

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.

```
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,
5                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6      if (from > to) {
7          arr[0] = from;
8          flag = 0;
9      }
10 }
11 else if ((end_idx-1)%2==0 && end_idx<N){
12     // printf("odd end_idx %d rank %d sendrecv rank %d\n", end_idx, rank, rank+1);
13     to = arr[end_idx-start_idx-1];
14     MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1,
15                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16     if (from < to) {
17         arr[end_idx-start_idx-1] = from;
18         flag = 0;
19     }
20 }

```

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, $p = 4, 8, \dots, 80$) and 20 different array sizes (from 50000 to 1000000 with increment 50000, $n = 5 \times 10^4, 10^5, \dots, 10^6$), 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

```

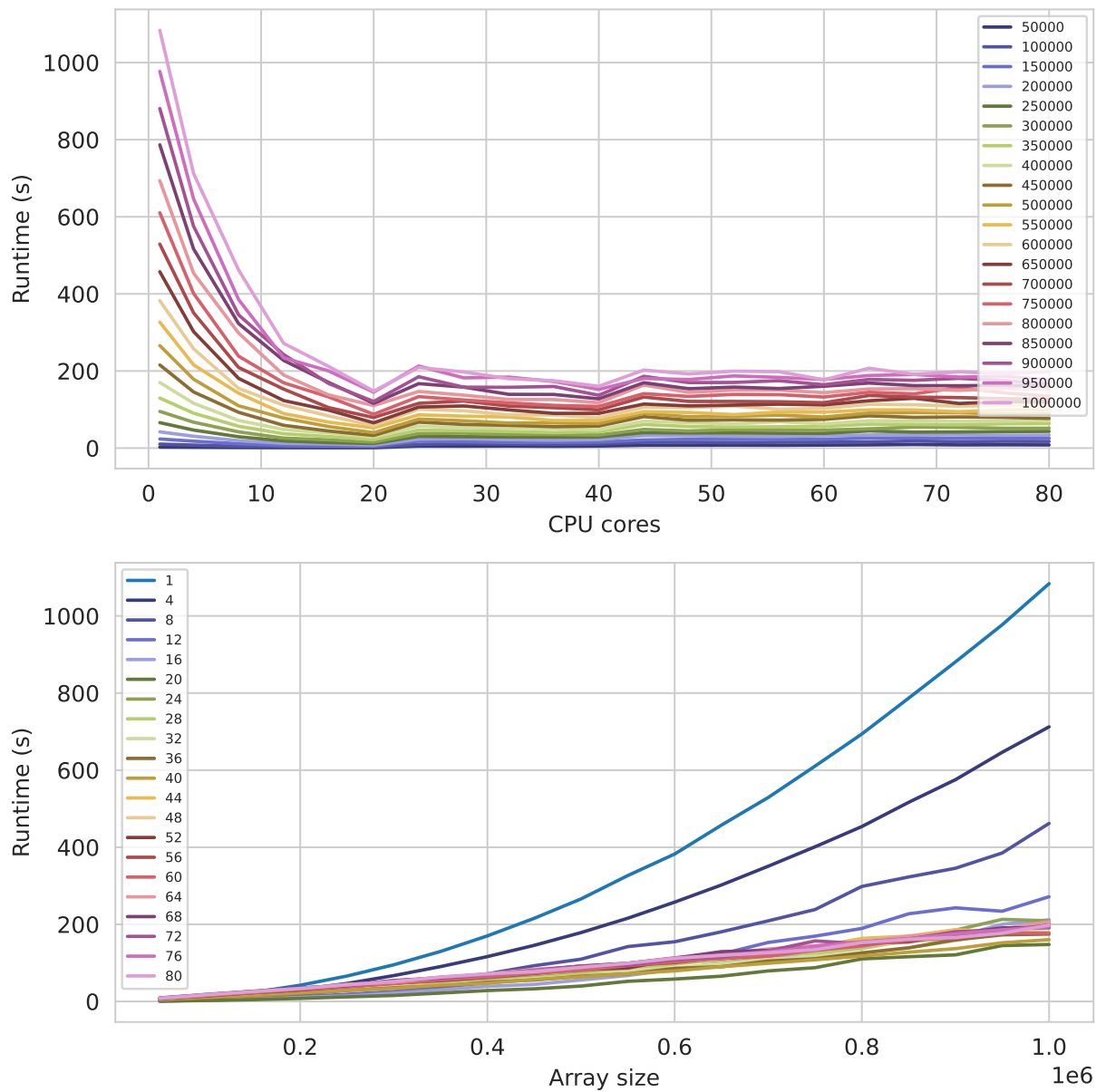


Figure 1: Running time versus the number of CPU cores

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,

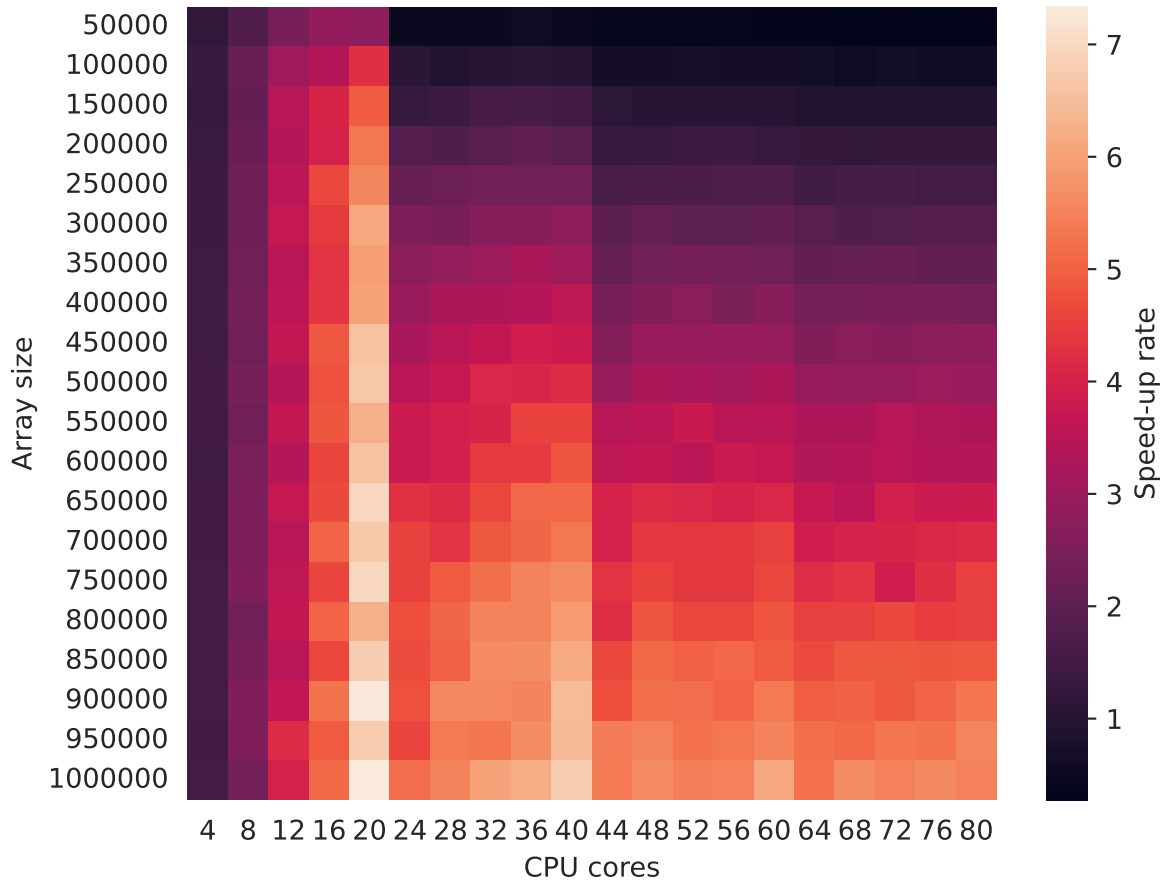


Figure 2: Heatmap of speed-up rate

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

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

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.

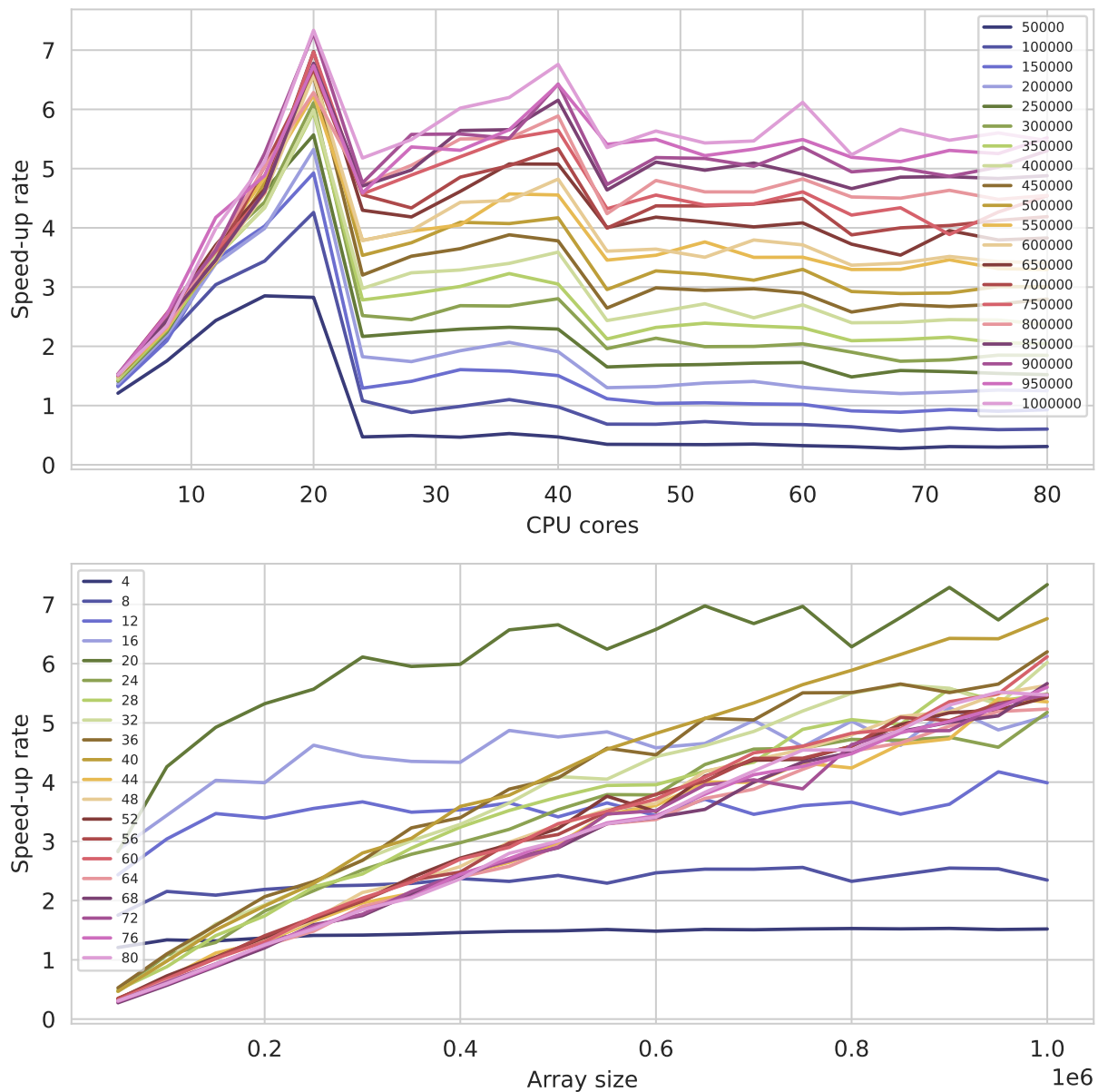


Figure 3: Speed up rate versus the number of CPU cores and array size

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 deduce the conclusion that the optimal MPI configuration under this cluster would be 20 cores. Moreover, speculation could be proposed that the optimal MPI

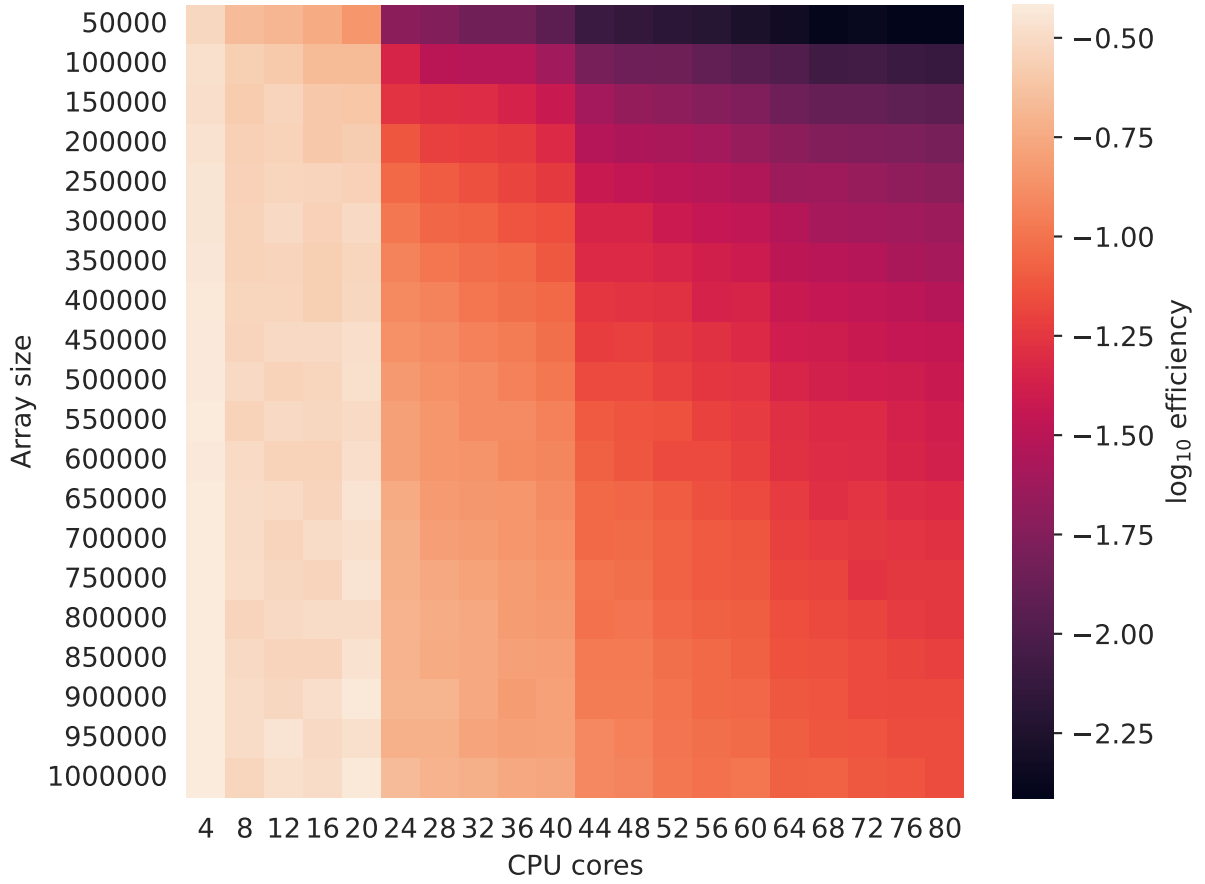


Figure 4: Heat map of $\log_{10}(\text{seq CPU time}/\text{par CPU time})$

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

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

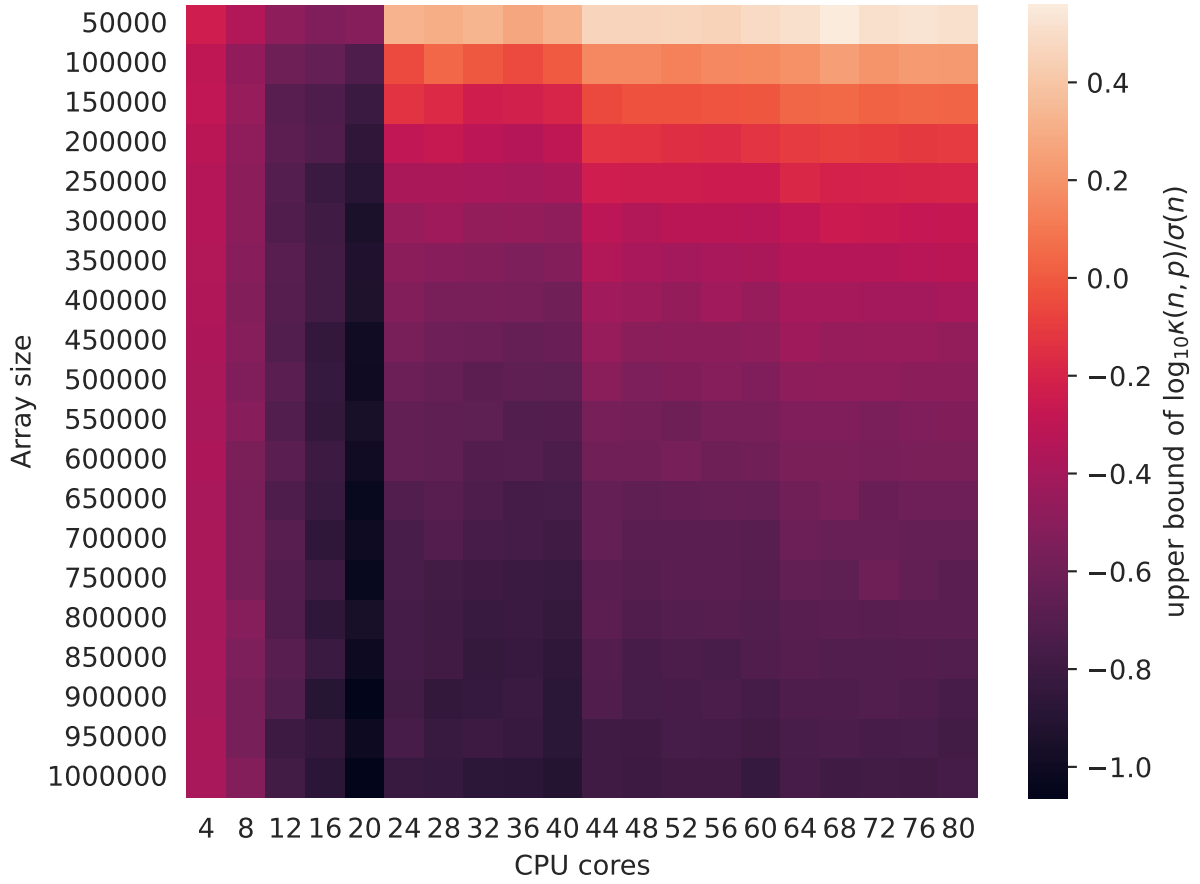


Figure 5: Heat map of upper bound of $\log_{10} \kappa(n, p)/\sigma(n)$.

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

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

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

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

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 $n = 4, \dots, 20$, 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

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

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

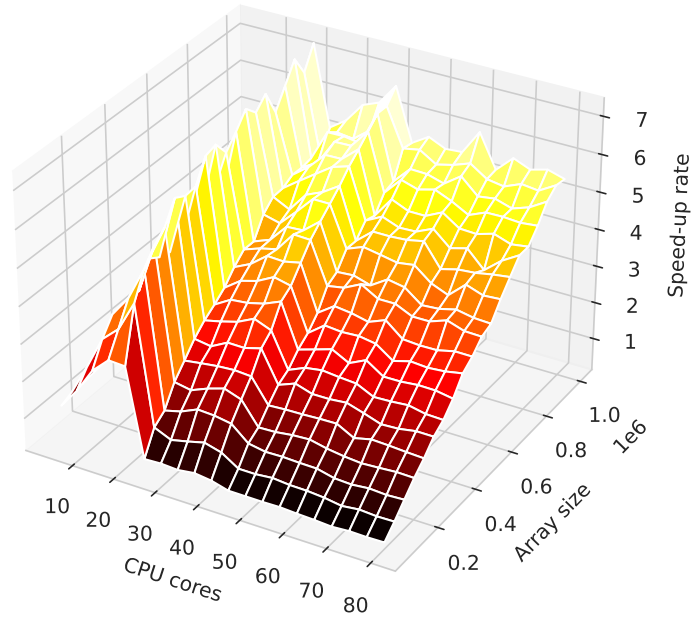


Figure A.1: 3D heatmap of speed-up rate

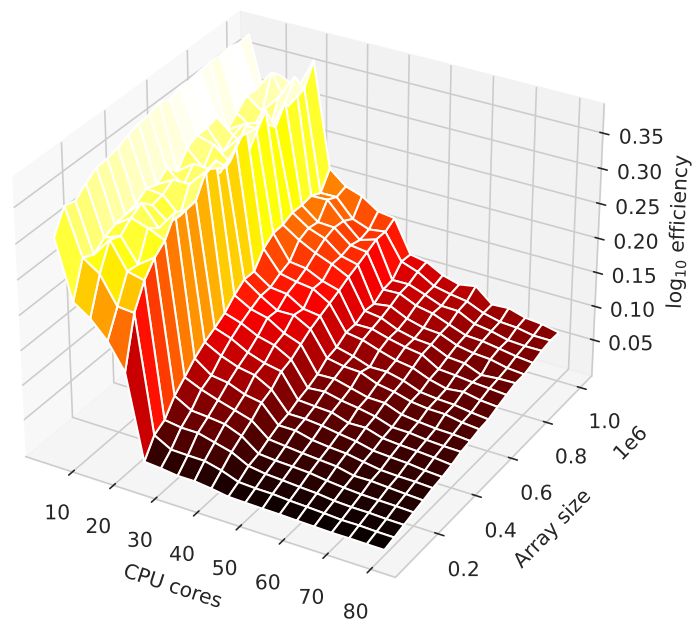


Figure A.2: 3D heatmap of log efficiency

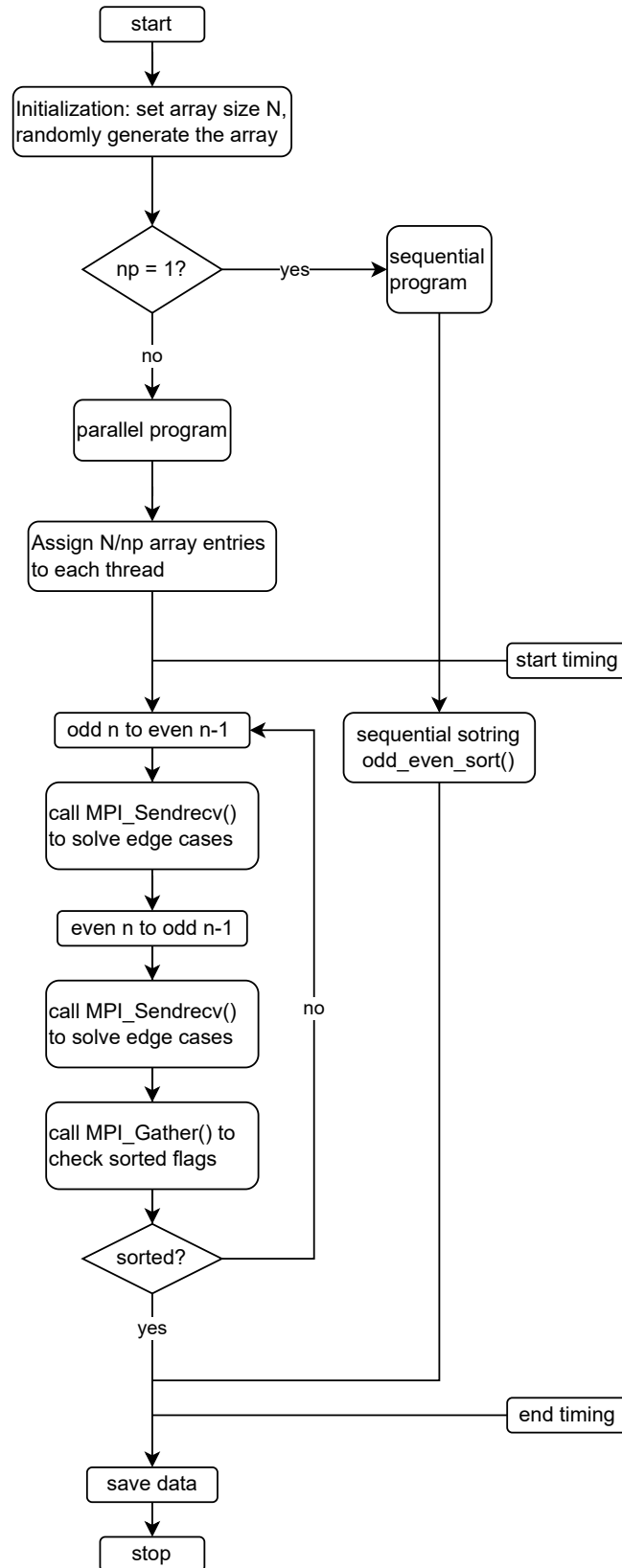


Figure A.3: Program flowchart

B Source code

main.cpp

```

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     int print = 0;
20     MPI_Comm_size(MPI_COMM_WORLD, &size);
21     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
22     // initialization, N = 10 default
23     int N = 10;
24     // parse argument
25     char buff[100];
26     for (int i = 0; i < argc; i++){
27         strcpy(buff, argv[i]);
28         if (strcmp(buff, "-n")==0){
29             std::string num(argv[i+1]);
30             N = std::stoi(num);
31         }
32         if (strcmp(buff, "--save")==0){
33             std::string num(argv[i+1]);
34             save = std::stoi(num);
35         }
36         if (strcmp(buff, "--print")==0){
37             std::string num(argv[i+1]);
38             print = std::stoi(num);
39         }
40     }
41
42     // determine start and end index
43     int *arr;
44     int *arr_;
45     int jobsize = N / size;
46     int start_idx = jobsize * rank;
47     int end_idx = start_idx + jobsize;
48     int *rbuf = (int *)malloc(sizeof(int) * size);
49     double *time_arr = (double *)malloc(sizeof(double) * size);
50     double t1, t2, t, t_sum;
51     int from, to;
52     int flag;
53     if (rank == size-1) end_idx = N;
54
55     // master proc array allocation
56     if (rank==0){
57         printf("Name: Haoran Sun\n");
58         printf("ID: 119010271\n");
59         printf("HW: Parallel Odd-Even Sort\n");
60         printf("Set N to %d.\n", N);
61         arr_ = (int *) malloc(sizeof(int) * N);
62         fill_rand_arr(arr_, N);

```

```

63         if (print==1) {
64             printf("Array:\n");
65             print_arr(arr_, N);
66         }
67     }
68     arr = (int *) malloc(sizeof(int) * (end_idx-start_idx));
69
70     // MAIN PROGRAM
71     // start time
72     t1 = MPI_Wtime();
73
74     // CASE 1: sequential
75     if (size==1) {
76         odd_even_sort(arr_, N, 0);
77     }
78
79     // CASE 2: parallel
80     else {
81         // STEP 1: data transfer master --> slave
82         if (rank==0){
83             for (int i = 1; i < size; i++){
84                 int start = i*jobsize;
85                 int end = start + jobsize;
86                 MPI_Request request;
87                 if (i==size-1) end += N*size;
88                 MPI_Send(arr_+start, end-start, MPI_INT, i, 0, MPI_COMM_WORLD);
89             }
90             for (int i = 0; i < jobsize; i++) arr[i] = arr_[i];
91         }
92         else
93             MPI_Recv(arr, end_idx-start_idx, MPI_INT, 0, 0, MPI_COMM_WORLD,
94                     MPI_STATUS_IGNORE);
95         MPI_Barrier(MPI_COMM_WORLD);
96         // print_arr(arr, end_idx-start_idx);
97
98         // STEP 2: main program
99         while (true){
100             flag = 1;
101
102             int a, b;
103             int from, to;
104             MPI_Request request = MPI_REQUEST_NULL;
105             MPI_Status status;
106             // STEP 2.1: odd loop
107             // inner odd loop
108             for (int i = 1; i < end_idx-start_idx; i++){
109                 if ((start_idx+i)%2==1){
110                     a = arr[i-1];
111                     b = arr[i];
112                     if (b < a){
113                         arr[i] = a;
114                         arr[i-1] = b;
115                         flag = 0;
116                     }
117                 }
118             }
119             // possible interexchange
120             if (start_idx>0 && start_idx%2==1){
121                 // printf("odd start_idx %d rank %d sendrecv rank %d\n", start_idx, rank
122                 , rank-1);
123                 to = arr[0];
124                 MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2,
125                             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
126                 if (from > to) {
127                     arr[0] = from;
128                     flag = 0;
129                 }
130             }
131         }
132     }

```

```

127     }
128     else if ((end_idx-1)%2==0 && end_idx<N){
129         // printf("odd end_idx %d rank %d sendrecv rank %d\n", end_idx, rank,
130             rank+1);
131         to = arr[end_idx-start_idx-1];
132         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1,
133             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
134         if (from < to) {
135             arr[end_idx-start_idx-1] = from;
136             flag = 0;
137         }
138     }
139     // MPI_Barrier(MPI_COMM_WORLD);
140     // STEP 2.2: even loop
141     // inner even loop
142     for (int i = 1; i < end_idx-start_idx; i++){
143         if ((start_idx+i)%2==0){
144             a = arr[i-1];
145             b = arr[i];
146             if (b < a){
147                 arr[i] = a;
148                 arr[i-1] = b;
149                 flag = 0;
150             }
151         }
152     }
153     // possible interexchange
154     if (rank%2==1 && start_idx>0 && start_idx%2==0){
155         // printf("even start_idx %d rank %d sendrecv rank %d\n", start_idx,
156             rank, rank-1);
157         to = arr[0];
158         MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2,
159             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
160         if (from > to) {
161             arr[0] = from;
162             flag = 0;
163         }
164     }
165     else if (rank%2==0 && (end_idx-1)%2==1 && end_idx<N){
166         // printf("even end_idx %d rank %d sendrecv rank %d\n", end_idx, rank,
167             rank+1);
168         to = arr[end_idx-start_idx-1];
169         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1,
170             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
171         if (from < to) {
172             arr[end_idx-start_idx-1] = from;
173             flag = 0;
174         }
175     }
176     // MPI_Barrier(MPI_COMM_WORLD);
177     if (rank%2==0 && start_idx>0 && start_idx%2==0){
178         // printf("even start_idx %d rank %d sendrecv rank %d\n", start_idx,
179             rank, rank-1);
180         to = arr[0];
181         MPI_Sendrecv(&to, 1, MPI_INT, rank-1, 1, &from, 1, MPI_INT, rank-1, 2,
182             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
183         if (from > to) {
184             arr[0] = from;
185             flag = 0;
186         }
187     }
188     else if (rank%2==1 && (end_idx-1)%2==1 && end_idx<N){
189         // printf("even end_idx %d rank %d sendrecv rank %d\n", end_idx, rank,
190             rank+1);
191         to = arr[end_idx-start_idx-1];

```

```

184         MPI_Sendrecv(&to, 1, MPI_INT, rank+1, 2, &from, 1, MPI_INT, rank+1, 1,
185                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
186         if (from < to) {
187             arr[end_idx-start_idx-1] = from;
188             flag = 0;
189         }
190     MPI_Barrier(MPI_COMM_WORLD);
191
192     // STEP 2.3: sending stop flag to master, master decide whether
193     // to continue
194     MPI_Gather(&flag, 1, MPI_INT, rbuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
195     if (rank==0) {
196         // print_arr(rbuf, size);
197         for (int i = 0; i < size; i++){
198             if (rbuf[i] != 1) {
199                 flag = 0;
200             }
201         }
202     }
203     MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
204     MPI_Barrier(MPI_COMM_WORLD);
205     // printf("2. rank %d flag = %d\n", rank, flag);
206     if (flag == 1) {
207         // odd_even_sort(arr, end_idx-start_idx, 0);
208         break;
209     }
210 }
211
212
213 // STEP 3: gather sorted array
214 MPI_Gather(arr, jobsize, MPI_INT, arr_, jobsize, MPI_INT, 0, MPI_COMM_WORLD);
215 MPI_Barrier(MPI_COMM_WORLD);
216 // tail case
217 if (N%size != 0) {
218     if (rank == size-1) MPI_Send(arr+jobsize, N%size, MPI_INT, 0, 2,
219                                 MPI_COMM_WORLD);
220     if (rank == 0) MPI_Recv(arr_+N/size*size, N%size, MPI_INT, size-1, 2,
221                             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
222 }
223
224 // master print result
225 if (rank==0 && print==1) {
226     printf("Sorted array:\n");
227     print_arr(arr_, N);
228 }
229
230 // end time
231 t2 = MPI_Wtime();
232 t = t2 - t1;
233 MPI_Gather(&t, 1, MPI_DOUBLE, time_arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
234 if (rank==0) {
235     t_sum = arr_sum(time_arr, size);
236     printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t_sum, size);
237     // check_sorted(arr_, N);
238 }
239
240 // free array
241 free(arr);
242 if (rank==0) free(arr_);
243
244 // print info to file
245 if (rank==0 && save==1) {
246     FILE* outfile;
247     if (size==1) outfile = fopen("data_seq.txt", "a");
248     else outfile = fopen("data.txt", "a");

```

```

248     fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t_sum);
249     fclose(outfile);
250 }
251 // this line is added to make sure that the data is correctly saved
252 std::this_thread::sleep_for(std::chrono::milliseconds(100));
253
254 return 0;
255 }

```

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     int print = 0;
16     // initialization, N = 10 default
17     int N = 10;
18     // parse argument
19     char buff[100];
20     for (int i = 0; i < argc; i++){
21         strcpy(buff, argv[i]);
22         if (strcmp(buff, "-n")==0){
23             std::string num(argv[i+1]);
24             N = std::stoi(num);
25         }
26         if (strcmp(buff, "--save")==0){
27             std::string num(argv[i+1]);
28             save = std::stoi(num);
29         }
30         if (strcmp(buff, "--print")==0){
31             std::string num(argv[i+1]);
32             print = std::stoi(num);
33         }
34     }
35
36     // determine start and end index
37     int *arr = (int *)malloc(sizeof(int) * N);
38     int *rbuf = (int *)malloc(sizeof(int) * size);
39
40     // master proc array allocation
41     printf("Name: Haoran Sun\n");
42     printf("ID: 119010271\n");
43     printf("HW: Parallel Odd-Even Sort\n");
44     printf("Set N to %d.\n", N);
45     fill_rand_arr(arr, N);
46     if (print==1) {
47         printf("Array:\n");
48         print_arr(arr, N);
49     }
50
51     // MAIN PROGRAM
52     // start time
53     auto t1 = std::chrono::system_clock::now();
54
55     // sequential sort
56     odd_even_sort(arr, N, 0);

```




```

57
58 // print array
59 if (print==1) {
60     printf("Sorted array:\n");
61     print_arr(arr, N);
62 }
63
64 // end time
65 auto t2 = std::chrono::system_clock::now();
66 auto dur = t2 - t1;
67 auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
68 double t = dur_.count();
69 printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t, size);
70
71 // free array
72 free(arr);
73
74 // print data info to file
75 if (save==1) {
76     FILE* outfile;
77     outfile = fopen("data_seq.txt", "a");
78     fprintf(outfile, "%10d %5d %10.2f %10.2f\n", N, size, t, t);
79     fclose(outfile);
80 }
81
82 // this line is added to make sure that the data is correctly saved
83 std::this_thread::sleep_for(std::chrono::milliseconds(100));
84
85 return 0;
86 }

```

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("%5d ", arr[i]);
70         if ((i+1)%10 == 0)
71             printf("\n");
72     }
73     if (N%10 != 0) printf("\n");
74 }
75
76 void check_sorted(int* arr, int N){
77     for (int i = 0; i < N-1; i++){
78         if (arr[i] > arr[i+1]) {
79             printf("Error at idx %d.\n", i);
80             return;
81         }
82     }
83     printf("Array sorted.\n");
84     return;
85 }
86
87
88 template <class T>
89 T arr_sum(T *arr, int N){
90     T sum = 0;
91     for (int i = 0; i < N; i++) sum += arr[i];
92     return sum;
93 }

```