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 implementation. 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 random array was generated in any integer in the range of $[0, 10000)$. 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;
    }
}
```

2.2 Usage

The program is compiled using CMake build system. One can have a look at `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.

For convenience, one can directly execute `./scripts/demo.sh` for both sequential and parallel odd-even sort on a $n = 20$ array.

```
cd hw01
./script/demo.sh
```

Sample output

```
-- The C compiler identification is GNU 8.5.0
-- The CXX compiler identification is GNU 8.5.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
```

```
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MPI_C: /home/shiroha/mpich/mpich-4.0.2/lib/libmpi.so (found version "4.0")
-- Found MPI_CXX: /home/shiroha/mpich/mpich-4.0.2/lib/libmpicxx.so (found version "4.0")
-- Found MPI: TRUE (found version "4.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/e/personal files/CUHK(SZ)/Lectures/SDS/CSC4005/hw01
    /build
[ 25%] Building CXX object src/CMakeFiles/main.dir/main.cpp.o
[ 50%] Linking CXX executable ../bin/main
[ 50%] Built target main
[ 75%] Building CXX object src/CMakeFiles/main.seq.dir/main.seq.cpp.o
[100%] Linking CXX executable ../bin/main.seq
[100%] Built target main.seq
=== SEQ ===
Name: Haoran Sun
ID:   119010271
HW:   Parallel Odd-Even Sort
Set N to 20.
Array:
 9305  6936  6308  3097  5589  7572  3029  4862  1541  6099
 3953  3149  4036  2275  1619  4847  1683  2446  4855  7615
Sorted array:
 1541  1619  1683  2275  2446  3029  3097  3149  3953  4036
 4847  4855  4862  5589  6099  6308  6936  7572  7615  9305
Execution time: 0.00s, cpu time: 0.00s, #cpu  1
===========

=== PAR ===
Name: Haoran Sun
ID:   119010271
HW:   Parallel Odd-Even Sort
Set N to 20.
Array:
 9305  6936  6308  3097  5589  7572  3029  4862  1541  6099
 3953  3149  4036  2275  1619  4847  1683  2446  4855  7615
Sorted array:
 1541  1619  1683  2275  2446  3029  3097  3149  3953  4036
 4847  4855  4862  5589  6099  6308  6936  7572  7615  9305
Execution time: 0.00s, cpu time: 0.00s, #cpu  4
===========
```

2.3 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, $p = 4, 8, \ldots, 80$) and 20 different array sizes (from 50000 to 1000000 with increment 50000, $n = 5 \times 10^4, 10^5, \ldots, 10^6$), total of 400 cases was 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.

```
./build/bin/main.seq -n 20 --save 0 --print 0 # omit array output
./build/bin/main.seq -n 20 --save 0 --print 0 # omit array output
```

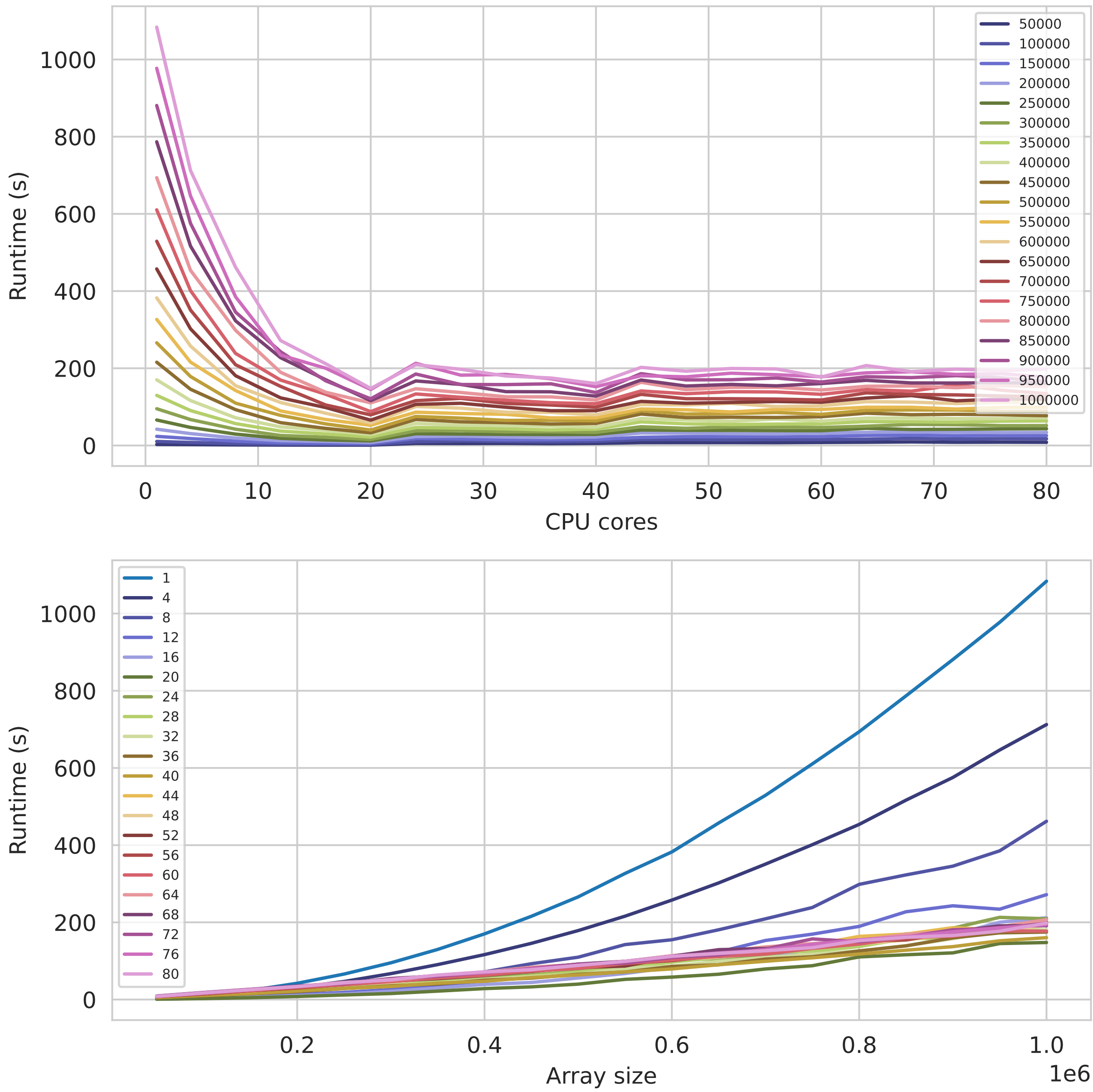

Figure 1: Running time versus the number of CPU cores

3 Result and Discussion

3.1 Running time

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,

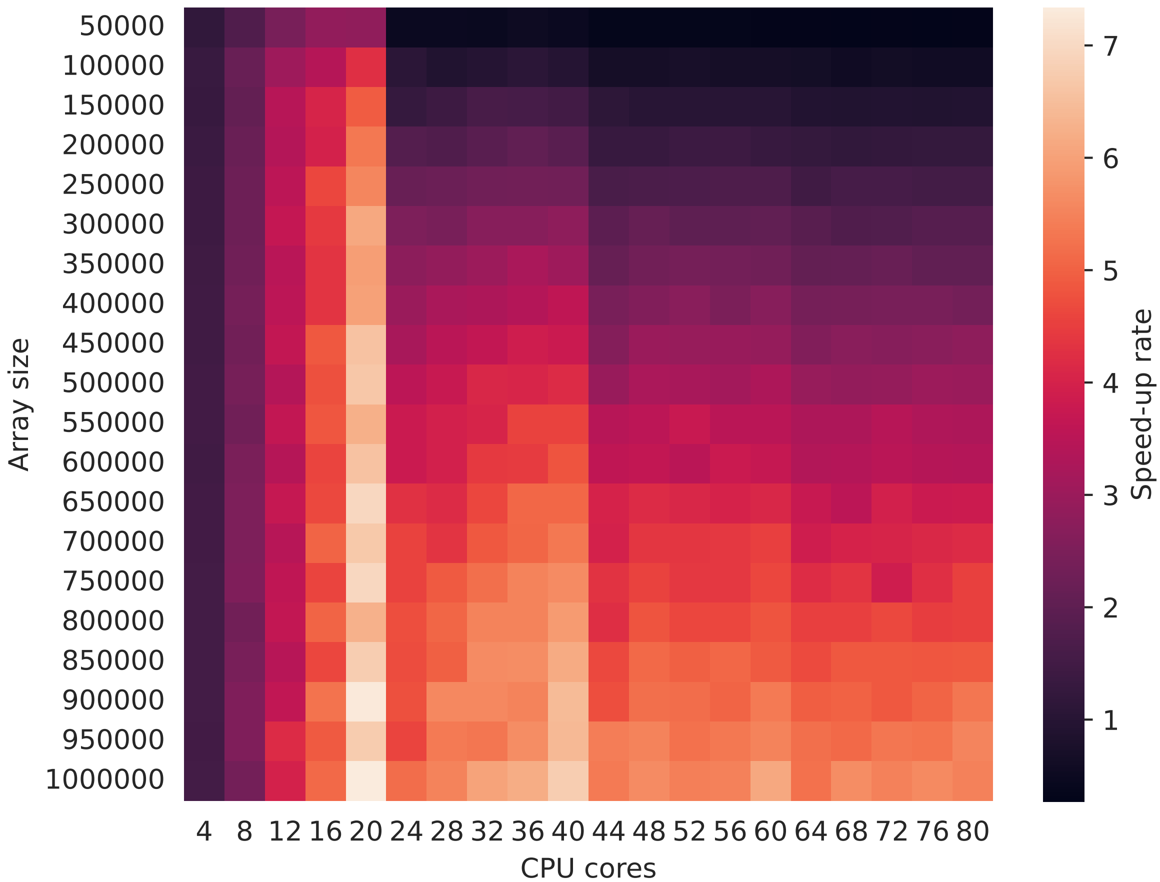

Figure 2: Heatmap of speed-up rate

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

The speed-up rate S is defined as the following ratio, where n denotes the array size and p is the CPU core numbers in a parallel program. Let $\sigma(n)$ denote the runtime of running a sequential sorting program on a size n array, and let $\sigma(n, p)$ denote the runtime of running a parallel sorting program on a size n array using p processors, then the speed-up rate could be defined as

$$S(n, p) = \frac{\sigma(n)}{\sigma(n, p)} \tag{1}$$

Therefore, we can calculate a speed-up rate for each sample (n, p). 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.

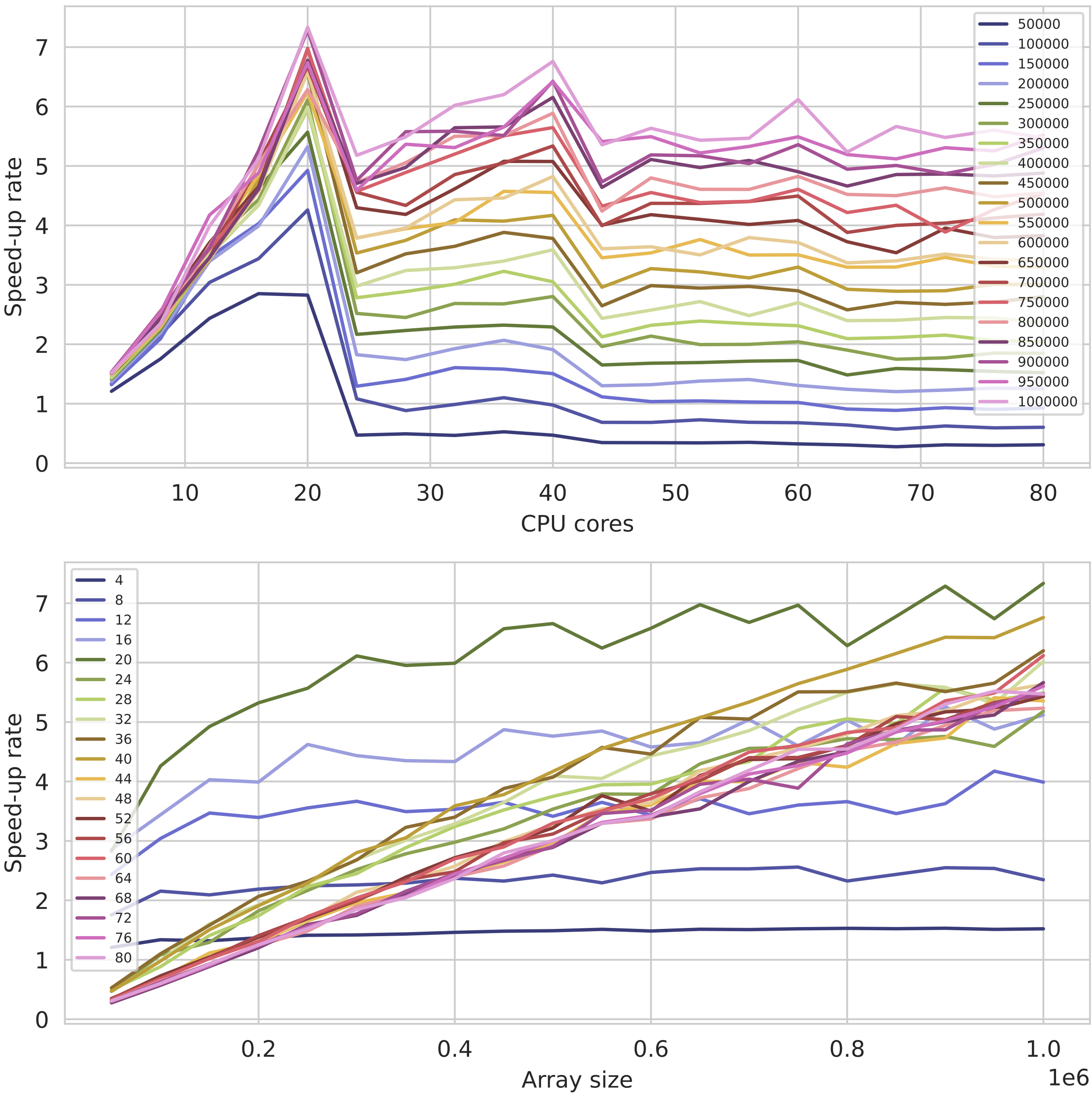

Figure 3: Speed up rate versus the number of CPU cores and array size

Noticeably, the maximum speed-up rate was achieved in $n = 1000000$ and $p = 20$. 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. Then this phenomenon 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

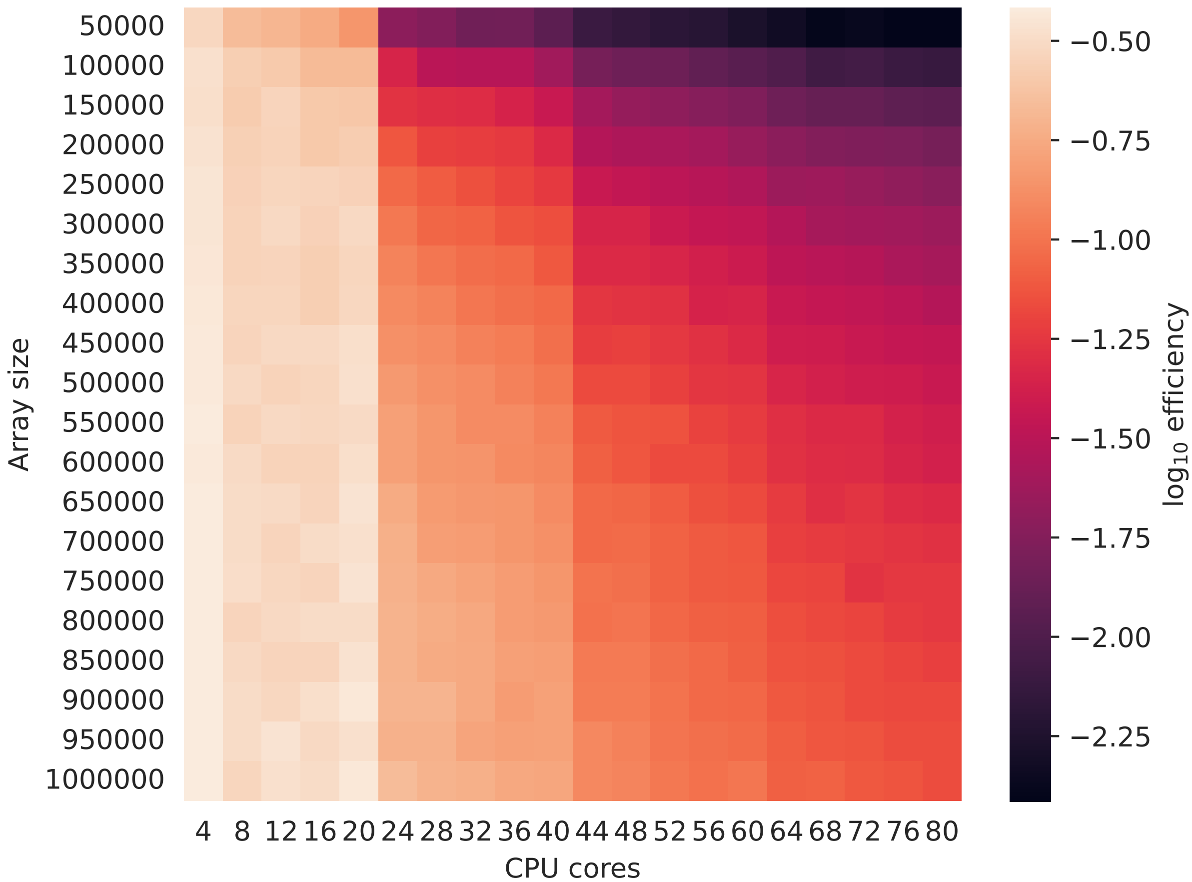

Figure 4: Heat map of $\log_{10}$(seq CPU time/par CPU time)

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.

The efficiency of execution can also illustrate this phenomenon. From the heatmap printed in Figure 4, there are three clear color gaps between CPU cores 20 and 24, 40 and 44, and 60 and 64, meaning the efficiency would drop significantly when inter-node communication occurs. To quantitatively illustrate the negative effect of communication, we can define some terms. Let $\kappa(n, p)$ denote the communication time of running a parallel sorting program on size n array using p processors. Let $\psi(n, p)$ denote the parallel execution time of running a parallel sorting program on size n array using p processors. Then we should have the following relation

$$S(n, p) \leq \frac{\sigma(n)}{\psi(n, p) + \kappa(n, p)} \tag{2}$$

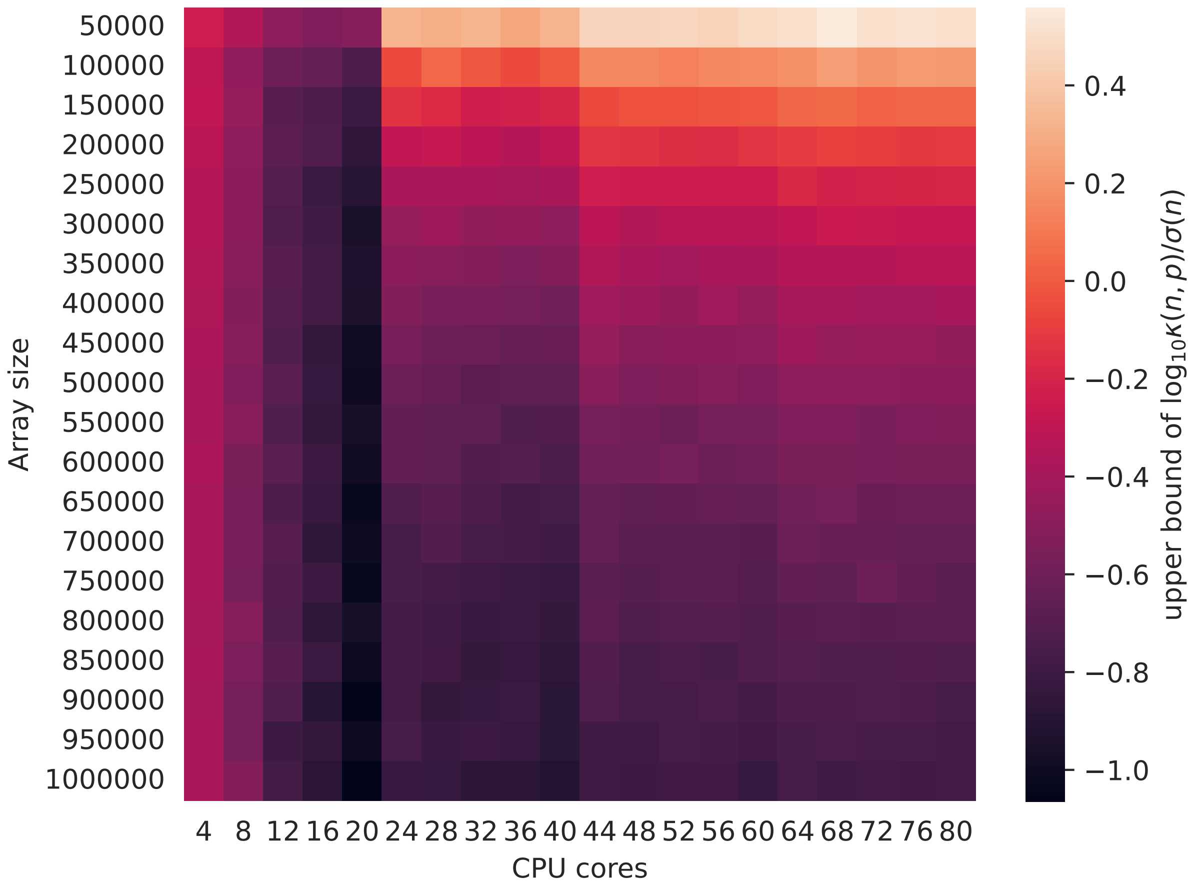

Figure 5: Heat map of upper bound of $\log_{10}\kappa(n,p)/\sigma(n)$.

Naively, we may assume that $\sigma(n) = p\psi(n,p)$, then we have

$$S(n,p) \leq \frac{p\sigma(n)}{\sigma(n) + p\kappa(n,p)} = \frac{p}{1 + p\frac{\kappa(n,p)}{\sigma(n)}} \tag{3}$$

Hence, we can get an upper bound of $\kappa(n,p)/\sigma(n)$ by

$$\frac{\kappa(n,p)}{\sigma(n)} \leq \frac{1}{S(n,p)} - \frac{1}{p} \tag{4}$$

According to the plot in Figure 5, we can see that the ratio of the communication time would increase with p and decrease with n. From the data of $n = 4, \ldots, 20$, we can see the communication latency in a single node is proportional to the array size n. This also illustrates that inter-node communication could be extremely expensive. Naively, we may guess an expression for the speed-up rate

$$S(n,p) = \frac{k_1 n^2}{k_1 n^2/p + pnk_2(p)} = \frac{p}{1 + \frac{k_2(p)p^2}{k_1 n}} \tag{5}$$

where $\sigma(n) = k_1 n^2$ and $\kappa(n, p) = pnk_2(p)$

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

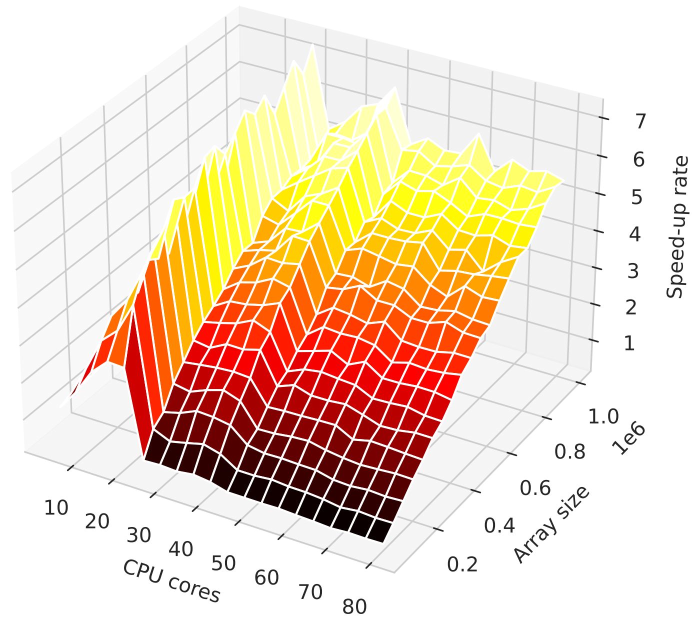

Figure A.1: 3D heatmap of speed-up rate

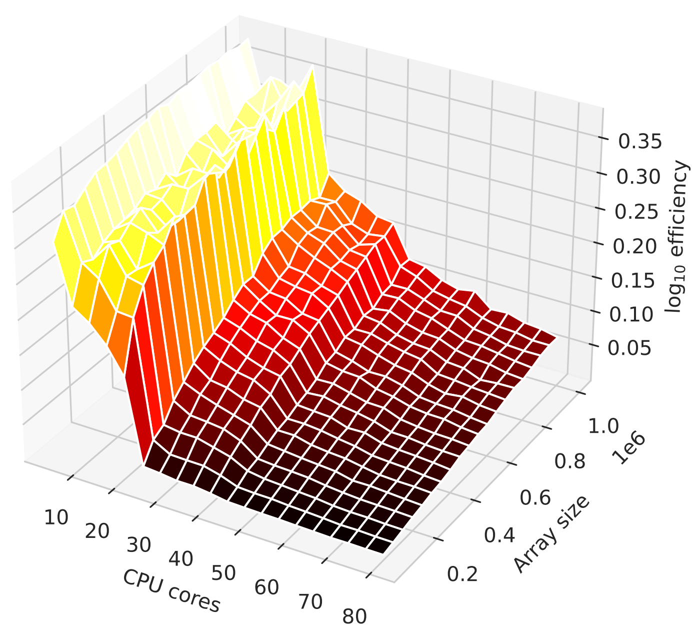

Figure A.2: 3D heatmap of log efficiency

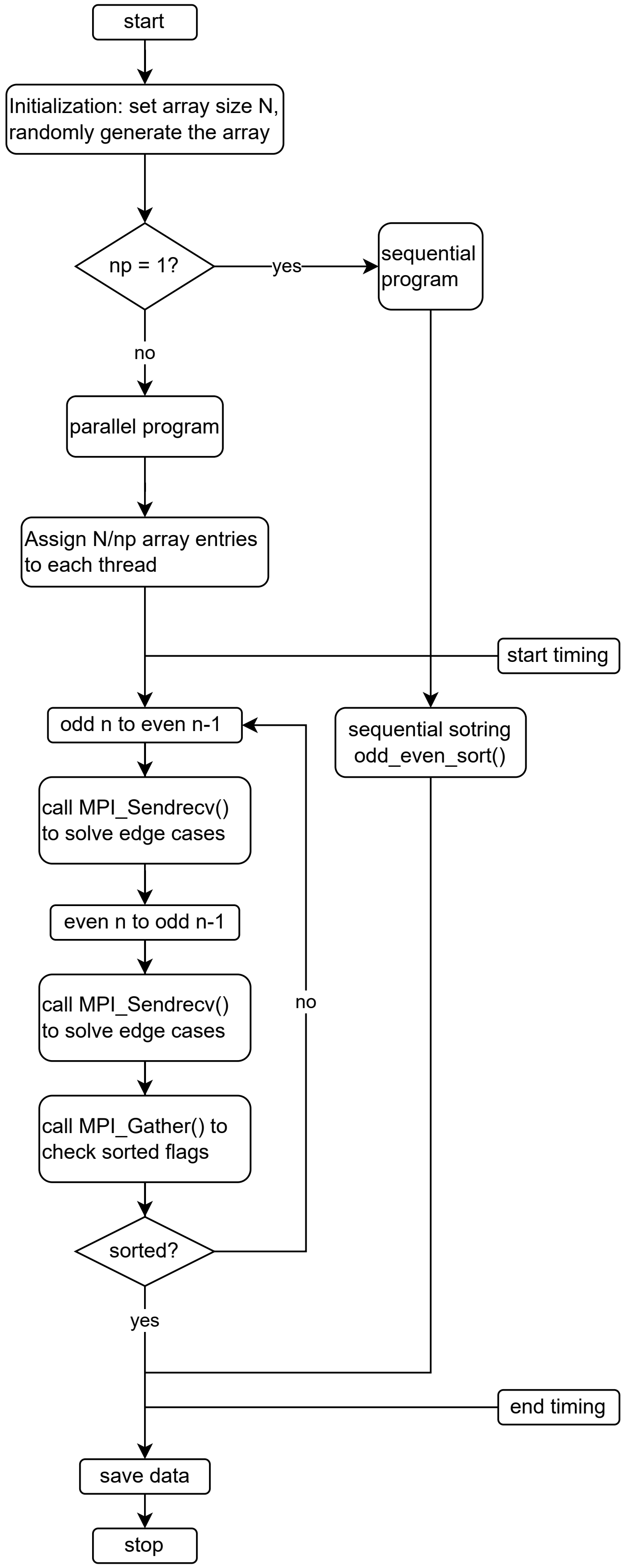

Figure A.3: Program flowchart

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;
    int print = 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);
        }
        if (strcmp(buff, ”--print”)==0){
            std::string num(argv[i+1]);
            print = 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);
```

```
        if (print==1) {
            printf("Array:\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);
        }
    }

    // master print result
    if (rank==0 && print==1) {
        printf(”Sorted array:\n”);
        print_arr(arr_, N);
    }

    // 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);
    }

    // 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;
    int print = 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);
        }
        if (strcmp(buff, ”--print”)==0){
            std::string num(argv[i+1]);
            print = 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);
    if (print==1) {
        printf(”Array:\n”);
        print_arr(arr, N);
    }

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

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

```

    // print array
    if (print==1) {
        printf("Sorted array:\n");
        print_arr(arr, N);
    }

    // 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
    if (save==1) {
        FILE* outfile;
        outfile = fopen("data_seq.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;
    }
}

// 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("%5d ", arr[i]);
        if ((i+1)%10 == 0)
            printf("\n");
    }
    if (N%10 != 0) 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;
}
```