2МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №6

по дисциплине «Параллельные алгоритмы»

Тема: Умножение матриц

Студент гр. 3381 _____ Иванов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2025

Цель работы.

Изучить принципы создания виртуальной топологии параллельной программы на примере декартовой решётки. Реализовать программу на языке программирования C в соответствии с заданием.

Задание.

Выполнить задачу умножения двух квадратных матриц A и B размера $m \times m$, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Вариант 1: Ленточный алгоритм 1 (горизонтальные полосы).

Выполнение работы.

Тип данных matrix_t

Для хранения матрицы реализована отдельный тип данных matrix_t и соответствующие методы matrix_* для работы с ней. Методы обеспечивают возможность инициализации матрицы (matrix_init), освобождения памяти (matrix_free), считывания значений матрицы из файлового дескриптора (matrix_read), генерации матрицы по заданному количеству столбцов и строк (matrix_gen).

Последовательный алгоритм

Последовательный алгоритм представляет из себя самый простой алгоритм умножения матриц с помощью тройного цикла. Его временная сложность равна $o(n^3)$, где n — максимум из размерностей умножаемых матриц. Эта сложность подтверждается графиком (n обозначает размер матрицы).

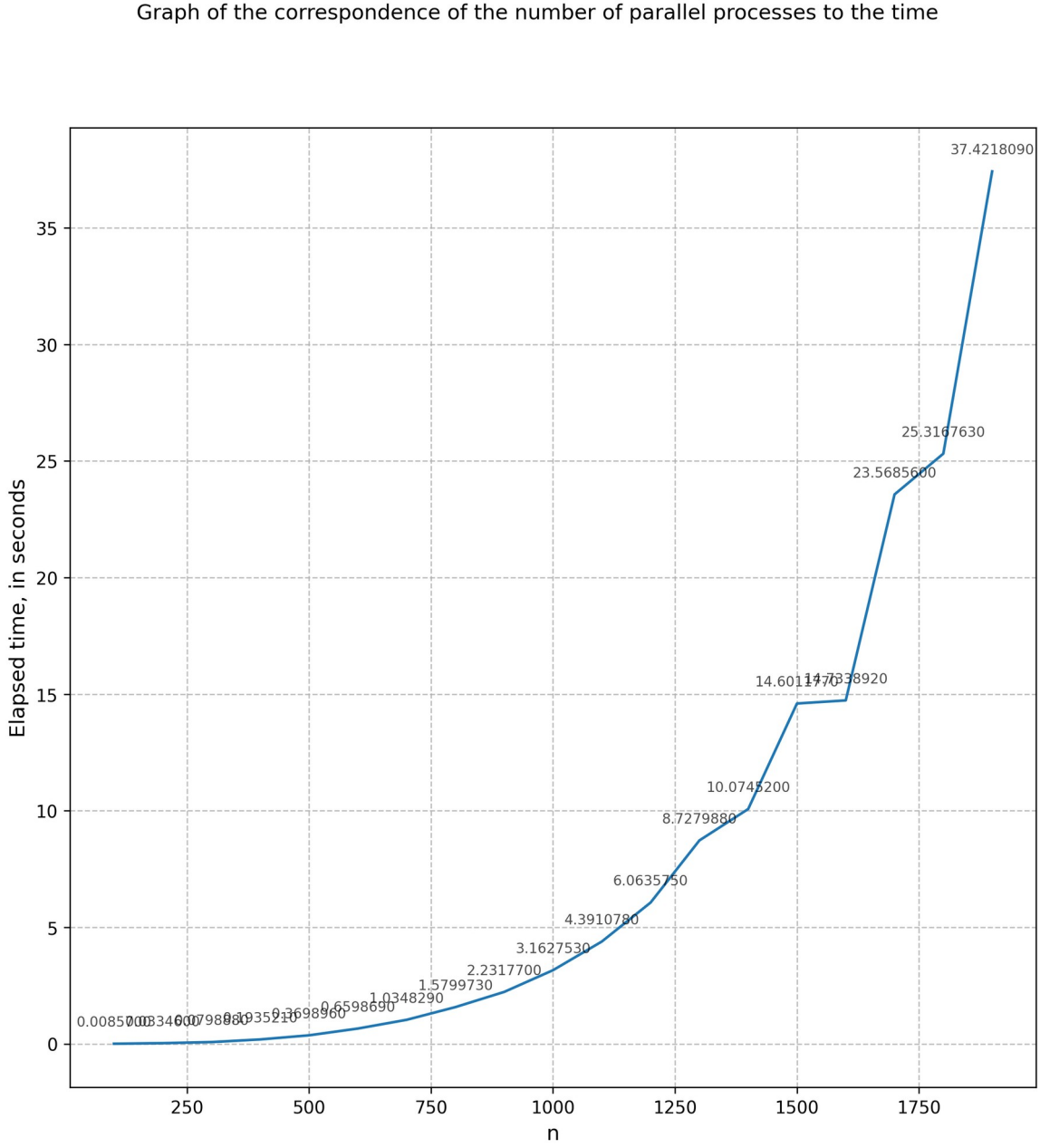

Рисунок 1 — График зависимости времени выполнения последовательного алгоритма от размера матрицы.

Параллельный алгоритм

Параллельный алгоритм представляет из себя ленточный метод (горизонтальные полосы), где матрица A делится по строкам между процессами, а матрица B копируется. Для определённости p обозначается количество процессов, а m — размерность умножаемых квадратных матриц (например, это будет 5 для матрицы 5 на 5). При $p \geq m$ каждому процессу

достаётся строка одна строка матрицы A. Иначе сначала вычисляется оценка $\text{estimation} = \lfloor \frac{m}{p} \rfloor$ количества строк матрицы A, которая достанется каждому процессу и избыток $\text{overhead} = \text{m} - \lfloor \frac{m}{p} \rfloor \cdot \text{world\_size}$, который равномерно распределяется между процессами. В итоге получается overhead процессов, которые получают $\text{estimation} + 1$ строк матрицы A, а остальные процессы получают estimation строк матрицы A. Все матрицы генерируются в процессе с рангом 0; результат уможения также аггрегируется здесь. Временная сложность алгоритма равна $o\left(\frac{m^3}{p}\right)$, так как каждый процесс вычисляет в общей сложности $\frac{m}{p} \times m \times m$ элементов матрицы.

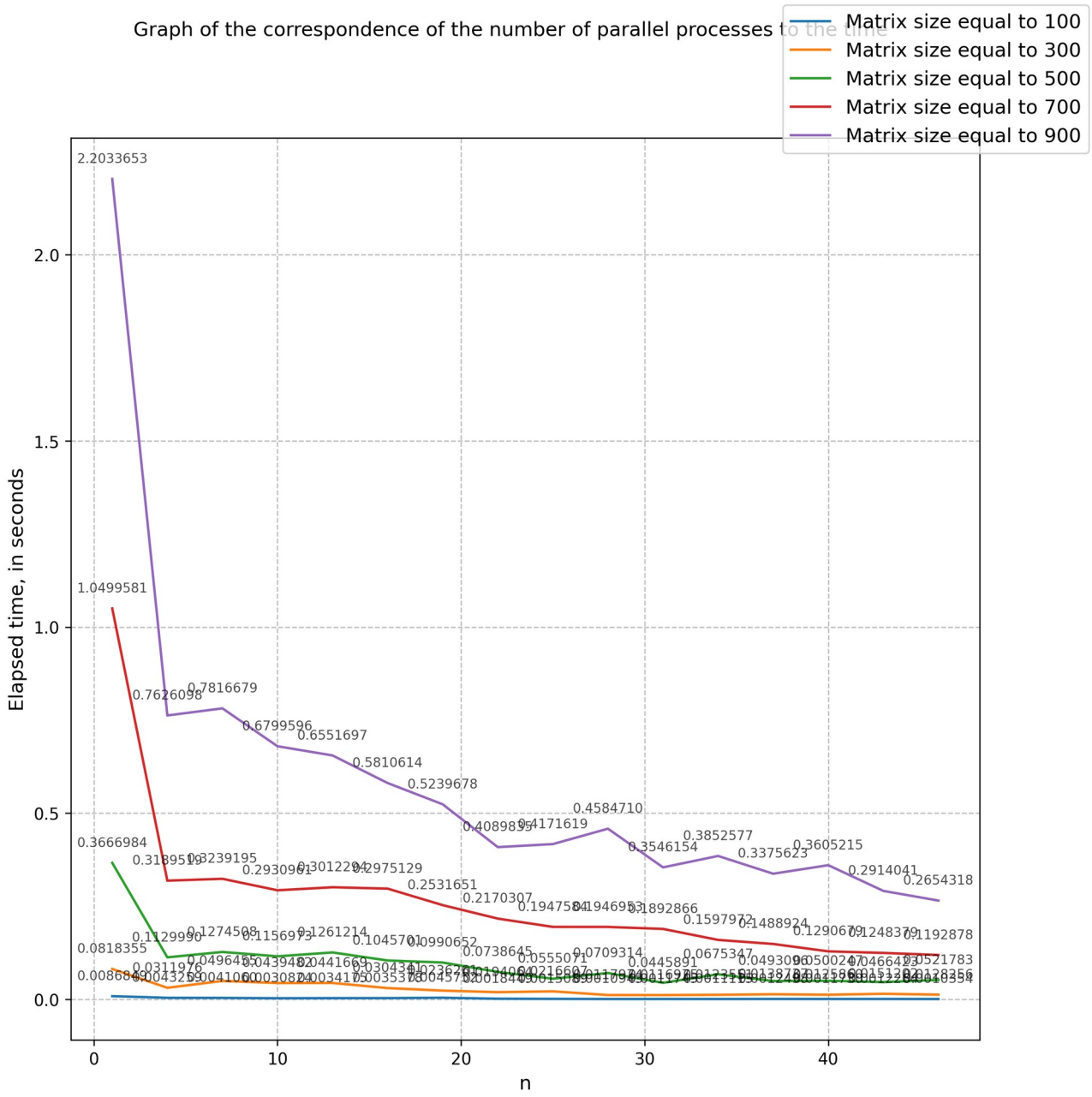

Рисунок 2 — График зависимости времени выполнения от количества параллельных процессов

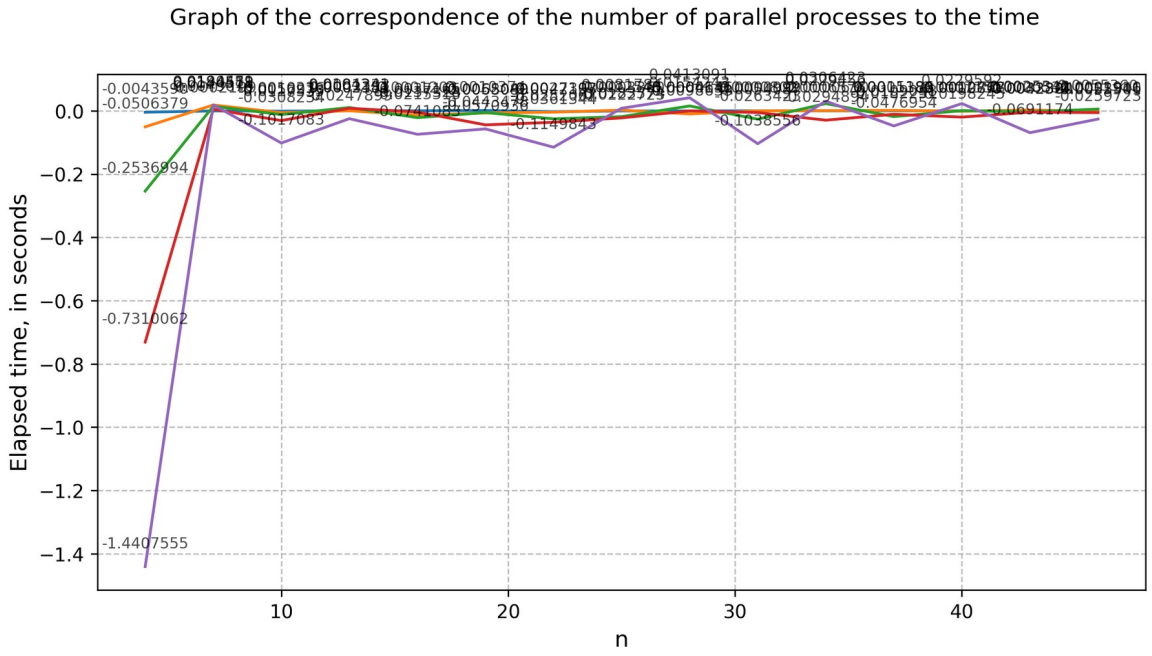

Рисунок 3 — График ускорения времени выполнения в зависости от количества параллельных процессов

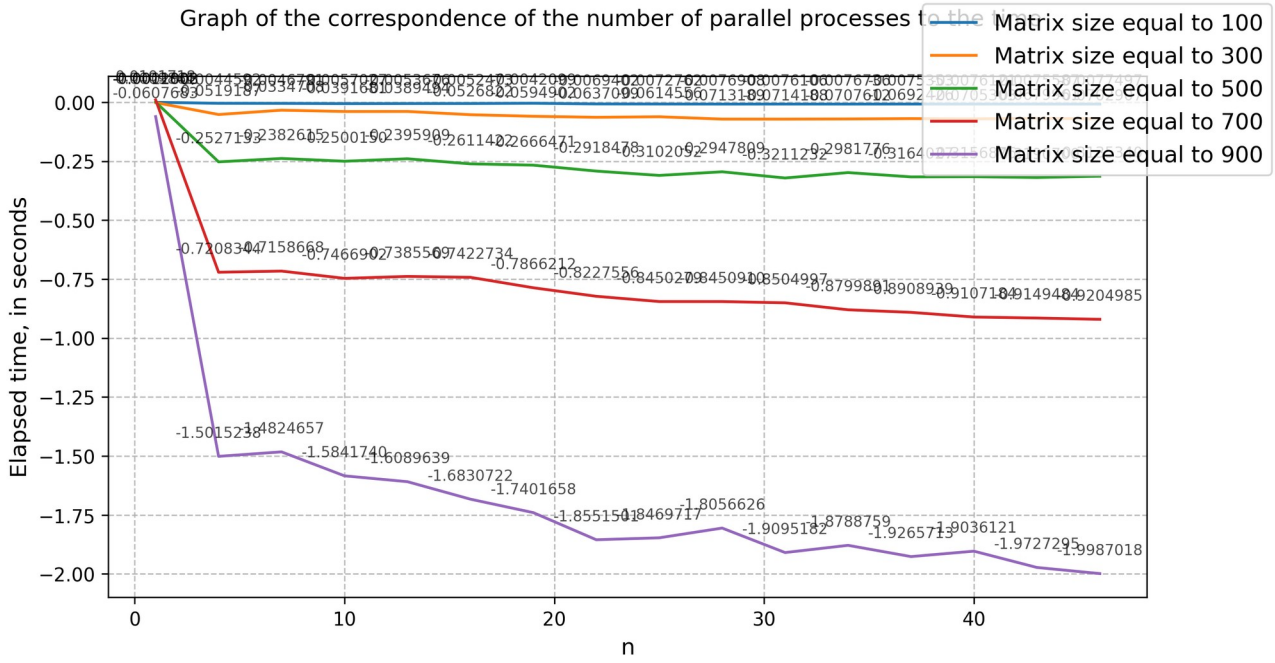

Рисунок 4 — График разницы времени выполнения параллельного и последовательного алгоритмов

Теоретически при подобном распараллеливании умножения работа программы должна до определённого момент немного ускоряться при

изменении числа параллельных процессов. Также параллельная программа должна быть быстрее своего последовательного аналога.

Эти предположения подтверждаются графиками.

Виртуальная топология параллельного алгоритма

Для алгоритма не была создана никакая виртуальная топология наподобие декартовой решётки или графа. В данном случае оказалось достаточно обычной цепочки процессов.

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 создания виртуальной топологии параллельной программы на примере декартовой решётки. Была реализована программа на языке программирования С для ленточного (по строкам) умножения двух квадратных матриц. Проведён сравнительный анализ последовательного и параллельного алгоритмов.

ПРИЛОЖЕНИЕ А
СЕТЬ ПЕТРИ

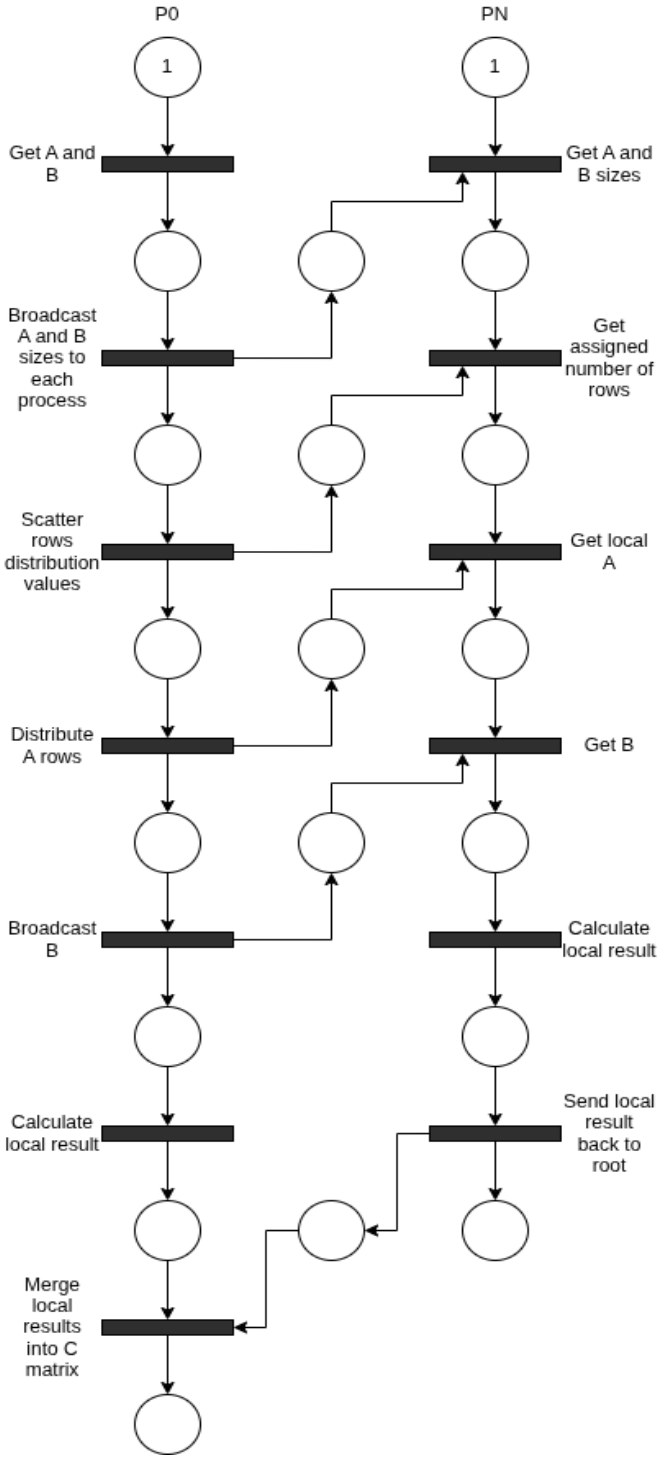

Рисунок А.1 — Сеть Петри

ПРИЛОЖЕНИЕ В
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

Умножение матриц 3 на 3 в последовательной программе:

```
 → mpirun -n 1 ./sequential_task
3 3
1 0 1
2 9 7
-9 5 6
3 3
9 9 9
1 6 8
9 12 11
18 21 20
90 156 167
-22 21 25

 → mpirun -n 1 ./sequential_task 3 3
5306 8087 6861
38 4493 1653
210 7269 2788

 → mpirun -n 5 ./sequential_task 3 3
This program cannot be run with two and more processes
...
```

Умножение матриц 3 на 3 в параллельной программе:

```
 → mpirun -n 3 ./task
3 3
9 5 1
1 2 2
99 5 1
3 3
90909 2 1
4 1 2
0 9 9
818201 32 28
90917 22 23
9000011 212 118


 → mpirun -n 5 ./task 3 3
6839 3717 4461
429 342 436
7032 4678 6219

 → mpirun -n 1 ./task 3 3
5386 5888 6769
7276 7905 13212
1686 1664 6791
```

При определённой директиве MEASURE_TIME:

```
 → mpirun -n 1 ./sequential_task 3 3
0.000016
```

ПРИЛОЖЕНИЕ С
ЛИСТИНГ

Реализация задачи.

Название файла: lab6/src/sequential_task.c

```
#include <stdio.h>

#include "matrix.h"

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

int task(int world_size, int world_rank, int n_args, char* args[])
{
    matrix_t A, B, C;

    if (world_size != 1) {
        if (world_rank == 0) {
            fprintf(stderr,
                    "This program cannot be run with two and more processes\n");
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    if (n_args == 3) {
        int n = atoi(args[1]);
        int m = atoi(args[2]);
        if (n <= 0 || m <= 0) {
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }
        matrix_gen(&A, n, m);
        matrix_gen(&B, n, m);
    } else {
        matrix_read(&A, stdin);
        matrix_read(&B, stdin);
    }

    matrix_init(A.n, B.m, &C);

    matrix_multiply(&A, &B, &C);

#ifndef MEASURE_TIME
    matrix_print(&C);
#endif

    matrix_free(&A);
    matrix_free(&B);
    matrix_free(&C);

    return 0;
}
```

Название файла: lab6/src/task.c

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

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

```
#include <stdlib.h>
#include <string.h>

#include "matrix.h"

int task(int world_size, int world_rank, int n_args, char* args[])
{
    matrix_t A = {0}, B = {0}, C = {0};

    /*
    Amount of elements which this process will recieve
    This var necessary to store sendcounts scatterv result
    */
    int recvcount = 0;

    int* sendcounts = NULL;
    int* displs = NULL;

    if (world_rank == 0) {
        // Get matrices
        if (n_args == 3) {
            int n = atoi(args[1]);
            int m = atoi(args[2]);
            if (n <= 0 || m <= 0) {
                MPI_Abort(MPI_COMM_WORLD, 1);
                return 1;
            }
            matrix_gen(&A, n, m);
            matrix_gen(&B, n, m);
        } else {
            matrix_read(&A, stdin);
            matrix_read(&B, stdin);
        }

        if (A.m != B.n) {
            fprintf(stderr, "Matrices cannot be multiplied\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        matrix_init(A.n, B.m, &C);

        sendcounts = malloc(world_size * sizeof(int));
        displs     = malloc(world_size * sizeof(int));
        memset(sendcounts, 0, world_size * sizeof(int));
        memset(displs, 0, world_size * sizeof(int));

        // Distribute rows of A
        int estimation = A.n / world_size;
        int overhead   = A.n % world_size;
        int row        = 0;

        for (int i = 0; i < world_size; ++i) {
            int rows = estimation + (i < overhead ? 1 : 0);
            sendcounts[i] = rows * A.m;
            displs[i]     = row * A.m;
            row += rows;
        }
    }

    // Broadcast matrix sizes
    MPI_Bcast(&A.n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&A.m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&B.n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    MPI_Bcast(&B.m, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Scatter a row counts
    MPI_Scatter(sendcounts, 1, MPI_INT,
                &recvcount, 1, MPI_INT,
                0, MPI_COMM_WORLD);

    int local_rows = recvcount / A.m;

    // Recieve local A
    matrix_t A_local;
    matrix_init(local_rows, A.m, &A_local);

    MPI_Scatterv(
        A.data, sendcounts, displs, MPI_INT,
        A_local.data, recvcount, MPI_INT,
        0, MPI_COMM_WORLD
    );

    // Broadcast B
    if (world_rank != 0) {
        matrix_init(B.n, B.m, &B);
    }

    MPI_Bcast(B.data, B.n * B.m, MPI_INT, 0, MPI_COMM_WORLD);

    // Local multiplication
    matrix_t C_local;
    matrix_init(local_rows, B.m, &C_local);
    memset(C_local.data, 0, local_rows * B.m * sizeof(int));

    matrix_multiply(&A_local, &B, &C_local);

    // Gather results
    int* recvcounts_C = NULL;
    int* displs_C = NULL;

    if (world_rank == 0) {
        recvcounts_C = malloc(world_size * sizeof(int));
        displs_C     = malloc(world_size * sizeof(int));

        for (int i = 0; i < world_size; ++i) {
            recvcounts_C[i] = sendcounts[i] / A.m * B.m;
            displs_C[i]     = displs[i] / A.m * B.m;
        }
    }

    MPI_Gatherv(
        C_local.data, local_rows * B.m, MPI_INT,
        C.data, recvcounts_C, displs_C, MPI_INT,
        0, MPI_COMM_WORLD
    );

#ifndef MEASURE_TIME
    // Print result
    if (world_rank == 0) {
        matrix_print(&C);
    }
#endif

    // Perform cleanup
    matrix_free(&A_local);
    matrix_free(&C_local);
```

```
    matrix_free(&B);

    if (world_rank == 0) {
        matrix_free(&A);
        matrix_free(&C);
        free(sendcounts);
        free(displs);
        free(recvcounts_C);
        free(displs_C);
    }

    return 0;
}
```

Название файла: lab6/src/matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <stdio.h>
#include <time.h>     // To initialize random number generator
#include <stdlib.h>

typedef struct {
    int   n;
    int   m;
    int* data;
} matrix_t;

int  matrix_init     (int n, int m, matrix_t* matrix);
int  matrix_read     (matrix_t* matrix, FILE* stream);
int  matrix_multiply (const matrix_t* A, const matrix_t* B, matrix_t* C);
void matrix_print    (const matrix_t* matrix);
void matrix_free     (matrix_t* matrix);
int  matrix_gen      (matrix_t* matrix, int n, int m);

#endif  // MATRIX_H
```

Название файла: lab6/src/matrix.c

```
#include "matrix.h"

int matrix_init(int n, int m, matrix_t* matrix)
{
    matrix->n = n;
    matrix->m = m;

    if (n == 0 && m == 0) {
        matrix->data = NULL;
        return 0;
    }

    matrix->data = (int*)malloc(n * m * sizeof(int*));
    if (!matrix->data) { return 1; }

    return 0;
}

int matrix_read(matrix_t* matrix, FILE* stream)
{
    int n = 0, m = 0;
```

```
    if (!fscanf(stream, "%d", &n) || n <= 0) {
        fprintf(stderr, "Invalid matrix rows count value");
        return 1;
    }
    if (!fscanf(stream, "%d", &m) || m <= 0) {
        fprintf(stderr, "Invalid matrix cols count value");
        return 1;
    }
    matrix_init(n, m, matrix);

    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < m; ++j) {
            if (!fscanf(stream, "%d", &(matrix->data[i * m + j]))) {
                return 1;
            }
        }
    }

    return 0;
}

int matrix_multiply(const matrix_t* A, const matrix_t* B, matrix_t* C)
{
    if (A->m != B->n) {
        fprintf(stderr, "A and B cannot be multiplied");
        return 1;
    }

    for (size_t i = 0; i < A->n; ++i) {
        for (size_t j = 0; j < B->m; ++j) {
            for (size_t k = 0; k < A->m; ++k) {
                 C->data[i * B->m + j] += A->data[i * A->m + k] * B->data[k * B-
>m + j];
            }
        }
    }

    return 0;
}

void matrix_print(const matrix_t* matrix)
{
    for (size_t i = 0; i < matrix->n; ++i) {
        for (size_t j = 0; j < matrix->m; ++j) {
            printf("%d", matrix->data[i * matrix->m + j]);
            if (j == matrix->m - 1) { putchar('\n'); }
            else { putchar(' '); }
        }
    }
}

void matrix_free(matrix_t* matrix)
{
    free(matrix->data);

    matrix->n = 0;
    matrix->m = 0;
}

int matrix_gen(matrix_t* matrix, int n, int m)
{
    if (matrix_init(n, m, matrix)) {
        return 1;
```

```
    }

    srand(time(NULL));

    for (size_t i = 0; i < matrix->n; ++i) {
        for (size_t j = 0; j < matrix->m; ++j) {
            matrix->data[i * matrix->m + j] = rand() % 100;
        }
    }
    return 0;
}
```

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: lab6/src/make_chart_sequential.py

```
import chart_tools
import matplotlib.pyplot as plt

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

    fig, ax = plt.subplots(1, figsize=(10, 10))
    chart_tools.setup_axes(fig, ax)

    X, Y = [], []
    for i in range(100, 2_000, 100):
        _, loc_Y = chart_tools.collect_time_stats(
            f"{args.argv} {i} {i}",
            args.averaging_parameter,
            n_values=[1]
        )
        X.append(i)
        Y.append(loc_Y[0])

    chart_tools.plot_chart(ax, X, Y)
    fig.legend(fontsize=12)
    fig.savefig("chart.png", dpi=300, bbox_inches="tight", facecolor="white")
```

Название файла: lab6/src/make_chart.py

```
import chart_tools
import matplotlib.pyplot as plt

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

    fig, ax = plt.subplots(1, figsize=(10, 10))
    chart_tools.setup_axes(fig, ax)
```

```
    fig_ac, ax_ac = plt.subplots(1, figsize=(10, 5))
    chart_tools.setup_axes(fig_ac, ax_ac)

    fig_ac_seq, ax_ac_seq = plt.subplots(1, figsize=(10, 5))
    chart_tools.setup_axes(fig_ac_seq, ax_ac_seq)

    for i in range(100, 1_000, 200):
        X_seq, Y_seq = chart_tools.collect_time_stats(
            f"./sequential_task {i} {i}",
            args.averaging_parameter,
            n_values=(1,)
        )
        X, Y = chart_tools.collect_time_stats(
            f"./task {i} {i}",
            args.averaging_parameter,
            n_values=(list(range(1, 48 + 1, 3)))
        )
        chart_tools.plot_chart(ax, X, Y, label=f"Matrix size equal to {i}")

        X_ac, Y_ac = chart_tools.calculate_acceleration(X, Y)
        chart_tools.plot_chart(ax_ac, X_ac, Y_ac)

        X_ac_seq, Y_ac_seq = X, [Y[i] - Y_seq[0] for i in range(len(Y))]
           chart_tools.plot_chart(ax_ac_seq, X_ac_seq, Y_ac_seq, label=f"Matrix
size equal to {i}")

    fig.legend(fontsize=12)
    fig.savefig("chart.png", dpi=300, bbox_inches="tight", facecolor="white")
    fig_ac.legend(fontsize=12)
            fig_ac.savefig("acceleration.png",   dpi=300,   bbox_inches="tight",
facecolor="white")
    fig_ac_seq.legend(fontsize=12)
       fig_ac_seq.savefig("acceleration_seq.png", 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();
}
```