

# **CSC 4005 Project 1**

## **Odd & Even Transposition Sort**

### **Project Introduction**

There are currently many different types of sorting algorithms, each with unique benefits and drawbacks. But none of them can be faster than  $O(n \log n)$  in terms of time complexity. This is disappointing because we need to sort through a ton of data in this big data era. Although there is a way to make our computer run faster, the limits of present technology are virtually reached. Thus, parallel computing presents itself as a better notion for a solution. Even if we might not have better computer power, we can combine some power to provide a higher performance outcome.

This computing assignment tries to familiarize us with parallel programming. We will have to use MPI to sequentially and parallel program an Odd-Even transposition sort. The kernel version I am currently using is version **3.10.0-862.el7.x86\_64**.

### **Design Approach**

#### **Assumptions:**

Related to the Bubble Sort Algorithm, we cycle over an integer array in the Odd-Even Transposition Sort Algorithm and exchange the items based on whether they are smaller or larger than their neighbors. The main distinction is that we swap the elements in the array alternatively between those with odd and even indexed positions, as opposed to starting at the top of the array and working our way down. The algorithm must traverse the entire array at each iteration and evaluate each neighbor before deciding whether to swap or not. An element can only move one step left or right, at most, as a result. In the worst situation, an element from the array's tail

ends up in the head of the array due to a misplacement. This will require  $N$  iterations sequentially because the algorithm must traverse the array  $N$ . Keep in mind that  $N$  represents the array's element count (data input size). As a result, the runtime ( $O(N^2)$ ) grows quadratically to the input size. The code will compare the right neighbors rather than the left ones in both sequential and parallel versions of the technique.

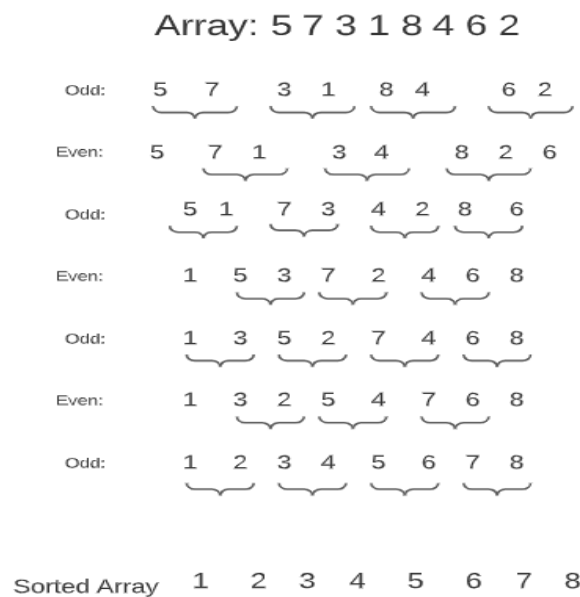


Figure 2.1 Odd Even Transposition Sort Example

### Sequential Sorting Implementation:

It's pretty simple to implement sequential sorting. It works in a manner similar to bubble sort. Two for loops, one nesting inside the other, are used in its sorting mechanism. Inside the inner for loop, it compares the adjacent items (the elements being compared depend on whether they are in the odd or even phase) and moves them about if necessary. When  $N$  is the number of data elements in the array, this is performed  $N$  times. This implementation begins with elements with odd indexes and proceeds on to those with even ones. The sequential variant first goes through every element in the array with an odd index before switching to an element with an even index and alternating between the two until  $N$  iterations have been reached. The

algorithm modulates the current iteration number ( $i$ ) by 2 to check the current phase. This is in odd phase if it leaves a leftover, and vice versa. It is denoted in the code as  $j = (i \% 2)$ . This continues until all  $N$  data items have been traversed by the algorithm for loop.

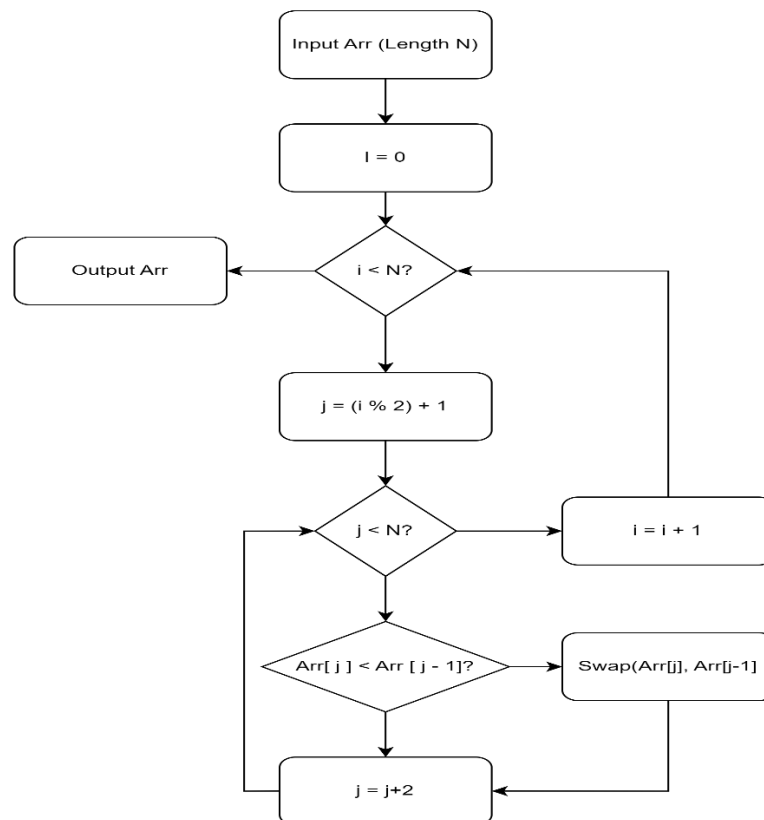


Figure 2.2 Sequential Sorting Flow Diagram

```

// Sorting Process

i = 0;
while(i < num_elements) {
    for (int j = (i % 2) + 1; j < num_elements; j += 2) {
        if(compare(&elements[j], &elements[j - 1]))
            std::swap(elements[j], elements[j - 1]);
    }
    i++;
}
  
```

Figure 2.3 Sequential Sorting Process Algorithm

### Parallel Sorting Implementation:

Similar logic governs both the sequential and parallel implementations. The consistency between both implementations is maintained by using the same logic and

comparison technique (compares next element), which should result in a more accurate comparison of the two techniques. The dividing of data to give to the other processes and its parallel execution are the primary differences between a sequential and a parallel program. The distribution of the input array to the other produced processes kicks off the parallel implementation. It accomplishes this by splitting the input array's element count in half and, if possible, distributes the remaining components evenly among all other processes, including the master process. The procedure will distribute one of the remaining data items to each additional process, starting with the master, until there are no more remainders (allocates  $(N \% \text{num\_proc} + 1)$  data to the first  $(N \% \text{num\_proc})$ , in the event that the input array size is not divisible by the number of processes. This distribution technique guarantees that any process's input data deviate by no more than one element.

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 information (which index does their local array start at in the original input array).

The starting index of each process' local arrays are kept in a separate array called displacement, and the cycle it is currently on is kept in a Boolean variable called phase. The beginning position of the local array in the initial input array can be obtained by calling `displacement[rank]` (where rank is the process's rank). It can be determined whether a process needs to send or receive data to or from other processes

by comparing these two variables (if starting index parity does not match with the phase, then it would need to send data). As there are no processes that send data forward to the master process, the master process is not allowed to receive any data, and the final process is not allowed to send any data (no other process to send data to).

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 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 displacement, and add them to the output array known as sorted elements.

```

MPI_Scatterv(elements, tally_array, displace_array, MPI_INT, my_elements, num_my_elements, MPI_INT, 0, MPI_COMM_WORLD);

if(rank == 0) delete[] elements;
bool level = 0;
while (i < num_elements) {

    sort(my_elements, num_my_elements, displace_array, level, rank);

    if(rank != world_size - 1 && not_match(level, rank + 1, displace_array)) {
        int n;
        MPI_Recv(&n, 1, MPI_INT, rank + 1, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(compare(&n, &my_elements[num_my_elements - 1]))
            std::swap(n, my_elements[num_my_elements - 1]);
        MPI_Send(&n, 1, MPI_INT, rank + 1, i, MPI_COMM_WORLD);
    }

    if(rank != 0 && not_match(level, rank, displace_array)) {
        MPI_Send(&my_elements[0], 1, MPI_INT, rank - 1, i, MPI_COMM_WORLD);
        MPI_Recv(&my_elements[0], 1, MPI_INT, rank - 1, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    level = !level;
    MPI_Barrier(MPI_COMM_WORLD);
    i++;
}

if(rank == 0) sorted_elements = new int[num_elements];
MPI_Gatherv(my_elements, num_my_elements, MPI_INT, sorted_elements, displace_array, MPI_INT, 0, MPI_COMM_WORLD);

```

Figure 3.1 Parallel Sorting Process Code

```

while(i < world_size) {
    tally_array[i] = (i < remainder) ? num_elements / world_size + 1 : num_elements / world_size;
    displace_array[i] = store;
    store += tally_array[i];
    i++;
}

```

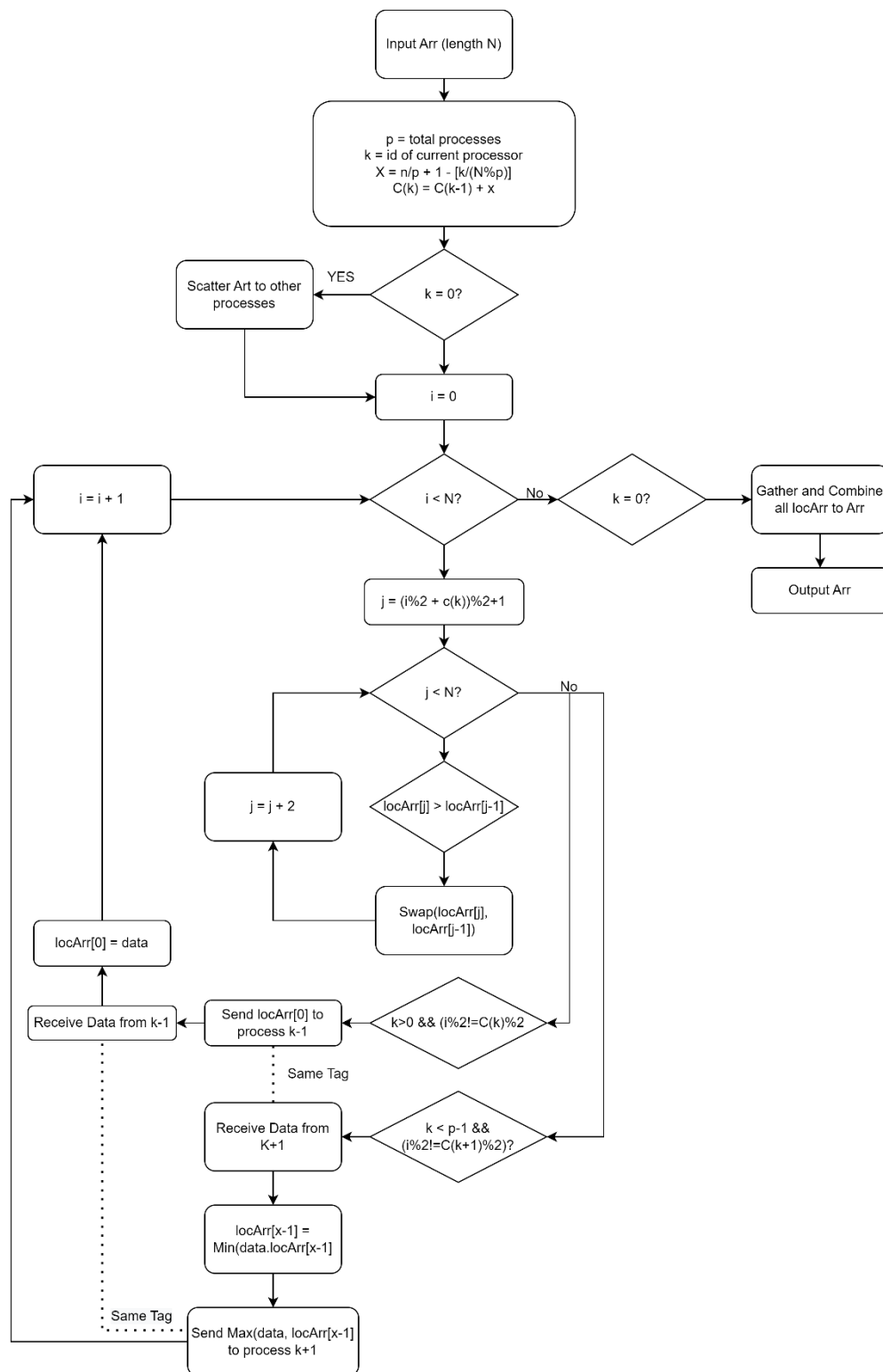


Figure 3.3 Parallel Sort Flow Diagram

## Code Experiment and Data Visualization

The data experiment is done in HPC. User are required to login to their ssh and run Sbatch\_template.sh to run each processor and input concurrently. There are 2 CPU Chips assigned for every program running. The input for every data will be saved in an in.a file and the output will be send to a parallel.out or seq.out file. The test\_data\_generator.cpp is used for the source code to generate the random input numbers. In this case I use 4 different data input including 100, 1000, 10000 and 20000. Each data input will be tested with 5 different processor: 1, 5, 10, 20, 30 processors.

### *Sequential Sorting (Per Second)*

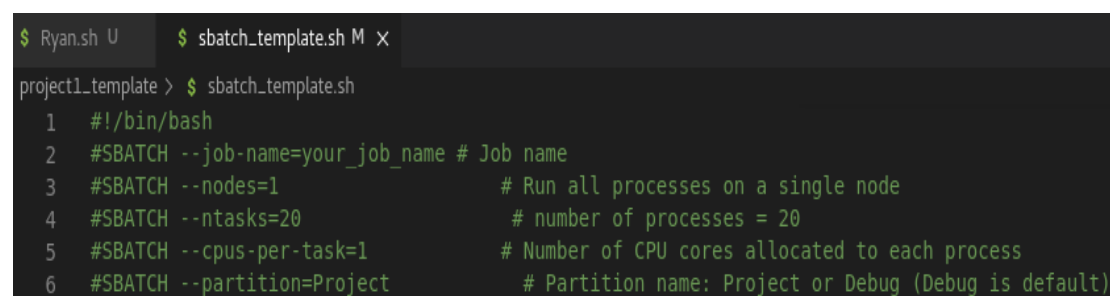
#PROCESSOR	100	1000	10000	200000
<b>1</b>	0.000160055	0.0123523	0.512418	215.293

### *Parallel Sorting (Per Second)*

#PROCESSOR	100	1000	10000	200000
<b>5</b>	0.00166495	0.0106119	0.0930189	16.3646
<b>10</b>	0.00316179	0.0147058	0.119906	11.0075
<b>20</b>	0.00243518	0.0197678	0.157561	9.82814
<b>30</b>	4.09398	55.396	N/A	N/A

Note:

An N/A result appeared due to a large run time for one task which exceeded the time limit given by the rule.



```

$ Ryan.sh U    $ sbatch_template.sh M x
project1_template > $ sbatch_template.sh
1  #!/bin/bash
2  #SBATCH --job-name=your_job_name # Job name
3  #SBATCH --nodes=1                # Run all processes on a single node
4  #SBATCH --ntasks=20              # number of processes = 20
5  #SBATCH --cpus-per-task=1        # Number of CPU cores allocated to each process
6  #SBATCH --partition=Project      # Partition name: Project or Debug (Debug is default)

```

Figure 4.1 SBatch Template to run concurrently

## Data Analysis

From the data in the first table (Sequential Sorting / 1 Processor) we can see that for small data such as 100 and 1000 sequential sorting is very efficient. It took only a small amount of time to finish the task. However, for bigger data it took much slower time to finish each task compared to the second table (Parallel Sorting/ >1 Processor). The fact that it takes longer to communicate between the processors because it is anticipated that they would take a constant amount of time is what is responsible for this effect. The sequential sorting algorithm, meanwhile, starts sorting immediately when the timer starts because it doesn't need to interact. Due to the small problem size, the parallel sorting method takes longer than the sequential implementation since the communication time lost between processors in the parallel implementation surpasses the time saved by the extra processors.

We can see that the parallel approach has surpassed the sequential implementation in execution time by focusing on the results for the 10000-problem size. Comparing the parallel algorithm to the sequential algorithm, the difference in speed is roughly a factor of 5. As a result, we might speculate that the crucial input size is in the range of 1000 data elements. It is significant to observe the tendency toward parallel execution as processor counts rise. Both table demonstrates that the execution times for problems with sizes of 1000 and 10,000 continue to generally increase as the number of processors increases. The contrast between the sequential implementation and the parallel implementation for smaller problem sizes can similarly explain this result. The amount of communication time needed increases along with the number of processors. As a result, because the input size is still too tiny in these situations, the additional benefit of more processors is sufficient to offset and overcome the communication time that more processors provide.



For problems with a size of 200000, the last column shows that the sequential algorithm's execution time is beginning to rise sharply. This is because the sequential implementation of the sorting algorithm has a quadratic run time (as discussed in the design section).

When compared to a sequential solution, it is substantially lower when considering parallel sorting techniques. Here, the advantage of the speedup brought on by many processors is seen very clearly. Additionally, it is clear that, in general, the execution time will decrease when the number of processors is increased in this issue size range. This indicates that the advantages of using many processors have finally outweighed their increased communication time.

### **Project Summary**

This assignment enabled us to comprehend how parallel computing outperforms sequential computing in terms of performance. The study of using different numbers of processors and input sizes to run the odd-even transposition sorting algorithm managed to identify the input size breakpoints at which the parallel sorting algorithm begins to outperform the sequential implementation and at which adding more processors improves execution time rather than slows it down (increasing execution time). To determine which implementation (sequential vs. parallel) and how many processors would result in the fastest execution time, it is crucial to understand the context of where the program will run depending on the input sizes to process.

## APPENDIX

```

[csc4005@localhost project1_template]$ g++ -std=c++11 odd_even_sequential_sort.cpp -o ssort
[csc4005@localhost project1_template]$ ./ssort 1000 ./test_data/1000a.in
actual number of elements:1000
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 0.00990778 seconds
Input Size: 1000
Process Number: 1
[csc4005@localhost project1_template]$ g++ check_sorted.cpp -o check
[csc4005@localhost project1_template]$ ./check 1000 ./test_data/1000a.in.seq.out
Sorted.
[csc4005@localhost project1_template]$ █

```

Figure 5.1 Sequential Sort with Input Size 1000

```

[csc4005@localhost project1_template]$ g++ -std=c++11 odd_even_sequential_sort.cpp -o ssort
[csc4005@localhost project1_template]$ ./ssort 10000 ./test_data/10000a.in
actual number of elements:10000
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 0.973483 seconds
Input Size: 10000
Process Number: 1
[csc4005@localhost project1_template]$ g++ check_sorted.cpp -o check
[csc4005@localhost project1_template]$ ./check 10000 ./test_data/10000a.in.seq.out
Sorted.
[csc4005@localhost project1_template]$ █

```

Figure 5.2 Sequential Sort Input Size 10000

```

100 n = 10
actual number of elements:100
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 0.00316179 seconds
Input Size: 100
Process Number: 10

```

Figure 5.3 Parallel N=10 100 Size 100

```

200000 n = 10
actual number of elements:200000
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 11.0075 seconds
Input Size: 200000
Process Number: 10

```

Figure 5.4 Parallel N=10 Size 200000

```

10000 n = 20
actual number of elements:10000
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 0.157561 seconds
Input Size: 10000
Process Number: 20

```

Figure 5.5 Parallel N=20 Size 10000

```

200000 n = 20
actual number of elements:200000
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 9.82814 seconds
Input Size: 200000
Process Number: 20

```

Figure 5.6 Parallel N=20 Size 200000

```
100 n = 30
actual number of elements:0
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 4.09398 seconds
Input Size: 100
```

Figure 5.7 Parallel N=30 Size 100

```
1000 n = 30
actual number of elements:0
Student ID: 119010507
Name: Ryan Christopher
Assignment 1
Run Time: 55.396 seconds
Input Size: 1000
Process Number: 30
```

Figure 5.8 Parallel N=30 Size 1000

```
10000 n = 30
actual number of elements:0
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** JOB 9484 ON node22 CANCELLED AT 2022-10-12T06:47:53 DUE TO TIME LIMIT ***
slurmstepd: error: *** STEP 9484.14 ON node22 CANCELLED AT 2022-10-12T06:47:53 DUE TO TIME LIMIT ***
```

Figure 5.9 Parallel N=30 Size 10000 Result Time Exceed N/A