CSC4005HW1: Parallel Odd-Even Transposition Sort & Analysis

118010202 / LIU Zhixuan

October 13, 2021

1 Abstract

Odd-Even sort is basically a variation of bubble-sort. This algorithm is divided into two phases- Odd and Even Phase. The algorithm runs until the array elements are sorted and in each iteration two phases occurs- Odd and Even Phases. In the odd phase, a bubble sort is performed on odd indexed elements and in the even phase, a bubble sort is performed on even indexed elements.

In this assignment, I performed both parallel version of Odd-Even transposition sort and sequential transposition sort, and the corresponding experiments and some efficient analysis is performed on both sorting algorithms.

2 Introduction to Odd Even Sort

In this section, we briefly introduce sequential odd-even sort and how to implement it using C++ and some complexity analysis of it.

In computing, an odd–even sort or odd–even transposition sort (also known as brick sort, is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the figure 1: list is sorted.

This process can be expressed by the following

By observing that, for iteration i and the index of the number j in the array, if (i + j) can be divided by 2, then this number will compare with its following number as a base. For example, if the it is an odd phase, than all the number with an odd index will be a base. By the above observation, the sequential implementation of odd-even sort and be written in the following codes using C++:

```
1 void oddEvenSeq(int * global_array, int global_number){
2     for (int i=0; i<global_number; i++){
3         // TRANSPORT INSIDE THE PROCESS
4         for(int j=0; j<global_number-1; j++){
5             int tar_index = j;
6             if((tar_index+i)%2 == 0){
7                 if(global_array[j] > global_array[j+1]){
```

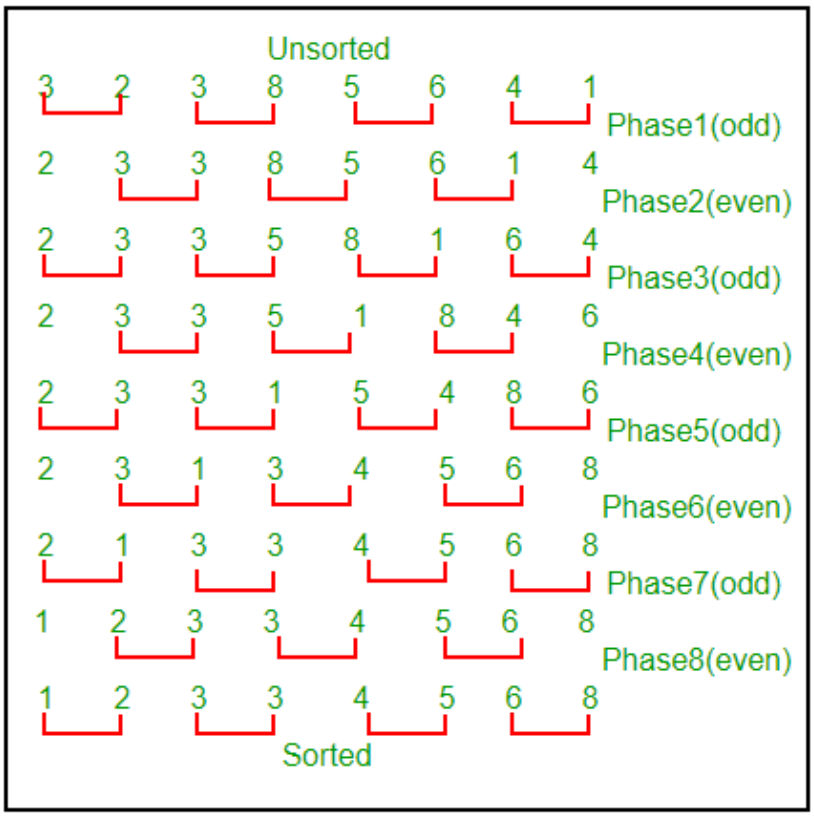

Figure 1: Sequential Odd-Even Sort

```
8                   int temp = global_array[j];
9                   global_array[j] = global_array[j+1];
10                  global_array[j+1] = temp;
11              }
12          }
13      }
14  }
15 }
```

Now, let's evaluate the complexity if sequential odd-even sort algorithm. The time complexity : O(N^2) where, N = Number of elements in the input array. This is because to finally sort this array, we will have N iteration and in each iteration there will be N comparison, therefore, the total amount is O(N^2). As for the auxiliary space complexity, it is O(1), because just like bubble sort this is an in-place algorithm.

3 Parallel Odd-Even Sort

It is obvious that sequential odd-even sort does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The requirement of this assignment is:

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.

Repeat 1-4 until the numbers are sorted.

The process for this distributed computing is shown in the figure 2 below:

The main idea of this process can be explained by the diagram 3: firstly, the array of numbers is distributed to different nodes. In each nodes, the odd-even sort is operated. The boundary check also appears in each phase. Here we also consider the possibility of the number of element in the array cannot be divided by the number of nodes. We use Scatterv and Gatherv to deal with this problem.

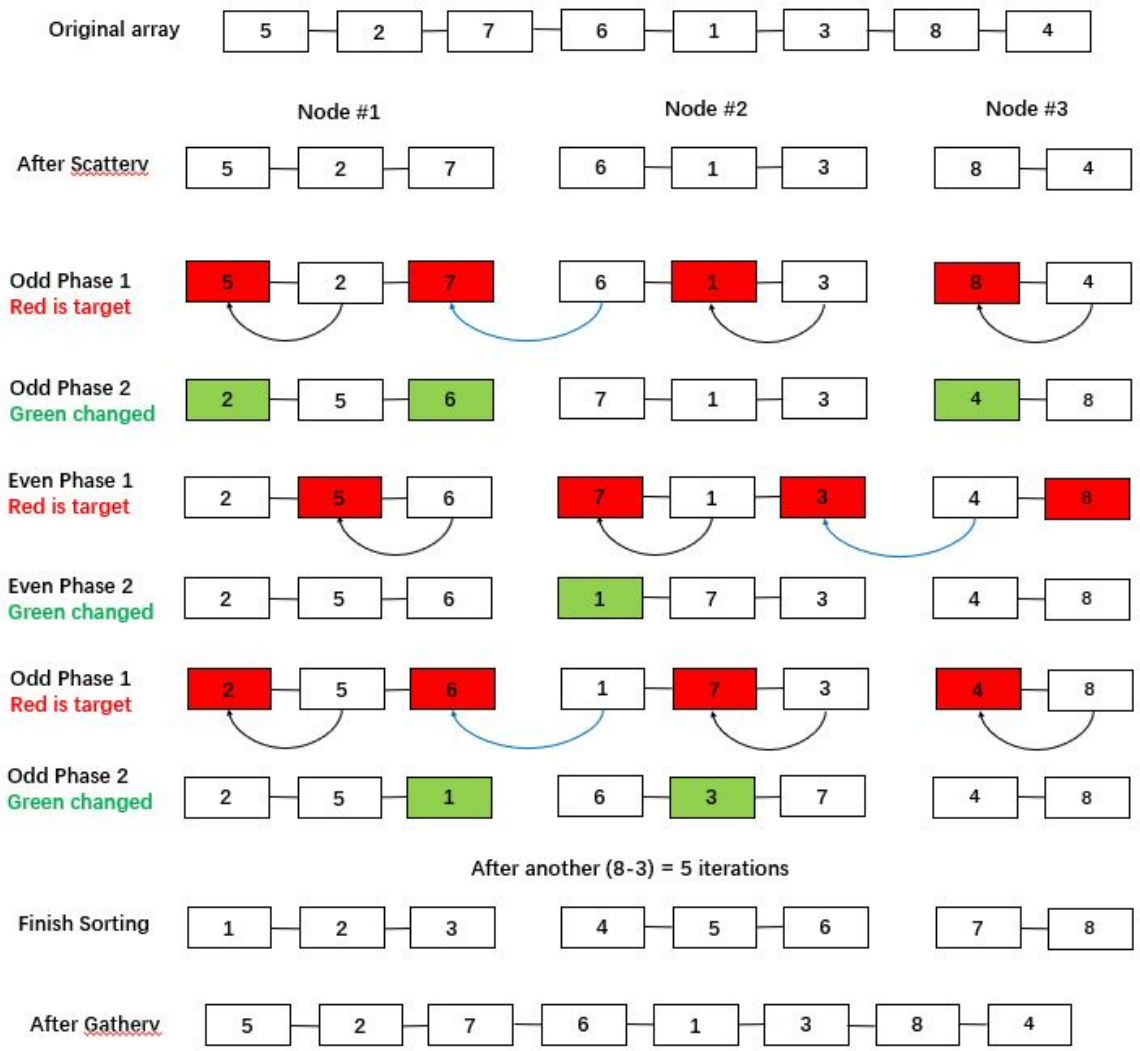

Figure 2: Process of Parallel Odd-Even Sort

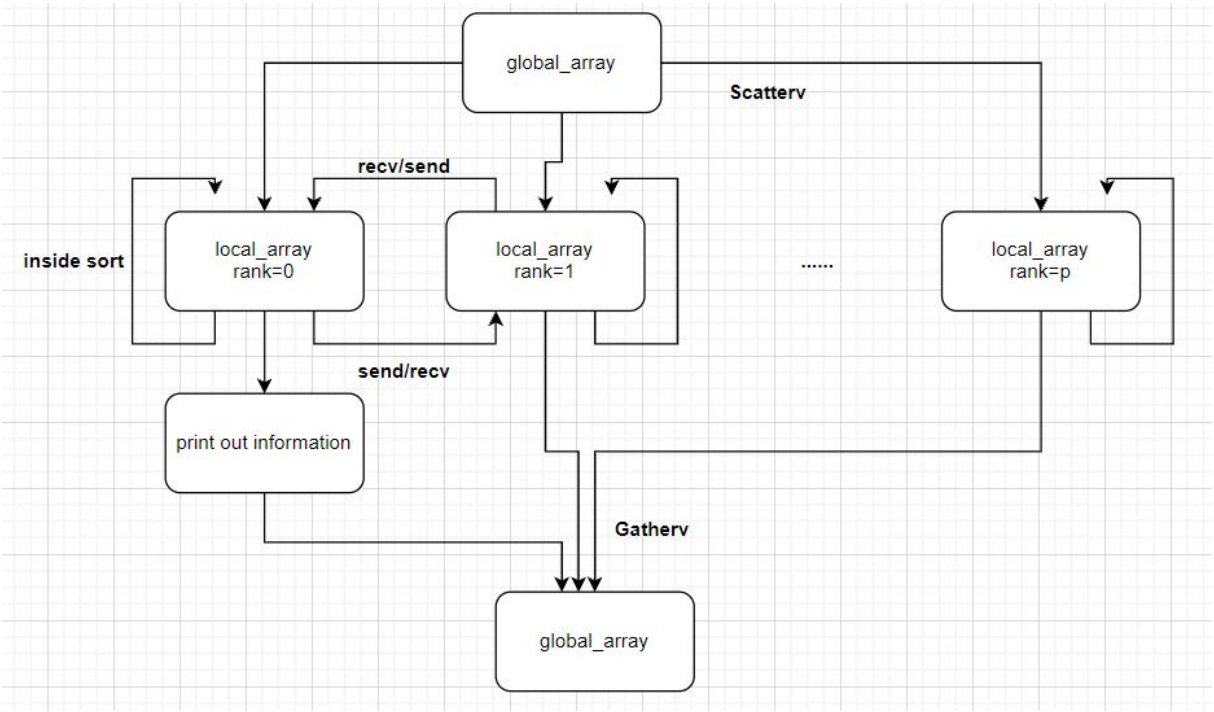

Figure 3: Diagram of Parallel Odd-Even Sort

3.1 Complexity Analysis of Paralleled Odd-Even Sort

Assume there are total n numbers and p processes. To figure out the time complexity of Paralleled Odd-Even Sort, we need to take two parts into consideration:

3.1.1 Computation complexity

There will be $\frac{n}{p}$ numbers in each process. Since inside the process is totally odd even sort, and there will be n rounds until the process stops. Therefore, the computational complexity is $\frac{n^2}{p}$. Therefore, the computation is $O(N^2)$

3.1.2 Communication complexity

Let's denote $T_{startup}$ and T_{data}.

For Scatterv, $T_{comm\_scatter} = p \times (T_{startup} + \frac{n}{p} \times T_{data}) = O(N)$.

Similarly for Gatherv, $T_{comm\_gather} = p \times (T_{startup} + \frac{n}{p} \times T_{data}) = O(N)$.

In the worst, each phase will operate a boundary change. Therefore, $T_{comm} = N \times p \times (T_{startup} + T_{data}) = O(pN)$.

Therefore, the total complexity is $O((\frac{n}{p})^2 + pn)$

3.2 Implementation

First thing to do is to init MPI:

```
1     MPI_Comm comm;
2     MPI_Init(&argc, &argv);
3     comm = MPI_COMM_WORLD;
4     MPI_Comm_size(comm, &p);
5     ierr = MPI_Comm_rank(comm, &my_rank);
```

```
    if (MPI_SUCCESS != ierr) {
        throw std::runtime_error("failed to get MPI world rank");
    }
```

After get the number, I will randomly generate the array based on the given number in rank 0. The global number is the user input.

```
    if (my_rank == 0){
        // ALLOC SPACE AND RANDOMNLY GENERATE NEW NUMBERS FOR THE GLOBAL ARRAY
        // GLOBAL_ARRAY = (INT*)MALLOC(SIZEOF(INT)*GLOBAL_NUM);
        std::cout << "The number of element is: " << global_num << "." << std::endl
            ;
        std::cout << "The array data before sorting is: " << std::endl;
        srand(time(0));
        for(int i=0; i<global_num; i++){
            global_array[i] = rand() % 1000;
            std::cout << global_array[i] << " ";
        }
        std::cout<<"."<<std::endl;
    }
```

Next is to scatter the number to different nodes. To deal with the problem that each node cannot be assigned to the same number of numbers, I use scatterv. The previous node will always get more numbers than the latter nodes. For example, 11 numbers for 4 nodes will be 4+4+4+3 for each node.

```
    // DEALING WITH THE PROBLEM IF # CANNOT BE DIVIDED BY # OF PROCESS
    int scatter_count[p];
    int scatter_dsp[p];
    int ccount = 0;
    for (int j=0; j<p ; j++){
        if( j <= remainder-1){
            scatter_count[j] = basic+1;
        }else{
            scatter_count[j] = basic;
        }
        scatter_dsp[j] = ccount;
        ccount += scatter_count[j];
    }
```

Now lets begin the mpi odd-even sort in each process. The first thing is the inside odd-even sort. At this time, we do not consider the boundary check. Only odd numbers in the odd phase will be checked.

```
for (int i=0; i<global_number; i++){
        // TRANSPORT INSIDE THE PROCESS
        for(int j=0; j<local_number-1; j++){
            int tar_index = base_num + j;
            if((tar_index+i)%2 == 0){
                if(local_array[j] > local_array[j+1]){
                    int temp = local_array[j];
                    local_array[j] = local_array[j+1];
                    local_array[j+1] = temp;
                }
            }
        }
```

For the mpi odd-even sort part, we should consider the difference between the frist node and the last node.

The first node does not send the boundary to the previous node because it does not have a previous node. The last node does not receive the boundary because it cannot receive one. Also the last node does not do the comparison process.

By observing that, only in the odd phase (i) and the number with odd index (j) or in the even phase (i) and the number with even index (j) need to have boundary check, and (i + j) % 2 == 0 always holds when check is implemented. Therefore, I use this condition to decide the exchange situation.

In each node, I will decide whether this node need to send or receive numbers from its adjacent nodes.

The boundary exchange will be implemented by mpi_Send() and mpi_Recv().

The code implementation can be:

```
1     s1=0;
2     r1=0;
3
4     if (my_rank == 0){
5
6         if((last_index+i)%2 == 0){
7             MPI_Recv(&r1, 1, MPI_INT, my_rank+1, 0, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
8             s1 = (local_array[local_number-1] > r1) ? local_array[local_number-1] :
                  r1;
9             local_array[local_number-1] = (local_array[local_number-1] > r1) ? r1 :
                  local_array[local_number-1];
10            MPI_Send(&s1, 1, MPI_INT, my_rank+1, 1, MPI_COMM_WORLD);
11        }
12    }else if(my_rank == last_p-1){
13        if((first_index+i)%2 != 0){
14            s1 = local_array[0];
15            MPI_Send(&s1, 1, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD);
16            MPI_Recv(&r1, 1, MPI_INT, my_rank-1, 1, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
17            local_array[0] = r1;
18        }
19    }else if (my_rank>0 && my_rank < last_p-1){
20        if((first_index+i)%2 != 0){
21            s1 = local_array[0];
22            MPI_Send(&s1, 1, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD);
23            MPI_Recv(&r1, 1, MPI_INT, my_rank-1, 1, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
24            local_array[0] = r1;
25        }
26        if((last_index+i)%2 == 0){
27            MPI_Recv(&r1, 1, MPI_INT, my_rank+1, 0, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
28            s1 = (local_array[local_number-1] > r1) ? local_array[local_number-1] :
                  r1;
29            local_array[local_number-1] = (local_array[local_number-1] > r1) ? r1 :
                  local_array[local_number-1];
30            MPI_Send(&s1, 1, MPI_INT, my_rank+1, 1, MPI_COMM_WORLD);
31        }
32    }
```

4 Experiment

4.1 Compile and Run

4.1.1 Sequential

To compile my sequential codes, use

```
g++ oddEvenSortSeq.cpp
```

To run my sequential codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o N1n1.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1

mpirun a.out 10
mpirun a.out 100
mpirun a.out 1000
mpirun a.out 10000
```

4.1.2 Parallel

I use C++ to implement this algorithm. To compile my codes, use this command:

```
mpic++ oddEvenSort.cpp
```

Then, you will get the a.out. I use sbatch to run my code on the cluster. User needs to enter the number they want to sort. For example, it they want to sort 3, 10 and 20 numbers, the shell codes are like this:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o N1n4.out
#SBATCH -N 1
#SBATCH -n 4
#SBATCH -t 1

srun hostname -s | sort -n >slurm.hosts

mpirun a.out 3
mpirun a.out 10
mpirun a.out 20
```

4.2 Sample outputs

4.2.1 Normoal Case

We set nodes to be 4 and number to be 20, where all the nodes have the same number of elements. The sample output is in figure 4:

```
21  Name: Zhixuan Liu
22  Student ID: 118010202
23  Assignment 1, Paralleled-Odd-Even Sort
24  The total number of element is: 20.
25  The array data before sorting is:
26  241 715 643 308 256 537 235 828 633 552 398 934 692 496 796 616 673 674 659 175 .
27  final sorted data:
28  175 235 241 256 308 398 496 537 552 616 633 643 659 673 674 692 715 796 828 934 .
29  Time used to sort 20 numbers is: 0.000000.
30
```

Figure 4: Sort 20 elements in 4 nodes

4.2.2 Unequally Assigned Case

We set nodes to be 4 and number to be 10, where all the nodes do not have the same number of elements. In this case, the assignment is 3+3+2+2. The sample output is in figure 5:

```
12  Name: Zhixuan Liu
13  Student ID: 118010202
14  Assignment 1, Paralleled-Odd-Even Sort
15  The total number of element is: 10.
16  The array data before sorting is:
17  51 472 741 822 107 197 17 596 933 816 .
18  final sorted data:
19  17 51 107 197 472 596 741 816 822 933 .
20  Time used to sort 10 numbers is: 0.000000.
```

Figure 5: Sort 10 elements in 4 nodes

4.2.3 Number Smaller than Process Case

We set nodes to be 4 and number to be 3, where there will be one does not have the any of the elements. In this case, the assignment is 1+1+1+0. The sample output is in figure 6:

```
3   Name: Zhixuan Liu
4   Student ID: 118010202
5   Assignment 1, Paralleled-Odd-Even Sort
6   The total number of element is: 3.
7   The array data before sorting is:
8   51 472 741 .
9   final sorted data:
10  51 472 741 .
11  Time used to sort 3 numbers is: 0.000000.
```

Figure 6: Sort 3 elements in 4 nodes

4.3 Results

4.3.1 Sequential: Time vs Job Size

This is the time vs Job Size figure in Sequential Program.

JobSize	1000	10000	20000	30000	40000	50000	60000	70000	80000	>100000
Seqtime/s	0	0.81	3.27	7.41	13.22	20.76	29.21	41.18	53.28	>60

Figure 7: Sequential: Time vs Job Size

4.3.2 Parallel: Time vs Job Size

This is the time vs Job Size figure in Parallel Program.

JobSize\Nodes	n2	n3	n4	n5	6n	n8	n9	n10	n12	n16	n20	n24
10000	0.41	0.32	0.21	0.16	0.12	0.11	0,10	0.09	0.08	0.06	0.04	0.01
20000	1.65	1.13	0.86	0.69	0.6	0.44	0.41	0.35	0.3	0.22	0.19	0.05
30000	3.73	2.59	1.97	1.59	1.33	1.02	0.91	0.82	0.68	0.51	0.41	0.16
40000	6.64	4.56	3.52	2.83	2.41	1.84	1.64	1.49	1.24	0.94	0.73	0.34
50000	10.42	7.28	5.53	4.45	3.51	2.9	2.59	2.33	1.97	1.48	1.17	0.62
60000	15.06	10.5	8.01	6.46	5.48	4.2	3.76	3.39	2.84	2.13	1.71	0.97
70000	20.52	14.33	10.91	8.79	7.49	5.76	5.14	4.64	3.89	2.95	2.35	1.41
80000	29.21	18.73	14.27	11.52	9.8	7.55	6.76	6.09	5.12	3.88	3.1	1.96
90000	38.82	23.32	18.07	14.59	12.45	9.58	8.57	7.75	6.5	4.9	3.93	2.58
100000	50.32	29.21	22.31	18.09	15.41	11.85	10.62	9.59	8.06	6.1	4.86	3.26

Figure 8: Parallel: Time vs Job Size

4.3.3 Speed-Up Factor: Speed-Up factor S(p) where ts and tp are the time spent by sequential pro-grams and that by parallel programs on p processors: $\frac{TIMEforSeq}{TIMEforPara}$

JobSize\Nodes	n2	n3	n4	n5	6n	n8	n9	n10	n12	n16	n20	n24
10000	1.9756	2.5313	3.8571	5.0625	6.75	7.364	8.1	9	10.13	13.5	20.3	81
20000	1.9818	2.8938	3.8023	4.7391	5.45	7.432	7.9756	9.3429	10.9	14.9	17.2	65.4
30000	1.9866	2.861	3.7614	4.6604	5.571	7.265	8.1429	9.0366	10.9	14.5	18.1	46.3
40000	1.991	2.8991	3.7557	4.6714	5.485	7.185	8.061	8.8725	10.66	14.1	18.1	38.9
50000	1.9923	2.8516	3.7541	4.6652	5.915	7.159	8.0154	8.9099	10.54	14	17.7	33.5
60000	1.9396	2.7819	3.6467	4.5217	5.33	6.955	7.7686	8.6165	10.29	13.7	17.1	30.1
70000	2.0068	2.8737	3.7745	4.6849	5.498	7.149	8.0117	8.875	10.59	14	17.5	29.2
80000	1.824	2.8446	3.7337	4.625	5.437	7.057	7.8817	8.7488	10.41	13.7	17.2	27.2

Figure 9: Speed-Up Factor

4.3.4 System Efficiency: $\frac{SpeedUpFactor}{p}$

This figure 10 shows the System Efficiency divided by 100% for simplicity.

JobSize\Nodes	n2	n3	n4	n5	6n	n8	n9	n10	n12	n16	n20	n24
10000	0.9878	0.8438	0.9643	1.0125	1.125	0.92	0.9	0.9	0.8438	0.8438	1.0125	3.375
20000	0.9909	0.9646	0.9506	0.9478	0.908	0.929	0.8862	0.9343	0.9083	0.929	0.8605	2.725
30000	0.9933	0.9537	0.9404	0.9321	0.929	0.908	0.9048	0.9037	0.9081	0.9081	0.9037	1.9297
40000	0.9955	0.9664	0.9389	0.9343	0.914	0.898	0.8957	0.8872	0.8884	0.879	0.9055	1.6201
50000	0.9962	0.9505	0.9385	0.933	0.986	0.895	0.8906	0.891	0.8782	0.8767	0.8872	1.3952
60000	0.9698	0.9273	0.9117	0.9043	0.888	0.869	0.8632	0.8617	0.8571	0.8571	0.8541	1.2547
70000	1.0034	0.9579	0.9436	0.937	0.916	0.894	0.8902	0.8875	0.8822	0.8725	0.8762	1.2169
80000	0.912	0.9482	0.9334	0.925	0.906	0.882	0.8757	0.8749	0.8672	0.8582	0.8594	1.1327

Figure 10: System Efficiency

5 Analysis

5.1 Sequential Analysis

This figure 11 shows the relationship between the running time and the data size when sequential. We can roughly see that this curve is approximately regarded as a quadratic function, and this result is consistent with our previous analysis of its complexity $O(N^2)$.

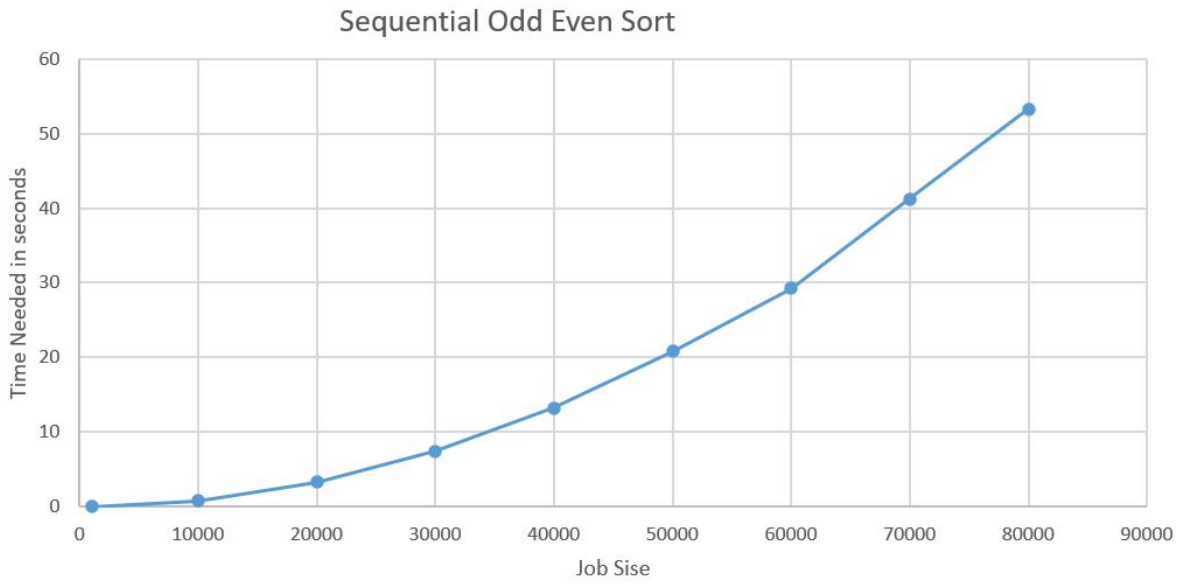

Figure 11: Sequential Analysis

5.2 Parallel Analysis

This time, we compare the Node size, Job size and the time spent in figure 12.

5.2.1 Cores and Job numbers in Parallel Program

According to the figure above, if you only focus on one of the curves, then the trend of this curve is in the form of a quadratic function. And when we look at the entire image macro, we find that when the number of processes is larger, the slope of the curve is

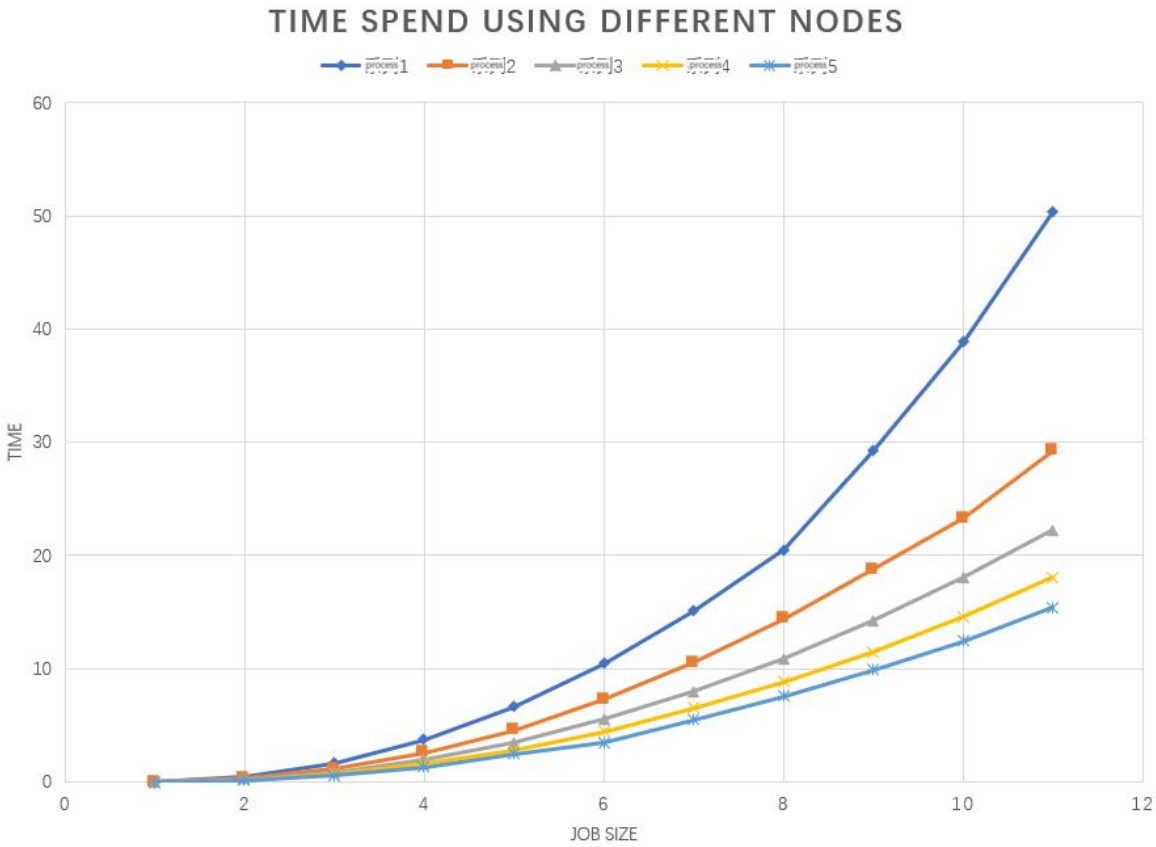

Figure 12: Node size, Job size and the time spent

smaller. We can roughly see from this figure that when the job size is the same, the time used is inversely proportional to the number of processes.

As we mentioned in the previous section, the complexity of Odd Even sort should be $O((\frac{n}{p})^2)$. Based on our observations, our results are consistent with previous conclusions.

The reason why my results are excellent is because I only use one node and specify different processes on this node. In this way, the influence of the communication between node and node before will not be produced in our experiment. So our experimental results are excellent and satisfy the theory.

5.2.2 MPI vs Sequential

As we can see above, the sequential process, which is the blue curve in figure 12, is the sequential process. It takes much more times in each job size than the parallel process. This is because the time complexity is $O((\frac{n}{p})^2)$ and for sequential process, the n is 1.

5.3 Speed Up Factor vs Number of Process

These two figures show the relationship between the speed up factor and the number of processes.

We can see in the picture that within the allowable range of error, when problem size stays fixed but the number of process elements are increased, it is a strong scaling picture, and also a linear scaling.

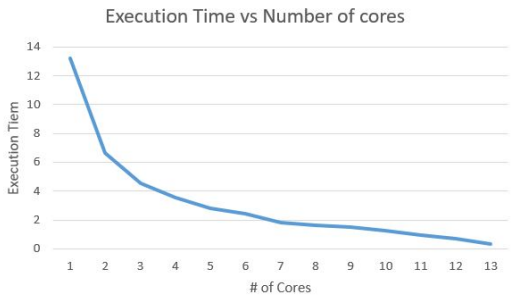

Figure 13: Execution time vs Number of processes

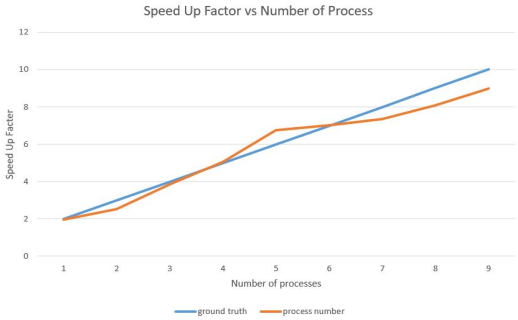

Figure 14: Concrete and Constructions

And by observing figure 15, I found that, when process number is fixed, speed up factor does not change too much with the job size.

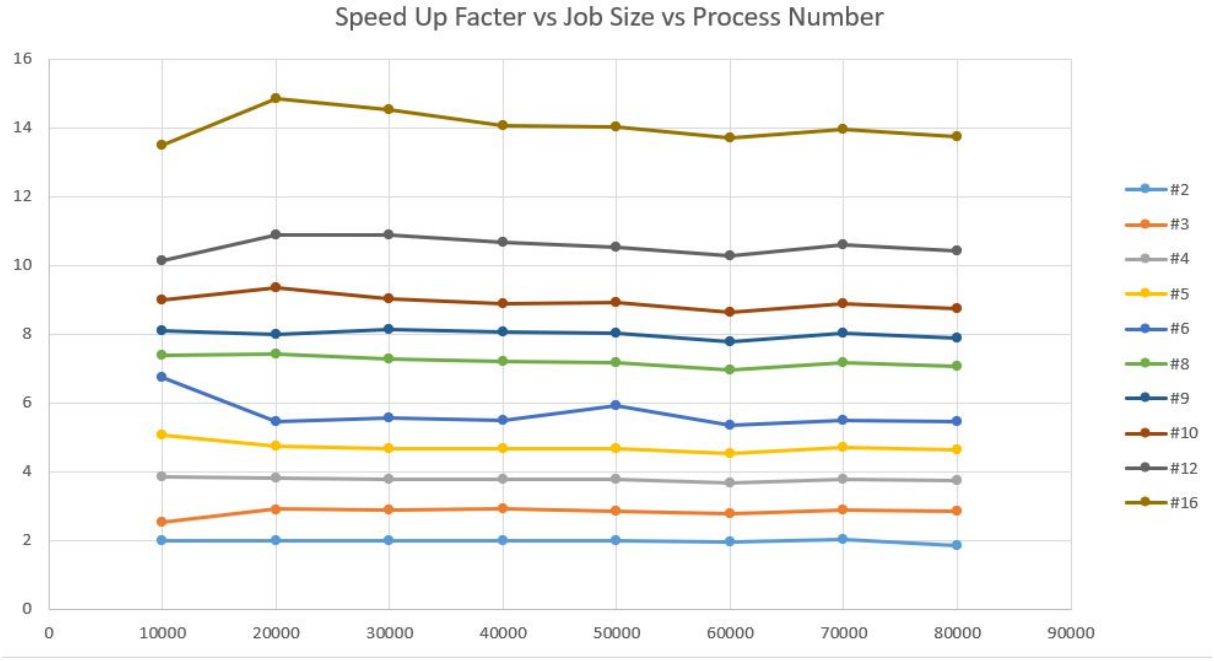

Figure 15: Node size, Job size and the time spent

6 Conclusion and Future work

This project uses sequential and parallel parity sorting to experiment Regarding their performance. I obvserved that if there are more processes, the acceleration overhead will increase. From the experiment, I found that when the process number is increased, parallel programs do provide faster performance.

Moreover, by looking at the speed up factor and the processes number, I found out that it is a strong scaling. Thus, it can be used to find a "sweet spot" that allows the computation to complete in a reasonable amount of time, yet does not waste too many cycles due to parallel overhead.