

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.

```
1 // binary sort
2 void odd_even_sort(int* arr, int N, int f){
3     if (f==1) return;
4     int a, b;
5     int flag = 1;
6     // odd loop
7     for (int i = 1; i < N; i += 2){
8         a = arr[i-1];
9         b = arr[i];
10        if (b < a){
11            arr[i] = a;
12            arr[i-1] = b;
13            flag = 0;
14        }
15    }
16    // even loop
17    for (int i = 2; i < N; i += 2){
18        a = arr[i-1];
19        b = arr[i];
20        if (b < a){
21            arr[i] = a;
22            arr[i-1] = b;
23            flag = 0;
24        }
25    }
26    return odd_even_sort(arr, N, flag);
27 }
```

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().

```

1  if (start_idx>0 && start_idx%2==1){
2      // printf("odd start_idx %d rank %d sendrecv rank %d\n", start_idx, rank,
        rank-1);
3      to = arr[0];
4      MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5      if (from > to) {
6          arr[0] = from;
7          flag = 0;
8      }
9  }
10 else if ((end_idx-1)%2==0 && end_idx<N){
11     // printf("odd end_idx %d rank %d sendrecv rank %d\n", end_idx, rank, rank
        +1);
12     to = arr[end_idx-start_idx-1];
13     MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
14     if (from < to) {
15         arr[end_idx-start_idx-1] = from;
16         flag = 0;
17     }
18 }

```

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

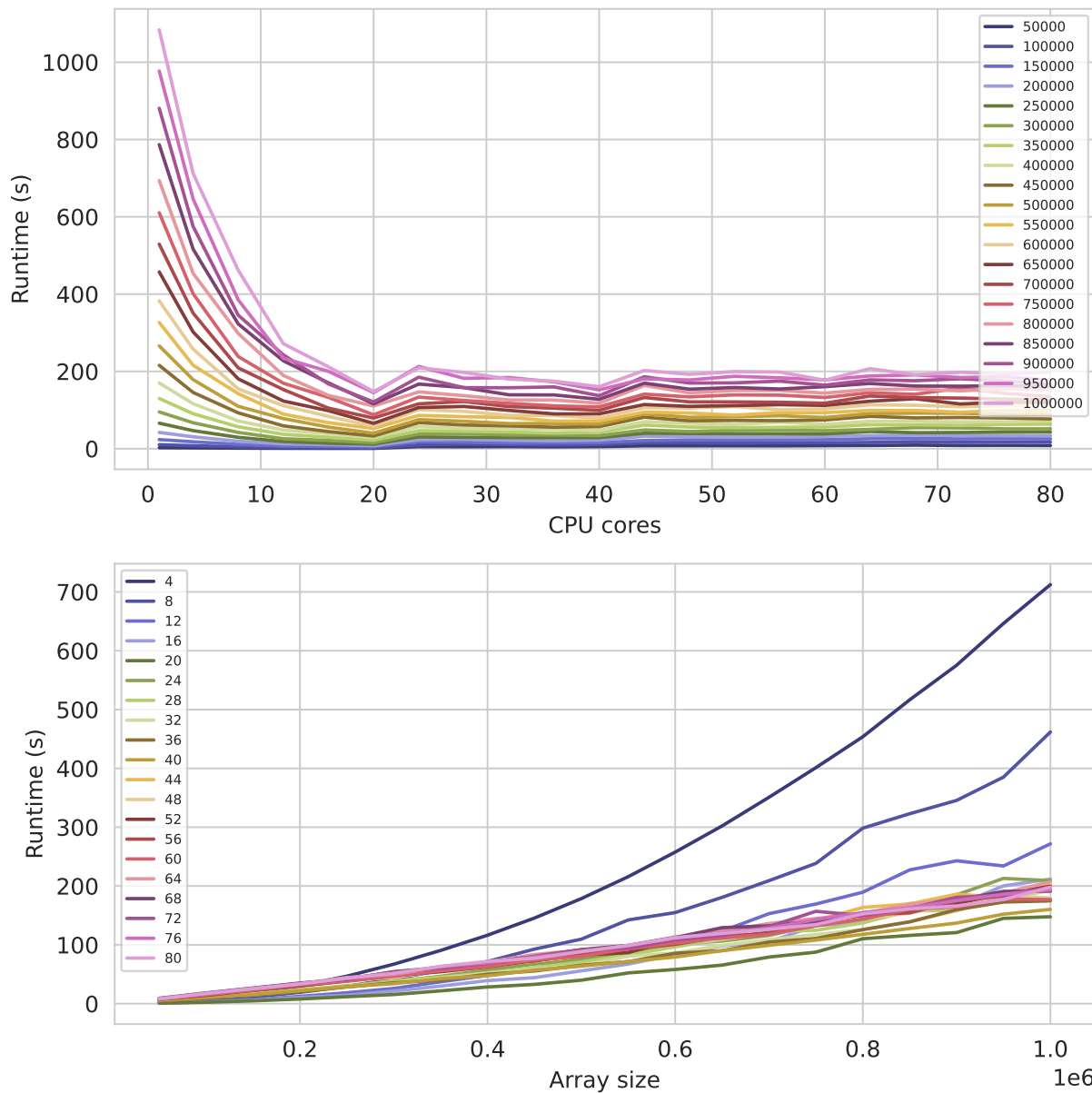


Figure 1: Running time versus the number of CPU cores

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

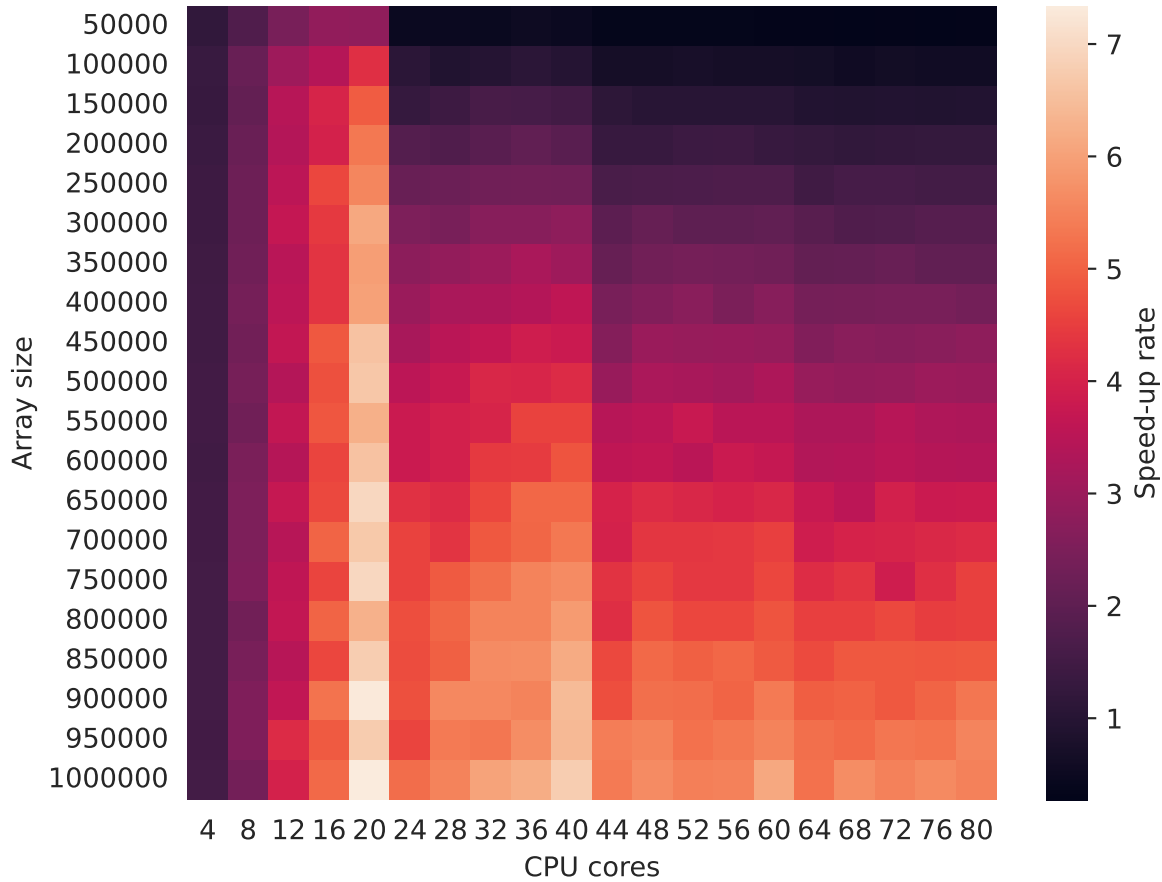


Figure 2: Heatmap of speed-up rate

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.

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

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.

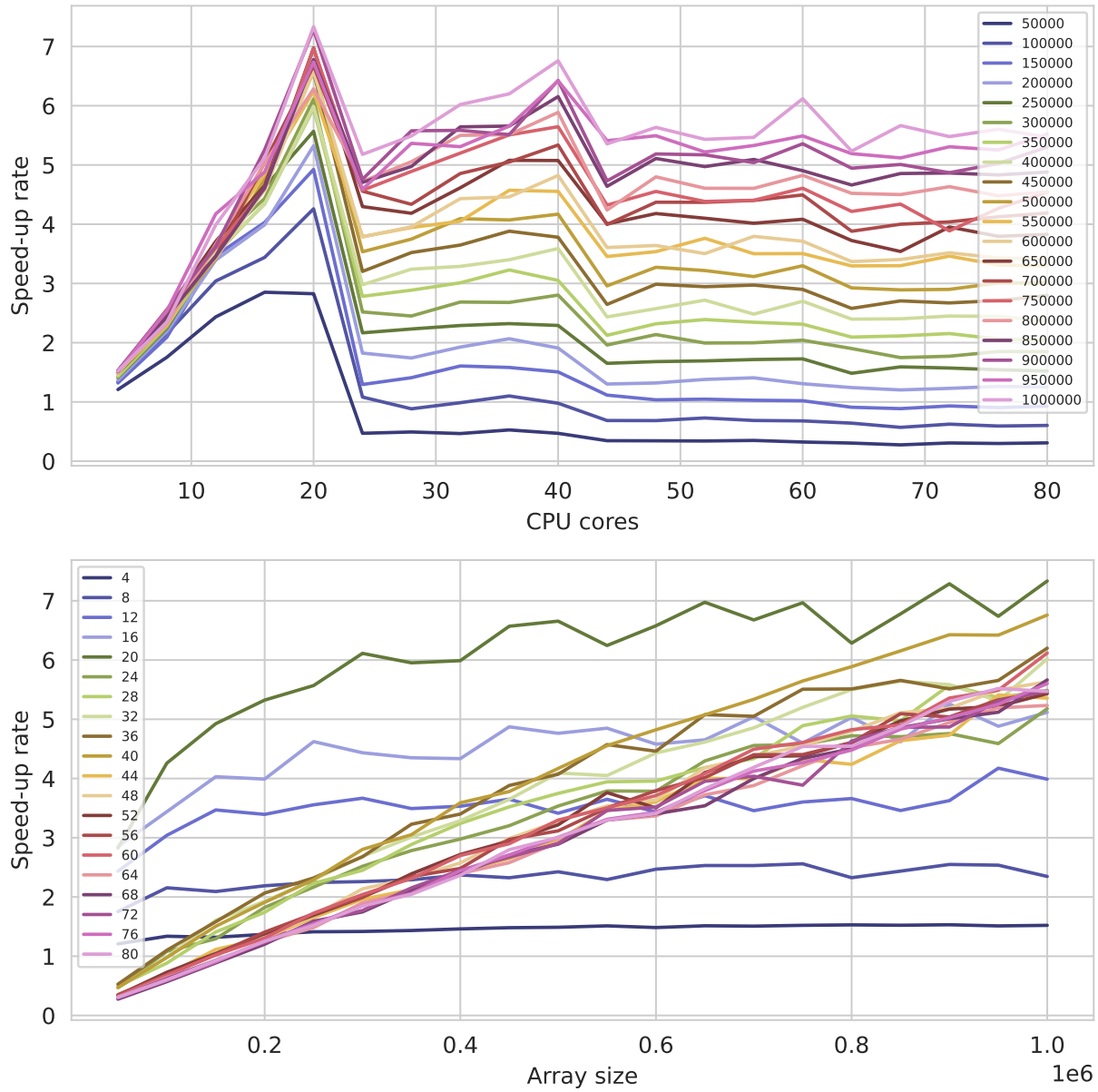


Figure 3: Speed up rate versus the number of CPU cores and array size

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 deduce 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

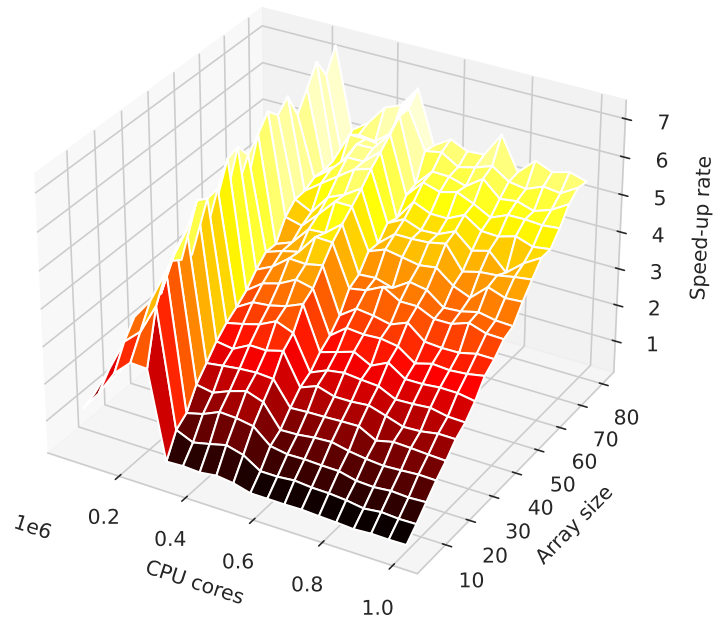


Figure A.1: 3D heatmap of speed-up rate

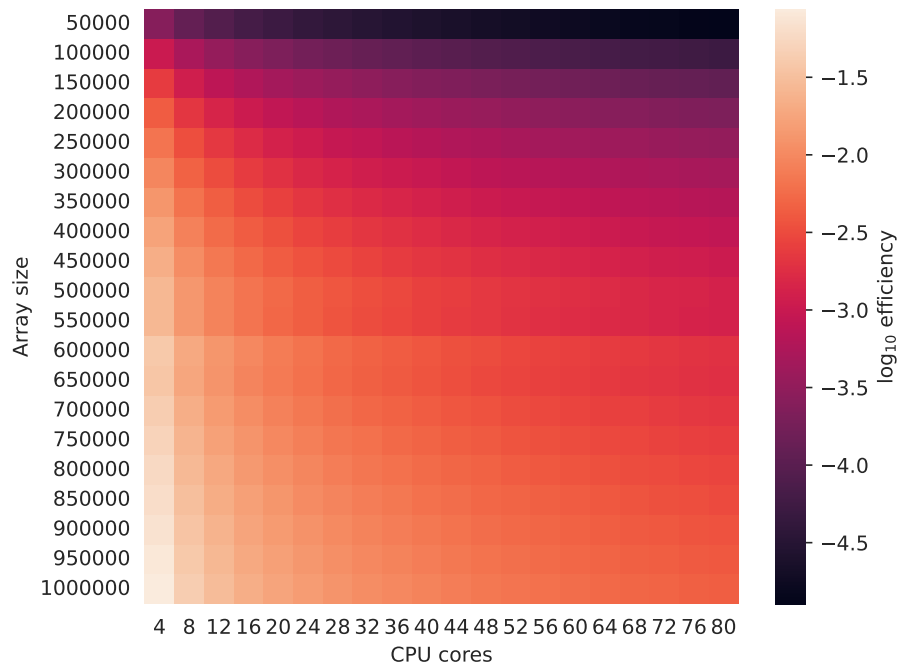


Figure A.2: Heat map of $\log_{10}(\text{seq CPU time}/\text{par CPU time})$

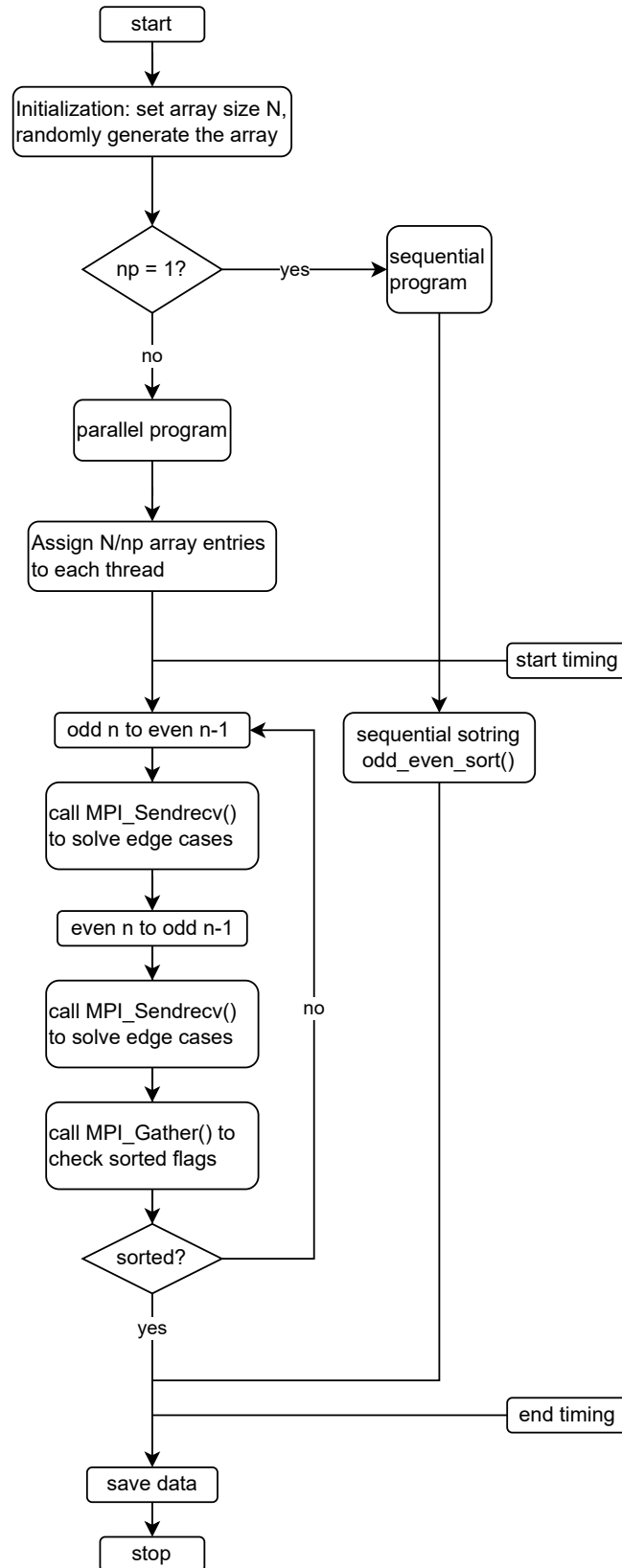


Figure A.3: Program flowchart

B Source code

main.cpp

```

1  #include <stdio.h>
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <string.h>
6  #include <mpi.h>
7  #include <chrono>
8  #include <thread>
9  #include "utils.h"
10
11
12  int main(int argc, char* argv[]) {
13      // mpi initialize
14      MPI_Init(NULL, NULL);
15
16      // fetch size and rank
17      int size, rank;
18      int save = 0;
19      MPI_Comm_size(MPI_COMM_WORLD, &size);
20      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
21      // initialization, N = 10 default
22      int N = 10;
23      // parse argument
24      char buff[100];
25      for (int i = 0; i < argc; i++){
26          strcpy(buff, argv[i]);
27          if (strcmp(buff, "-n")==0){
28              std::string num(argv[i+1]);
29              N = std::stoi(num);
30          }
31          if (strcmp(buff, "--save")==0){
32              std::string num(argv[i+1]);
33              save = std::stoi(num);
34          }
35      }
36
37      // determine start and end index
38      int *arr;
39      int *arr_;
40      int jobsize = N / size;
41      int start_idx = jobsize * rank;
42      int end_idx = start_idx + jobsize;
43      int *rbuf = (int *)malloc(sizeof(int) * size);
44      double *time_arr = (double *)malloc(sizeof(double) * size);
45      double t1, t2, t, t_sum;
46      int from, to;
47      int flag;
48      if (rank == size-1) end_idx = N;
49
50      // master proc array allocation
51      if (rank==0){
52          printf("Name: Haoran Sun\n");
53          printf("ID: 119010271\n");
54          printf("HW: Parallel Odd-Even Sort\n");
55          printf("Set N to %d.\n", N);

```

```

56     arr_ = (int *) malloc(sizeof(int) * N);
57     fill_rand_arr(arr_, N);
58     // print_arr(arr_, N);
59 }
60 arr = (int *) malloc(sizeof(int) * (end_idx-start_idx));
61
62 // MAIN PROGRAM
63 // start time
64 t1 = MPI_Wtime();
65
66 // CASE 1: sequential
67 if (size==1) {
68     odd_even_sort(arr_, N, 0);
69 }
70
71 // CASE 2: parallel
72 else {
73     // STEP 1: data transfer master --> slave
74     if (rank==0){
75         for (int i = 1; i < size; i++){
76             int start = i*jobsize;
77             int end = start + jobsize;
78             MPI_Request request;
79             if (i==size-1) end += N%size;
80             MPI_Send(arr_+start, end-start, MPI_INT, i, 0, MPI_COMM_WORLD);
81         }
82         for (int i = 0; i < jobsize; i++) arr[i] = arr_[i];
83     }
84     else
85         MPI_Recv(arr, end_idx-start_idx, MPI_INT, 0, 0, MPI_COMM_WORLD,
86                 MPI_STATUS_IGNORE);
87     MPI_Barrier(MPI_COMM_WORLD);
88     // print_arr(arr, end_idx-start_idx);
89
90     // STEP 2: main program
91     while (true){
92         flag = 1;
93
94         int a, b;
95         int from, to;
96         MPI_Request request = MPI_REQUEST_NULL;
97         MPI_Status status;
98         // STEP 2.1: odd loop
99         // inner odd loop
100        for (int i = 1; i < end_idx-start_idx; i++){
101            if ((start_idx+i)%2==1){
102                a = arr[i-1];
103                b = arr[i];
104                if (b < a){
105                    arr[i] = a;
106                    arr[i-1] = b;
107                    flag = 0;
108                }
109            }
110        }
111        // possible interexchange
112        if (start_idx>0 && start_idx%2==1){
113            // printf("odd start_idx %d rank %d sendrecv rank %d\n",
114                    start_idx, rank, rank-1);

```

```

113         to = arr[0];
114         MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank
115                     -1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
116         if (from > to) {
117             arr[0] = from;
118             flag = 0;
119         }
120     }
121     else if ((end_idx-1)%2==0 && end_idx<N){
122         // printf("odd end_idx %d rank %d sendrecv rank %d\n", end_idx,
123             rank, rank+1);
124         to = arr[end_idx-start_idx-1];
125         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank
126                     +1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
127         if (from < to) {
128             arr[end_idx-start_idx-1] = from;
129             flag = 0;
130         }
131     }
132     // MPI_Barrier(MPI_COMM_WORLD);
133
134     // STEP 2.2: even loop
135     // inner even loop
136     for (int i = 1; i < end_idx-start_idx; i++){
137         if ((start_idx+i)%2==0){
138             a = arr[i-1];
139             b = arr[i];
140             if (b < a){
141                 arr[i] = a;
142                 arr[i-1] = b;
143                 flag = 0;
144             }
145         }
146     }
147     // possible interexchange
148     if (rank%2==1 && start_idx>0 && start_idx%2==0){
149         // printf("even start_idx %d rank %d sendrecv rank %d\n",
150             start_idx, rank, rank-1);
151         to = arr[0];
152         MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank
153                     -1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
154         if (from > to) {
155             arr[0] = from;
156             flag = 0;
157         }
158     }
159     else if (rank%2==0 && (end_idx-1)%2==1 && end_idx<N){
160         // printf("even end_idx %d rank %d sendrecv rank %d\n", end_idx,
161             rank, rank+1);
162         to = arr[end_idx-start_idx-1];
163         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank
164                     +1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
165         if (from < to) {
166             arr[end_idx-start_idx-1] = from;
167             flag = 0;
168         }
169     }
170     // MPI_Barrier(MPI_COMM_WORLD);
171     if (rank%2==0 && start_idx>0 && start_idx%2==0){

```

```

165         // printf("even start_idx %d rank %d sendrecv rank %d\n",
166             start_idx, rank, rank-1);
167         to = arr[0];
168         MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank
169             -1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
170         if (from > to) {
171             arr[0] = from;
172             flag = 0;
173         }
174     }
175     else if (rank%2==1 && (end_idx-1)%2==1 && end_idx<N){
176         // printf("even end_idx %d rank %d sendrecv rank %d\n", end_idx,
177             rank, rank+1);
178         to = arr[end_idx-start_idx-1];
179         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank
180             +1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
181         if (from < to) {
182             arr[end_idx-start_idx-1] = from;
183             flag = 0;
184         }
185     }
186     MPI_Barrier(MPI_COMM_WORLD);
187
188     // STEP 2.3: sending stop flag to master, master decide whether
189     // to continue
190     MPI_Gather(&flag, 1, MPI_INT, rbuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
191     if (rank==0) {
192         // print_arr(rbuf, size);
193         for (int i = 0; i < size; i++){
194             if (rbuf[i] != 1) {
195                 flag = 0;
196             }
197         }
198     }
199     MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
200     MPI_Barrier(MPI_COMM_WORLD);
201     // printf("2. rank %d flag = %d\n", rank, flag);
202     if (flag == 1) {
203         // odd_even_sort(arr, end_idx-start_idx, 0);
204         break;
205     }
206 }
207
208 // STEP 3: gather sorted array
209 MPI_Gather(arr, jobsize, MPI_INT, arr_, jobsize, MPI_INT, 0,
210     MPI_COMM_WORLD);
211 MPI_Barrier(MPI_COMM_WORLD);
212 // tail case
213 if (N%size != 0) {
214     if (rank == size-1) MPI_Send(arr+jobsize, N%size, MPI_INT, 0, 2,
215         MPI_COMM_WORLD);
216     if (rank == 0) MPI_Recv(arr+N/size*size, N%size, MPI_INT, size-1,
217         2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
218 }
219 }
220
221 // end time
222 t2 = MPI_Wtime();

```

```

217     t = t2 - t1;
218     MPI_Gather(&t, 1, MPI_DOUBLE, time_arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
219     if (rank==0) {
220         t_sum = arr_sum(time_arr, size);
221         printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t_sum,
                size);
222         // check_sorted(arr_, N);
223     }
224
225     // // master print result
226     // if (rank==0) print_arr(arr_, jobsizes);
227
228     // free array
229     free(arr);
230     if (rank==0) free(arr_);
231
232     // print info to file
233     if (rank==0 && save==1) {
234         FILE* outfile;
235         if (size==1) outfile = fopen("data_seq.txt", "a");
236         else outfile = fopen("data.txt", "a");
237         fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t_sum);
238         fclose(outfile);
239     }
240     // this line is added to make sure that the data is correctly saved
241     std::this_thread::sleep_for(std::chrono::milliseconds(100));
242
243     return 0;
244 }
245

```

main.seq.cpp

```

1  #include <stdio.h>
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <string.h>
6  #include <chrono>
7  #include <thread>
8  #include "utils.h"
9
10
11 int main(int argc, char* argv[]) {
12     // fetch size and rank
13     int size = 1, rank = 0;
14     int save = 0;
15     // initialization, N = 10 default
16     int N = 10;
17     // parse argument
18     char buff[100];
19     for (int i = 0; i < argc; i++){
20         strcpy(buff, argv[i]);
21         if (strcmp(buff, "-n")==0){
22             std::string num(argv[i+1]);
23             N = std::stoi(num);
24         }
25         if (strcmp(buff, "--save")==0){
26             std::string num(argv[i+1]);

```

```

27         save = std::stoi(num);
28     }
29 }
30
31 // determine start and end index
32 int *arr = (int *)malloc(sizeof(int) * N);
33 int *rbuf = (int *)malloc(sizeof(int) * size);
34
35 // master proc array allocation
36 printf("Name: Haoran Sun\n");
37 printf("ID: 119010271\n");
38 printf("HW: Parallel Odd-Even Sort\n");
39 printf("Set N to %d.\n", N);
40 fill_rand_arr(arr, N);
41
42 // MAIN PROGRAM
43 // start time
44 auto t1 = std::chrono::system_clock::now();
45
46 // sequential sort
47 odd_even_sort(arr, N, 0);
48
49 // end time
50 auto t2 = std::chrono::system_clock::now();
51 auto dur = t2 - t1;
52 auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
53 double t = dur_.count();
54 printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t, size);
55
56 // free array
57 free(arr);
58
59 // print data info to file
60 FILE* outfile;
61 if (size==1) outfile = fopen("data_seq.txt", "a");
62 else outfile = fopen("data.txt", "a");
63 fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t);
64 fclose(outfile);
65
66 // this line is added to make sure that the data is correctly saved
67 std::this_thread::sleep_for(std::chrono::milliseconds(100));
68
69 return 0;
70 }

```

utils.h

```

1 #include <stdio.h>
2 #include <iostream>
3 #include <cstdlib>
4 #include <string.h>
5
6 // fill random array
7 void fill_rand_arr(int* arr, int N){
8     std::srand(time(0));
9     for (int i = 0; i < N; i++){
10         arr[i] = std::rand() % 10000;
11     }
12 }

```

```
13
14 // binary sort
15 void odd_even_sort(int* arr, int N, int f){
16     if (f==1) return;
17     int a, b;
18     int flag = 1;
19     // odd loop
20     for (int i = 1; i < N; i += 2){
21         a = arr[i-1];
22         b = arr[i];
23         if (b < a){
24             arr[i] = a;
25             arr[i-1] = b;
26             flag = 0;
27         }
28     }
29     // even loop
30     for (int i = 2; i < N; i += 2){
31         a = arr[i-1];
32         b = arr[i];
33         if (b < a){
34             arr[i] = a;
35             arr[i-1] = b;
36             flag = 0;
37         }
38     }
39     return odd_even_sort(arr, N, flag);
40 }
41 // binary sort single iteration
42
43 // min-max
44 void min_max(int *arr, int N) {
45     int min_idx = 0;
46     int max_idx = 0;
47     int min = arr[0];
48     int max = arr[0];
49
50     for (int i = 1; i < N; i++) {
51         if (arr[i] > max) {
52             max = arr[i];
53             max_idx = i;
54         }
55         if (arr[i] < min) {
56             min = arr[i];
57             min_idx = i;
58         }
59     }
60
61     arr[min_idx] = arr[0];
62     arr[max_idx] = arr[N-1];
63     arr[0] = min;
64     arr[N-1] = max;
65 }
66
67 void print_arr(int* arr, int N){
68     for (int i = 0; i < N; i++){
69         printf("%d ", arr[i]);
70     }
71     printf("\n");
```

```
72 }
73
74 void check_sorted(int* arr, int N){
75     for (int i = 0; i < N-1; i++){
76         if (arr[i] > arr[i+1]) {
77             printf("Error at idx %d.\n", i);
78             return;
79         }
80     }
81     printf("Array sorted.\n");
82     return;
83 }
84
85
86 template <class T>
87 T arr_sum(T *arr, int N){
88     T sum = 0;
89     for (int i = 0; i < N; i++) sum += arr[i];
90     return sum;
91 }
```