The Chinese University of Hong Kong, Shenzhen
CSC4005 Distributed and Parallel Computing
Homework 1

Name: Tian Min

Student ID: 116010168

Date: 2019.10

I. INTRODUCTION

In this homework, you are required to write a parallel odd-even transposition sort by using MPI. A parallel odd-even transposition sort is performed as follows:

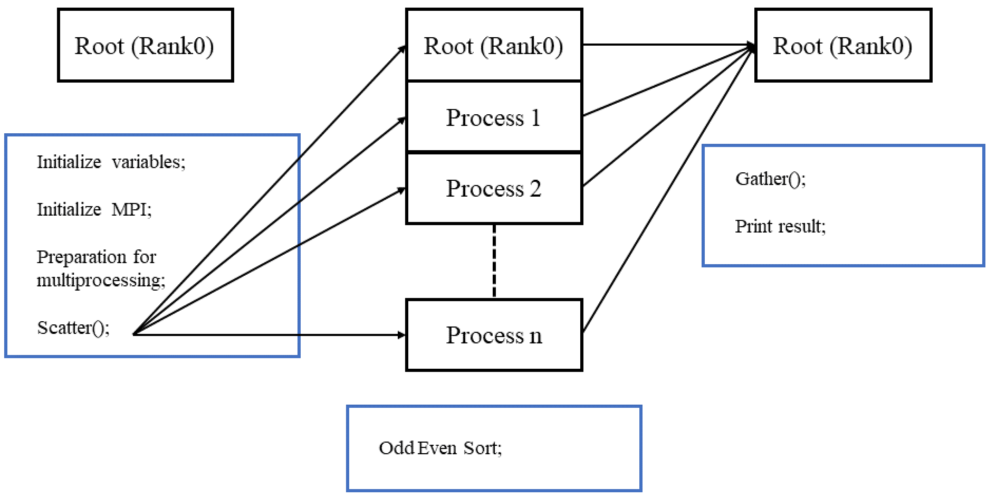
/* Initially, m numbers are distributed to n processes, respectively. */

1. Insides each process, compare the odd element with the posterior even element in odd iteration, or the even element with the posterior odd element in even iteration respectively. Swap the elements if the posterior element is smaller.
2. If the current process rank is P, and there some elements that are left over for comparison in step 1, Compare the boundary elements with process with rank P-1 and P+1. If the posterior element is smaller, swaps them.
3. Repeat 1-2 until the numbers are sorted.

You need to use MPI to design the program. The number of processors used to execute the program is n that is much less than m.

II. DESIGN

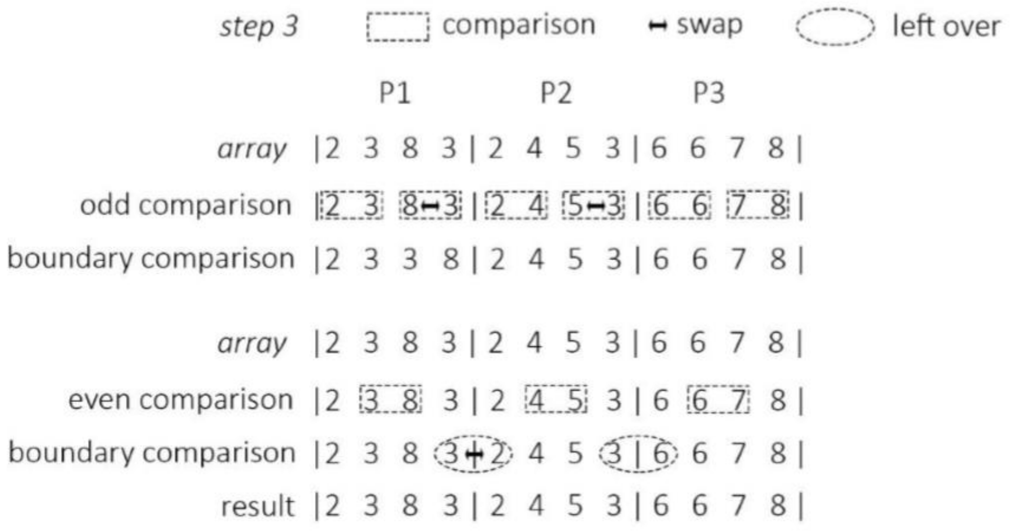
1. Program Flow



From left to right, there are mainly three steps designed in the program:

(a) First part: the program goes into the main function, it initializes the variables including random array generation function's configuration, the storage used to contain global and local array, their lengths and other information. Then, it does the MPI initialization, including space allocation, specifying communicator, and get the rank & processor number. At the end of this part, the root process called the Scatter function in order to distribute the unsorted array to sub-processes.

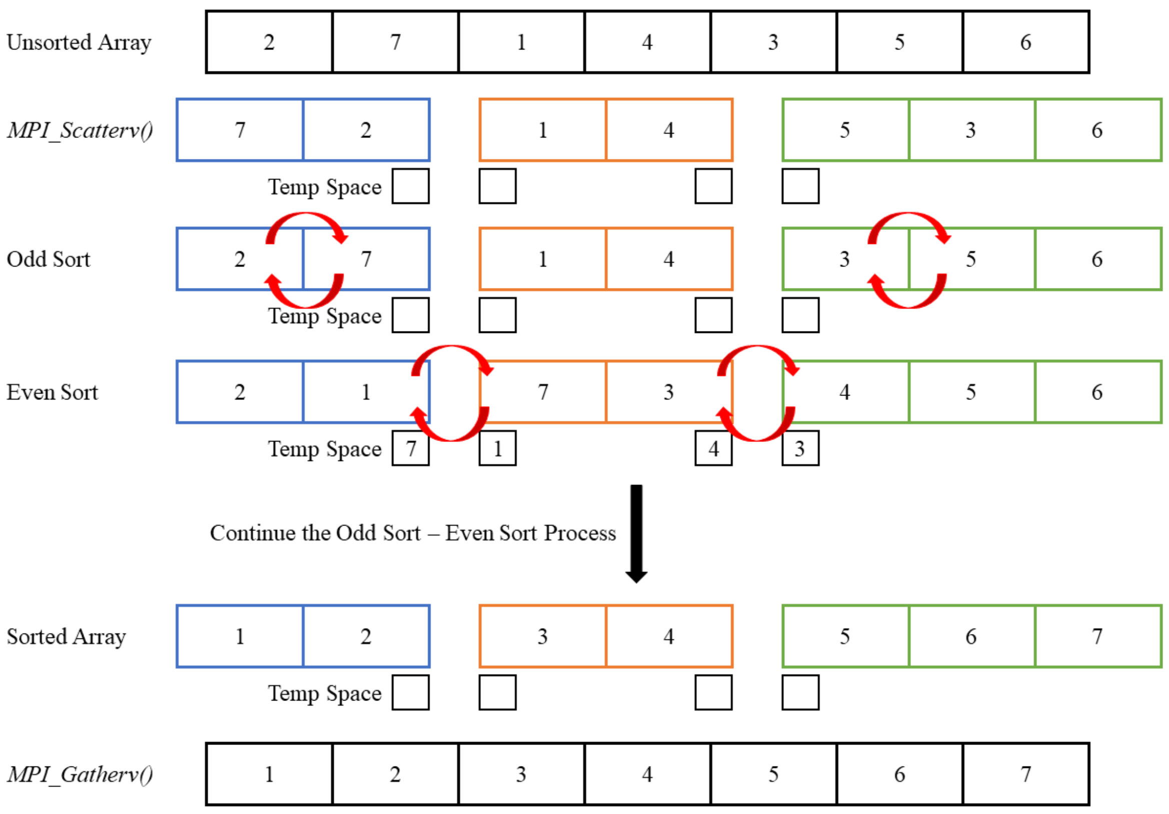
(b) Second part: p processes are going to do the odd even sort together, in the way demonstrated in the HW requirement:


step 3
comparison
swap
left over
P1
P2
P3
array |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
odd comparison |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
boundary comparison |2 3 3 8 | 2 4 5 3 | 6 6 7 8 |
array |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
even comparison |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
boundary comparison |2 3 8 3 2 4 5 3 | 6 6 7 8 |
result |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |

As showing above, the *OddEvenSort()* function should not only implement the comparison algorithm, but also the *MPI_Send()* and *MPI_Recv()* so that the processes can communicate with each other.

(c) Third part: the root process call the *MPI_Gather(),* all the sub-processes submit sorted array to the root. Then print the result.

2. Data Structure Design



An array is used to contain the numbers. After scatter, there's a mechanism that test if the current process is holding the sub-array on both ends or in the middle. The program will allocate different number of buffer spaces depends on the sub-array's position: sub-arrays on the ends get one; sub-arrays in the middle get two.

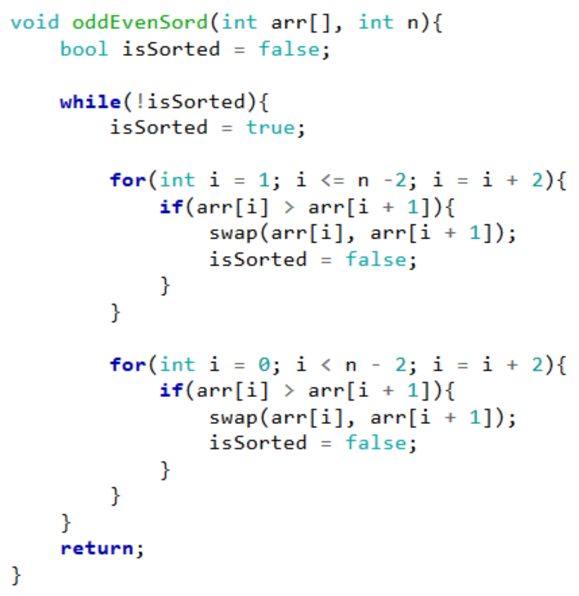
3. Algorithm – Odd Even Transposition Sort

Odd-even transposition sort works by iterating through the list, comparing adjacent elements and swapping them if they're in the wrong order. The complexity analysis is shown in the chart below:

Worst Case	Best Case	Average Case	Space

$O(n^2)$	$\Theta(n)$	$O(n^2)$	$O(1)$ auxiliary

C++ implementation:



```
void oddEvenSord(int arr[], int n){
    bool isSorted = false;

    while(!isSorted){
        isSorted = true;

        for(int i = 1; i <= n -2; i = i + 2){
            if(arr[i] > arr[i + 1]){
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }

        for(int i = 0; i < n - 2; i = i + 2){
            if(arr[i] > arr[i + 1]){
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
    return;
}
```

Suppose we have *n* numbers array and *p* process, in MPI version of odd even sort, each process sorts its $\frac{n}{p}$ numbers; performs *p* passes of odd-even interchanges; finally, each process has its sorted array. The complexity in the parallel Odd Even Sort can be calculated as follow:

Each process can sort in $O(\frac{n}{p}\log\frac{n}{p})$

At each phase(odd/even):

The communication is $O(\frac{n}{p})$

Merging of two sequence is $O(\frac{n}{p})$

Since we have p phases, the total work of communication and merging should be $O(n)$

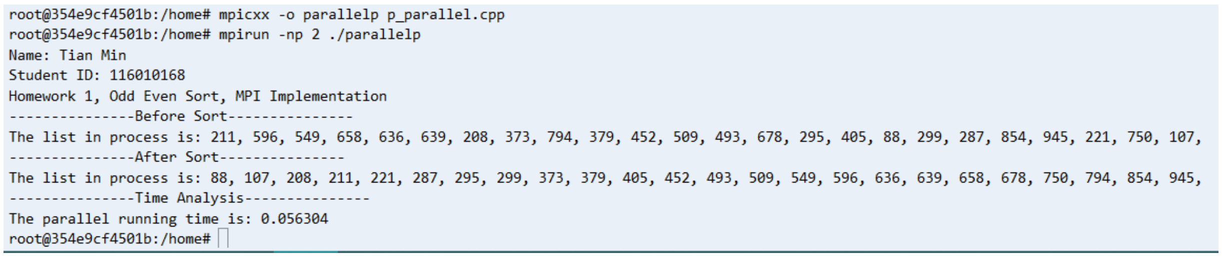
Hence, the parallel time should be $O\left(\frac{n}{p}log\frac{n}{p}\right) + O(n)$

Here is the pseudo code for processes communication:

```
Sort local keys;
for(phase = 0; phase < comm_size; phase++){
    partner = Compute_partner(phase, my_rank);
    if(I am not idle){
        send my keys to partner;
        receive keys from partner;
        if(my_rank < partner)
            keep smaller keys;
        else
            keep larger keys;
    }
}
```

III. Program Execute

Expected output:



```
root@354e9cf4501b:/home# mpicxx -o parallelp p_parallel.cpp
root@354e9cf4501b:/home# mpirun -np 2 ./parallelp
Name: Tian Min
Student ID: 116010168
Homework 1, Odd Even Sort, MPI Implementation
---------------Before Sort---------------
The list in process is: 211, 596, 549, 658, 636, 639, 208, 373, 794, 379, 452, 509, 493, 678, 295, 405, 88, 299, 287, 854, 945, 221, 750, 107,
---------------After Sort---------------
The list in process is: 88, 107, 208, 211, 221, 287, 295, 299, 373, 379, 405, 452, 493, 509, 549, 596, 636, 639, 658, 678, 750, 794, 854, 945,
---------------Time Analysis---------------
The parallel running time is: 0.056304
root@354e9cf4501b:/home#
```

To Run the program, first, compile it under the same directory, and then, run it with *mpirun*:

mipcxx -o parallel parallel.cpp

mpirun -np **numbers_of_process** ./ parallel

To Run the sequential program:

g++ sequential.cpp -o sequential and ./sequential

IV. Performance Analysis

1. Test Machine

Due to the jam on the public server, the perform was analysed on personal server with AWS Cloud9 as IDE:

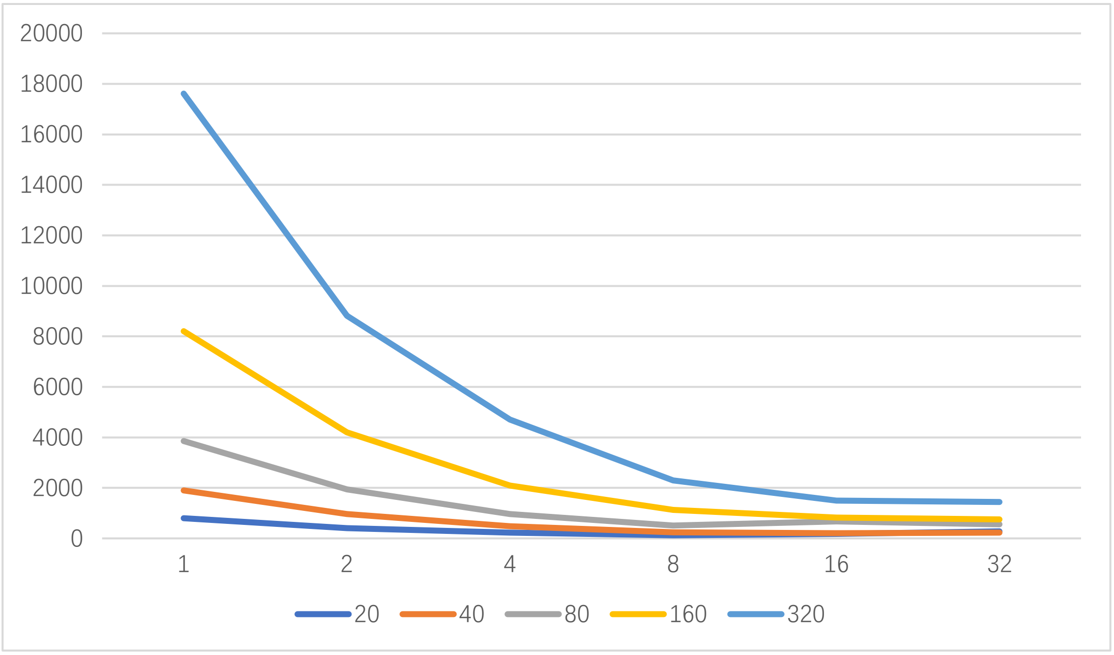
AWS EC2 a1.xlarge[1] 4vCPU 8GiB
ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server
Using the dockerfile provided in the tutorial.

[1] https://aws.amazon.com/jp/ec2/instance-types/?nc1=h_ls

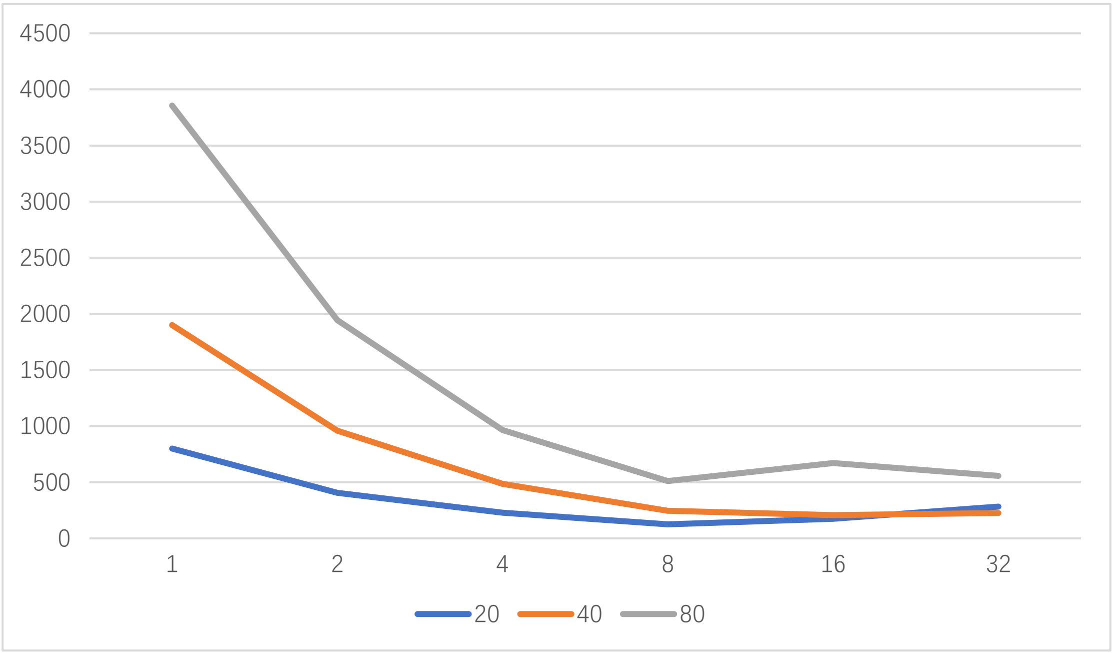
2. Test Design

Array Size: 20K, 40K, 80K, 160K, 320K

Process Number: 1, 2, 4, 8, 16, 32



(Figure1: Chart with All Array Length)

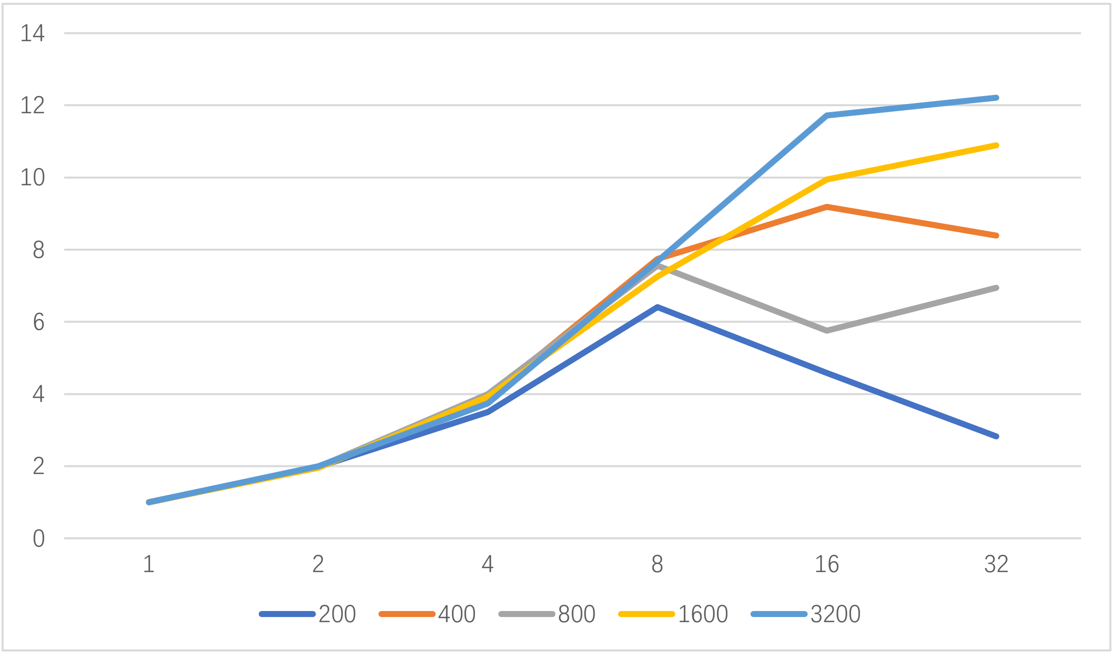


(Figure 2: Chart Without 160K & 320K, Time in Millisecond)

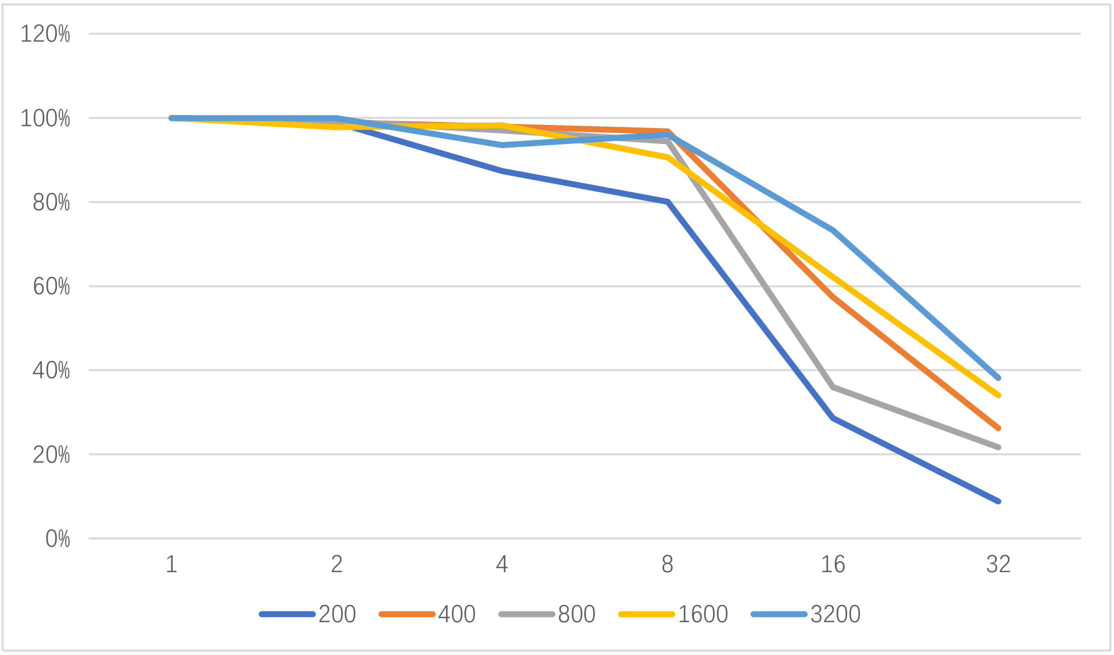
Due to the time limitation, all the data were collected after a single execution rather than several run’s average value. Figure 1 & 2 illustrate the time spent to sort the array. Macroscopically, regardless of the length of the array, increasing the number of threads can reduce runtime to a certain extent. However, when the number of process larger than 16, its speed-up capability is significantly limited. It is worth noting that when the array length is 20K and 80K, the running time is slower as the number of threads increases from 8 to 32. The reason behind this is the communication overhead, if we test it with 64 processes or 128 processes, there must be a more significant bottom rebound. However, we can amplify the impact of this process overhead by calculating speedup and efficiency. According to the speedup and efficiency formula:

$$Speedup = 1 - \frac{1}{serialTime + \frac{parallelTime}{p}}$$

$$Ep = \frac{serialTime}{p * parallelTime}$$



(Figure3: Speedup)



(Figure 4: Efficiency)

Figure 3 illustrate the speedup of the parallel program. We can see that when the data size is much larger than the process number (1600K & 3200K), there is a stable increment on the speedup. As for the efficiency, the situation is similar: when the process number larger than 8, efficiency drop steeply.

V. Conclusion & Improvement

To sum up, this HW1 experiment can support the relationship between the parallel computation process and the operating rate mentioned in the lecture. For the potential improvement, when the number of array and the process number are not divisible, the last process would handle more numbers, this could be improved by share the extra number to other processes. The performance should be better than handling by itself. There are few potential reasons listed for the decrement on speedup:

1. Parallel Overhead:

   Inter-processor communication: increase data locality to minimize communication.

   Load imbalance: distribution of work (load) is not uniform. Inherent parallelism of the algorithm is not sufficient.

   Extra-computation: modify the best sequential algorithm may result in extra-computation.

2. Machine Performance Limitation:

   Didn’t get enough fund for a c5.4xlarge (computation speciality machine) on AWS. The performance of EC2 entities is unstable. I get great variance even testing the program with only 2 processes.

VI. Source Code

Parallel Version

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <time.h>
#include <iomanip>
#include <bits/stdc++.h>
using namespace std;

const int RMAX = 1000;
void GenerateList(int local_array[], int local_num);
void PrintList(int* Arr, int n);
int OddEvenSort(int local_array[], const int local_num, const int
my_rank, int p, MPI_Comm comm);

int main(int argc, char* argv[]) {
    MPI_Comm comm;
    int rank;
    int processor_num;
    int global_num = 20;
    int local_num;
    int* Arr = 0;
    int n = 0;
    clock_t start_time;
    clock_t end_time;
    double time;

    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processor_num);

    int* global_array = (int*)malloc(sizeof(int) * global_num);
    local_num = global_num / processor_num;

    if (rank == 0)
    {
        GenerateList(global_array, global_num);
        cout << "Name: Tian Min" << endl << "Student ID: 116010168" <<
endl << "Homework 1, Odd Even Sort, MPI Implementation" << endl;
        cout << "---------------Before Sort---------------" << endl;
        PrintList(global_array, global_num);
    }

    int* local_array = (int*)malloc(sizeof(int) *local_num);
    start_time = clock();
    MPI_Scatter(global_array, local_num, MPI_INT, local_array,
local_num, MPI_INT, 0, comm);
    OddEvenSort(local_array,local_num, rank, processor_num, comm);
    MPI_Gather(local_array, local_num, MPI_INT, global_array,
local_num, MPI_INT, 0, comm);
    end_time = clock();
    time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    if (rank == 0)
    {

        cout << "---------------After Sort---------------" << endl;
        PrintList(global_array, global_num);
        cout << "---------------Time Analysis-------------" << endl;
        cout <<"The execution time is: " << setprecision(3) << time <<
" seconds" << endl;
        time = 0;
    }
    MPI_Finalize();
    return 0;
}



void GenerateList(int global_array[], int global_num) {
    int i;
    srand(clock());
    for (i = 0; i < global_num; i++) global_array[i] = rand() % RMAX;
}

void PrintList(int* Arr, int n){
    cout << "The list is: ";
    for (int count = 0; count < n; count++) {
        cout << Arr[count] << " | ";
    }
    cout << endl;
```

```
}


int OddEvenSort(int local_array[],const int local_num,const int
my_rank,int p, MPI_Comm comm)
{
    int n = local_num;
    int temp = 0;
    int send_temp = 0;
    int recv_temp = 10001;
    int rightrank = (my_rank + 1) % p;
    int leftrank = (my_rank + p - 1) % p;

    for (int k = 0; k < p * n; k++)
    {
        if (k % 2 == 0)
        {
            for (int j = n - 1; j > 0; j -= 2)
            {
                if (local_array[j] < local_array[j - 1])
                {
                    swap(local_array[j], local_array[j - 1]);
                }
            }
        }
        else
        {
            for (int j = n - 2; j > 0; j -= 2)
            {
                if (local_array[j] < local_array[j - 1])
                {
                    swap(local_array[j], local_array[j - 1]);
                }
            }
            if (my_rank != 0)
            {
                send_temp = local_array[0];
                MPI_Send(&send_temp, 1, MPI_INT, leftrank, 0, comm);
                MPI_Recv(&recv_temp, 1, MPI_INT, leftrank, 0, comm,
MPI_STATUS_IGNORE);
                if (recv_temp > local_array[0]) local_array[0] =
recv_temp;
            }
            if (my_rank != p - 1) {
                int send_buff = local_array[local_num - 1];
                MPI_Recv(&recv_temp, 1, MPI_INT, rightrank, 0, comm,
MPI_STATUS_IGNORE);
                MPI_Send(&send_buff, 1, MPI_INT, rightrank, 0, comm);
                if (recv_temp < local_array[local_num - 1])
local_array[local_num - 1] = recv_temp;
            }
        }
    }
    return 0;
}
```

Sequential Version

```cpp
#include <iostream>
#include <bits/stdc++.h>
#include <time.h>
using namespace std;

const int RMAX = 1000;

void oddEvenSord(int arr[], int n){
    bool isSorted = false;

    while(!isSorted){
        isSorted = true;

        for(int i = 1; i <= n -2; i = i + 2){
            if(arr[i] > arr[i + 1]){
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }

        for(int i = 0; i < n - 2; i = i + 2){
            if(arr[i] > arr[i + 1]){
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
    return;
}

void printArray(int arr[], int n){
    for(int i = 0; i < n; i++){
        cout << arr[i] << " ";
    }
   cout << endl;
}

void Generate_list(int Gobal_A[], int gobal_n) {
    srand(clock());
    for (int i = 0; i < gobal_n; i++) Gobal_A[i] = rand() % RMAX;
}
```

```
void printInform(){
    cout << "Name: Tian Min" << endl  << "Student ID: 116010168" << endl
<< "Homework 1, Odd Even Sort, Sequential Implementation" << endl;
}

int main( )
{
   clock_t start, end;
   int n = 20;
   int num_list[n];
   Generate_list(num_list, n);

   printInform();
   cout << "---------------Before Sort---------------" << endl;
   printArray(num_list, n);

   start = clock();
   oddEvenSord(num_list, n);
   end = clock();

   cout << "---------------Result---------------" << endl;
   printArray(num_list, n);
   cout << "Runtime: " << (double)(end-start)/CLOCKS_PER_SEC << "
seconds" << endl;

   return(0);
}
```

VII. Reference

[1] “Odd-even sort,” Growing with the Web, 03-Oct-2016. [Online]. Available: https://www.growingwiththeweb.com/2016/10/odd-even-sort.html. [Accessed: 06-Oct-2019].

[2]“Odd Even Transposition Sort” Odd Even Transposition Sort - MPI Programming. [Online]. Available: http://selkie-macalester.org/csinparallel/modules/MPIProgramming/build/html/oddEvenSort/oddEven.html#. [Accessed: 06-Oct-2019].

[3]B. Chanaka, "Overhead of Parallelism", Medium, 2019. [Online]. Available: https://medium.com/@chanakadkb/overhead-of-parallelism-d1d3c43abadd. [Accessed: 06- Oct-2019].