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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Параллельные алгоритмы»

Тема: Группы процессов и коммуникаторы

Студент гр. 3381 _____ Иванов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2025

Цель работы.

Написать параллельную программу на языке C с использованием функций группового обмена внутри собственных коммуникаторов.

Задание.

Вариант 6. В главном процессе дано целое число K и набор из K вещественных чисел, в каждом подчиненном процессе дано целое число N, которое может принимать два значения: 0 и 1 (количество подчиненных процессов с $N = 1$ равно K). Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать по одному вещественному числу из главного процесса в каждый подчиненный процесс с $N = 1$ и вывести в этих подчиненных процессах полученные числа.

Указание. При вызове функции MPI_Comm_split в процессах, которые не требуется включать в новый коммуникатор, в качестве параметра color следует указывать константу MPI_UNDEFINED.

Выполнение работы.

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

1. Главный процесс проверяет число, которое было передано аргументом командной строки;
2. Рассылка результатов валидации входного аргумента с помощью MPI_Bcast: если валидация провалилась, то все процессы завершаются с ошибкой;
3. Определение значения K во всех процессах с помощью MPI_Bcast: значение K рассылается между процессами;
4. Сравнение K с 0: если K равно нулю, значит в новый коммуникатор не попадёт ни один процесс, а значит можно сразу же завершить выполнение программы;

5. Генерация массива нулей и единиц с помощью алгоритма тасования Фишера-Йетса: этот массив представляет собой интерпретацию расстановки значений N в рабочих процессах. Массив генерируется только для рабочих процессов, но буфер выделяется для всех процессов. В первый индекс всегда подставляется 1 чтобы всегда добавлять главный процесс в новый коммуникатор, остальные значения генерируется функцией;
6. Рассылка значений N между процессами с помощью MPI_Scatter: сгенерированные значения N рассылаются между процессами;
7. Определение принадлежности процесса новому коммуникатору: если в процессе хранится значение N равное 1, то он добавляется в новый коммуникатор, иначе не добавляется;
8. Генерация массива вещественных чисел в главном процессе;
9. Рассылка вещественных чисел рабочим процессам: распределение происходит с помощью MPI_Scatterv. MPI_Scatterv используется так как главному процессу вещественное число отправлять не надо;
10. Вывод сообщения о получении вещественного числа;
11. Очистка выделенных ресурсов.

По результатам многократного запуска полученной программы получены следующий графики (K всегда бралось равное world_size — 1).

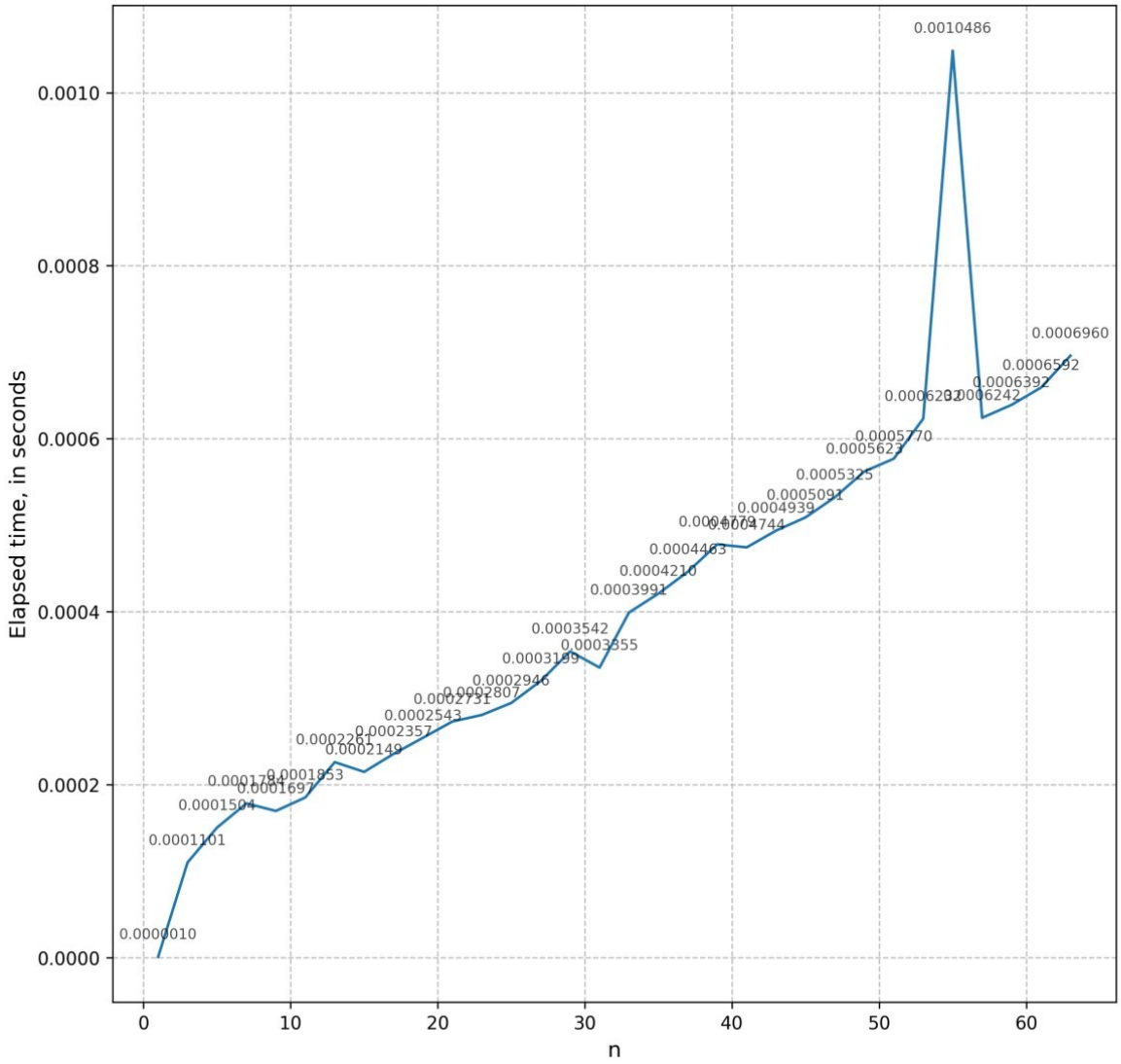

Рисунок 1 — Зависимость времени от количества процессов

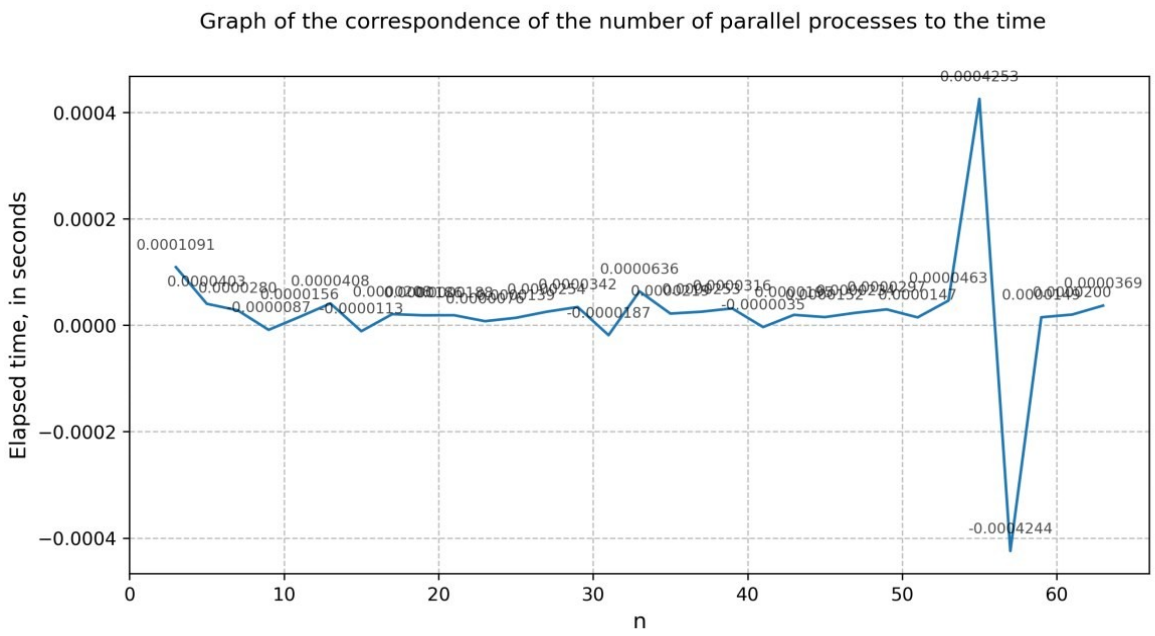

Рисунок 2 — Зависимость ускорения от количества процессов

Каждый замер произведён по 200 раз, из этих замеров отсеяны значения меньше 95-го квантиля. Если логически поразмыслить, форма графика должна быть примерно линейной (по крайней мере в первое время), так как здесь нет распараллеливания какой-то единой задачи и в данном конкретном случае измерения проводились при линейно увеличивающемся значении K. Эта оценка подтверждается графиком.

Причём интересно что на мобильном Intel i5 12450H на 12 ядер график даже при 500 замерах выглядел странно (возрастание до $n = 20$ и дальнейшее убывание). Данный же график получен на двух процессорах Intel Xeon Gold 6338 которые вместе имеют 64 ядра.

Сеть Петри для реализованного алгоритма приведена в приложении A.

Пример работы написанной программы приведён в приложении B.

Листинг программ приведён в приложении C.

Выводы.

В ходе лабораторной работы были изучены принципы создания собственных коммуникаторов и их использования в программах OpenMPI.

Для применения полученных теоретических знаний была написана программа на C в соответствии с заданием. Эта программа случайно генерирует подмножества процессов, которые будут взаимодействовать между собой в новом коммуникаторе. Далее для распределения тестовых вещественных чисел между процессами в новом коммуникаторе используются уже изученные в прошлых лабораторных работах коллективные операции.

ПРИЛОЖЕНИЕ А
СЕТЬ ПЕТРИ

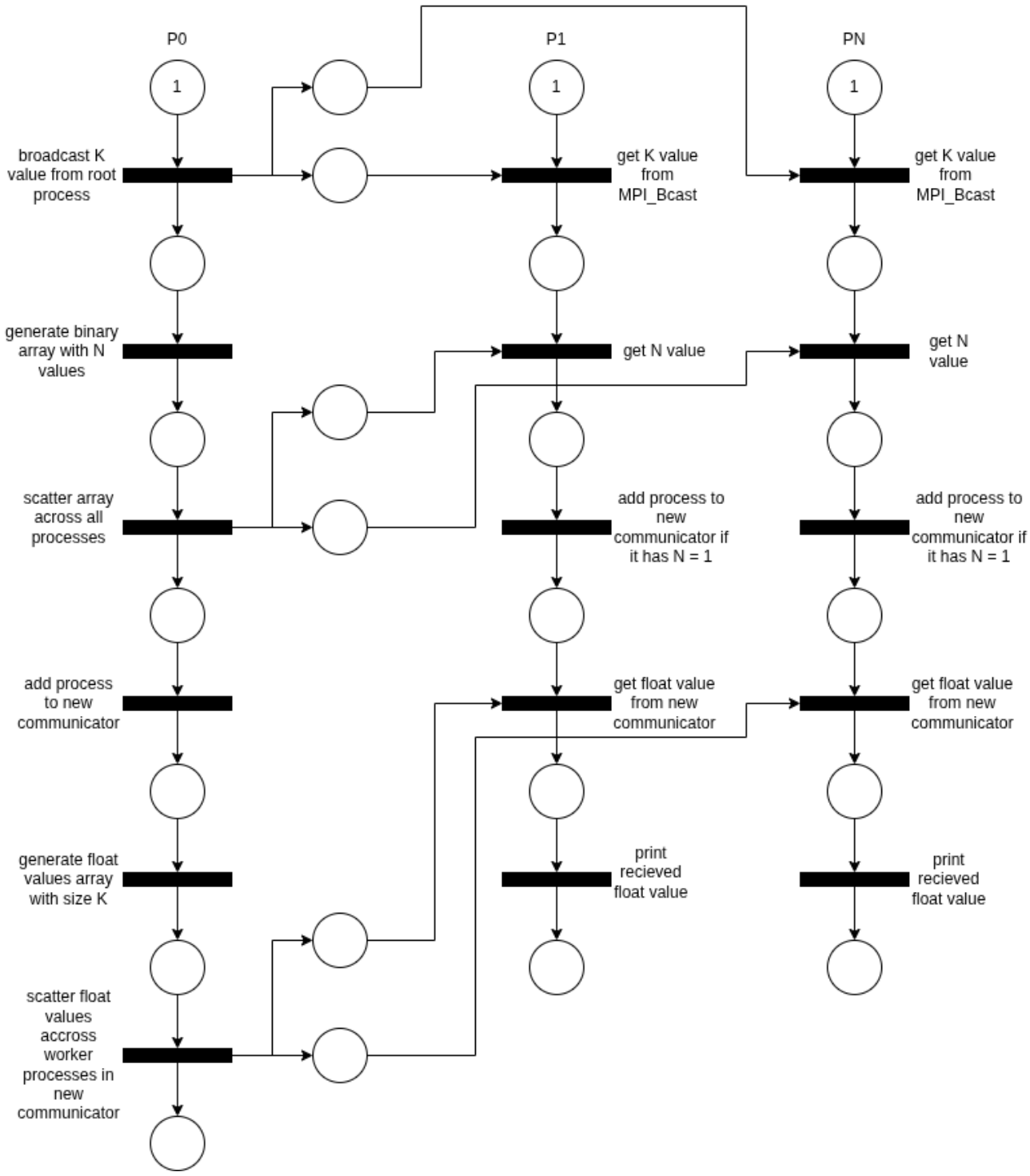

Рисунок А.1 — Сеть Петри

ПРИЛОЖЕНИЕ В
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

При нормальных условиях.

```
→ mpirun -n 8 ./exe 5
Process with world_rank=1 (local_rank=1) recieved value=0.580717
Process with world_rank=3 (local_rank=2) recieved value=0.977938
Process with world_rank=5 (local_rank=4) recieved value=0.188320
Process with world_rank=4 (local_rank=3) recieved value=0.747985
Process with world_rank=6 (local_rank=5) recieved value=0.256901

→ mpirun -n 8 ./exe 0
```

При некорректом значении K

```
→ mpirun -n 8 ./exe -1
Cannot convert input argument to positive interger
Unable to recover from error in task

→ mpirun -n 8 ./exe 8
Error: K value must be less or equal than workers count
Unable to recover from error in task

→ mpirun -n 8 ./exe A
Cannot convert input argument to positive interger
Unable to recover from error in task

→ mpirun -n 8 ./exe
Usage: ./exe <K>
Unable to recover from error in task
```

При определённой директиве MEASURE_TIME

```
→ mpirun -n 8 ./exe
0.000320
```

ПРИЛОЖЕНИЕ С
ЛИСТИНГ

Реализация задачи.

Название файла: lab4/src/task.c

```
#include "../../task.h"

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

static void usage(char* name)
{
    printf("Usage: %s <K>\n", name);
}

void generate_double_array(double* array, size_t array_size, int seed) {
    if (array == NULL || array_size == 0) {
        return;
    }

    srand(seed);

    for (size_t i = 0; i < array_size; ++i) {
        array[i] = (double)rand() / RAND_MAX;
    }
}

// Fisher-Yates
void generate_binary_array(int* array, size_t array_size, int K, int seed)
{
    if (K > array_size || array_size == 0 || array == NULL) {
        array = NULL;
        return;
    }

    srand(seed);

    memset(array, 0, array_size * sizeof(int));

    int placed_count = 0;
    for (int i = 0; i < array_size && placed_count < K; ++i) {
        int remaining_positions = array_size - i;
        int remaining_ones = K - placed_count;

        if (rand() % remaining_positions < remaining_ones) {
            array[i] = 1;
            ++placed_count;
        }
    }
}

int task(int world_size, int world_rank, int n_args, char* args[])
{
    int retcode = 0;

    // Count of subordinate processes
    int workers_count = world_size - 1;
```

```
    int K = 0;
    if (world_rank == 0) {
        #ifndef MEASURE_TIME
        // Check command line arguments count
        if (n_args != 2) {
            retcode = -1;
            usage(args[0]);
        // Try to convert argument to positive number
        } else if (sscanf(args[1], "%d", &K) != 1 || K < 0) {
            retcode = errno;
            fprintf(stderr,
                    "Cannot convert input argument to positive interger\n");
        // Program needs to have K less or equal workers count to work properly
        } else if (K > workers_count) {
            retcode = -1;
            fprintf(stderr,
                           "Error: K value must be less or equal than workers
count\n");
        }
        #else
        // Use this version to simplify time measurements
        // srand(time(NULL));
        // K = rand() % world_size;
        K = world_size - 1;
        #endif
    }
    MPI_Barrier(MPI_COMM_WORLD);

    #ifndef MEASURE_TIME
    // Send validation result to all processes
    MPI_Bcast(&retcode, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (retcode) {
        return retcode;
    }
    #endif

    MPI_Bcast(&K, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // If K is 0 then there is no processes for new communicator
    if (K == 0) {
        return retcode;
    }

    int* ones_placement = NULL;
    if (world_rank == 0) {
        ones_placement = (int*)malloc(world_size * sizeof(int));
        ones_placement[0] = 1;  // Root process must be in new communicator
        generate_binary_array(&ones_placement[1], workers_count, K, time(NULL));
    }

    int N = 0;
    MPI_Scatter(ones_placement, 1, MPI_INT, &N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (ones_placement) free(ones_placement);

    int color = (N == 1) ? (0) : (MPI_UNDEFINED);

    MPI_Comm local_comm = MPI_COMM_NULL;
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &local_comm);

    double *sendbuf = NULL;
    double recvbuf = 0.0F;
    if (world_rank == 0) {
        sendbuf = (double*)malloc(K * sizeof(double));
        if (!sendbuf) {
```

```
            retcode = errno;
            perror("Cannot allocate memory for sendbuf to send array\n");
            return retcode;
        }

        generate_double_array(sendbuf, K, time(NULL));
    }

    // Distribute double values between processes in local_comm
    if (local_comm != MPI_COMM_NULL) {
        int local_size = 0;
        MPI_Comm_size(local_comm, &local_size);
        int local_rank = 0;
        MPI_Comm_rank(local_comm, &local_rank);

        // Create prerequisites to ignore process with world rank 0 on Scatter
        int *send_counts = NULL;
        int *displs = NULL;
        if (world_rank == 0) {
            send_counts = (int*)malloc(local_size * sizeof(int));
            displs = (int*)malloc(local_size * sizeof(int));

            for (size_t i = 0; i < local_size; ++i) {
                send_counts[i] = (i == 0) ? (0) : (1);
                displs[i] = (i == 0) ? (0) : (i - 1);
            }
        }

        MPI_Scatterv(sendbuf, send_counts, displs, MPI_DOUBLE, &recvbuf, 1,
                     MPI_DOUBLE, 0, local_comm);

        #ifndef MEASURE_TIME
        if (world_rank != 0) {
                    printf("Process with world_rank=%d (local_rank=%d) recieved
value=%lf\n",
                    world_rank, local_rank, recvbuf);
        }
        #endif

        MPI_Comm_free(&local_comm);

        if (send_counts) free(send_counts);
        if (displs) free(displs);
    }

    if (sendbuf) free(sendbuf);

    return retcode;
}
```

Построение графиков.

Название файла: chart_tools.py

```
import argparse
import subprocess
import statistics

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)

        if len(time_values) > 1:
            # Calculate 95th quantile of measured time values
            time_values_95th_quantile = statistics.quantiles(
                time_values, n=100
            )[94]
                        time_values = [t for t in time_values if t <
time_values_95th_quantile]

        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 calculate_acceleration(X: int, Y: float):
    # 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])
    return (X_ac, Y_ac)

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)):
        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"
    )

    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("chart.png", dpi=300, bbox_inches="tight", facecolor="white")
```

Название файла: lab4/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}")
        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: task.h

```
#ifndef TASK_H
#define TASK_H

#include <mpi.h>
#include <errno.h>

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

#endif
```

Название файла: timer.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)) {
        if (world_rank == 0) {
            fprintf(stderr, "Unable to recover from error in task\n");
        }
        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();
}
```