**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №1**

**по дисциплине «Параллельные алгоритмы»**

Тема: Основы работы с потоками

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Иванов Д. М. |
| Преподаватель |  | Сергеева Е. И. |

Санкт-Петербург

2025

## Задание

Реализовать многопоточное умножение матриц с блочным разбиением.

1.1 Использовать std::thread для создания потоков. Провести исследование метрик производительности в зависимости от количества потоков и в зависимости от размера задачи, выбрать количество потоков при котором достигается оптимальная производительность.

1.2 Использовать std::async и std::future.

Реализация должна содержать тесты на корректность вычислений: состоящие в сравнение с однопоточной реализацией перемножения матриц. Метрики производительности приводить с осреднением, в отчете построить графики.

**Выполнение работы**

Алгоритм на вход будет получать квадратные матрицы, которые необходимо разбить на блоки определенного размера. Итоговая матрица также разбивается на такие же подматрицы, каждая из которых получается, как сумма произведений блоков входных матриц.

Пусть матрицы A, B разбились на q\*q блоков. Аналогично с матрицей C. Блоки будем обозначать, как Aij (0 < i < q, 0 < j < q). Тогда в результате перемножения будет получаться: Cij = Σ(Ais \* Bsj), где s перебирается от 0 до q-1.

Так как блоки вычисляются независимо, то распределим равномерно их по определенному количеству потоков. Мы поедлим количество блоков q\*q на количество потоков, именно такое количество подматриц возьмет на себя каждый, будет идти перебор по сетке потоков размера q\*q. Если нацело делиться не будет, то остаток перейдет последнему потоку.

После всех результатов блоки скопируются в итоговую матрицу C.

**Написание программы для реализации алгоритма бисекции**

Реализуем алгоритм на языке C++. Потоки будем создавать через std::thread. Для решения задачи создадим отдельный класс, в который будем передавать размеры матриц, размер блока. В конструкторе будет генерация входных матриц случайными числами.

mBaseMultiply – функция для выполнения обычного умножения входных матриц в 1 потоке.

mSumMatrix – функция для суммирования подматриц.

mGetMatrixBlock – функция для вывода самого блока матрицы по расположению начальных и конечных индексов.

mGetBlockByIndex — функция, вызывающая предыдущий метод и возвращающая блок по его расположению в сетке блоков (0 < i < q, 0 < j < q).

mGetNextBlockIndex — получение следующего блока из сетки для его вычисления в потоке

parallelMultiply — сам алгоритм блочного умножения по определенному количеству потоков. Сначала высчитывается, сколько блоков будет отведено каждому потоку. Потом в вектор по очереди добавляются потоки. В каждом из них получаются индексы блоков, которые необходимо посчитать и поток запускается для их расчета. После их создания основной поток ждет их окончания через функцию join.

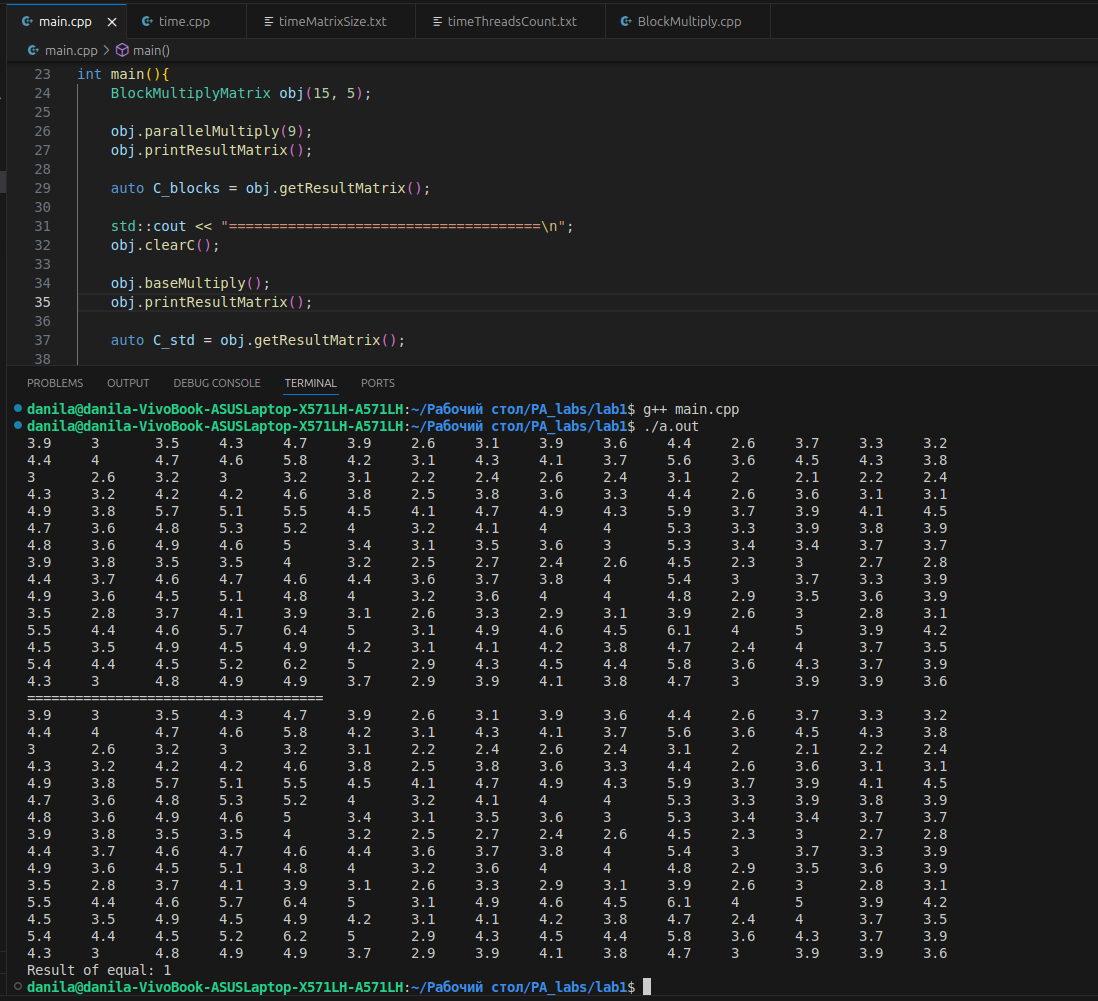
Для тестирования напишем отдельную функцию checkCorrectMultyply, которая принимает 2 итоговые матрицы: посчитанную блочным умножением и посчитанную обычным однопоточным образом. Они поэлементно сранвиваются.

Тестирование программы приведено в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | BlockMultiplyMatrix obj(55, 11);  obj.parallelMultiply(8);  auto C\_blocks = obj.getResultMatrix();  obj.clearC();  obj.baseMultiply();  auto C\_std = obj.getResultMatrix();  std::cout << "Result of equal: " << checkCorrectMultyply(C\_blocks, C\_std) << std::endl; | Result of equal: 1 | В конструкторе создаем 2 матрицы размером 55\*55, размер блока указываем 11. В параллельном умножении создаем 8 потоков, сохраняем результат. После считаем заново по однопоточному умножению. Сравниваем итоговые матрицы и получаем на выходе, что матрицы равны. |
|  | BlockMultiplyMatrix obj(100, 4);  obj.parallelMultiply(22);  auto C\_blocks = obj.getResultMatrix();  obj.clearC();  obj.baseMultiply();  auto C\_std = obj.getResultMatrix();  std::cout << "Result of equal: " << checkCorrectMultyply(C\_blocks, C\_std) << std::endl; | Result of equal: 1 | Аналогично с другими параметрами. |
|  | BlockMultiplyMatrix obj(15, 5);  obj.parallelMultiply(9);  auto C\_blocks = obj.getResultMatrix();  obj.clearC();  obj.baseMultiply();  auto C\_std = obj.getResultMatrix();  std::cout << "Result of equal: " << checkCorrectMultyply(C\_blocks, C\_std) << std::endl; | Result of equal: 1 | Аналогично с другими параметрами. |

Для зрительной проверки корректной работы, последний тест выведим на экран и посмотрим, что обе матрицы действительно вычислянются.

Рисунок 1 – результат работы одного из тестов через вывод итоговой матрицы и сравнения с обычным умножением

**Исследование зависимости времени от числа потоков**

Для исследование данной зависимости напишем программу, которая будет перебирать различные значения количества потоков для матриц размера 1408\*1408 и рамера блока 128. Расчеты будут проводиться на 3 запуска программы и считать по итогу среднее значение на каждый случай.

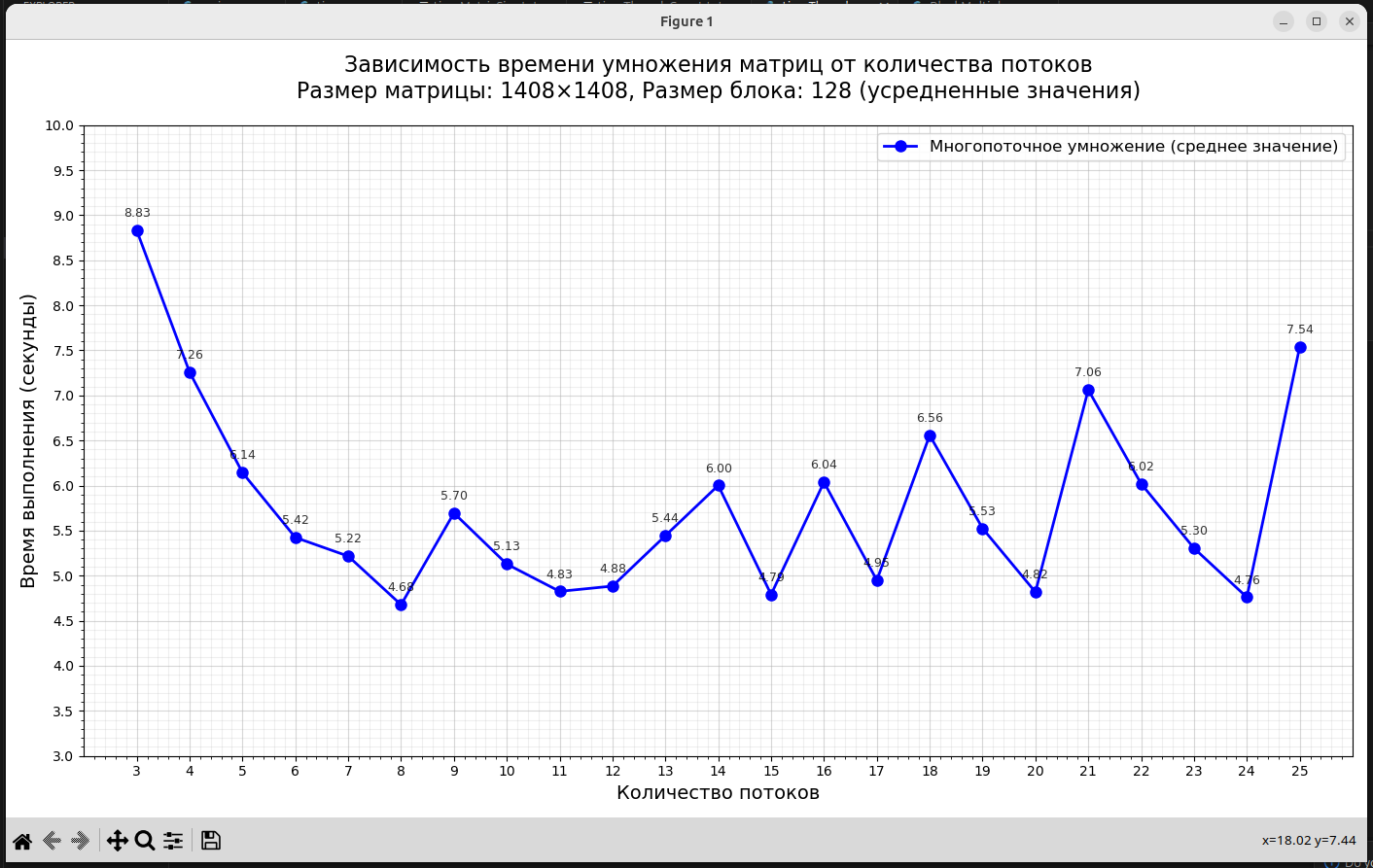
Рисунок 2 – график зависимости времени алгоритма от количества потоков

Рисунок 1 – результаты зависимости количества итераций от Eps

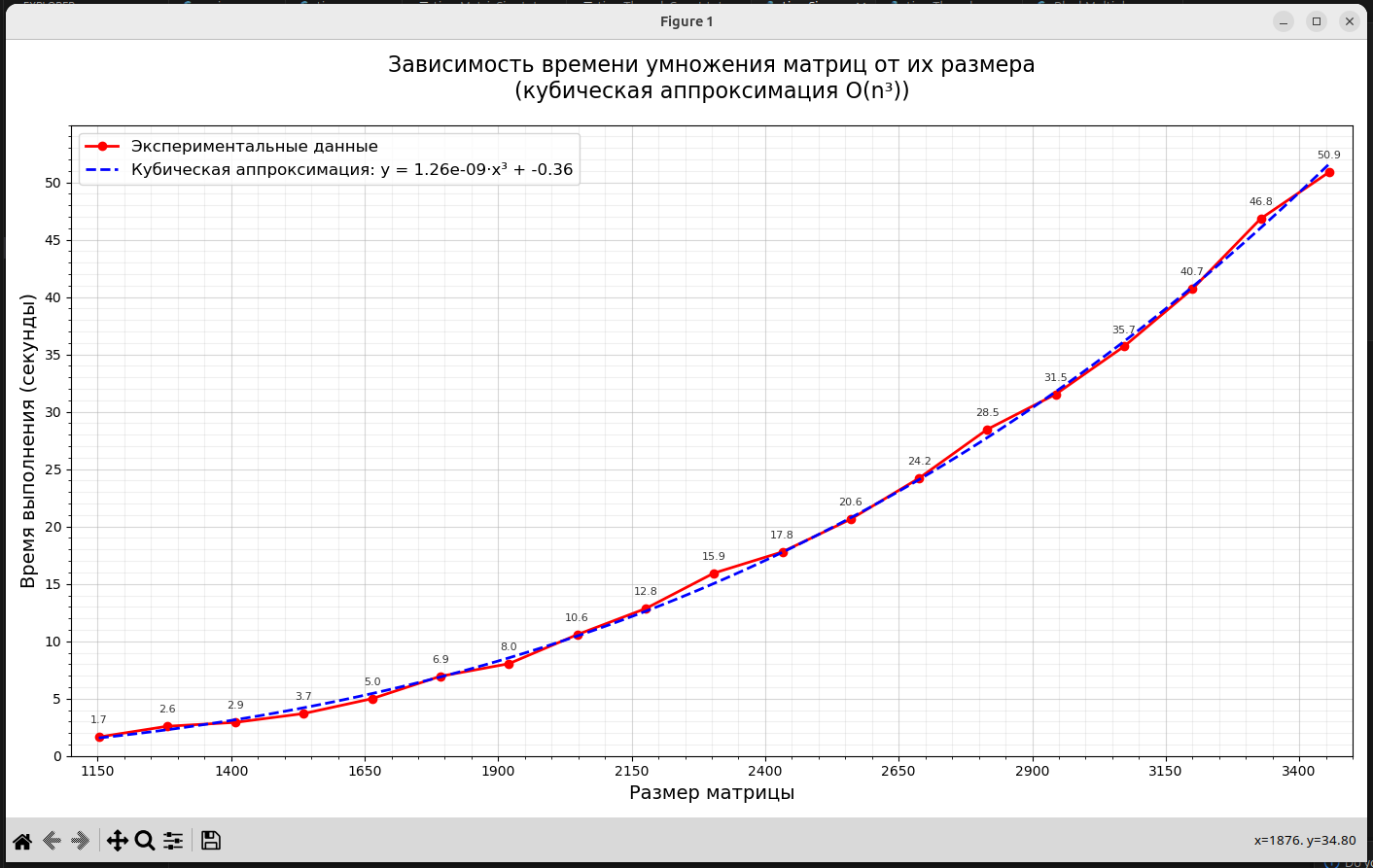
По этим данным видно, что самое оптимальное количество потоков для данной программы находится в промежутке 8-12, а лучшше время именно на 8-ми потоках. Это связано с архитектурой процессора на данной машине. Через lscpu получили информацию, что система содержит 8 ядер. Это значит, что самое оптимальное состояние алгоритма достигается в том случае. Когда каждый из ядер работает со своим потоком и проводит параллельное вычисления. При сильно маленьких значениях слабо ощущается преимущество блочного умножения, а в при сильно больших — начинается конкурениця за ресурсы и каждое ядро переходит с выполнения одного потока на другой, из-за чего возникает потеря времени.

Также на время может влиять тот факт, как поделились блоки на каждый из потоков. Если число потоков не кратно количеству блоков, то последнему потоку может достаться сильно больше блоков, чем остальным. Идеальный вариант, когда блоки поделились равномерно.

Для сравнения замерилось время перемножения матриц того же размера путем обычного умножения. Получили результат 32.612 секунд. Поэтому ускорение в сравнении с лучшим результатом (4.678 на 8 потоков) будет 6.971. А итоговая эффективность будет равна: 0.8714.

**Исследование зависимости времени от размера матриц**

Для выполнения данного исследование через цикл будем подавать программе различные размеры матриц. При этом оставлять постоянными число блоков и количество потоков. После получения зависимости через библиотеку python нашлись параметры для кубической апроксимации данных. То есть нашли такие значения, при которых функция y=a\*x^3 + b была бы максимально приближенно к нашим данным.

Рисунок 3 – график зависимости времени алгоритма от размеров матриц

По графику видно, что зависимость дейтсвительно вяглядит кубической, что соответствует сложности перемножения матриц.

## Замена на std::future и std::async

Существует в языке C++ еще один метод создания потоков. Добавим еще одно функцию parallelMultiplyFuture. Сам алгоримт останется тот же, изменится сам тип потока. В векторе теперь буду храниться объекты std::future<void>, которые нужны для получения результатов из асинхронных задач, void говорит о том, что задачи ничего возвращать не будут. Создание потока будет проиcходить через std::async, std::launch::async говорит о том, тчо гарантированно будет создан новый поток. Потом также ождание окончания работы каждого потока через wait.

Проверили данную реализацию на тех же тестах и получили аналогичный результат.

В отличие от thread, future предназначен для задач более высокого уровня. Здесь происходит автоматическое управление потоком: легкая обработка исключения, возврат результата — также поддерживается пулл потоков. Однако thread обладает большей проивзодительностью.

## Вывод

В ходе лабораторной работы была написана программа на языке C++, выполняющая алгоримт блочного умножения матриц. Были реализованы 2 варианта: через std::thread и через std::future. Для проверки работоспособности также была написана функция обычного матричного умножения дл проверки корректности работы. На тестовых данных был сделан вывод, что алгоримт работает корректно.

Также была проведена работа по анализу зависимости времени работы алгоритма от количества потоков и размеров входных матриц.

# Приложение А Исходный код программы

Название файла: BlockMultiplyMatrix.cpp

#include <thread>

#include <future>

#include <iostream>

#include <iomanip>

#include <chrono>

#include <vector>

#include <cmath>

class BlockMultiplyMatrix{

private:

std::vector<std::vector<double>> A, B, C;

int sizeMatrix;

int blockSize;

int blocksOnLine;

void mFillMatrix(){

srand(time(0));

for (int i = 0; i < sizeMatrix; i++){

for (int j = 0; j < sizeMatrix; j++){

A[i][j] = ((double)rand()) / RAND\_MAX;

B[i][j] = ((double)rand()) / RAND\_MAX;

C[i][j] = 0;

}

}

}

std::vector<std::vector<double>> mBaseMultiply(const std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B){

int n = A[0].size();

int rowCount = A.size();

int columnCount = B[0].size();

std::vector<std::vector<double>> result(rowCount, std::vector<double>(columnCount));

for (int i = 0; i < rowCount; i++){

for (int j = 0; j < columnCount; j++){

double sum = 0;

for (int x = 0; x < n; x++){

sum += A[i][x] \* B[x][j];

}

result[i][j] = sum;

}

}

return result;

}

void mSumMatrix(std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B){

for (int i = 0; i < A.size(); i++){

for (int j = 0; j < A[i].size(); j++){

A[i][j] += B[i][j];

}

}

}

std::vector<std::vector<double>> mGetMatrixBlock(const std::vector<std::vector<double>>& matrix, int startX, int endX, int startY, int endY){

std::vector<std::vector<double>> result(endY - startY, std::vector<double>(endX - startX));

for (int y = startY; y < endY; y++){

for (int x = startX; x < endX; x++){

result[y - startY][x - startX] = matrix[y][x];

}

}

return result;

}

std::vector<std::vector<double>> mGetBlockByIndex(const std::vector<std::vector<double>>& matrix, int y, int x, int rowsPerBlock, int columnsPerBlock){

return mGetMatrixBlock(matrix, columnsPerBlock \* x, columnsPerBlock \* (x + 1), rowsPerBlock \* y, rowsPerBlock \* (y + 1));

}

void mGetNextBlockIndex(int& blockX, int& blockY, int blocksOnLine) {

blockX++;

if (blockX >= blocksOnLine) {

blockX = 0;

blockY++;

}

}

void mFillC(int blockX, int blockY, int blocksPerThread, const std::vector<std::vector<double>>& C\_block){

for (int i = blockY \* blockSize; i < (blockY + 1) \* blockSize; i++){

for (int j = blockX \* blockSize; j < (blockX + 1) \* blockSize; j++){

C[i][j] = C\_block[i - blockY \* blockSize][j - blockX \* blockSize];

}

}

}

public:

BlockMultiplyMatrix(int sizeMatrix, int blockSize): sizeMatrix(sizeMatrix), blockSize(blockSize){

A.resize(sizeMatrix, std::vector<double>(sizeMatrix));

B.resize(sizeMatrix, std::vector<double>(sizeMatrix));

C.resize(sizeMatrix, std::vector<double>(sizeMatrix));

mFillMatrix();

blocksOnLine = sizeMatrix / blockSize;

}

void baseMultiply(){

C = mBaseMultiply(A, B);

}

const std::vector<std::vector<double>>& getResultMatrix(){return C;}

void printResultMatrix(){

for (auto arr: C){

for (double element: arr){

std::cout << std::setprecision(2) << element << '\t';

}

std::cout << std::endl;

}

}

void clearC(){

for (int i = 0; i < sizeMatrix; i++){

for (int j = 0; j < sizeMatrix; j++){

C[i][j] = 0;

}

}

};

void parallelMultiply(int threadsCount){

std::vector<std::thread> threadArr;

int blocksPerThread = blocksOnLine \* blocksOnLine / threadsCount;

int extraBlocks = blocksOnLine \* blocksOnLine % threadsCount;

int indexThread = 0;

for (int i = 0; i < threadsCount \* blocksPerThread; i += blocksPerThread){

threadArr.emplace\_back([this, blocksPerThread, i, threadsCount, indexThread, extraBlocks](){

int blockX = i % blocksOnLine;

int blockY = i / blocksOnLine;

int iterationsCount = blocksPerThread;

if (indexThread == threadsCount - 1){

iterationsCount += extraBlocks;

}

for (int j = 0; j < iterationsCount; j++){

std::vector<std::vector<double>> C\_block (blockSize, std::vector<double>(blockSize, 0));

for (int t = 0; t < blocksOnLine; t++){

auto A\_block = mGetBlockByIndex(A, blockY, t, blockSize, blockSize);

auto B\_block = mGetBlockByIndex(B, t, blockX, blockSize, blockSize);

mSumMatrix(C\_block, mBaseMultiply(A\_block, B\_block));

}

mFillC(blockX, blockY, blocksPerThread, C\_block);

mGetNextBlockIndex(blockX, blockY, blocksOnLine);

}

});

indexThread++;

}

for (auto& th: threadArr){

th.join();

}

}

void parallelMultiplyFuture(int threadsCount){

std::vector<std::future<void>> threadArr;

int blocksPerThread = blocksOnLine \* blocksOnLine / threadsCount;

int extraBlocks = blocksOnLine \* blocksOnLine % threadsCount;

int indexThread = 0;

for (int i = 0; i < threadsCount \* blocksPerThread; i += blocksPerThread){

threadArr.emplace\_back(std::async(

std::launch::async, [this, blocksPerThread, i, threadsCount, indexThread, extraBlocks](){

int blockX = i % blocksOnLine;

int blockY = i / blocksOnLine;

int iterationsCount = blocksPerThread;

if (indexThread == threadsCount - 1){

iterationsCount += extraBlocks;

}

for (int j = 0; j < iterationsCount; j++){

std::vector<std::vector<double>> C\_block (blockSize, std::vector<double>(blockSize, 0));

for (int t = 0; t < blocksOnLine; t++){

auto A\_block = mGetBlockByIndex(A, blockY, t, blockSize, blockSize);

auto B\_block = mGetBlockByIndex(B, t, blockX, blockSize, blockSize);

mSumMatrix(C\_block, mBaseMultiply(A\_block, B\_block));

}

mFillC(blockX, blockY, blocksPerThread, C\_block);

mGetNextBlockIndex(blockX, blockY, blocksOnLine);

}

}));

indexThread++;

}

for (auto& th: threadArr){

th.wait();

}

}

};