CSC4005 FA22 HW01

Haoran Sun (haoransun@link.cuhk.edu.cn)

1 Introduction

Similar to bubble sort, the odd-even sort is a sorting algorithm with complexity $O(n^2)$. In this assignment, the MPI library was utilized to improve the sorting speed with the benefit of multi-processing. A sequential and a parallel version of the sorting algorithm were implemented. The program was tested under different array sizes and numbers of CPU cores. The speed-up factor and CPU efficiency were also analyzed.

2 Method

2.1 Program design and implementation

The sequential and parallel sorting programs were implemented using the C++ programming language. MPICH library was used for the parallel part. The parallel version was written in `src/main.cpp`, while the sequential version was written in `src/main.seq.cpp`. Some important functions were written in `src/utils.h`.

For the flowchart, please refer to Figure A.3. The sequential sorting function is printed as the following c++ function.

```cpp
// binary sort
void odd_even_sort(int* arr, int N, int f){
    if (f==1) return;
    int a, b;
    int flag = 1;
    // odd loop
    for (int i = 1; i < N; i += 2){
        a = arr[i-1];
        b = arr[i];
        if (b < a){
            arr[i]   = a;
            arr[i-1] = b;
            flag = 0;
        }
    }
    // even loop
    for (int i = 2; i < N; i += 2){
        a = arr[i-1];
        b = arr[i];
        if (b < a){
            arr[i]   = a;
            arr[i-1] = b;
            flag = 0;
        }
    }
    return odd_even_sort(arr, N, flag);
}
```

The parallel sorting scheme is similar. However, when the odd-even comparison involves numbers in two processes, `MPI_Sendrecv` were called. For example, the following code performs the edge case by calling `MPI_Sendrecv()`.

```
if (start_idx>0 && start_idx%2==1){
    // printf(”odd start_idx %d rank %d sendrecv rank %d\n”, start_idx, rank, rank-1);
    to = arr[0];
    MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (from > to) {
        arr[0] = from;
        flag = 0;
    }
}
else if ((end_idx-1)%2==0 && end_idx<N){
    // printf(”odd end_idx %d rank %d sendrecv rank %d\n”, end_idx, rank, rank+1);
    to = arr[end_idx-start_idx-1];
    MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (from < to) {
        arr[end_idx-start_idx-1] = from;
        flag = 0;
    }
}
```

The program is compiled using CMake build system. One can have a look in `CMakeLists.txt` and `src/CMakeLists.txt` to check compilation requirements. If one wants to build the program, he can run the following commands to configure and start compilation under `hw01` directory. The compiled programs are placed in `build/bin` directory.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release # write configure files in ./build
cmake --build build                       # build the program in ./build
```

After the building process is finished, for example, one can run the program using the following commands.

```
./build/bin/main.seq -n 20 --save 0 --print 1           # sequential program
mpirun -np 10 ./build/bin/main -n 20 --save 0 --print 1 # parallel program
```

`-n 10` means set array size to 10, `--save 0` means do not save any runtime data, and `--print 1` means output the randomly generated array at first and output the sorted array at the end.

2.2 Performance evaluation

In order to evaluate the parallel code, the program was executed under different configurations. With 20 different CPU core numbers (from 4 to 80 with increment 4, $n = 4, 8, \dots, 80$) and 20 different array sizes (from 50000 to 1000000 with increment 50000, $N = 50000, 100000, \dots, 1000000$), overall, 400 cases were sampled. Recorded runtime and CPU time were analyzed through the Numpy package in Python. Figures were plotted through the Matplotlib and the Seaborn packages in Python. Analysis code were written in `analysis/main.ipynb`. It is highly recommended to set `--print 0` when the array size is large.

3 Result and Discussion

3.1 Running time

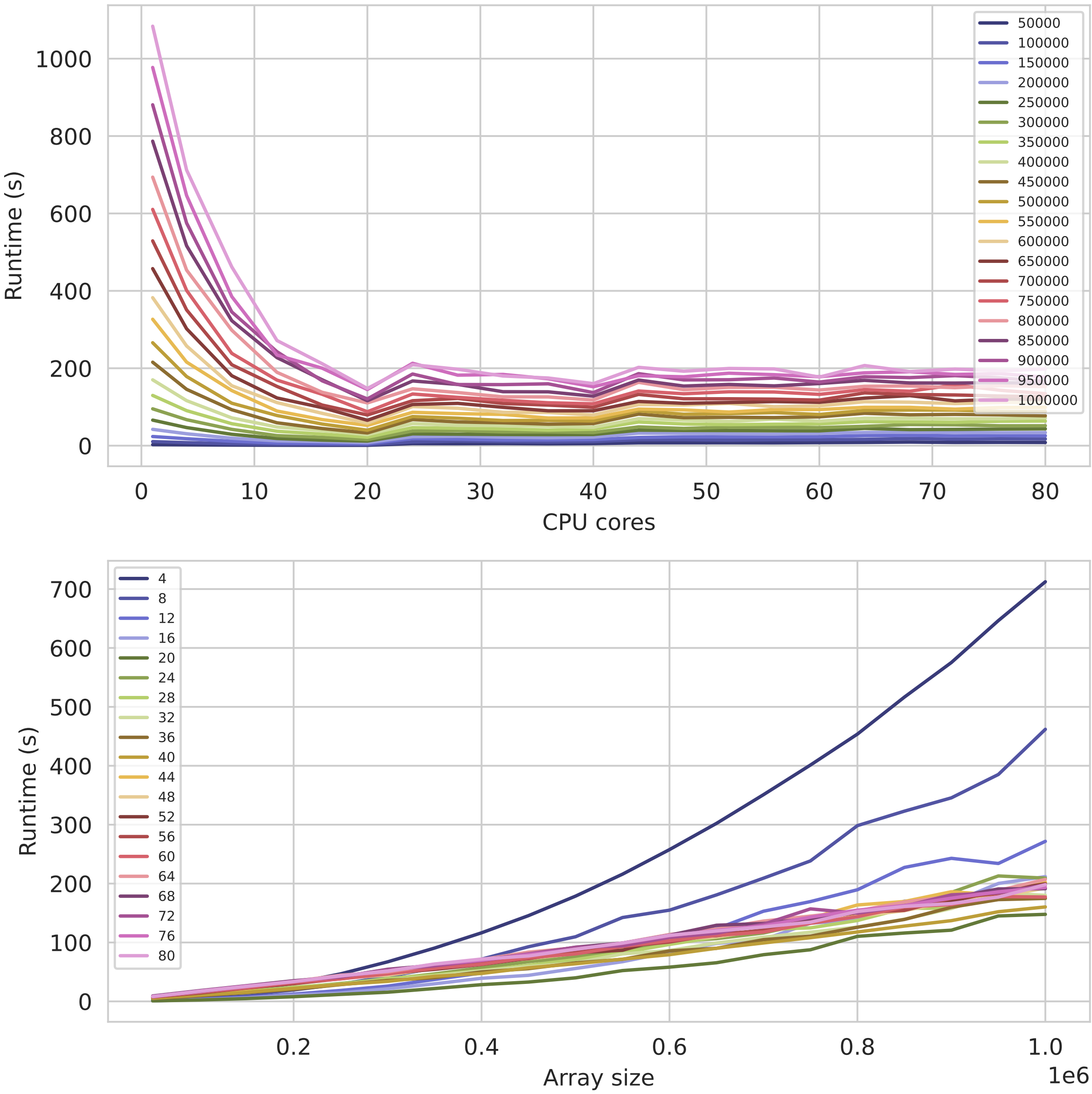

Figure 1: Running time versus the number of CPU cores

The graph of running time versus CPU cores and versus array size were plotted in Figure 1. From this figure, we can see that when the array size is small, the parallelism will not make the program faster but will slow down the execution time. However, when the array size is large enough (e.g., 150000), we can observe a strong speed-up effect if we use more than one core. From Figure 1, we can observe an obvious $O(n^2)$ complexity of the algorithm. It also should be noted that in the case of a large array, with the increase in CPU cores, the running time keeps dropping until the number of cores exceeds 20.

3.2 Performance analysis

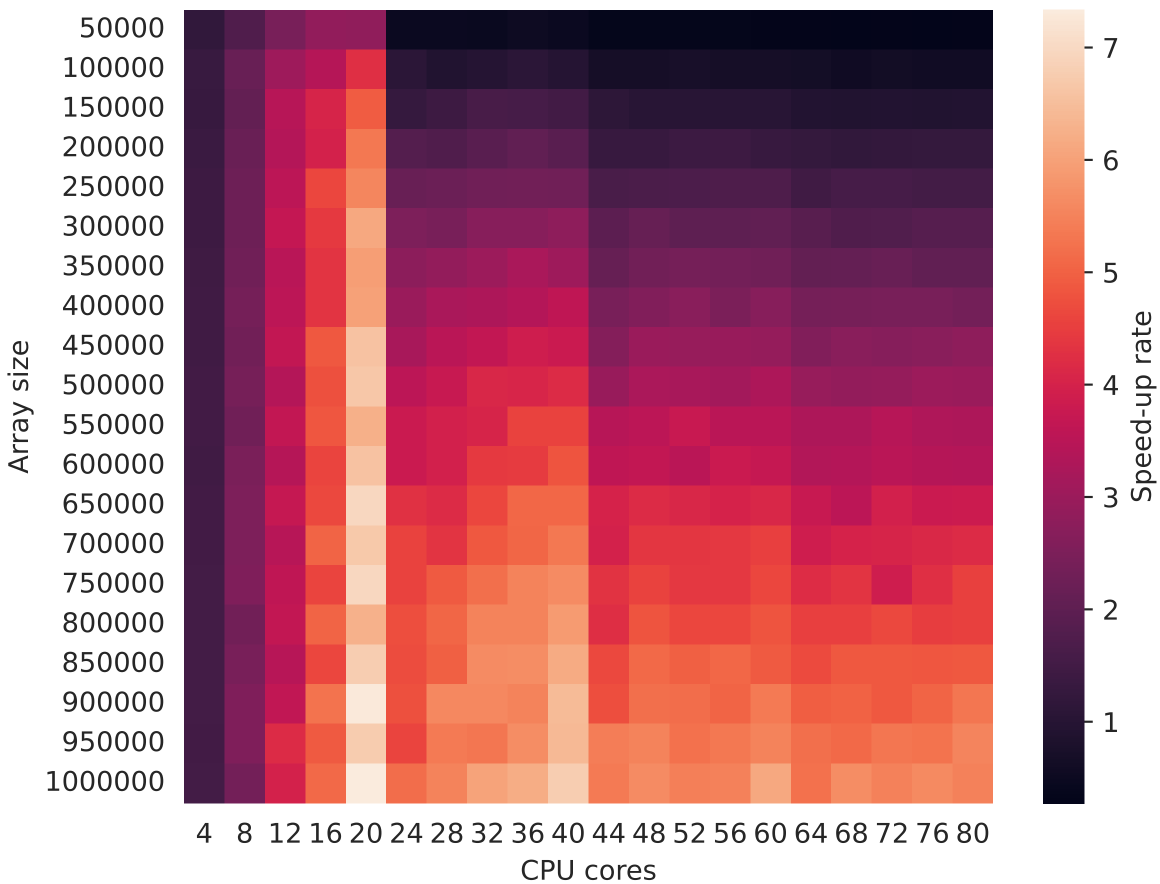

Figure 2: Heatmap of speed-up rate

The speed-up rate is defined as the following ratio, where N denotes the array size and n is the CPU core numbers in a parallel program.

$$r(N,n) = \frac{\text{runtime of sequential sorting a size } N \text{ array}}{\text{runtime of parallel sorting a size } N \text{ array using } n \text{ processes}} \tag{1}$$

Therefore, we can calculate a speed-up rate for each sample (N,n). The heatmap of the speed-up rate is plotted in Figure 2. A 3D version of the heatmap is also provided in Figure A.1 for better visualization. The speed-up rate versus array size and CPU core number is plotted in Figure 3.

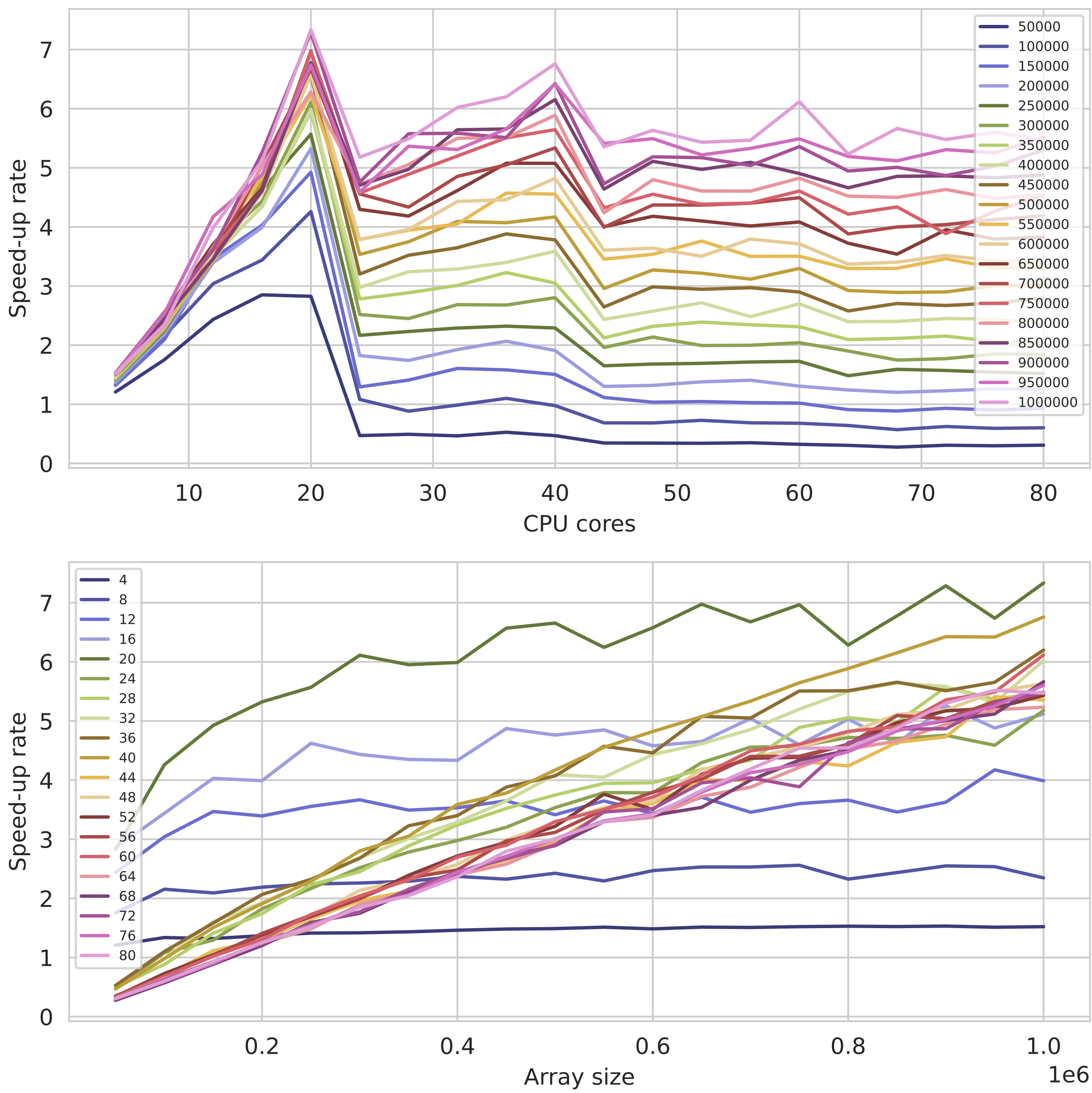

Figure 3: Speed up rate versus the number of CPU cores and array size

Noticeably, the maximum speed-up rate was achieved in $N = 1000000$ and $n = 20$. Interestingly, in Figure 3, for a fixed array size, the speed-up rate will have a significant drop if the CPU core number just exceeds 20, 40, and 60. The reason for this could be highly caused by the configuration of the cluster. In the CSC4005 computing cluster, it is easy to discover that each node contains 20 CPU cores. If the CPU cores exceed 20, the program would involve more than one node. Obviously, the primary influence of multi-node computing would be communication latency. Since MPI is not designed for the shared memory system, data must be transferred during the calculation. Therefore, we can deduct the conclusion that the optimal MPI configuration under this cluster would be 20 cores. Moreover, speculation could be proposed that the optimal MPI configuration for any cluster is one node with all its available CPU cores.

Also, according to the plot of speed-up rate versus array size in Figure 3, we can see that if we only use one computing node, the speed-up rate is not significantly affected by the array size. However, as long as the calculation involves two or more nodes, the speed-up rate dropped notably when the array size is relatively small. Also, the speed-up rate is no longer sensitive to the increment of total CPU cores used in sorting. That is because when the array size becomes large, the latency caused by the odd-even transition would be less dominant.

4 Conclusion

In conclusion, a parallel odd-even sorting algorithm was implemented and its performance was evaluated. When the array size is small, the sequential program is more efficient. While the parallel program is more efficient when the array size is large. However, it is better not to use more than one node since the latency caused by inter-node communication could significantly increase the runtime.

A Supplementary figures

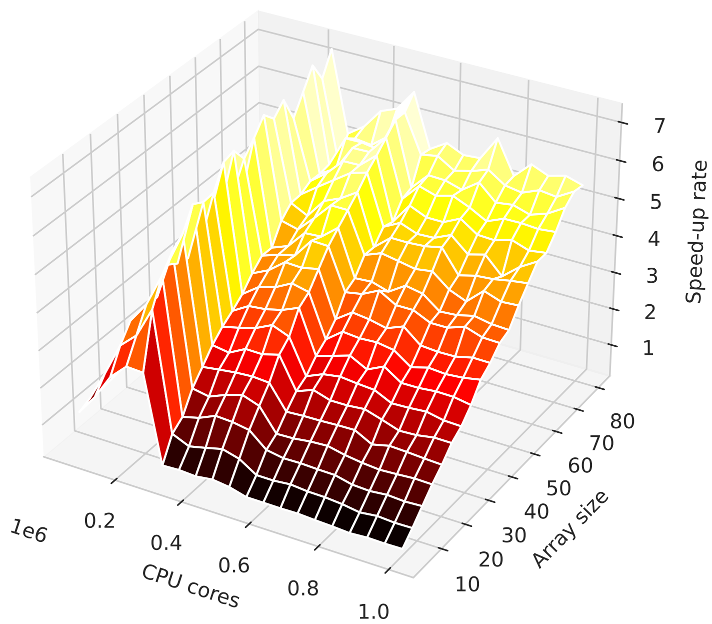

Figure A.1: 3D heatmap of speed-up rate

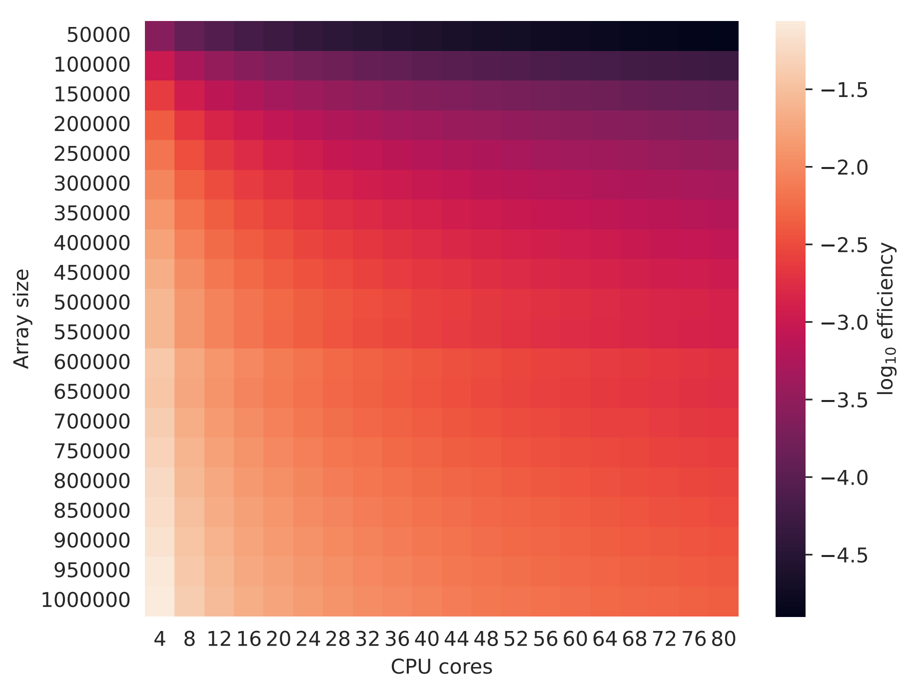

Figure A.2: Heat map of $\log_{10}$(seq CPU time/par CPU time)

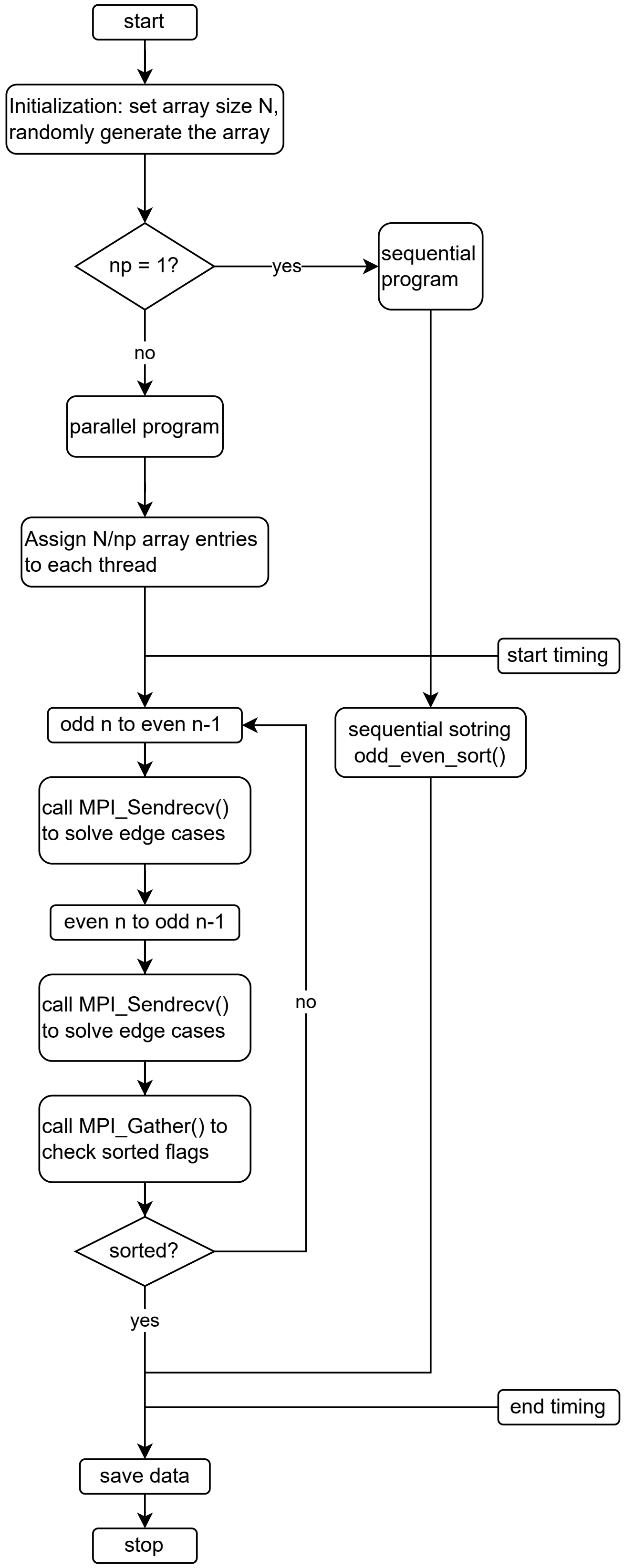

Figure A.3: Program flowchart

B Source code

main.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <mpi.h>
#include <chrono>
#include <thread>
#include "utils.h"


int main(int argc, char* argv[]) {
    // mpi initialize
    MPI_Init(NULL, NULL);

    // fetch size and rank
    int size, rank;
    int save = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // initializiation, N = 10 default
    int N = 10;
    // parse argument
    char buff[100];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, "-n")==0){
            std::string num(argv[i+1]);
            N = std::stoi(num);
        }
        if (strcmp(buff, "--save")==0){
            std::string num(argv[i+1]);
            save = std::stoi(num);
        }
    }

    // determine start and end index
    int *arr;
    int *arr_;
    int jobsize = N / size;
    int start_idx = jobsize * rank;
    int end_idx   = start_idx + jobsize;
    int *rbuf = (int *)malloc(sizeof(int) * size);
    double *time_arr = (double *)malloc(sizeof(double) * size);
    double t1, t2, t, t_sum;
    int from, to;
    int flag;
    if (rank == size-1) end_idx = N;

    // master proc array allocation
    if (rank==0){
        printf("Name: Haoran Sun\n");
        printf("ID:   119010271\n");
        printf("HW:   Parallel Odd-Even Sort\n");
        printf("Set N to %d.\n", N);
```

```
        arr_ = (int *) malloc(sizeof(int) * N);
        fill_rand_arr(arr_, N);
        // print_arr(arr_, N);
    }
    arr = (int *) malloc(sizeof(int) * (end_idx-start_idx));

    // MAIN PROGRAM
    // start time
    t1 = MPI_Wtime();

    // CASE 1: sequential
    if (size==1) {
        odd_even_sort(arr_, N, 0);
    }

    // CASE 2: parallel
    else {
        // STEP 1: data transfer master --> slave
        if (rank==0){
            for (int i = 1; i < size; i++){
                int start = i*jobsize;
                int end   = start + jobsize;
                MPI_Request request;
                if (i==size-1) end += N%size;
                MPI_Send(arr_+start, end-start, MPI_INT, i, 0, MPI_COMM_WORLD);
            }
            for (int i = 0; i < jobsize; i++) arr[i] = arr_[i];
        }
        else
            MPI_Recv(arr, end_idx-start_idx, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Barrier(MPI_COMM_WORLD);
        // print_arr(arr, end_idx-start_idx);

        // STEP 2: main program
        while (true){
            flag = 1;

            int a, b;
            int from, to;
            MPI_Request request = MPI_REQUEST_NULL;
            MPI_Status status;
            // STEP 2.1: odd loop
            // inner odd loop
            for (int i = 1; i < end_idx-start_idx; i++){
                if ((start_idx+i)%2==1){
                    a = arr[i-1];
                    b = arr[i];
                    if (b < a){
                        arr[i]   = a;
                        arr[i-1] = b;
                        flag = 0;
                    }
                }
            }
            // possible interexchange
            if (start_idx>0 && start_idx%2==1){
                // printf("odd start_idx %d rank %d sendrecv rank %d\n", start_idx, rank, rank-1);
```

```
                to = arr[0];
                MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from > to) {
                    arr[0] = from;
                    flag = 0;
                }
            }
            else if ((end_idx-1)%2==0 && end_idx<N){
                // printf(”odd end_idx %d rank %d sendrecv rank %d\n”, end_idx, rank, rank+1);
                to = arr[end_idx-start_idx-1];
                MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from < to) {
                    arr[end_idx-start_idx-1] = from;
                    flag = 0;
                }
            }
            // MPI_Barrier(MPI_COMM_WORLD);

            // STEP 2.2: even loop
            // inner even loop
            for (int i = 1; i < end_idx-start_idx; i++){
                if ((start_idx+i)%2==0){
                    a = arr[i-1];
                    b = arr[i];
                    if (b < a){
                        arr[i]   = a;
                        arr[i-1] = b;
                        flag = 0;
                    }
                }
            }
            // possible interexchange
            if (rank%2==1 && start_idx>0 && start_idx%2==0){
                // printf(”even start_idx %d rank %d sendrecv rank %d\n”, start_idx, rank, rank-1);
                to = arr[0];
                MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from > to) {
                    arr[0] = from;
                    flag = 0;
                }
            }
            else if (rank%2==0 && (end_idx-1)%2==1 && end_idx<N){
                // printf(”even end_idx %d rank %d sendrecv rank %d\n”, end_idx, rank, rank+1);
                to = arr[end_idx-start_idx-1];
                MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from < to) {
                    arr[end_idx-start_idx-1] = from;
                    flag = 0;
                }
            }
            // MPI_Barrier(MPI_COMM_WORLD);
            if (rank%2==0 && start_idx>0 && start_idx%2==0){
```

```
                // printf(”even start_idx %d rank %d sendrecv rank %d\n”, start_idx, rank, rank-1);
                to = arr[0];
                MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from > to) {
                    arr[0] = from;
                    flag = 0;
                }
            }
            else if (rank%2==1 && (end_idx-1)%2==1 && end_idx<N){
                // printf(”even end_idx %d rank %d sendrecv rank %d\n”, end_idx, rank, rank+1);
                to = arr[end_idx-start_idx-1];
                MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (from < to) {
                    arr[end_idx-start_idx-1] = from;
                    flag = 0;
                }
            }
            MPI_Barrier(MPI_COMM_WORLD);

            // STEP 2.3: sending stop flag to master, master decide whether
            // to continue
            MPI_Gather(&flag, 1, MPI_INT, rbuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
            if (rank==0) {
                // print_arr(rbuf, size);
                for (int i = 0; i < size; i++){
                    if (rbuf[i] != 1) {
                        flag = 0;
                    }
                }
            }
            MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
            // printf(”2. rank %d flag = %d\n”, rank, flag);
            if (flag == 1) {
                // odd_even_sort(arr, end_idx-start_idx, 0);
                break;
            }
        }


        // STEP 3: gather sorted array
        MPI_Gather(arr, jobsize, MPI_INT, arr_, jobsize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        // tail case
        if (N%size != 0) {
            if (rank == size-1) MPI_Send(arr+jobsize, N%size, MPI_INT, 0, 2, MPI_COMM_WORLD);
            if (rank == 0) MPI_Recv(arr_+N/size*size, N%size, MPI_INT, size-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }

    // end time
    t2 = MPI_Wtime();
```

```
    t = t2 - t1;
    MPI_Gather(&t, 1, MPI_DOUBLE, time_arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (rank==0) {
        t_sum = arr_sum(time_arr, size);
        printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t_sum,
            size);
        // check_sorted(arr_, N);
    }


    // // master print result
    // if (rank==0) print_arr(arr_, jobsize);

    // free array
    free(arr);
    if (rank==0) free(arr_);

    // print info to file
    if (rank==0 && save==1) {
        FILE* outfile;
        if (size==1) outfile = fopen("data_seq.txt", "a");
        else  outfile = fopen("data.txt", "a");
        fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t_sum);
        fclose(outfile);
    }
    // this line is added to make sure that the data is correctly saved
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    return 0;
}
```

main.seq.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <chrono>
#include <thread>
#include "utils.h"


int main(int argc, char* argv[]) {
    // fetch size and rank
    int size = 1, rank = 0;
    int save = 0;
    // initializiation, N = 10 default
    int N = 10;
    // parse argument
    char buff[100];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, "-n")==0){
            std::string num(argv[i+1]);
            N = std::stoi(num);
        }
        if (strcmp(buff, "--save")==0){
            std::string num(argv[i+1]);
```

```
            save = std::stoi(num);
        }
    }

    // determine start and end index
    int *arr = (int *)malloc(sizeof(int) * N);
    int *rbuf = (int *)malloc(sizeof(int) * size);

    // master proc array allocation
    printf("Name: Haoran Sun\n");
    printf("ID:   119010271\n");
    printf("HW:   Parallel Odd-Even Sort\n");
    printf("Set N to %d.\n", N);
    fill_rand_arr(arr, N);

    // MAIN PROGRAM
    // start time
    auto t1 = std::chrono::system_clock::now();

    // sequential sort
    odd_even_sort(arr, N, 0);

    // end time
    auto t2 = std::chrono::system_clock::now();
    auto dur = t2 - t1;
    auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
    double t = dur_.count();
    printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t, size);

    // free array
    free(arr);

    // print data info to file
    FILE* outfile;
    if (size==1) outfile = fopen("data_seq.txt", "a");
    else  outfile = fopen("data.txt", "a");
    fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t);
    fclose(outfile);

    // this line is added to make sure that the data is correctly saved
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    return 0;
}
```

utils.h

```
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <string.h>

// fill random array
void fill_rand_arr(int* arr, int N){
    std::srand(time(0));
    for (int i = 0; i < N; i++){
        arr[i] = std::rand() % 10000;
    }
}
```

```c

// binary sort
void odd_even_sort(int* arr, int N, int f){
    if (f==1) return;
    int a, b;
    int flag = 1;
    // odd loop
    for (int i = 1; i < N; i += 2){
        a = arr[i-1];
        b = arr[i];
        if (b < a){
            arr[i]   = a;
            arr[i-1] = b;
            flag = 0;
        }
    }
    // even loop
    for (int i = 2; i < N; i += 2){
        a = arr[i-1];
        b = arr[i];
        if (b < a){
            arr[i]   = a;
            arr[i-1] = b;
            flag = 0;
        }
    }
    return odd_even_sort(arr, N, flag);
}
// binary sort single iteration

// min-max
void min_max(int *arr, int N) {
    int min_idx = 0;
    int max_idx = 0;
    int min = arr[0];
    int max = arr[0];

    for (int i = 1; i < N; i++) {
        if (arr[i] > max) {
            max = arr[i];
            max_idx = i;
        }
        if (arr[i] < min) {
            min = arr[i];
            min_idx = i;
        }
    }

    arr[min_idx] = arr[0];
    arr[max_idx] = arr[N-1];
    arr[0] = min;
    arr[N-1] = max;
}

void print_arr(int* arr, int N){
    for (int i = 0; i < N; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
}

void check_sorted(int* arr, int N){
    for (int i = 0; i < N-1; i++){
        if (arr[i] > arr[i+1]) {
            printf("Error at idx %d.\n", i);
            return;
        }
    }
    printf("Array sorted.\n");
    return;
}


template <class T>
T arr_sum(T *arr, int N){
    T sum = 0;
    for (int i = 0; i < N; i++) sum += arr[i];
    return sum;
}
```