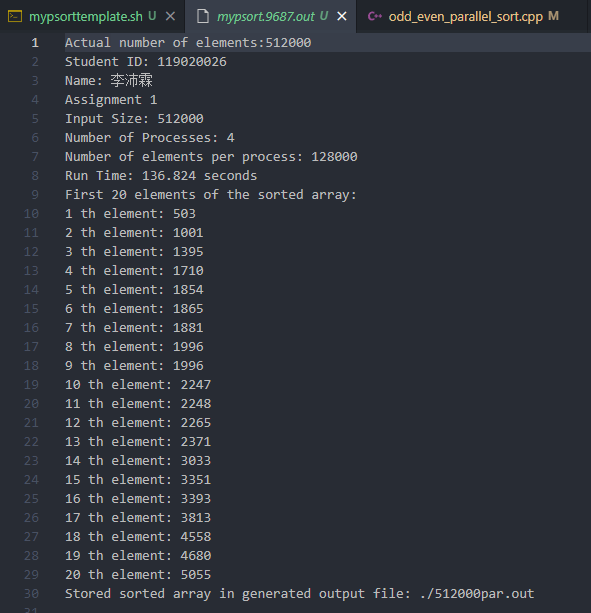


### 512000 inputs

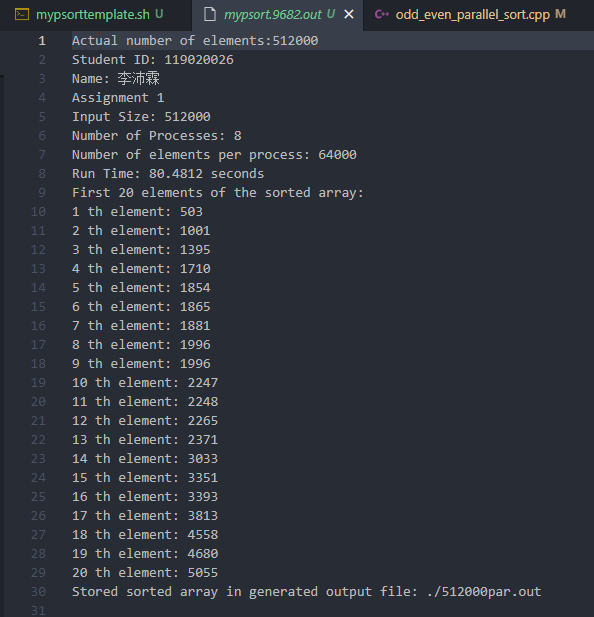
#### 2 processes



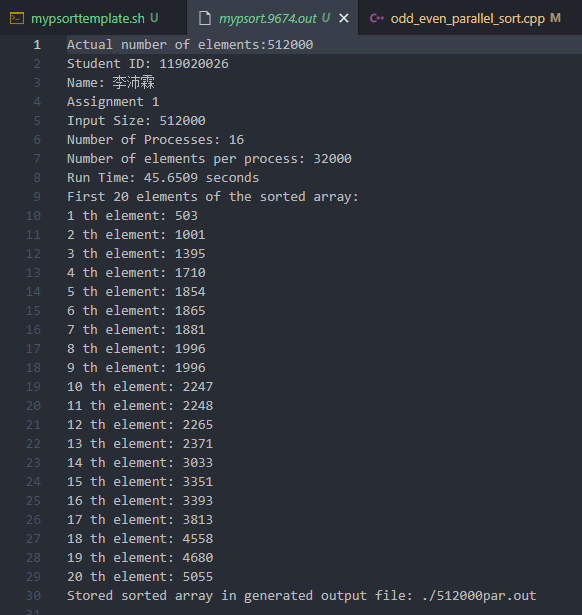
#### 4 processes



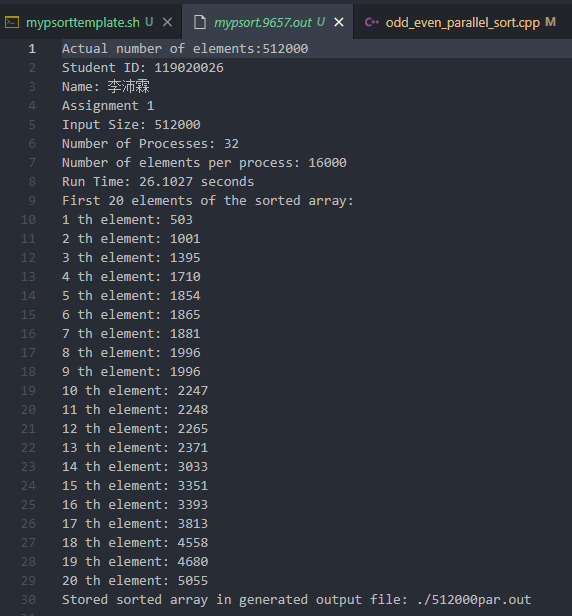
#### 8 processes



#### 16 processes



#### 32 processes



### 1024000 inputs

#### 2 processes

#### 4 processes

#### 8 processes

#### 16 processes

#### 32 processes

