

# ECE445: Parallel and Distributed Computing

## Winter Semester 2024-2025

### 3rd set of Exercises

Team #15

Athanasios Kastoras 03101,  
Spyridon Stamoulis 03775

University of Thessaly  
{akastoras,spstamoulis}@uth.gr

## 1 Exercise 1

The code is in the bottom section.  
Here is a screenshot of the execution:

```
(kali@kali)~/Desktop/par/hw3
$ mpirun -np 4 ./ex1
Processor name: kali
Number of tasks: 4
Hello. This is the master node.
Hello. This is node 1.
Hello. This is node 3.
Hello. This is node 2.
Average time for int: 20.741759 microseconds
Average time for float: 23.842604 microseconds
Average time for double: 28.386486 microseconds

(kali@kali)~/Desktop/par/hw3
$
```

The table describing the times of each communication:

Number of tasks	int	float	double
Time (sec)	20.7417	23.8426	28.3864

Table 1: Execution time Point2Point communication.

## 2 Exercise 2

The code with the implementation of a broadcast of an integer using MPI\_Bcast and using a custom implementation with MPI\_Send/MPI\_Recv is listed in section B. On Table 2 we can see the communication times measured by our program.

Number of tasks		2	4	6	8	10	12	20
Time (sec)	MPI Function	7.207e-08	1.575e-07	2.189e-07	3.615e-07	3.79e-07	4.221e-07	1.141e-06
	My Implementation	4.194e-07	7.14e-07	1.477e-06	1.229e-06	1.977e-06	4.217e-06	2.713e-05

Table 2: Mean communication time for broadcasting an integer across all tasks.

### 3 Exercise 3

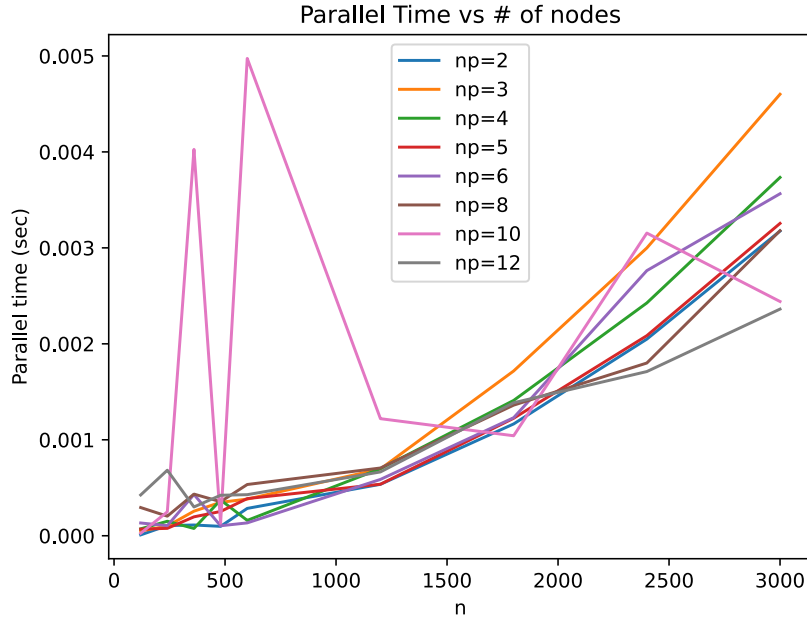


Fig. 1: Parallel times in relation to num of tasks.

MPI Tasks ( $np$ ) / Size ( $n$ )	120	240	360	480	600	1200	1800	2400	3000
2	0.009	0.106	0.112	0.098	0.285	0.536	1.165	2.049	3.175
3	0.049	0.101	0.256	0.348	0.382	0.696	1.717	2.999	4.601
4	0.072	0.151	0.076	0.373	0.160	0.699	1.410	2.426	3.734
5	0.069	0.077	0.197	0.253	0.387	0.537	1.224	2.085	3.255
6	0.133	0.105	0.430	0.104	0.134	0.587	1.230	2.763	3.564
8	0.294	0.204	0.434	0.355	0.534	0.706	1.361	1.800	3.179
10	0.026	0.249	4.026	0.131	4.974	1.220	1.042	3.154	2.441
12	0.425	0.683	0.300	0.422	0.428	0.665	1.386	1.711	2.362

Table 3: Execution times (in milliseconds) for different problem sizes ( $n$ ) and number of MPI tasks ( $np$ )

### 4 Exercise 4

The algorithm consists of the following steps:

1. Sort local arrays using serial Odd-Even sorting algorithm, which takes  $\Theta((\frac{n}{p})^2)$  operations.
2. Doing  $p$  steps of compare-split operations between all nodes.
3. A compare split operation includes  $\Theta(\frac{n}{p})$  communication steps and  $\Theta(\frac{n}{p})$  computation steps on each node.

Therefore the final theoretical time for our algorithm is  $\Theta((\frac{n}{p})^2) + p \times \Theta(\frac{n}{p}) = \Theta((\frac{n}{p})^2) + \Theta(n)$ . The measured execution times for this algorithm are shown in Table 4.

Problem Size	Serial (s)	MPI (Time   Speedup)			
		2	4	5	10
10000	0.048	0.0075 (6.4x)	0.0078 (6.2x)	0.0073 (6.6x)	0.005 (9.6x)
50000	1.248	0.21 (5.9x)	0.085 (14.7x)	0.057 (21.9x)	0.014 (89.1x)
100000	5.927	1.252 (4.7x)	0.386 (15.4x)	0.245 (24.2x)	0.091 (65.1x)
200000	26.018	5.939 (4.4x)	1.892 (13.8x)	1.145 (22.7x)	0.282 (92.3x)
1000000	679.069	167.335 (4.1x)	55.1 (12.3x)	38.516 (17.6x)	9.62 (70.6x)

Table 4: Execution time and speedup of various array sizes for parallel executions with 2, 4, 5 and 10 parallel tasks.

The expected speedup for 2, 4, 5 and 10, is correspondingly 4, 16, 25 and 100 since the performance is inversely proportional to the square of  $p$ . Of course, we would expect the speedup to be a bit lower than that, and to be closer to that value for larger sizes. We observe, that this assumption is correct except for when using 2 MPI tasks where the speedup is higher than 4x. This anomaly can be explained due to inaccuracies in the theoretical analysis of the algorithm.

## A Code of exercise 1

```

1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  #define NUM_ITERATIONS 1000
7
8  int main(int argc, char** argv) {
9      int rank, size;
10     char processor_name[MPI_MAX_PROCESSOR_NAME];
11     int name_len;
12     double avint, avfloat, avdoubl;
13
14     MPI_Init(&argc, &argv);
15     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16     MPI_Comm_size(MPI_COMM_WORLD, &size);
17     MPI_Get_processor_name(processor_name, &name_len);
18
19     // Synchronize output before printing messages to ensure order
20     MPI_Barrier(MPI_COMM_WORLD);
21
22     if (rank == 0) {
23         printf("Processor name: %s\n", processor_name);
24         printf("Number of tasks: %d\n", size);
25         printf("Hello. This is the master node.\n");
26     }else{
27
28         printf("Hello. This is node %d.\n", rank);
29     }
30
31
32     //count p2p communication time
33     if (rank < 2) { // Only rank 0 and rank 1 will participate
34         MPI_Barrier(MPI_COMM_WORLD); // Synchronize communication
35
36         // Timing for int, float, double
37         double start_time, end_time;
38
39         if (rank == 0) {
40             int int_data = 42;
41             float float_data = 42.42;
42             double double_data = 42.4242;
43
44             // Measure time for int
45             start_time = MPI_Wtime();
46             for (int i = 0; i < NUM_ITERATIONS; i++) {
47                 MPI_Sendrecv(&int_data, 1, MPI_INT, 1, 0, &int_data, 1, MPI_INT
, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
48             }
49             end_time = MPI_Wtime();
50             avint= (end_time - start_time) / NUM_ITERATIONS * 1e6;
51
52             // Measure time for float
53             start_time = MPI_Wtime();
54             for (int i = 0; i < NUM_ITERATIONS; i++) {
55                 MPI_Sendrecv(&float_data, 1, MPI_FLOAT, 1, 0, &float_data, 1,
MPI_FLOAT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
56             }
57             end_time = MPI_Wtime();
58             avfloat = (end_time - start_time) / NUM_ITERATIONS * 1e6;
59

```

```

60         // Measure time for double
61         start_time = MPI_Wtime();
62         for (int i = 0; i < NUM_ITERATIONS; i++) {
63             MPI_Sendrecv(&double_data, 1, MPI_DOUBLE, 1, 0, &double_data,
1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
64         }
65         end_time = MPI_Wtime();
66         avdoubl = (end_time - start_time) / NUM_ITERATIONS * 1e6;
67
68     } else if (rank == 1) {
69         int int_data;
70         float float_data;
71         double double_data;
72
73         for (int i = 0; i < NUM_ITERATIONS; i++) {
74             MPI_Sendrecv(&int_data, 1, MPI_INT, 0, 0, &int_data, 1, MPI_INT
, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
75         }
76
77         for (int i = 0; i < NUM_ITERATIONS; i++) {
78             MPI_Sendrecv(&float_data, 1, MPI_FLOAT, 0, 0, &float_data, 1,
MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
79         }
80
81         for (int i = 0; i < NUM_ITERATIONS; i++) {
82             MPI_Sendrecv(&double_data, 1, MPI_DOUBLE, 0, 0, &double_data,
1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
83         }
84     }
85     } else {
86         // Rank 2 and rank 3 don't do any communication, so they just wait to
finalize
87         MPI_Barrier(MPI_COMM_WORLD); // Synchronize before exiting
88     }
89
90     if (rank==0){
91         printf("Average time for int: %lf microseconds\n",avint);
92         printf("Average time for float: %lf microseconds\n",avfloat);
93         printf("Average time for double: %lf microseconds\n", avdoubl);
94     }
95
96     MPI_Finalize();
97     return 0;
98 }

```

Listing 1.1: Code of Exercise 1.

## B Code of exercise 2

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #include "mpi.h"
6
7 #define MASTER 0
8 #define SAMPLES 1000000
9
10 int main (int argc, char *argv[])
11 {
12     int    numtasks, taskid, len, meaning_of_life = 0;
13     char   hostname[MPI_MAX_PROCESSOR_NAME];
14     double tstart, tend;
15
16     MPI_Init(&argc, &argv);
17     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18     MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
19     MPI_Get_processor_name(hostname, &len);
20
21     // A
22     if (taskid == MASTER) {
23         printf("Processor name: %s. Number of MPI tasks: %d\n", hostname, numtasks)
24     };
25
26     // B
27     if (taskid == MASTER) {
28         printf("Hello, this is the Master node.\n");
29     } else {
30         printf("Hello this is task %d\n", taskid);
31     }
32     MPI_Barrier(MPI_COMM_WORLD);
33
34     // C
35     if (taskid == MASTER) {
36         meaning_of_life = 42;
37         printf("MASTER: Broadcasting meaning of life to all nodes %d times using
38             Bcast.\n", SAMPLES);
39         tstart = MPI_Wtime();
40     }
41     for (int i = 0; i < SAMPLES; i++) {
42         MPI_Bcast(&meaning_of_life, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
43     }
44
45     // E
46     if (taskid == MASTER) {
47         tend = MPI_Wtime();
48         printf("MASTER: Took %lf seconds or %e seconds per time.\n", SAMPLES, tend
49             - tstart, (tend - tstart) / SAMPLES);
50     }
51     #ifdef DEBUG
52     else {
53         printf("Task %d received the meaning of life: %d\n", taskid,
54             meaning_of_life);
55     }
56     #endif
57     MPI_Barrier(MPI_COMM_WORLD);
58
59     // D
60     if (taskid == MASTER) {

```

```

58     printf("MASTER: Broadcasting meaning of life to all nodes %d times using
Send/Recv.\n", SAMPLES);
59     tstart = MPI_Wtime();
60 }
61 for (int i = 0; i < SAMPLES; i++) {
62     custom_bcast(&meaning_of_life, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
63 }
64
65 // E
66 if (taskid == MASTER) {
67     tend = MPI_Wtime();
68     printf("MASTER: Took %lf seconds or %e seconds per time.\n", SAMPLES, tend
- tstart, (tend - tstart) / SAMPLES);
69 }
70 #ifdef DEBUG
71 else {
72     printf("Task %d received the meaning of life: %d\n", taskid,
meaning_of_life);
73 }
74 #endif
75 MPI_Finalize();
76 }
77
78 void custom_bcast(void *data, int count, MPI_Datatype datatype, int root,
MPI_Comm communicator) {
79     int numtasks, taskid;
80     MPI_Comm_size(communicator, &numtasks);
81     MPI_Comm_rank(communicator, &taskid);
82     MPI_Status status;
83
84     if (taskid == root) {
85         for (int i = 0; i < numtasks; i++) {
86             if (i != root) {
87                 MPI_Send(data, count, datatype, i, 1, communicator);
88             }
89         }
90     } else {
91         MPI_Recv(data, count, datatype, root, 1, communicator, &status);
92     }
93     MPI_Barrier(MPI_COMM_WORLD);
94 }

```

Listing 1.2: Code of Exercise 2.

## C Code of exercise 3

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  // Function to perform matrix-vector multiplication
6  void rowMVMult(int n, double *localA, double *localb, double *localy, MPI_Comm
    comm) {
7      int i, j, rank, size;
8      MPI_Comm_rank(comm, &rank);
9      MPI_Comm_size(comm, &size);
10
11     int local_n = n / size;
12
13     // Initialize localy to 0
14     for (i = 0; i < local_n; i++) {
15         localy[i] = 0.0;
16     }
17
18     // Perform local matrix-vector multiplication
19     for (i = 0; i < local_n; i++) {
20         for (j = 0; j < n; j++) {
21             localy[i] += localA[i * n + j] * localb[j];
22         }
23     }
24 }
25
26 int main(int argc, char *argv[]) {
27     int i, j, n, rank, size;
28     double *A, *b, *y, *localA, *localb, *localy;
29     double start_time, end_time;
30
31     MPI_Init(NULL, NULL);
32     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33     MPI_Comm_size(MPI_COMM_WORLD, &size);
34
35     // Get matrix dimension from user (assume n is divisible by size)
36     if (rank == 0) {
37         scanf("%d", &n);
38     }
39     // Broadcast n to all processes
40     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
41
42     // Calculate local block size
43     int local_n = n / size;
44
45
46     // Allocate memory for local arrays
47     localA = (double *)malloc(local_n * n * sizeof(double));
48     localb = (double *)malloc(n * sizeof(double));
49     localy = (double *)malloc(local_n * sizeof(double));
50
51     // Master process initializes A and b
52     if (rank == 0) {
53         A = (double *)malloc(n * n * sizeof(double));
54         y = (double *)malloc(n * sizeof(double));
55
56         for (i = 0; i < n; i++) {
57             for (j = 0; j < n; j++) {
58                 A[i * n + j] = (i + j) * 10.0;
59             }
60             localb[i] = 10.0;

```



```

61     }
62
63     // Scatter rows of A to all processes
64     MPI_Scatter(A, local_n * n, MPI_DOUBLE, localA, local_n * n, MPI_DOUBLE, 0,
65               MPI_COMM_WORLD);
66
67     // Broadcast b to all processes
68     MPI_Bcast(localb, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
69
70     start_time = MPI_Wtime();
71 } else {
72     // Receive rows of A from master
73     MPI_Scatter(NULL, local_n * n, MPI_DOUBLE, localA, local_n * n, MPI_DOUBLE,
74               0, MPI_COMM_WORLD);
75
76     // printf("test %d\n",rank);
77     // Receive b from master (broadcast)
78     MPI_Bcast(localb, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
79 }
80
81 // Barrier synchronization
82 MPI_Barrier(MPI_COMM_WORLD);
83 // Perform local matrix-vector multiplication
84 rowMVMult(n, localA, localb, localy, MPI_COMM_WORLD);
85
86 if (rank == 0) {
87     // Master process receives the gathered data
88     MPI_Gather(localy, local_n, MPI_DOUBLE, y, local_n, MPI_DOUBLE, 0,
89               MPI_COMM_WORLD);
90 } else {
91     // Other processes only send data
92     MPI_Gather(localy, local_n, MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0,
93               MPI_COMM_WORLD);
94 }
95
96 // Master process prints results and execution time
97 if (rank == 0) {
98     end_time = MPI_Wtime();
99
100    printf("Resulting vector y:\n");
101    for (i = 0; i < n; i++) {
102        printf("%f ", y[i]);
103    }
104    printf("\n");
105
106    printf("Execution time: %f seconds\n", end_time - start_time);
107    // printf("%lf", end_time-start_time);
108    free(A);
109    //free(b);
110    free(y);
111 }
112
113 // Free local memory
114 free(localA);
115 free(localb);
116 free(localy);
117
118 MPI_Finalize();
119 return 0;
120 }

```

Listing 1.3: Code of Exercise 3.

## D Code of exercise 4

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5 #include <stdbool.h>
6 #include <assert.h>
7
8 #include "mpi.h"
9
10 #define MASTER 0
11 #define SAMPLES 1000000
12
13 bool is_sorted(int *arr, int n);
14 int compare_exchange(int *a, int *b);
15 int *comparesplit_merge(int *sorted, int *arr, int n, bool max);
16 int oddevenser_phase(int *array, int n, int phase);
17 int *oddevenpar(int taskid, int numtasks, int task_n, int *sorted);
18
19 int main (int argc, char *argv[])
20 {
21     int    numtasks, taskid, len, task_n, n;
22     char   hostname[MPI_MAX_PROCESSOR_NAME];
23     double tstart, tend;
24
25     n = atoi(argv[1]);
26     MPI_Init(&argc, &argv);
27     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
28     MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
29     MPI_Get_processor_name(hostname, &len);
30
31     if (taskid == MASTER) {
32         printf("Processor name: %s. Number of MPI tasks: %d\n",
33             hostname, numtasks);
34     }
35
36     task_n = n / numtasks;
37     // Initialize an array of size n with random numbers
38     int *unsorted = (int *)malloc(task_n * sizeof(int));
39     int *sorted = (int *)malloc(task_n * sizeof(int));
40     srand(time(NULL) + taskid);
41     for (int i = 0; i < task_n; i++) {
42         unsorted[i] = rand() % 10000;
43     }
44
45     #ifdef DEBUG
46     // Print the unsorted array
47     printf("Task %d - Unsorted array:\n", taskid);
48     for (int i = 0; i < task_n; i++) {
49         printf("%4d ", unsorted[i]);
50     }
51     printf("\n");
52     #endif
53
54     if (taskid == MASTER) {
55         tstart = MPI_Wtime();
56     }
57
58     // Sort the local array
59     oddevenser(unsorted, task_n, sorted);
60     assert(is_sorted(sorted, task_n));
61

```

```

62 // Compare-split the sorted arrays
63 sorted = oddevenpar(taskid, numtasks, task_n, sorted);
64
65 MPI_Barrier(MPI_COMM_WORLD);
66 if (taskid == MASTER) {
67     tend = MPI_Wtime();
68     printf("Took %lf seconds\n", tend - tstart);
69 }
70
71 int *global_sorted = NULL;
72 if (taskid == MASTER) {
73     global_sorted = (int *)malloc(n * sizeof(int));
74 }
75 MPI_Gather(sorted, task_n, MPI_INT, global_sorted, task_n, MPI_INT, MASTER,
76           MPI_COMM_WORLD);
77
78 if (taskid == MASTER) {
79     assert(is_sorted(global_sorted, n));
80     // Print the sorted array
81     printf("Sorted array correctly\n");
82     #ifdef DEBUG
83     for (int i = 0; i < n; i++) {
84         printf("%4d ", global_sorted[i]);
85     }
86     printf("\n");
87     #endif
88     free(global_sorted);
89 }
90 MPI_Finalize();
91 }
92
93 // Odd-even parallel sort
94 int *oddevenpar(int taskid, int numtasks, int task_n, int *sorted)
95 {
96     int *temp = (int *)malloc(task_n * sizeof(int));
97
98     for (int i = 0; i < numtasks; i++) {
99         MPI_Barrier(MPI_COMM_WORLD);
100         if (i % 2 == 0) { // even
101             if (taskid % 2 == 0) {
102                 if (taskid + 1 < numtasks) {
103                     MPI_Send(sorted, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD);
104                     MPI_Recv(temp, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD,
105                             MPI_STATUS_IGNORE);
106                     int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, false);
107                     free(sorted);
108                     sorted = temp_new_sorted;
109                 }
110             } else if (taskid - 1 >= 0) {
111                 MPI_Recv(temp, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD,
112                         MPI_STATUS_IGNORE);
113                 MPI_Send(sorted, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD);
114                 int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, true);
115                 free(sorted);
116                 sorted = temp_new_sorted;
117             }
118         } else {
119             if (taskid % 2 == 0) {
120                 if (taskid - 1 >= 0) {
121                     MPI_Recv(temp, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD,
122                             MPI_STATUS_IGNORE);

```

```

120     MPI_Send(sorted, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD);
121     int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, true)
122 ;
123     free(sorted);
124     sorted = temp_new_sorted;
125 }
126 } else if (taskid + 1 < numtasks) {
127     MPI_Send(sorted, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD);
128     MPI_Recv(temp, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD,
129 MPI_STATUS_IGNORE);
130     int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, false);
131     free(sorted);
132     sorted = temp_new_sorted;
133 }
134 }
135 free(temp);
136 return sorted;
137 }
138
139 // Keep n/2 max or min values of both arrays in sorted
140 int *comparesplit_merge(int *sorted, int *arr, int n, bool max)
141 {
142     int s_i, a_i;
143     int *temp = (int *)malloc(n * sizeof(int));
144     if (!max) {
145         s_i = a_i = 0;
146         while (s_i + a_i < n) {
147             // No need to check if s_i or a_i is out of bounds
148             if (sorted[s_i] < arr[a_i]) {
149                 temp[s_i + a_i] = sorted[s_i];
150                 s_i++;
151             } else {
152                 temp[s_i + a_i] = arr[a_i];
153                 a_i++;
154             }
155         }
156     } else {
157         s_i = a_i = n - 1;
158         while (s_i + a_i > n - 2) {
159             // No need to check if s_i or a_i is out of bounds
160             if (sorted[s_i] > arr[a_i]) {
161                 temp[s_i + a_i - (n - 2) - 1] = sorted[s_i];
162                 s_i--;
163             } else {
164                 temp[s_i + a_i - (n - 2) - 1] = arr[a_i];
165                 a_i--;
166             }
167         }
168     }
169     return temp;
170 }
171
172 void oddevenser(int *unsorted, int n, int *sorted)
173 {
174     int phase, i;
175     int changes = 0;
176     bool prev_changes = true;
177
178     memcpy(sorted, unsorted, n * sizeof(int));
179
180     for (phase = 0; phase < n; phase++) {

```

```
181     changes = oddevenser_phase(sorted, n, phase);
182
183     // Check for sorted array in early phase
184     if (changes == 0) {
185         if (prev_changes == false) {
186             break;
187         } else {
188             prev_changes = false;
189         }
190     } else {
191         prev_changes = true;
192     }
193 }
194 }
195
196 int oddevenser_phase(int *array, int n, int phase)
197 {
198     int changes = 0;
199     for (int i = phase % 2; i + 1 < n; i += 2) {
200         changes += (int)compare_exchange(&array[i], &array[i + 1]);
201     }
202     return changes;
203 }
204
205 int compare_exchange(int *a, int *b)
206 {
207     if (*a > *b) {
208         int t = *a;
209         *a = *b;
210         *b = t;
211         return 1;
212     }
213     return 0;
214 }
215
216 bool is_sorted(int *arr, int n)
217 {
218     for (int i = 0; i + 1 < n; i++) {
219         if (arr[i] > arr[i + 1]) {
220             return false;
221         }
222     }
223     return true;
224 }
```

Listing 1.4: Code of Exercise 4.