CSC4005 Assignment1 Report

Introduction

The objective of this assignment is to write a **Parallel Odd-Even Transposition Sort** program by using MPI.

In this experiment, the sequential version and the parallel version of Odd-Even Transposition Sort were implemented respectively. This report is going to introduce the implementations of this sort method and analyse the results we generated.

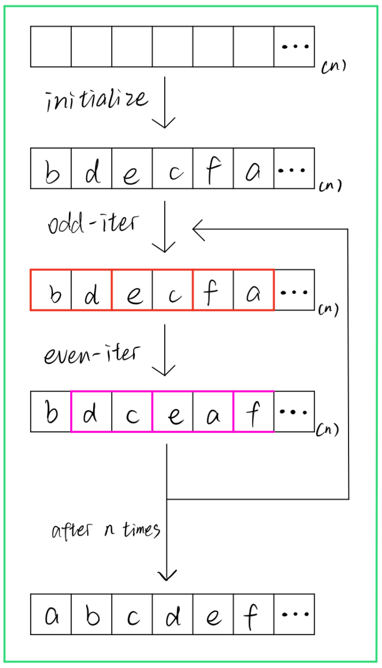
Program Design

Sequential Implementation

As we know, the Sequential Odd-Even Transposition Sort could sort an array of numbers with the following steps:

1. In odd iteration, compare the odd element with the posterior even element, and swap them if the posterior even element is smaller.
2. In even iteration, compare the even element with the posterior odd elements, and swap them if the posterior odd element is smaller.
3. Repeat the step 1-2 until the array is sorted.

It is simple enough so we could just follow these steps to implement our program, as the following logic figure. (The comparisons happen in the red and the purple frames)

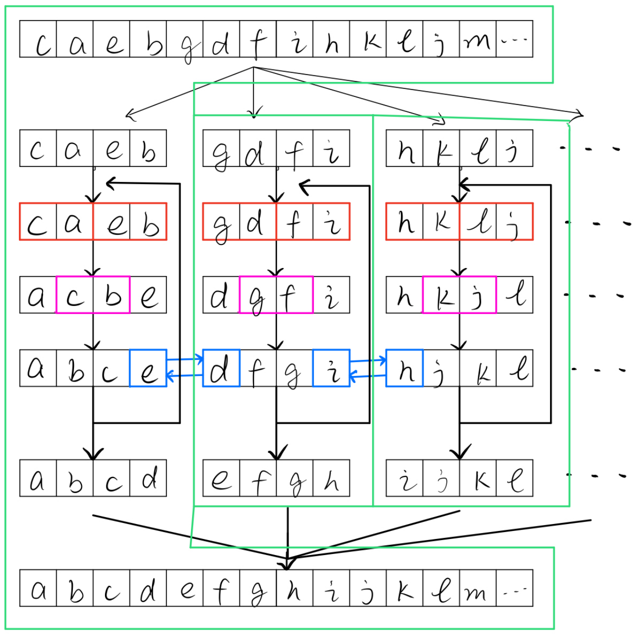


(For more details, please refer to the source code attached to this report)

Parallel Implementation

According to the 3 steps introduced in sequential implementation, it is easy to find that, in step 1 and step 2, each pair-wise comparison is independent with all other comparisons, which means, each group of pair-wise compairisons is independent with all other groups of comparisions.

Therefore, we could group the numbers and assign each group a process to do these independent things and then bound them together to get the final result, which is the parallel implementation of Odd-Even Transposition Sort, as the following figure shows:



where the red and purple frames represent pair-wise comparisons, the green frames stand for different processes (the big C-shape green frame is just the master process with rank 0) and the blue frames represent the MPI communications between different processes.

This implementation is more complicated than the sequential one, so here we get into details a little bit about the code:

1. Initialize the MPI Communicator:

```
MPI_Init(&argc, &argv);
int taskid,      /* task ID - also used as seed number */
  numtasks;   /* number of tasks */
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
```

2. Prepare the empty array and subarrays:

```
int subarray_size;
if (SIZE % numtasks == 0){
  subarray_size = SIZE / numtasks;
} else {
  subarray_size = SIZE / numtasks + 1;
}
int subarray[subarray_size];
int array_size = subarray_size * numtasks;
int array[array_size];
```

In the following steps, we are goinng to use MPI_Scatter to scatter the data, but the size of the array may not be the multiple of the number of processes. Therefore, here we extend the array size and fill them with INT_MAX later, so that we could evenly distribute the numbers.

3. In the master process, initialize the array with random number between 0 - 10000, and print out the array. Also start timing.

```
if (taskid == 0) {
  srand((int)time(NULL));
  // To let the initial array be ditributed evenly,
  // append some INT_MAX at the end of the array.
  for (int i = 0; i < array_size; i ++) {
    if (i < SIZE) {
      array[i] = rand() % 10000;
    } else {
      array[i] = INT_MAX;
    }
  }
// print out the initial array
  printf("Initial array: \n");
  for (int i = 0; i < SIZE; i ++) {
    printf("%d\t", array[i]);
  }
  printf("\n");
  clock_t begin_time = clock();
}
```

4. Scatter the initial array to the subarrays in each process:

```
 MPI_Scatter(array, subarray_size, MPI_INT, subarray, subarray_size,
MPI_INT, 0, MPI_COMM_WORLD);
```

5. In all process:
    1. Do the simple odd-even sort by one iteration like sequential one.
        - Here, we just implement the sort by using odd-even-bound instead of odd-bound-even-bound. This could somehow speed up the sorting since the 'bound' step has

a higher time cost than 'odd' and 'even' step. However, this might lead to some mistakes when the size of subarray is only one, since in this case, the same number would be sent to two processes and some data might be messed up. This problem should be considered especially when execute the code with small array size. (The sample code is 20-dim, so please do not run it by using over 20 processes)

2. Using the send buffer and the receive buffer to deliver the first and the last value of each subarray to the adjacent processes, and swap value if necessary:

```
// send to the left subarray
if (taskid != 0){
  int sl = MPI_Send(&left_send_buff, 1, MPI_INT, taskid - 1, i,
MPI_COMM_WORLD);
}

// send to the right subarray
if (taskid < numtasks - 1){
  int sr = MPI_Send(&right_send_buff, 1, MPI_INT, taskid + 1, i,
MPI_COMM_WORLD);
}

// receive from the left subarray
if (taskid != 0){
  int rl = MPI_Recv(&left_recev_buff, 1, MPI_INT, taskid - 1, i,
MPI_COMM_WORLD, &status1);
  if (left_recev_buff > left_send_buff) subarray[0] = left_recev_buff;
}

// receive from the right subarray
if (taskid < numtasks - 1){
  int rr = MPI_Recv(&right_recev_buff, 1, MPI_INT, taskid + 1, i,
MPI_COMM_WORLD, &status2);
  if (right_recev_buff < right_send_buff)
    subarray[subarray_size-1] = right_recev_buff;
}
```

6. Gather the sorted numbers from subarrays to the array in the master process after SIZE (the data size of the array) times of iterations, since the array will be sorted after at most SIZE times of iterations (note that here the code does not judge whether the sorting is finished, it just do enough iterations to ensure the array is sorted) :

```
  MPI_Gather(subarray, subarray_size, MPI_INT, array, subarray_size,
MPI_INT, 0,  MPI_COMM_WORLD);
```

7. The master process print out the sorted result and relevent information.

(The above are the concise implementation from the code respect, more details please refer to the source code attached to this report)

Steps to run the code

sequential_odd_even_sort.c

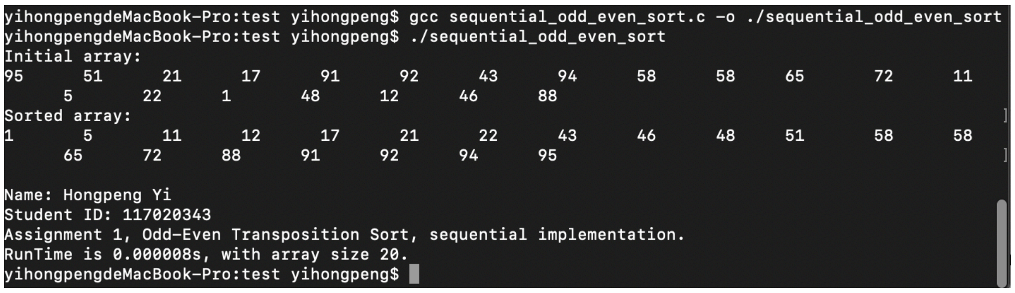
1. In the directory of *sequential version*, type the following command and enter to compile the file:

```
$ gcc sequential_odd_even_sort.c -o ./sequential_odd_even_sort
```

2. In the same directory, type the following command to run the code:

```
$ ./sequential_odd_even_sort
```

3. Then the output would be printed on the console:



```
yihongpengdeMacBook-Pro:test yihongpeng$ gcc sequential_odd_even_sort.c -o ./sequential_odd_even_sort
yihongpengdeMacBook-Pro:test yihongpeng$ ./sequential_odd_even_sort
Initial array:
95      51      21      17      91      92      43      94      58      58      65      72      11
      5       22      1       48      12      46      88
Sorted array:
1       5       11      12      17      21      22      43      46      48      51      58      58
      65      72      88      91      92      94      95

Name: Hongpeng Yi
Student ID: 117020343
Assignment 1, Odd-Even Transposition Sort, sequential implementation.
RunTime is 0.000008s, with array size 20.
yihongpengdeMacBook-Pro:test yihongpeng$
```

parallel_odd_even_sort.c

1. Put the file to a virtual machine with enough cores to run the parallel program, and ensure the MPI has been intalled in this machine.
2. In the directory of *parallel version*, type the following command and enter to compile the program:

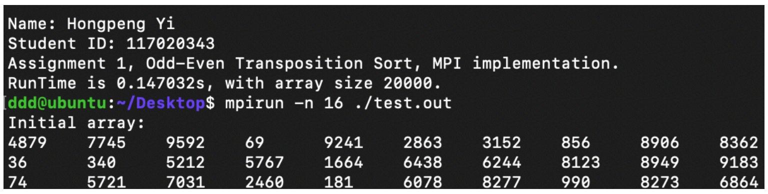
```
$ mpicc parallel_odd_even_sort.c -o ./parallel_odd_even_sort.out
```

3. In the same directory, type the following command and enter to run the code:

```
$ mpirun -np ./parallel_odd_even_sort.out
```

   where -np is the number of processes to be runned.

4. The output should be similar as the following partly screen shot:



```
Name: Hongpeng Yi
Student ID: 117020343
Assignment 1, Odd-Even Transposition Sort, MPI implementation.
RunTime is 0.147032s, with array size 20000.
ddd@ubuntu:~/Desktop$ mpirun -n 16 ./test.out
Initial array:
4879    7745    9592    69      9241    2863    3152    856     8906    8362
36      340     5212    5767    1664    6438    6244    8123    8949    9183
74      5721    7031    2460    181     6078    8277    990     8273    6864
```

(Here the array size is 20000, different with the size 20-dim of the attached source code)

Performance Analysis

Running Time Measurement

To analyse the performance of the two versions of sorting, it is important to set the running time measurement properly. In this assignment, the function *clock()* was adopted to both programs:

```
clock_t begin_time = clock(); // begin timming
...
clock_t end_time = clock(); // stop timming
double running_time = (double)(end_time-begin_time)/CLOCKS_PER_SEC;
```

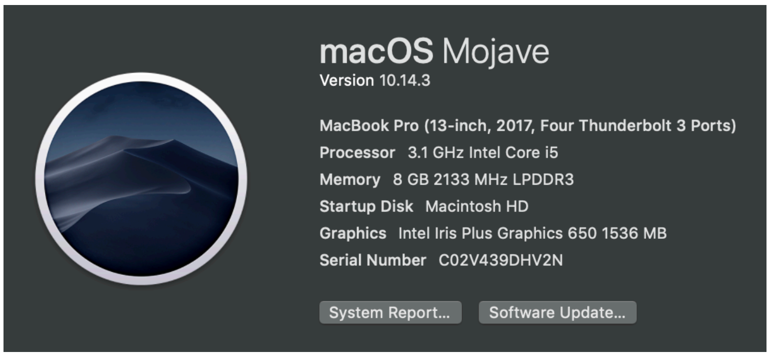
To ensure fairness between the two versions:

- For the sequential version, the timing will begin right after the array data was initialized and will end right after the sorting was finished.
- For the parallel version. the running time will be measured by the master process, since the master process is in charge of the scattering and gathering. The timing will begin right after the array data was initialized in the master process and will end right after the data was gathered by the master process.

Sequential Version

running environment

Personal laptop:



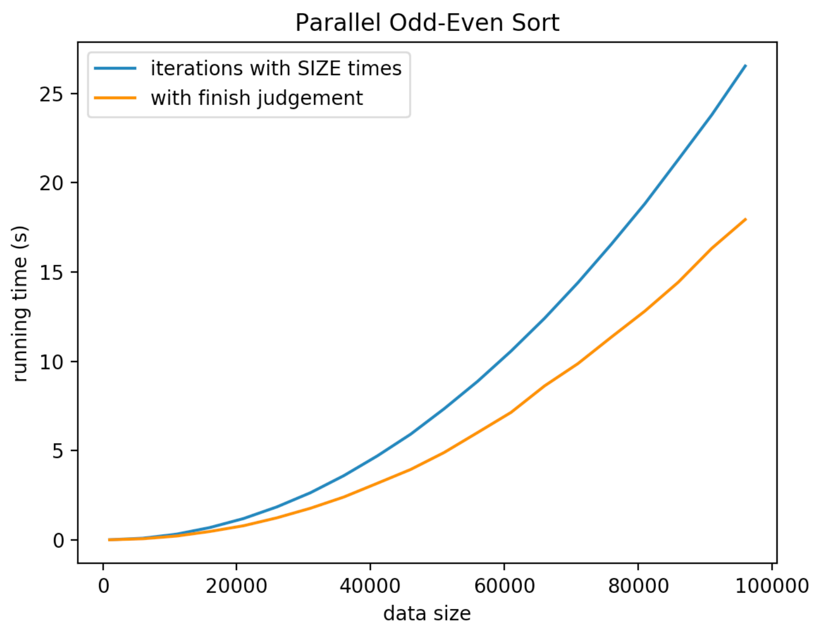
macOS Mojave
Version 10.14.3

MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)
Processor 3.1 GHz Intel Core i5
Memory 8 GB 2133 MHz LPDDR3
Startup Disk Macintosh HD
Graphics Intel Iris Plus Graphics 650 1536 MB
Serial Number C02V439DHV2N

performance

According to the program design of the parallel version, the parallel program does not have judgement on whether the sorting is finished, so it will run the iterations SIZE (the data size of the array) times to ensure the sorting will be finished. To let the later comparison make more sense, we generate a new version of sequential odd-even sort that also run the iterations SIZE times. And actually, this is the version of the source code attached.

To get some insights about the sequential odd-even sort, we test the runnning time with increasing data size (from 1000 to 96000 with interval 5000), and plot the test results by python:



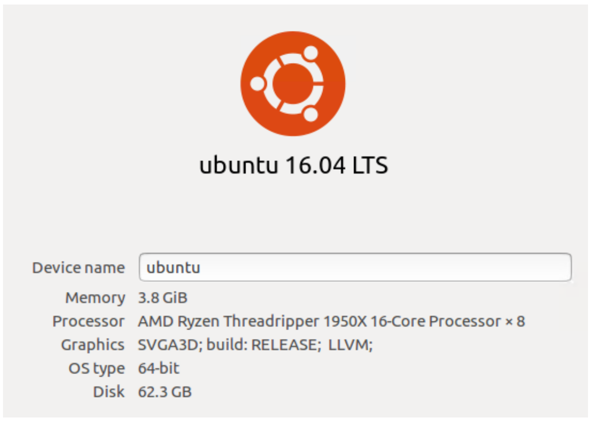
It is easy to find that, as the data size increases, the running time increases very fast, and the time complexity of this sequential sortinng method is like O(n^2). Therefore, it is not recommended to use Sequential Odd-Even when the data size is large. It cost too much time due to the time complexity of the algorithm.

Also, comparing the two implementations, doing iterations enough times without finishing judgement is a inefficient way to sort, especially when the data is large. The same, if we could implement the parallel version of the sorting with finish judgement, the efficiency could be improved a lot. But due to the time limit, we just use enough iterations to finish the sorting now.

Parallel Version

running environment

For the prallel version, we runned it under a virtual machine provided by a classmate, the information of the machine is as follows:

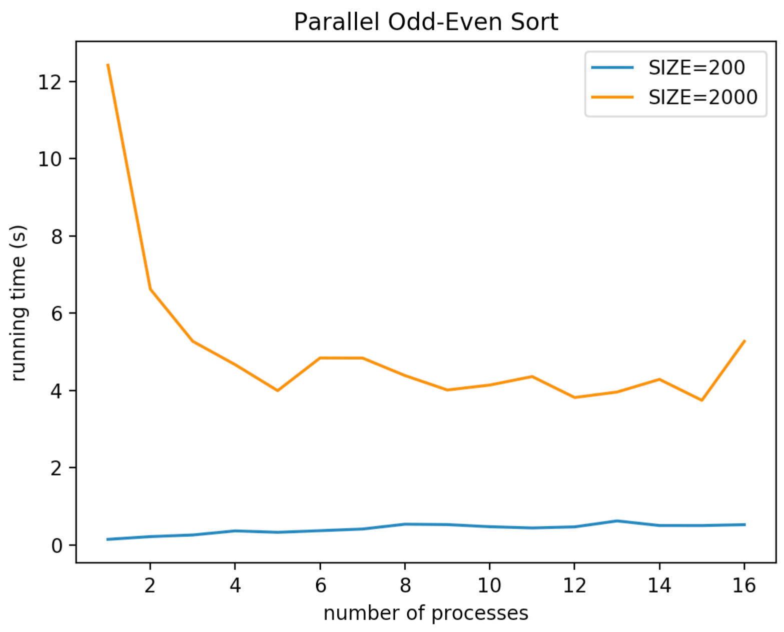

ubuntu 16.04 LTS

Device name	ubuntu
Memory	3.8 GiB
Processor	AMD Ryzen Threadripper 1950X 16-Core Processor × 8
Graphics	SVGA3D; build: RELEASE; LLVM;
OS type	64-bit
Disk	62.3 GB

which is similar to the running environment of the machine provided by the course.

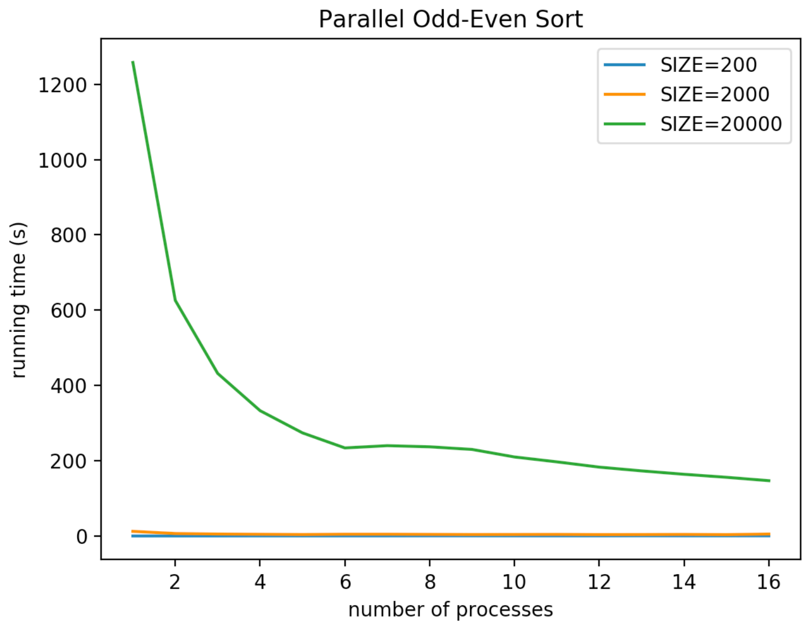
performance

The program has been tested by three data sizes: 200 (small), 2000 (medium) and 20000 (large), with different number of processes: from 1 to 16. The data set of running time has been recorded and presented by the following figures.



According to the above output data graph with small and medium data size, we could find that:

1. According the the blue curve, when the data size is small, the increase of the process number could hardly improve the running time.
2. When the data size is medium, the increase of the process number could obviously improve the running time when the number of processes is under 4, but fluctuate at a certain level when the number of processes become larger.



However, when we add the results of large data size, as what is shown above, the curves with small data size and medium data size are flaten a lot, and we could find that:

1. According to the green curve, when the data size is large, the increase of the process number lead to the improvement of the running time, and especially has a great improvement when the number of processes is under 4.
2. Comparing the three curves with different data sizes, when the data size becomes large, the increased running time with more processes is much smaller than that with less processes.

Combine some of the data from previous test:

data size	sequential	parallel (1 process)	parallel (4 processes)
200	0.257 ms	0.143 ms	0.362 ms
2000	12.760 ms	12.416 ms	4.664 ms
20000	1290 ms	1258 ms	333 ms

According to the chart, we could find that, the performance of the parallel version with 1 process is similar to sequential version and it does make sense since 1 process is just like sequential sorting.

And comparing the sequential version with 4-process parallel version, the parallel version only took around 1/4 running time, which means the speed up is 4. Therefore, we could almost say that the Parallel Odd-Even Sorting mainly consists of parallel parts.