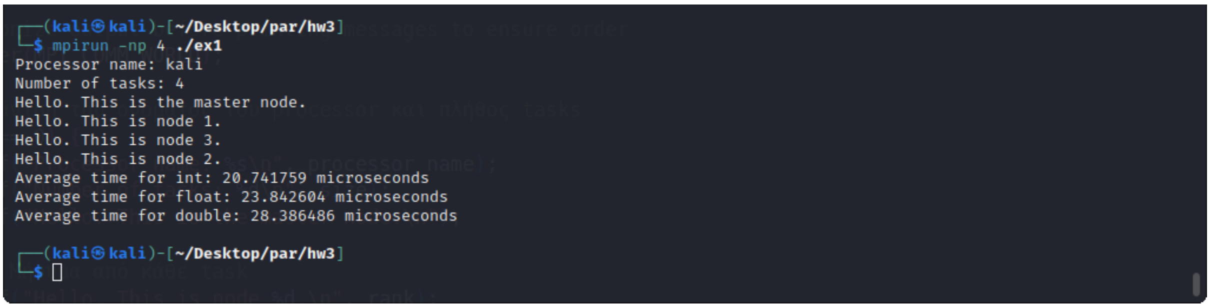
ECE445: Parallel and Distributed Computing
Winter Semester 2024-2025
3rd set of Exercises

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:



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

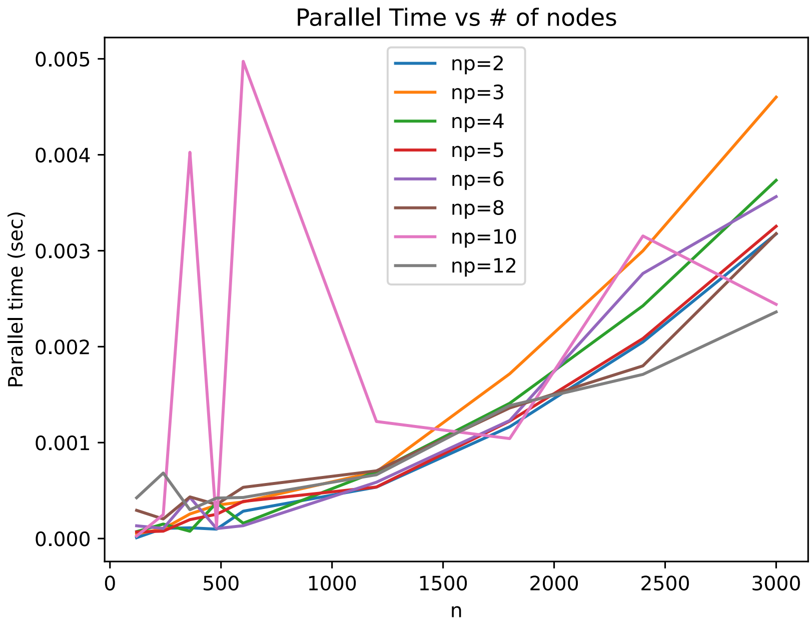


Fig. 1: Parallel times in relation to num of tasks.

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 inversly 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

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_ITERATIONS 1000

int main(int argc, char** argv) {
    int rank, size;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    double avint, avfloat, avdoubl;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &name_len);

    // Synchronize output before printing messages to ensure order
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Processor name: %s\n", processor_name);
        printf("Number of tasks: %d\n", size);
        printf("Hello. This is the master node.\n");
    }else{

        printf("Hello. This is node %d.\n", rank);
    }


    //count p2p communication time
    if (rank < 2) { // Only rank 0 and rank 1 will participate
        MPI_Barrier(MPI_COMM_WORLD); // Synchronize communication

        // Timing for int, float, double
        double start_time, end_time;

        if (rank == 0) {
            int int_data = 42;
            float float_data = 42.42;
            double double_data = 42.4242;

            // Measure time for int
            start_time = MPI_Wtime();
            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&int_data, 1, MPI_INT, 1, 0, &int_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            end_time = MPI_Wtime();
            avint= (end_time - start_time) / NUM_ITERATIONS * 1e6;

            // Measure time for float
            start_time = MPI_Wtime();
            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&float_data, 1, MPI_FLOAT, 1, 0, &float_data, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            end_time = MPI_Wtime();
            avfloat = (end_time - start_time) / NUM_ITERATIONS * 1e6;

```

```c
            // Measure time for double
            start_time = MPI_Wtime();
            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&double_data, 1, MPI_DOUBLE, 1, 0, &double_data, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            end_time = MPI_Wtime();
            avdoubl = (end_time - start_time) / NUM_ITERATIONS * 1e6;

        } else if (rank == 1) {
            int int_data;
            float float_data;
            double double_data;

            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&int_data, 1, MPI_INT, 0, 0, &int_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&float_data, 1, MPI_FLOAT, 0, 0, &float_data, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            for (int i = 0; i < NUM_ITERATIONS; i++) {
                MPI_Sendrecv(&double_data, 1, MPI_DOUBLE, 0, 0, &double_data, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
    } else {
        // Rank 2 and rank 3 don't do any communication, so they just wait to finalize
        MPI_Barrier(MPI_COMM_WORLD);  // Synchronize before exiting
    }

    if (rank==0){
        printf("Average time for int: %lf microseconds\n",avint);
        printf("Average time for float: %lf microseconds\n",avfloat);
        printf("Average time for double: %lf microseconds\n", avdoubl);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 1.1: Code of Exercise 1.

B Code of exercise 2

```c

#include <stdio.h>
#include <stdlib.h>

#include "mpi.h"

#define  MASTER 0
#define SAMPLES 1000000

int main (int argc, char *argv[])
{
  int   numtasks, taskid, len, meaning_of_life = 0;
  char hostname[MPI_MAX_PROCESSOR_NAME];
  double tstart, tend;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  MPI_Get_processor_name(hostname, &len);

  // A
  if (taskid == MASTER) {
    printf("Processor name: %s. Number of MPI tasks: %d\n", hostname, numtasks);
  }

  // B
  if (taskid == MASTER) {
    printf("Hello, this is the Master node.\n");
  } else {
    printf("Hello this is task %d\n", taskid);
  }
  MPI_Barrier(MPI_COMM_WORLD);

  // C
  if (taskid == MASTER) {
    meaning_of_life = 42;
    printf("MASTER: Broadcasting meaning of life to all nodes %d times using Bcast.\n", SAMPLES);
    tstart = MPI_Wtime();
  }
  for (int i = 0; i < SAMPLES; i++) {
    MPI_Bcast(&meaning_of_life, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
  }

  // E
  if (taskid == MASTER) {
    tend = MPI_Wtime();
    printf("MASTER: Took %lf seconds or %e seconds per time.\n", SAMPLES, tend - tstart, (tend - tstart) / SAMPLES);
  }
  #ifdef DEBUG
  else {
    printf("Task %d received the meaning of life: %d\n", taskid, meaning_of_life);
  }
  #endif
  MPI_Barrier(MPI_COMM_WORLD);

  // D
  if (taskid == MASTER) {
```

```
    printf("MASTER: Broadcasting meaning of life to all nodes %d times using Send/Recv.\n", SAMPLES);
    tstart = MPI_Wtime();
  }
  for (int i = 0; i < SAMPLES; i++) {
    custom_bcast(&meaning_of_life, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
  }

  // E
  if (taskid == MASTER) {
    tend = MPI_Wtime();
    printf("MASTER: Took %lf seconds or %e seconds per time.\n", SAMPLES, tend - tstart, (tend - tstart) / SAMPLES);
  }
  #ifdef DEBUG
  else {
    printf("Task %d received the meaning of life: %d\n", taskid, meaning_of_life);
  }
  #endif
  MPI_Finalize();
}

void custom_bcast(void *data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator) {
  int numtasks, taskid;
  MPI_Comm_size(communicator, &numtasks);
  MPI_Comm_rank(communicator, &taskid);
  MPI_Status status;

  if (taskid == root) {
    for (int i = 0; i < numtasks; i++) {
      if (i != root) {
        MPI_Send(data, count, datatype, i, 1, communicator);
      }
    }
  } else {
    MPI_Recv(data, count, datatype, root, 1, communicator, &status);
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Listing 1.2: Code of Exercise 2.

C Code of exercise 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// Function to perform matrix-vector multiplication
void rowMVMult(int n, double *localA, double *localb, double *localy, MPI_Comm
    comm) {
  int i, j, rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  int local_n = n / size;

  // Initialize localy to 0
  for (i = 0; i < local_n; i++) {
    localy[i] = 0.0;
  }

  // Perform local matrix-vector multiplication
  for (i = 0; i < local_n; i++) {
    for (j = 0; j < n; j++) {
      localy[i] += localA[i * n + j] * localb[j];
    }
  }
}

int main(int argc, char *argv[]) {
  int i, j, n, rank, size;
  double *A, *b, *y, *localA, *localb, *localy;
  double start_time, end_time;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Get matrix dimension from user (assume n is divisible by size)
  if (rank == 0) {
    scanf("%d", &n);
  }
  // Broadcast n to all processes
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Calculate local block size
  int local_n = n / size;


  // Allocate memory for local arrays
  localA = (double *)malloc(local_n * n * sizeof(double));
  localb = (double *)malloc(n * sizeof(double));
  localy = (double *)malloc(local_n * sizeof(double));

  // Master process initializes A and b
  if (rank == 0) {
    A = (double *)malloc(n * n * sizeof(double));
    y = (double *)malloc(n * sizeof(double));

    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        A[i * n + j] = (i + j) * 10.0;
      }
      localb[i] = 10.0;
```

```
    }

    // Scatter rows of A to all processes
    MPI_Scatter(A, local_n * n, MPI_DOUBLE, localA, local_n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Broadcast b to all processes
    MPI_Bcast(localb, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    start_time = MPI_Wtime();
  } else {
    // Receive rows of A from master
    MPI_Scatter(NULL, local_n * n, MPI_DOUBLE, localA, local_n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // printf("test %d\n",rank);
    // Receive b from master (broadcast)
    MPI_Bcast(localb, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }

  // Barrier synchronization
  MPI_Barrier(MPI_COMM_WORLD);
  // Perform local matrix-vector multiplication
  rowMVMult(n, localA, localb, localy, MPI_COMM_WORLD);

  if (rank == 0) {
    // Master process receives the gathered data
        MPI_Gather(localy, local_n, MPI_DOUBLE, y, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    } else {
        // Other processes only send data
        MPI_Gather(localy, local_n, MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

  // Master process prints results and execution time
  if (rank == 0) {
    end_time = MPI_Wtime();

    printf("Resulting vector y:\n");
    for (i = 0; i < n; i++) {
      printf("%f ", y[i]);
    }
    printf("\n");

    printf("Execution time: %f seconds\n", end_time - start_time);
    // printf("%lf", end_time-start_time);
    free(A);
    //free(b);
    free(y);
  }

  // Free local memory
  free(localA);
  free(localb);
  free(localy);

  MPI_Finalize();
  return 0;
}
```

Listing 1.3: Code of Exercise 3.

D Code of exercise 4

```c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <assert.h>

#include "mpi.h"

#define  MASTER 0
#define SAMPLES 1000000

bool is_sorted(int *arr, int n);
int compare_exchange(int *a, int *b);
int *comparesplit_merge(int *sorted, int *arr, int n, bool max);
int oddevenser_phase(int *array, int n, int phase);
int *oddevenpar(int taskid, int numtasks, int task_n, int *sorted);

int main (int argc, char *argv[])
{
  int   numtasks, taskid, len, task_n, n;
  char hostname[MPI_MAX_PROCESSOR_NAME];
  double tstart, tend;

  n = atoi(argv[1]);
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  MPI_Get_processor_name(hostname, &len);

  if (taskid == MASTER) {
    printf("Processor name: %s. Number of MPI tasks: %d\n",
                        hostname, numtasks);
  }

  task_n = n / numtasks;
  // Initialize an array of size n with random numbers
  int *unsorted = (int *)malloc(task_n * sizeof(int));
  int *sorted = (int *)malloc(task_n * sizeof(int));
  srand(time(NULL) + taskid);
  for (int i = 0; i < task_n; i++) {
    unsorted[i] = rand() % 10000;
  }

  #ifdef DEBUG
  // Print the unsorted array
  printf("Task %d - Unsorted array:\n", taskid);
  for (int i = 0; i < task_n; i++) {
    printf("%4d ", unsorted[i]);
  }
  printf("\n");
  #endif

  if (taskid == MASTER) {
    tstart = MPI_Wtime();
  }

  // Sort the local array
  oddevenser(unsorted, task_n, sorted);
  assert(is_sorted(sorted, task_n));

```

```c
  // Compare-split the sorted arrays
  sorted = oddevenpar(taskid, numtasks, task_n, sorted);

  MPI_Barrier(MPI_COMM_WORLD);
  if (taskid == MASTER) {
    tend = MPI_Wtime();
    printf("Took %lf seconds\n", tend - tstart);
  }

  int *global_sorted = NULL;
  if (taskid == MASTER) {
    global_sorted = (int *)malloc(n * sizeof(int));
  }
  MPI_Gather(sorted, task_n, MPI_INT, global_sorted, task_n, MPI_INT, MASTER, MPI_COMM_WORLD);

  if (taskid == MASTER) {
    assert(is_sorted(global_sorted, n));
    // Print the sorted array
    printf("Sorted array correctly\n");
    #ifdef DEBUG
    for (int i = 0; i < n; i++) {
      printf("%4d ", global_sorted[i]);
    }
    printf("\n");
    #endif
    free(global_sorted);
  }

  MPI_Finalize();
}

// Odd-even parallel sort
int *oddevenpar(int taskid, int numtasks, int task_n, int *sorted)
{
  int *temp = (int*)malloc(task_n * sizeof(int));

  for (int i = 0; i < numtasks; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (i % 2 == 0) { // even
      if (taskid % 2 == 0) {
        if (taskid + 1 < numtasks) {
          MPI_Send(sorted, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD);
          MPI_Recv(temp, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, false);
          free(sorted);
          sorted = temp_new_sorted;
        }
      } else if (taskid - 1 >= 0) {
        MPI_Recv(temp, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(sorted, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD);
        int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, true);
        free(sorted);
        sorted = temp_new_sorted;
      }
    } else {
      if (taskid % 2 == 0) {
        if (taskid - 1 >= 0) {\
          MPI_Recv(temp, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
          MPI_Send(sorted, task_n, MPI_INT, taskid - 1, 0, MPI_COMM_WORLD);
          int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, true)
  ;
          free(sorted);
          sorted = temp_new_sorted;
        }
      } else if (taskid + 1 < numtasks) {
        MPI_Send(sorted, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD);
        MPI_Recv(temp, task_n, MPI_INT, taskid + 1, 0, MPI_COMM_WORLD,
  MPI_STATUS_IGNORE);
        int *temp_new_sorted = comparesplit_merge(sorted, temp, task_n, false);
        free(sorted);
        sorted = temp_new_sorted;
      }
    }
  }
  free(temp);
  return sorted;
}


// Keep n/2 max or min values of both arrays in sorted
int *comparesplit_merge(int *sorted, int *arr, int n, bool max)
{
  int s_i, a_i;
  int *temp = (int *)malloc(n * sizeof(int));
  if (!max) {
    s_i = a_i = 0;
    while (s_i + a_i < n) {
      // No need to check if s_i or a_i is out of bounds
      if (sorted[s_i] < arr[a_i]) {
        temp[s_i + a_i] = sorted[s_i];
        s_i++;
      } else {
        temp[s_i + a_i] = arr[a_i];
        a_i++;
      }
    }
  } else {
    s_i = a_i = n - 1;
    while (s_i + a_i > n - 2) {
      // No need to check if s_i or a_i is out of bounds
      if (sorted[s_i] > arr[a_i]) {
        temp[s_i + a_i - (n - 2) - 1] = sorted[s_i];
        s_i--;
      } else {
        temp[s_i + a_i - (n - 2) - 1] = arr[a_i];
        a_i--;
      }
    }
  }
  return temp;
}

void oddevenser(int *unsorted, int n, int *sorted)
{
  int phase, i;
  int changes = 0;
  bool prev_changes = true;

  memcpy(sorted, unsorted, n * sizeof(int));

  for (phase = 0; phase < n; phase++) {
```

```c
    changes = oddevenser_phase(sorted, n, phase);

    // Check for sorted array in early phase
    if (changes == 0) {
      if (prev_changes == false) {
        break;
      } else {
        prev_changes = false;
      }
    } else {
      prev_changes = true;
    }
  }
}

int oddevenser_phase(int *array, int n, int phase)
{
  int changes = 0;
  for (int i = phase % 2; i + 1 < n; i += 2) {
    changes += (int)compare_exchange(&array[i], &array[i + 1]);
  }
  return changes;
}

int compare_exchange(int *a, int *b)
{
  if (*a > *b) {
    int t = *a;
    *a = *b;
    *b = t;
    return 1;
  }
  return 0;
}

bool is_sorted(int *arr, int n)
{
  for (int i = 0; i + 1 < n; i++) {
    if (arr[i] > arr[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Listing 1.4: Code of Exercise 4.