Bernaldy Jullian

119010520

CSC4005 Assignment 1

Parallel Odd-Even Transposition Sort

I. Introduction

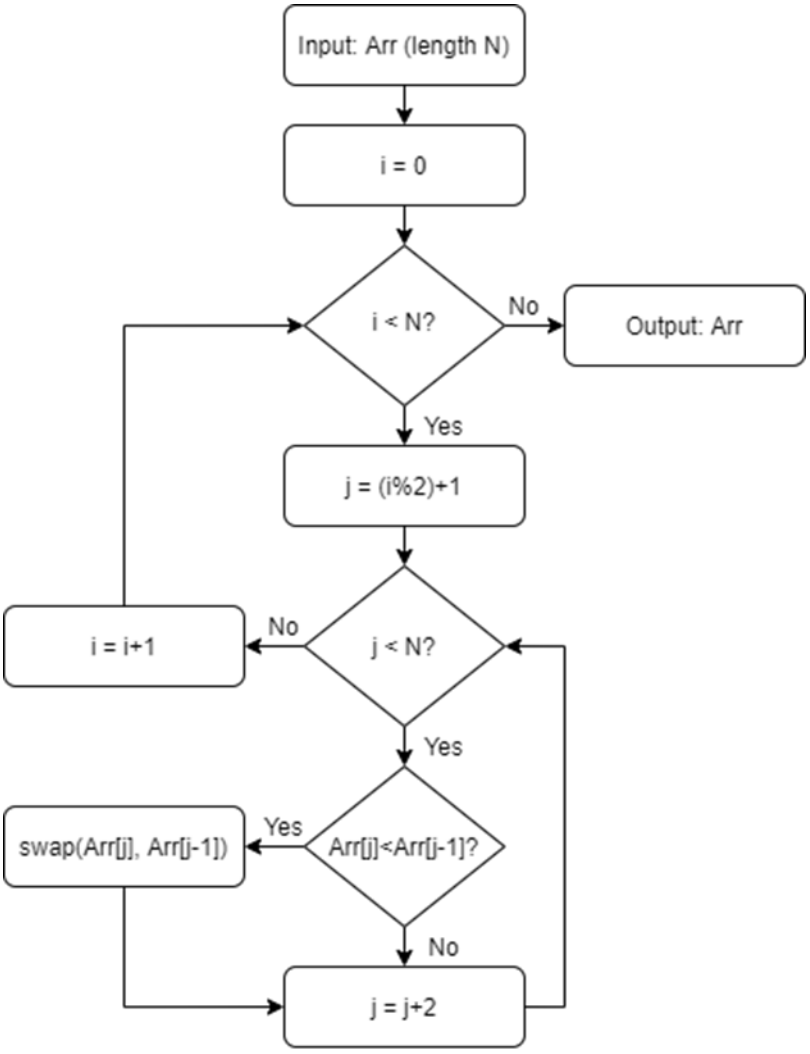
With how important data is in this era, it is essential to have the ability to sort through big data with the most effective and efficient method. Unfortunately, no sorting algorithm can reach a better result than a time complexity of O(n $log\ n$). Without a way to improve the time complexity of current sorting algorithms, we have to look for another way to improve efficiency. One of the alternative solutions is to use parallel computing. Instead of creating a better computing capability, we can combine several of the current capabilities in order to increase performance. In this report, we will try to utilize parallel computing to implement Odd-Even Transposition Sort algorithm. We will also try some parallel computing configurations and compare them with the sequential algorithm. In the end, we will know how well parallel computing compared to the common sequential method.

2. Design Approach

In the Parallel Odd-Even Transposition Sort algorithm, we need to swap two elements in the array alternately between odd-index elements with its left-neighbor elements and even-index elements with its left-neighbor elements until the array is sorted. Thus, for each iteration, an element can move at most 1 step either to the right (if it is bigger) or to the left (if it is smaller). That being said, the diameter or the farthest path an element can move is when an element is initially located at the far-left or far-right positions and it needs to go to the far-right or far-left position. This worst scenario will, hence, take N iterations to complete where N is the number of total elements in the array.

- Sequential Implementation

By following the assumption, the implemented sequential algorithm loops the swapping process for N times. The swapping process starts with the odd swapping of all elements in the array. Since this is a sequential algorithm, implementing the swapping process needs another loop that iterates through all the odd-indexes or even-indexes elements. After each swapping process, the algorithm will repeat the swapping process but with the other indexes (even after odd and odd after even). In the implemented sequential algorithm, this alternating process can be achieved by using the formula j=\left(i%2\right)+1 where j is the alternating value and i is the iteration value of the algorithm (the one that loops the swapping process). This swapping and alternating processes continue until the algorithm loop is finished (reach N times).



- Parallel Implementation

The parallel implementation follows a similar logic to the sequential implementation. Applying the same logic and comparison method (compares next element) maintains the consistency between both implementations and should give a better comparison between the two algorithms. The main difference between the sequential and parallel program is the splitting of data to give to the other processes and its parallel execution. The parallel implementation first starts with distribution of the input array to the other generated processes. It performs this task by dividing the number of elements of the input array and distributing it equally (if possible) to all other processes including the master process. In case where the input array size is indivisible with the number of processes, the algorithm will allocate one of the remainder data elements to the other processes, starting from the master until there is no more remainder (allocates (N % num_proc + 1) data to the first (N % num_proc). This distribution method ensures that the number of input data for any process differs by at most one element.

```
   int host_name_len, loc_arr_len, temp_var, temp_var_2;
char host_name[MPI_MAX_PROCESSOR_NAME];

MPI_Get_processor_name(host_name, &host_name_len);


int* send_count = (int*) malloc(world_size * sizeof(int));
int* dis_loc = (int*) malloc(world_size * sizeof(int));


loc_arr_len = num_elements/world_size + (rank<num_elements%world_size);
temp_var = 0;
for(int i = 0; i<world_size; i++){
 send_count[i] = num_elements/world_size + (i<num_elements%world_size);
 dis_loc[i] = temp_var;
 temp_var += send_count[i];
}

int* locArr = (int*) malloc(loc_arr_len * sizeof(int));


//distribute to other processes
temp_var = MPI_Scatterv(elements, send_count, dis_loc, MPI_INT, locArr, loc_arr_len, MPI_INT, 0, MPI_COMM_WORLD);
if(temp_var != MPI_SUCCESS){
 return -1;
}
```

The algorithm begins sorting in all of its operations after data dispersion. On each process' local input arrays, it sequentially performs the odd-even transposition sort. The processes may need to pass their edge data values—data at the end of the array—to the following process in each iteration after the local sorting in order to appropriately sort the array. By comparing the odd and even cycle against the initial index of their local array in the original input array, the processes determine when to send and receive data (which index does their local array start at in the original input array).

```
bool even = 0;
for(int i = 0; i<num_elements; i++){
 for(int j = (even+dis_loc[rank])%2+1; j<loc_arr_len; j+=2){
  if(locArr[j]<locArr[j-1]){
   temp_var = locArr[j];
   locArr[j] = locArr[j-1];
   locArr[j-1] = temp_var;
  }
 }
```

The cycle it is current on is stored in a Boolean variable called even while the starting index number of each process' local arrays are kept in a separate array called dis_loc. Calling dis_loc[rank] (where rank is the rank of the process) will give the starting position of the local array in the original input array. Comparing these two values is sufficient to tell whether said process will need to send or receive data to/from other processes (if starting index parity does not match with the phase, then it would need to send data). The master process is excluded from receiving any data (as there is no processes that sends forward to master) and the last process is excluded from sending any data (no other process to send data to).

```
if(rank!=0 && (even!=(dis_loc[rank]%2))){
 temp_var = MPI_Send(&locArr[0], 1, MPI_INT, rank-1, i, MPI_COMM_WORLD);
 if(temp_var != MPI_SUCCESS){
        return -1;
    }
}
```

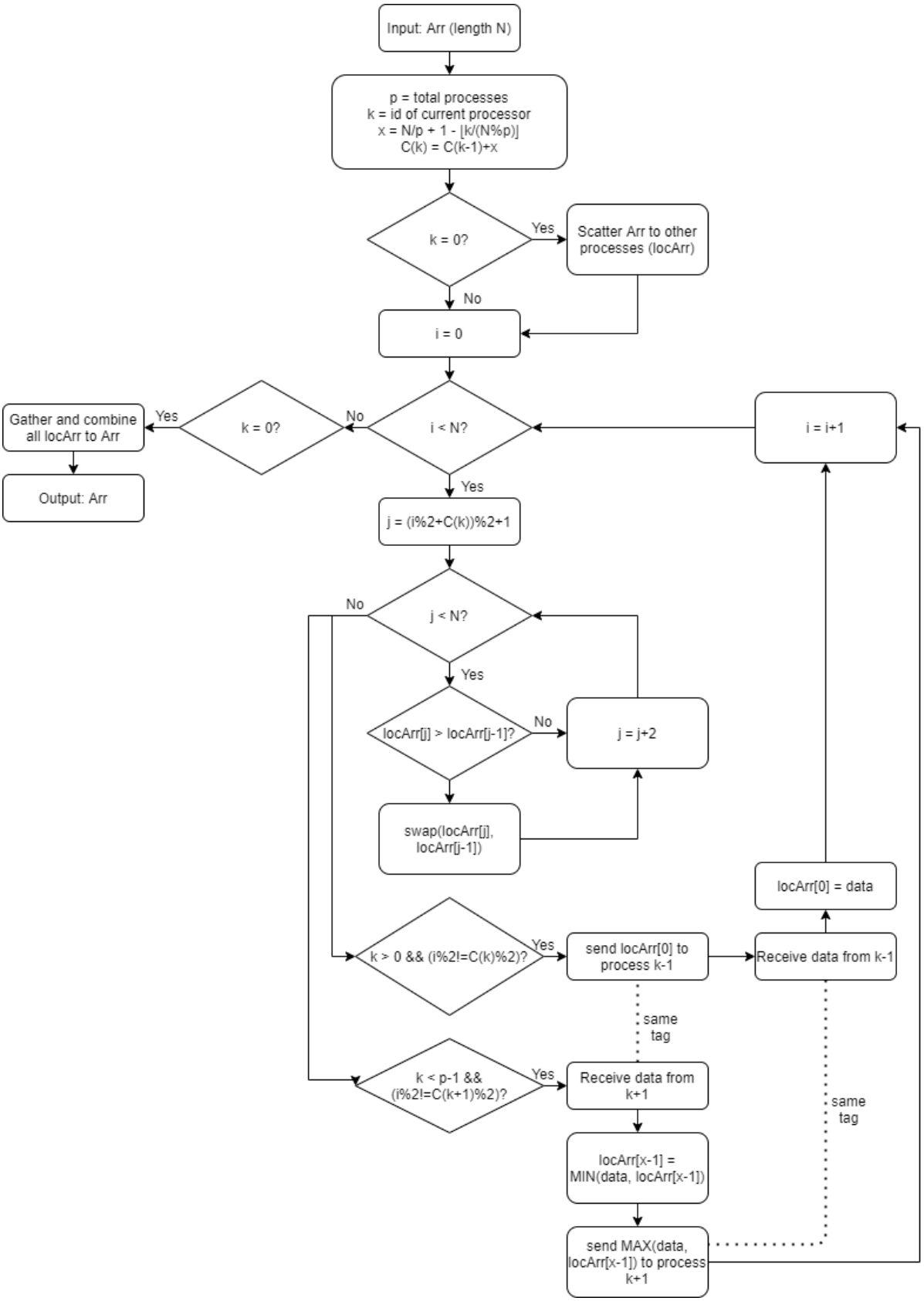
After comparing the integer supplied with the first element in their own local array, the process that receives data from another process will swap the two elements as necessary (if sent data is bigger, then swap the elements). The algorithm then operates simply, much like a sequential implementation, in which this procedure is looped N times, where N is the size of the initial input array in elements. The master process will then gather all the data from the other processes after the last iteration, rearrange them according to dis_loc, and add them to the output array known as sorted elements.

```
//receive, compare, and send the data from and to next process
if(rank!=world_size-1 && (even!=(dis_loc[rank+1]%2))) {
 temp_var = MPI_Recv(&temp_var_2, 1, MPI_INT, rank+1, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 if(temp_var != MPI_SUCCESS){
                return -1;
        }
 int temp_var = ((temp_var_2>locArr[loc_arr_len-1]) ? temp_var_2:locArr[loc_arr_len-1]);
 locArr[loc_arr_len-1] = ((temp_var_2<=locArr[loc_arr_len-1]) ? temp_var_2:locArr[loc_arr_len-1]);
 temp_var_2 = MPI_Send(&temp_var, 1, MPI_INT, rank+1, i, MPI_COMM_WORLD);
 if(temp_var_2 != MPI_SUCCESS){
                return -1;
    }
}

//receive data from previous process
if(rank!=0 && (even!=(dis_loc[rank]%2))){
 temp_var = MPI_Recv(&locArr[0], 1, MPI_INT, rank-1, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 if(temp_var != MPI_SUCCESS){
                return -1;
            }
}
even = (even+1)%2;
MPI_Barrier(MPI_COMM_WORLD);
}

//Collect and combine sorted local arrays
temp_var = MPI_Gatherv(locArr, loc_arr_len, MPI_INT, elements, send_count, dis_loc, MPI_INT, 0, MPI_COMM_WORLD);
if(temp_var!=MPI_SUCCESS){
 return -1;
}
```

Note that multiple variables and arrays are dynamically allocated for both the sequential and parallel implementation to be able to support large array numbers. Additionally, note that in the parallel implementation, the initial input array and the output sorted array is only dynamically allocated by the master process.


Input: Arr (length N)
p = total processes
k = id of current processor
x = N/p + 1 - [k/(N%p)]
C(k) = C(k-1)+x
k = 0?
Yes
Scatter Arr to other processes (locArr)
No
i = 0
i < N?
No
k = 0?
Yes
Gather and combine all locArr to Arr
Output: Arr
Yes
i = i+1
j = (i%2+C(k))%2+1
j < N?
No
Yes
locArr[j] > locArr[j-1]?
No
j = j+2
swap(locArr[j], locArr[j-1])
k > 0 && (i%2!=C(k)%2)?
Yes
send locArr[0] to process k-1
Receive data from k-1
locArr[0] = data
same tag
k < p-1 && (i%2!=C(k+1)%2)?
Yes
Receive data from k+1
same tag
locArr[x-1] = MIN(data, locArr[x-1])
send MAX(data, locArr[x-1]) to process k+1

3. Experiment

The experiment is done on the HPC cluster with slurm. The HPC environment and sbatch configurations are as shown below.

```
[119010520@node21 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
Stepping:              7
CPU MHz:               999.902
CPU max MHz:           3200.0000
CPU min MHz:           1000.0000
BogoMIPS:              4800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              14080K
NUMA node0 CPU(s):     0-9,20-29
NUMA node1 CPU(s):     10-19,30-39
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr p
rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dte
3dnowprefetch epb cat_l3 cdp_l3 intel_ppin intel_pt mba tpr_shadow vnmi f
t clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occ
l_stibp arch_capabilities
[119010520@node21 ~]$
```

- Data Collecting Method

During this experiment, the problems computed by both sequential and parallel algorithms are randomly generated. This random generation utilizes the available random function provided by "cstdlib" library. Hence, it may or may not use significant time that possibly affect the algorithms' performance. The time measurement for parallel algorithm is located before the distribution of the problems to other processes and after collecting the results from other processes into a single process. These timing methods for both algorithms are expected to give the best experiment results since they only evaluate the time needed for a data to be completely processed.

- Analysis
  - Number of processors vs. time

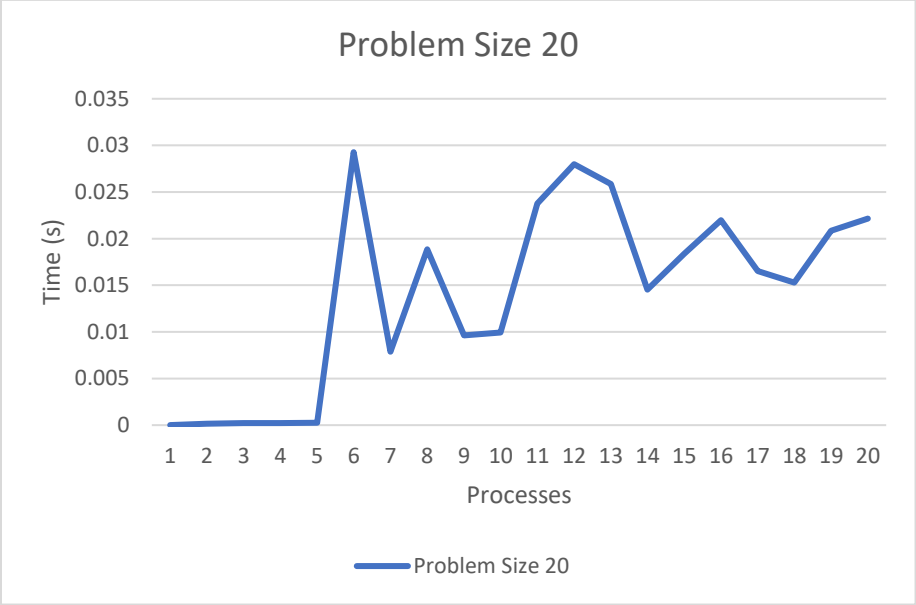


Figure 1. Number of processors vs. time (problem size 20)

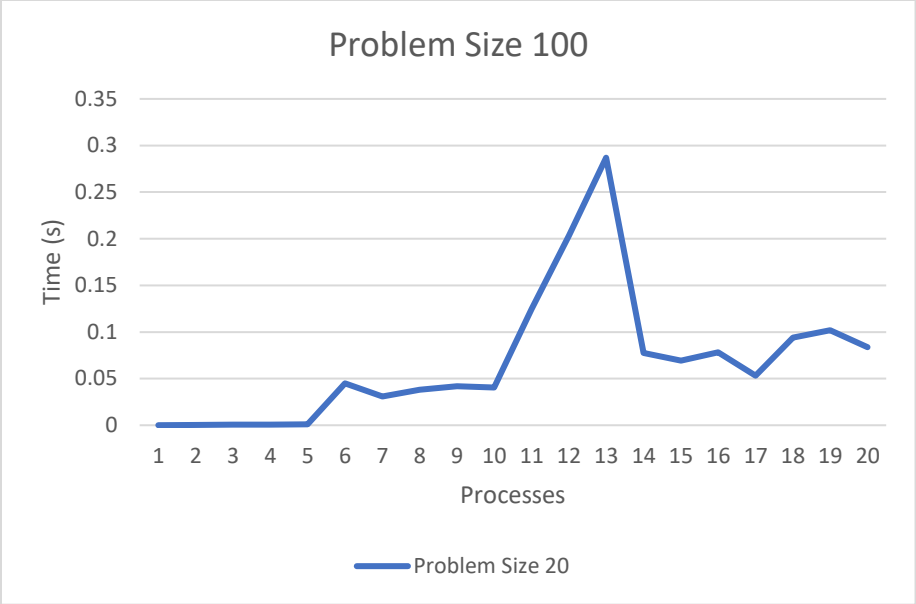


Figure 2. Number of processors vs. time (problem size 100)

Figures 1 and 2 show that the performance of a sequential approach is significantly superior for problems with 20 and 100 elements, respectively. The time spent in a sequential algorithm is only used for calculation, whereas time spent in a parallel approach is split between communication and computation. However, the time saved by distributing the issues cannot replace the time required for communication time (TCOMM > TCOMP) since the time required to communicate between processes is significantly longer than the computation time for small problem sizes. As a result, the parallel approach requires longer to complete than the sequential algorithm.

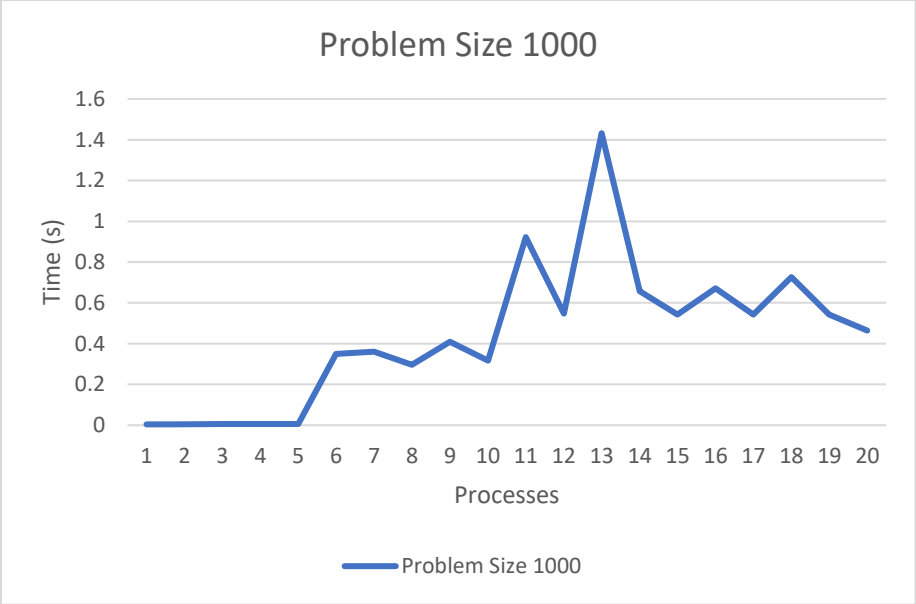


Figure 3. Number of processors vs. time (problem size 1000)

However, we see improvement for the parallel algorithm with problem size 1000. The average time with 2 processors is higher for about 0.0003 seconds or less than 8% than the time of the sequential one. Moreover, if we go back to the raw data, table 1 and table 2 show that the time for 2 processors are lesser than the sequential time. This shows that 1000 problem size may be the turning point where the parallel algorithm starts to show its advantage over the sequential algorithm ($T_{COMM} \approx T_{COMP}$).

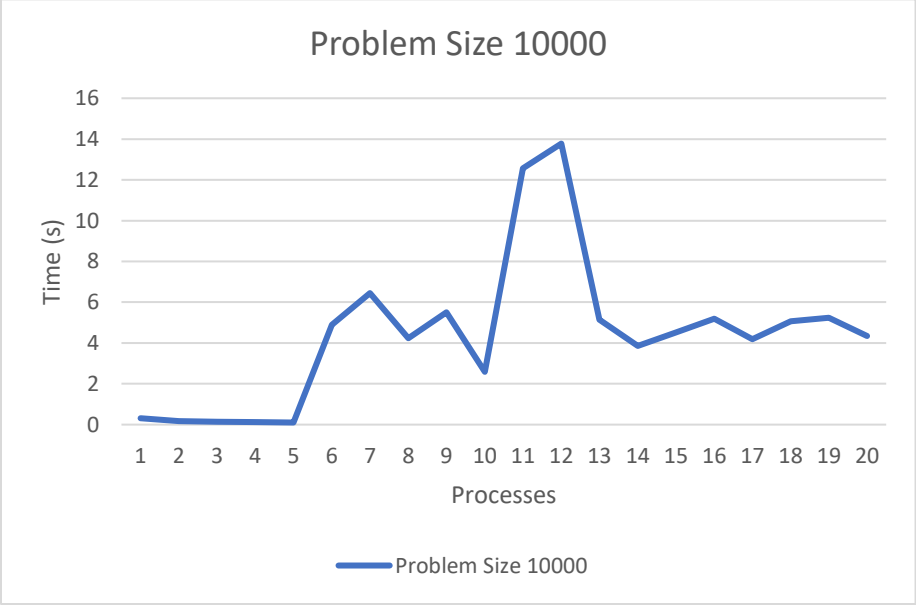


Figure 4. Number of processors vs. time (problem size 10000)

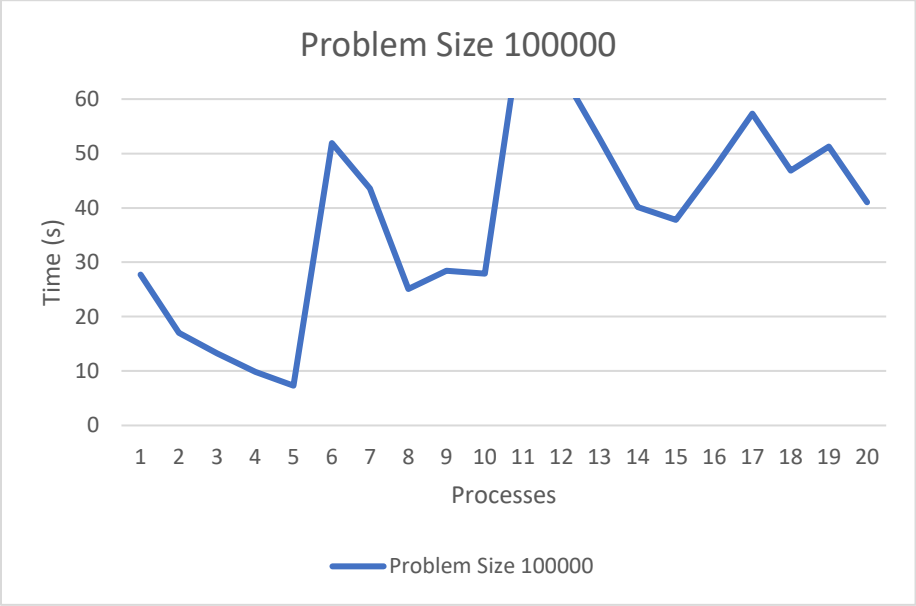


Figure 5. Number of processors vs. time (problem size 100000)

In the figure 4 and figure 5, we can finally see that the parallel algorithm can finally be faster (or even significantly faster in figure 5) with 2 to 5 processes than the sequential algorithm. This happens because the computation time, which is much larger than the communication time in these two cases ($T_{COMM} < T_{COMP}$), is distributed to some processes. Moreover, the time saved by distributing the problems is also larger than the T_{COMM}. Thus, the difference between the saved time and the T_{COMM} is what makes the parallel algorithm faster than the sequential algorithm.

Besides discussing about the comparison between sequential and parallel algorithms, it is also interesting to discuss about the pattern on the parallel algorithm. As we can see that in all figures, the line rises significantly from 5 processors to 6 processors. This is not just a coincidence, but it is caused by the architecture of the computer used to execute this algorithm (see figure 6).

IV. Conclusion

In the end, the limitation of computer power required for sorting algorithms can be addressed by parallel computing, especially when there are numerous little problems. We tested several parallel computing setups and discovered that parallel computing works better for solving large-scale computation challenges. We examined the performance of sequential and parallel algorithms and discovered that parallel computing can be used to speed up the sorting method and, as a result, get the output more quickly. However, when comparing the cost-effectiveness of parallel and sequential algorithms, the results are extremely discouraging because parallel algorithm's efficiency is far inferior to sequential algorithms. Therefore, it is still an interesting topic to find a better way in improving the parallel computing, such as developing a more efficient implementation of the algorithm.