МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Параллельные алгоритмы»
Тема: Использование функций обмена данными «Точка-Точка» в библиотеке MPI

Студент гр. 3381 _____ Иванов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2025

Цель работы.

Написать параллельную программу на языке C с использованием функций обмена данными «Точка-точка» из OpenMPI.

Задание.

Вариант 4. Процесс 0 генерирует массив и раздает его другим процессам для поиска максимума. Собрав локальные максимумы, ищет общий.

Выполнение работы.

В едином файле описана логика для главного процесса с рангом 0 и для всех остальных.

Логика главного процесса:

1. Для удобства определяются переменные для хранения количества рабочих процессов в общем и реальное их количество (если некоторые из них будут обрезаны);
2. Проверка количества рабочих процессов: если их количество равно 0, то выводится ошибка и происходит переход на метку finalize_redudant (описана в пункте n);
3. Считывание размера массива: программа ожидает получить на вход из командной строки число — размер массива, который необходимо сгенерировать. Здесь проверяются различные условия: наличие аргумента, его неотрицательность. В случае отсутствия аргумента выводится строка, которая подсказывает каким должен быть ввод;
4. Сравнение размера массива с 0: если размер массива равен 0, то программа переходит на метку finalize_redudant и завершает свою работу;
5. Формирование массива случайных чисел: под массив выделяется память и генерируется случайный массив. Если на этом шаге происходит ошибка выделения памяти то выводится ошибка и происходит переход на метку finalize_redudant;

6. Вычисление количества элементов массива, которые должны быть отправлены рабочему процессу: вычисляется оценка, которая не может быть менее 1. Если по оценке рабочему процессу предназначается менее 1 элемента массива, то это число жёстко задаётся на 1;
7. Проверка соотношения размера массива к количеству рабочих процессов: если размер массива меньше чем общее количество рабочих процессов, то лишние процессы «обрезаются» (им будет отправлен нулевой размер массива);
8. Вычисление количества элементов массива, которое предназначено рабочему процессу с рангом 1: количество элементов массива не всегда кратно количеству рабочих процессов, за счёт чего некоторым процессам достанется больше элементов массива чем другим. В данной программе все такие элементы отправляются первому процессу;
9. Отправка размера массива и самого массива процессу с рангом 1;
10. Отправка размера массива и самого массива остальным процессам;
11. Ожидание получения максимумов от каждого из рабочих процессов: после получения максимума от процесса он сравнивается с текущим найденным и значение максимума обновляется если условие выполнено;
12. Отправка нулевого размера всем лишним рабочим процессам: если в процессе выполнения программы некоторые рабочие процессы оказались лишними. Этот блок кода находится после метки finalize_redudant и на эту метку всегда переходит выполнение когда необходимо досрочно завершить выполнение.

Логика рабочего процесса:

1. Получение размера подмассива;
2. Получение элементов подмассива;
3. Вычисление максимума подмассива с помощью последовательного алгоритма;
4. Отправка массива обратно главному процессу.

График зависимости затраченного времени от количества процессов см. на рисунке 1. Каждый замер произведён 100 раз.

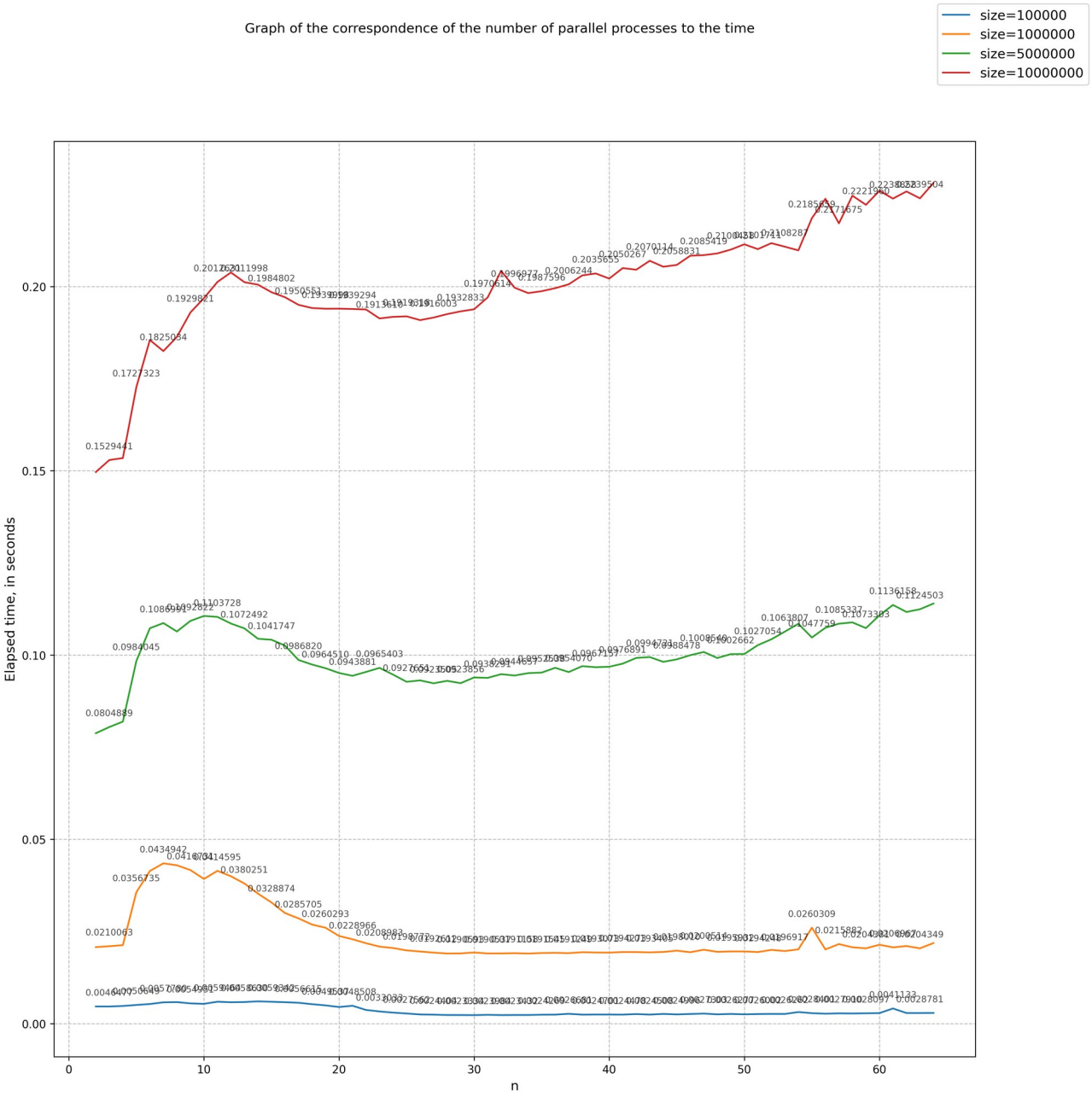

Рисунок 1 — Зависимость затраченного времени от количества процессов при $n \in \{2, 3, 4, ..., 64\}$

График ускорения см. на рисунке 2. Данный график построен на основе значений, полученных при построении графика на рис 1.

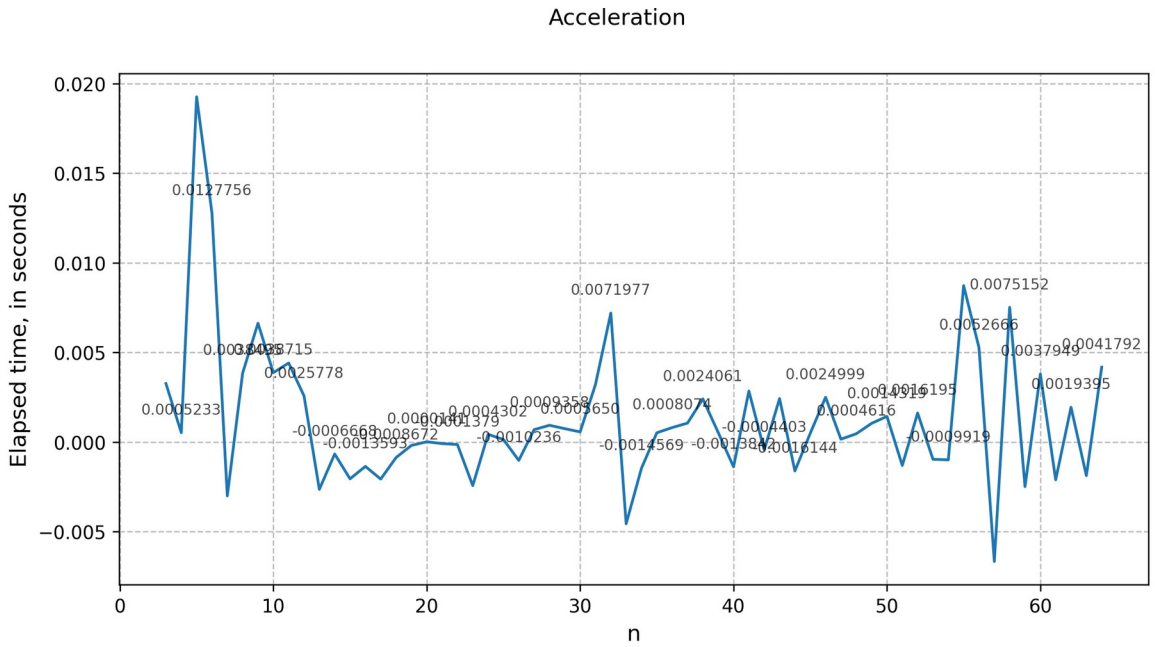

Рисунок 2 — Зависимость ускорения от количества процессов для массива размером 10.000.000

По графику на рис. 1 видно, что при $n \in \{8, 12\}$ наступает насыщение, после которого время падает и далее более менее стабилизируется. Сложно сказать почему графики именно такие. Графики при размере массива 100.000 и 1.000.000 после $n \approx 20$ становятся почти константными. Можно предположить, что время на вычисления и коммуникацию в данном случае настолько ничтожно что рост графика на данной области определения ничтожен. С другой стороны, на графиках для размера массива 5.000.000 и 10.000.000 время увеличивается более ощутимо. Скорее всего это связано с большим размером массива, за счёт чего время на коммуникацию и вычисление размера массива становится заметнее.

Сеть Петри для реализованного алгоритма приведена в приложении А.

Пример работы написанной программы приведён в приложении В.

Листинг программ приведён в приложении С.

Выводы.

В ходе лабораторной работы были изучены принципы взаимодействия «точка-точка» в OpenMPI. По результатам работы написана программа на языке программирования C, которая считает максимум массива параллельно. Для этого внутри программы происходит распределение нагрузки по расчётам между всеми процессами кроме главного. Гланый по ходу получения результатов вычислений подсчитывает итоговый максимум массива.

ПРИЛОЖЕНИЕ А
СЕТЬ ПЕТРИ

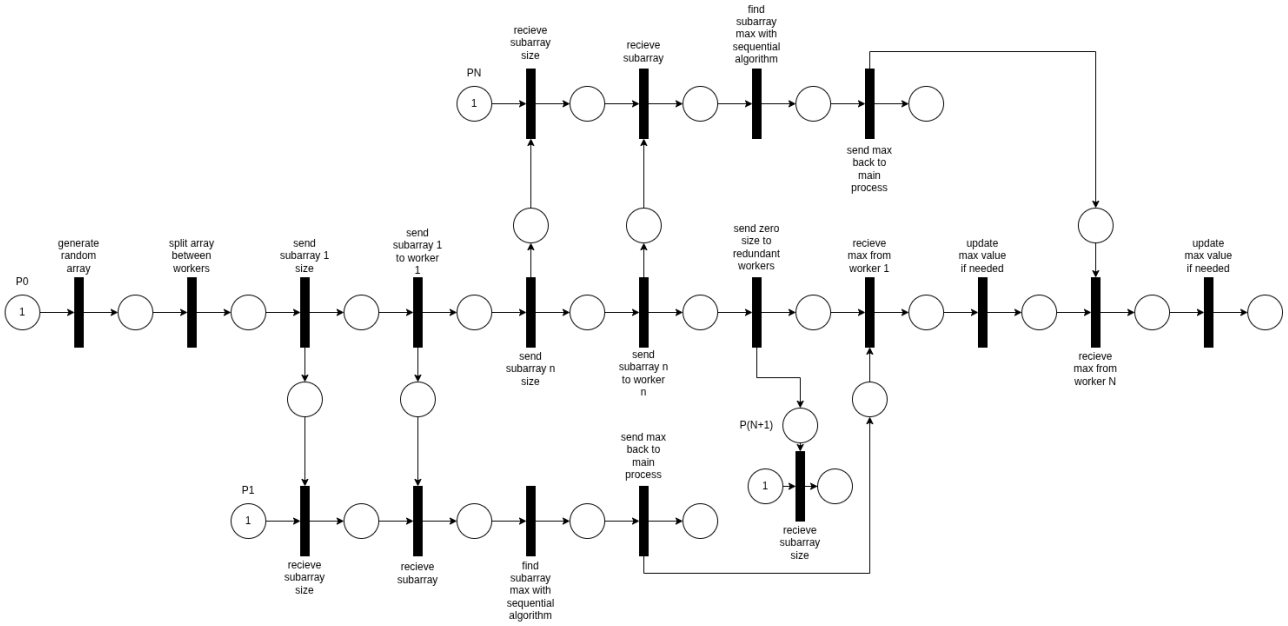

Рисунок А.1 — Сеть Петри

ПРИЛОЖЕНИЕ В
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

При n=8 и size=10.000 без определения директив.

```
 → mpicc -o exe timer.c common.c lab2/src/task.c
 → mpirun -n 8 ./exe 10000
999
```

При n=8 и size=15 с определённой директивой DEBUG.

```
 → mpicc -DDEBUG -o exe timer.c common.c lab2/src/task.c
 → mpirun -n 8 ./exe 15
Recieved array size: 15
Array elements: 211 192 465 103 613 271 509 808 884 7 752 88 549 208 124
Estimation: 2.142857
Array elements per process: 2
Array elements per process with overhead: 3
3 array elements were sent to worker 1: 211 192 465
2 array elements were sent to worker 2: 103 613
2 array elements were sent to worker 3: 271 509
2 array elements were sent to worker 4: 808 884
Recieved in 2 subarray size is 2
Recieved in 3 subarray size is 2
Recieved in 4 subarray size is 2
Recieved in 5 subarray size is 2
2 array elements were sent to worker 5: 7 752
2 array elements were sent to worker 6: 88 549
2 array elements were sent to worker 7: 208 124
Recieved in 6 subarray size is 2
Recieved in 7 subarray size is 2
Recieved in 1 subarray size is 3
Max value equal to 465 was received from process with rank 1
Max value equal to 613 was received from process with rank 2
Max value equal to 509 was received from process with rank 3
Max value equal to 884 was received from process with rank 4
Max value equal to 752 was received from process with rank 5
Max value equal to 549 was received from process with rank 6
Max value equal to 208 was received from process with rank 7
884
```

При MEASURE_TIME и DEBUG (когда включена директива MEASURE_TIME DEBUG становится неактивна).

```
 →  mpicc -DDEBUG -DMEASURE_TIME -o exe timer.c common.c lab2/src/task.c →
mpirun -n 8 ./exe 10000
0.000627
```

Построения графиков работы программы и ускорения.

```
 → python3 ./lab2/src/make_chart.py -p 1 --argv "./exe"
Start collecting time for array with size=100000
  1) n=2
  1) n=8
  1) n=14
```

```
  1) n=20
  1) n=26
  1) n=32
  1) n=38
  1) n=44
  1) n=50
Start collecting time for array with size=1000000
  1) n=2
  1) n=8
  1) n=14
  1) n=20
  1) n=26
  1) n=32
  1) n=38
  1) n=44
  1) n=50
Start collecting time for array with size=10000000
  1) n=2
  1) n=8
  1) n=14
  1) n=20
  1) n=26
  1) n=32
  1) n=38
  1) n=44
  1) n=50
```

ПРИЛОЖЕНИЕ С
ЛИСТИНГ

Реализация задачи.

Название файла: lab2/src/task.c

```
#include <mpi.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

#include "../../task.h"
#include "../../common.h"

#define MAX(i, j) (((i) > (j)) ? (i) : (j))

#ifdef MEASURE_TIME
#undef DEBUG
#endif

void usage(const char* exe)
{
    printf("Usage: %s <array_size>\n", exe);
}

// This task doesn't use root_rank value
int task(int world_size, int world_rank, int argc, char* argv[])
{
    int retval = 0;

    if (world_rank == 0) {
        int current_worker = 1;

        int workers_count = world_size - 1;
        int real_workers_count = workers_count;

        if (workers_count == 0) {
            retval = errno;
            perror("There are too few workers");
            goto finalize_redudant;
        }

        // Read array size from args list
        size_t array_size = 0;
        if (argc == 2) {
            if (argv[1] != NULL && sscanf(argv[1], "%lu", &array_size) == 1) {
                if (array_size < 0) {
                    retval = EINVAL;
                    perror("Array size is negative");
                    goto finalize_redudant;
                }
            } else {
                retval = EINVAL;
                perror("Invalid string argument (must contain array size)");
                goto finalize_redudant;
            }
        } else {
            retval = EINVAL;
            usage(argv[0]);
```

```
            goto finalize_redudant;
        }

        if (array_size == 0) {
            goto finalize_redudant;
        }

        #ifdef DEBUG
        printf("Recieved array size: %u\n", array_size);
        #endif

        // Generate array
        int* array = (int*)malloc(array_size * sizeof(int));
        if (!array) {
            retval = errno;
            perror("Can't allocate memory for array");
            goto finalize_redudant;
        }
        generate_array(array, array_size, time(NULL) + world_rank);

        #ifdef DEBUG
        printf("Array elements: ");
        for (size_t i = 0; i < array_size; ++i) {
            printf("%d ", array[i]);
        }
        putchar('\n');
        #endif

        double estimation = (double)array_size / workers_count;
        #ifdef DEBUG
        printf("Estimation: %lf\n", estimation);
        #endif

        /* It is not make sense no send less than 2 numbers to other processes.
           But it is good question what will be better: to send this two numbers
           to other process or calculate it by yourself... */
        // Calculate number of array elements for each process
        const size_t min_num_per_process = 1;
        size_t num_per_process = min_num_per_process;
        if ((unsigned)estimation > min_num_per_process) {
            num_per_process = (unsigned)estimation;
        }

        #ifdef DEBUG
        printf("Array elements per process: %u\n", num_per_process);
        #endif

        int* array_ptr = array;
        int max = array[0];

        // Array size is less than workers count => decrease workers count
        if (array_size < workers_count) {
            workers_count = array_size / 2 + 1;
            #ifdef DEBUG
            printf("Workers count decreased to %d\n", workers_count);
            #endif
        }

             size_t  overhead_num_per_process  =  num_per_process  +  array_size  %
workers_count;

        #ifdef DEBUG
                 printf("Array  elements  per  process  with  overhead:  %d\n",
```

```
overhead_num_per_process);
        #endif

        // Process with rank 1 will sort array part with overhead
        MPI_Send(&overhead_num_per_process, 1, MPI_UNSIGNED_LONG_LONG,
                 current_worker, 0, MPI_COMM_WORLD);
        MPI_Send(array_ptr, overhead_num_per_process, MPI_INT,
                 current_worker, 0, MPI_COMM_WORLD);

        #ifdef DEBUG
                  printf("%d  array  elements  were  sent  to  worker  %d:  ",
overhead_num_per_process, current_worker);
        for (int* tmp = array_ptr; tmp < array_ptr + overhead_num_per_process; +
+tmp) {
            printf("%d ", *tmp);
        }
        putchar('\n');
        #endif

        array_ptr = array_ptr + overhead_num_per_process;
        ++current_worker;

        for (; current_worker < workers_count + 1; ++current_worker) {
            MPI_Send(&num_per_process, 1, MPI_UNSIGNED_LONG_LONG,
                     current_worker, 0, MPI_COMM_WORLD);
            MPI_Send(array_ptr, num_per_process, MPI_INT,
                     current_worker, 0, MPI_COMM_WORLD);

            #ifdef DEBUG
                      printf("%d  array  elements  were  sent  to  worker  %d:  ",
num_per_process, current_worker);
            for (int* tmp = array_ptr; tmp < array_ptr + num_per_process; ++tmp)
{
                printf("%d ", *tmp);
            }
            putchar('\n');
            #endif

            array_ptr = array_ptr + num_per_process;
        }

        // Merge computation results
        int recieved_max = 0;
        for (size_t i = 1; i < workers_count + 1; ++i) {
                    MPI_Recv(&recieved_max,  1,  MPI_INT,  i,  0,  MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            max = MAX(max, recieved_max);

            #ifdef DEBUG
              printf("Max value equal to %d was received from process with rank
%ld\n", recieved_max, i);
            #endif
        }
        free(array);

        #ifndef MEASURE_TIME
        printf("%d\n", max);
        #endif


    // Send zero size for redudant workers
    finalize_redudant:
        // Send size 0 to redudant workers
```

```
        size_t zero_buf = 0;
        for (int i = current_worker; i < real_workers_count + 1; ++i) {
            MPI_Send(&zero_buf, 1, MPI_UNSIGNED_LONG, i, 0, MPI_COMM_WORLD);

            #ifdef DEBUG
            printf("Size equal to 0 was sent to process with rank %d\n", i);
            #endif
        }
    } else {
        // Get array size from main process
        size_t subarray_size = 0;
        MPI_Recv(&subarray_size, 1, MPI_UNSIGNED_LONG_LONG, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        #ifdef DEBUG
          fprintf(stderr, "Recieved in %d subarray size is %ld\n", world_rank,
subarray_size);
        #endif
        if (subarray_size != 0) {
            // Get array from main process
            int* subarray = (int*)malloc(subarray_size * sizeof(int));
            if (!subarray) {
                retval = errno;
                perror("Can't allocate memory for subarray");
                MPI_Abort(MPI_COMM_WORLD, retval);
            }
            MPI_Recv(subarray, subarray_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);

            // Find max of subarray
            int max = subarray[0];
            for (size_t i = 1; i < subarray_size; ++i) {
                max = MAX(max, subarray[i]);
            }

            // Send subarray max to main process
            MPI_Send(&max, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

            free(subarray);
        }
    }

    return retval;
}
```

Построение графиков.

Название файла: chart_tools.py

```
import argparse
import subprocess

import matplotlib.pyplot as plt

from functools import partial


def collect_time_stats(
    argv: str,
    avg_param: int,
    n_values: list = [1, 2, 4, 8, 16, 32, 64]) -> dict:

    time_stats = dict()
```

```
    for n in n_values:
        time_values = []
        for i in range(avg_param):
            print(f"{i+1:3}) n={n}")

            result = subprocess.run(
                f"mpirun --oversubscribe -n {n} {argv}".split(),
                encoding="utf-8",
                capture_output=True
            )
            if result.returncode != 0:
                    print(f"Error in subprocess run:\n\tstdout: {result.stdout}
\n\tstderr: {result.stderr}")
                exit(1)

            time_values.append(float(result.stdout))

        if len(time_values) == 0:
            print("Empty time values list, exiting...")
            exit(1)
        time_stats[n] = sum(time_values) / len(time_values)

    return (
        list(time_stats.keys()),
        list(time_stats.values())
    )

def setup_axes(
    fig,
    ax,
    suptitle="Graph of the correspondence of the number of parallel processes to
the time"
) -> None:
    fig.suptitle(suptitle)

    ax.grid(True, linestyle='--', alpha=0.8)

    # Setting up x axis
    ax.set_xlabel('n', fontsize=12)
    # Setting up y axis
    ax.set_ylabel('Elapsed time, in seconds', fontsize=12)

def plot_chart(ax, X: list, Y: list, label: str = None) -> None:
    ax.plot(X, Y, label=label)
    # Add this after the ax.plot(X, Y) line
    for i, (x, y) in enumerate(zip(X, Y)):
        if (i % 2 == 0):
            continue
        ax.annotate(f'{y:.7f}',
                    (x, y),
                    textcoords="offset points",
                    xytext=(0,10),
                    ha='center',
                    fontsize=8,
                    alpha=0.7)

def int_limited(arg: str, lower: int = None, upper: int = None):
    try:
        limited_int_arg = int(arg)
        if lower is not None and limited_int_arg < lower:
              raise argparse.ArgumentTypeError(f"Int value must be greater than
{lower}")
```

```
        if upper is not None and limited_int_arg > upper:
                raise argparse.ArgumentTypeError(f"Int value must be lower than
{upper}")

        return limited_int_arg
    except ValueError:
        raise argparse.ArgumentTypeError(f"Invalid int value: {arg}")

int_is_positive = partial(int_limited, lower=1)

def parse_cli():
    parser = argparse.ArgumentParser(description="Plot chart for MPI program")
    parser.add_argument(
        "-p",
        "--averaging-parameter",
        type=int_is_positive,
        required=True,
        help="The number of values for which the averaging is performed"
    )
    parser.add_argument(
        "-a",
        "--argv",
        type=str,
        required=True,
        help="argv of program to run with mpirun"
    )
    parser.add_argument(
        "-o",
        "--output",
        type=str,
        default="chart.png",
        help="Chart image filename"
    )

    return parser.parse_args()


if __name__ == "__main__":
    args = parse_cli()

    X, Y = collect_time_stats(args.executable, args.averaging_parameter)

    fig, ax = plt.subplots(1, figsize=(15, 15))

    setup_axes(fig, ax)
    plot_chart(ax, X, Y)

    fig.savefig(args.output, dpi=300, bbox_inches="tight", facecolor="white")
```

Название файла: lab2/src/make_chart.py

```
import os
import sys
sys.path.append(os.path.join(os.path.dirname(__file__), '../..'))

import chart_tools
import matplotlib.pyplot as plt


if __name__ == "__main__":
    args = chart_tools.parse_cli()
```

```
    fig, ax = plt.subplots(1, figsize=(15, 15))
    chart_tools.setup_axes(fig, ax)

    X, Y = [], []
    for size in [100_000, 1_000_000, 5_000_000, 10_000_000]:
        print(f"Start collecting time for array with size={size}")
            # n_values = [2, 8, 14, 20, 26, 32, 38, 44, 50, 56, 62]
        X, Y = chart_tools.collect_time_stats(
            f"{args.argv} {size}",
            args.averaging_parameter,
            n_values = list(range(2, 64 + 1))
        )
        chart_tools.plot_chart(ax, X, Y, label=f"size={size}")

    # Use last coordinates to plot acceleration chart
    X_ac, Y_ac = X[1:], []
    for i in range(1, len(X)):
        Y_ac.append(Y[i] - Y[i-1])
    fig_ac, ax_ac = plt.subplots(1, figsize=(10, 5))
    chart_tools.setup_axes(fig_ac, ax_ac, "Acceleration")
    chart_tools.plot_chart(ax_ac, X_ac, Y_ac)
    fig_ac.legend(fontsize=12)
            fig_ac.savefig("acceleration.png",   dpi=300,   bbox_inches="tight",
facecolor="white")

    fig.legend(fontsize=12)
    fig.savefig(args.output, dpi=300, bbox_inches="tight", facecolor="white")
```

Общая часть.

Название файла: common.h

```
#ifndef COMMON_H
#define COMMON_H

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

void generate_array(int* buf, size_t bufsize, unsigned int seed);
void print_array(int* buf, size_t bufsize);

#endif
```

Название файла: common.c

```
#include "common.h"

void generate_array(int* buf, size_t bufsize, unsigned int seed)
{
    if (buf == NULL) return;

    srand(seed);

    const int upper_bound = 1000;
    for (size_t i = 0; i < bufsize; ++i) {
        buf[i] = rand() % upper_bound;
    }
}

void print_array(int* buf, size_t bufsize)
```

```
{
    if (buf == NULL || bufsize == 0) return;

    int *tmp = buf;
    for (size_t i = 0; i < bufsize; ++i) {
        printf("%d%c", tmp[i], ((i != bufsize - 1) ? (' ') : ('\n')));
        ++tmp;
    }
}
```

Название файла: task.h

```
#ifndef TASK_H
#define TASK_H

// Variadic arguments behavior defined by task itself
int task(int world_size, int world_rank, int n_args, char* args[]);

#endif
```

Название файла: common.c

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

#include "task.h"

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get rank
    int world_rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get amount of processes
    int world_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int root_rank = 0;

    #ifdef MEASURE_TIME
    MPI_Barrier(MPI_COMM_WORLD);
    clock_t start_clock, end_clock;
    if (world_rank == root_rank) {
        start_clock = clock();
    }
    #endif

    if (task(world_size, world_rank, argc, argv)) {
        perror("Unable to recover from error in main task");
        goto finalize;
    }

    #ifdef MEASURE_TIME
    if (world_rank == root_rank) {
        end_clock = clock();
        printf("%lf\n", ((double)end_clock - start_clock) / CLOCKS_PER_SEC);
    }
    #endif
```

```
finalize:
    return MPI_Finalize();
}
```