**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

Параллельные вычисления

|  |
| --- |
| Знакомство с возможностями OpenMP |

Руководитель А.Г. Зотин

подпись, дата инициалы, фамилия

Обучающийся БПИ20-02, 201219047 Р.А. Сухачев

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2022 г.

# ЦЕЛЬ РАБОТЫ

Получить практические навыки разработки параллельных программ с использованием OpenMP. Ознакомиться с основными функциями и директивами OpenMP, позволяющими выполнять балансировку нагрузки при обработке данных.

# порядок выполнения работы

1. Ознакомиться с теоретическими сведениями.

2. Выполнить задания 1–3.

3. Подготовить отчет по лабораторной работе.

4.  Защитить лабораторную работу перед преподавателем.

# постановка задачи

Вариант 20

Задание 2.1. Разработайте консольное приложение, реализующее вычисление числа Pi разными алгоритмами:

1) последовательная реализация;

2) параллельные реализации с использованием возможностей по управлению нагрузки и синхронизации данных OpenMP;

Для каждой реализации функций вычисления числа Pi выполните расчет времени1 и сформируйте сводную таблицу. В качестве параметра для наборов данных (НД) выступает количество итераций (num\_steps), которое рекомендуется задавать в зависимости от мощности компьютера.

Задание 2.2. Разработайте консольное приложение, реализующее заполнение и перемножение матриц A и B (c размерностью M×N и N×K соответственно). Для перемножения матриц использовать классический алгоритм и один быстрый алгоритм. Каждый студент придумывает свои формулы заполнения для матриц A и B в зависимости от индексов. Необходимо осуществить следующие последовательную реализацию алгоритмов и параллельнык реализации с использованием различных видов распараллеливания.

При реализации перемножения матриц следует учитывать распределение элементов в памяти, и для одного алгоритма реализовать балансировку нагрузки. Также следует реализовать один из алгоритмов быстрого перемножения матриц.

Задание 2.3. Проведите экспериментального исследования по обработке данных с разными реализациями (выполнять не менее 20 запусков для каждой реализации). При проведении исследования используйте массивы вещественного типа размерностью от 800×800 до 2500×2500; четыре разных набора данных (НД). Для реализованных функций на основе полученных данных исследования заполнить таблицы.

# ХОД РАБОТЫ

Задание 2.1.

Код программы:

#include <fstream>

#include <iostream>

#include <omp.h>

#include <math.h>

#include <vector>

using namespace std;

typedef double(\*TestFunctTempl)(long&);

double Pi\_Posled(long num\_steps) {

double time = omp\_get\_wtime();

long i;

double step, pi, x, sum = 0.0;

step = 1.0 / (double)num\_steps;

for (i = 0; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum = sum + 4.0 / (1.0 + x \* x);

}

pi = step \* sum;

double time\_end = omp\_get\_wtime();

return time\_end - time;

}

double Pi\_Static(long num\_steps) {

double time = omp\_get\_wtime();

long i;

double step, pi, x, sum = 0.0;

step = 1.0 / (double)num\_steps;

#pragma omp parallel for schedule(static,num\_steps/30) private(x) reduction(+:sum)

for (i = 0; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum = sum + 4.0 / (1.0 + x \* x);

}

pi = step \* sum;

double time\_end = omp\_get\_wtime();

return time\_end - time;

}

double Pi\_dynamic(long num\_steps) {

double time = omp\_get\_wtime();

long i;

double step, pi, x, sum = 0.0;

step = 1.0 / (double)num\_steps;

#pragma omp parallel for schedule(dynamic,num\_steps/30) private(x) reduction(+:sum)

for (i = 0; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum = sum + 4.0 / (1.0 + x \* x);

}

pi = step \* sum;

double time\_end = omp\_get\_wtime();

return time\_end - time;

}

double Pi\_guided(long num\_steps) {

double time = omp\_get\_wtime();

long i;

double step, pi, x, sum = 0.0;

step = 1.0 / (double)num\_steps;

#pragma omp parallel for schedule(guided,num\_steps/30) private(x) reduction(+:sum)

for (i = 0; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum = sum + 4.0 / (1.0 + x \* x);

}

pi = step \* sum;

double time\_end = omp\_get\_wtime();

return time\_end - time;

}

double Pi\_Section(long num\_steps) {

double time = omp\_get\_wtime();

long i;

double step, pi, x, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0;

int n;

step = 1.0 / (double)num\_steps;

#pragma omp parallel

{

//omp\_set\_num\_threads(n = 4);

n = omp\_get\_max\_threads();

}

int nt = 0, nt1 = num\_steps / n, nt2 = num\_steps \* 2 / n, nt3 = num\_steps \* 3 / n;

#pragma omp parallel sections private (x)

{

#pragma omp section

{

for (i = nt; i < nt1; i++)

{

x = (i + 0.5) \* step;

sum1 += 4.0 / (1.0 + x \* x);

}

}

#pragma omp section

{

for (i = nt1; i < nt2; i++)

{

x = (i + 0.5) \* step;

sum2 += 4.0 / (1.0 + x \* x);

}

}

#pragma omp section

{

if (num\_steps > 2)

{

for (i = nt2; i < nt3; i++)

{

x = (i + 0.5) \* step;

sum3 += 4.0 / (1.0 + x \* x);

}

}

}

#pragma omp section

{

if (num\_steps > 3)

{

for (i = nt3; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum4 += 4.0 / (1.0 + x \* x);

}

}

}

}

pi = step \* (sum1 + sum2 + sum3 + sum4);

double time\_end = omp\_get\_wtime();

return time\_end - time;

}

double TestPi\_Posled(long& num\_steps) {

return Pi\_Posled(num\_steps);

}

double TestPi\_Static(long& num\_steps) {

return Pi\_Static(num\_steps);

}

double TestPi\_dynamic(long& num\_steps) {

return Pi\_dynamic(num\_steps);

}

double TestPi\_guided(long& num\_steps) {

return Pi\_guided(num\_steps);

}

double TestPi\_Section(long& num\_steps) {

return Pi\_Section(num\_steps);

}

double AvgTrustedInterval(double& avg, vector<double>& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

double TestIter(void\* Funct, long num\_steps)

{

double curtime = 0, avgTime = 0, avgTimeT = 0, correctAVG = 0;

int iterations = 100;

vector<double> Times(iterations);

for (int i = 0; i < iterations; i++)

{

curtime = (((TestFunctTempl)Funct)(num\_steps)) \* 1000;

Times[i] = curtime;

avgTime += curtime;

cout << "+";

}

cout << endl;

avgTime /= iterations;

cout << "AvgTime:" << avgTime << endl;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

cout << "AvgTimeTrusted:" << avgTimeT << endl;

return avgTimeT;

}

void test\_functions(void\*\* Functions, vector<string> fNames)

{

int nd = 0;

double times[4][5][3];

for (int num\_steps = 500000; num\_steps <= 2000000; num\_steps += 500000)

{

for (int threads = 1; threads <= 4; threads++)

{

omp\_set\_num\_threads(threads);

for (int alg = 0; alg <= 4; alg++)

{

if (threads == 1)

{

if (alg == 0) {

times[nd][alg][0] = TestIter(Functions[alg], num\_steps);

times[nd][alg][1] = times[nd][alg][0];

times[nd][alg][2] = times[nd][alg][0];

}

}

else

{

if (alg != 0)

{

times[nd][alg][threads - 2] = TestIter(Functions[alg], num\_steps);

}

}

}

}

nd++;

}

ofstream fout("output.txt");

fout.imbue(locale("Russian"));

for (int ND = 0; ND < 4; ND++)

{

switch (ND)

{

case 0:

cout << "\n----------500000 количество итераций----------" << endl;

break;

case 1:

cout << "\n----------1000000 количество итераций----------" << endl;

break;

case 2:

cout << "\n----------1500000 количество итераций----------" << endl;

break;

case 3:

cout << "\n----------2000000 количество итераций----------" << endl;

break;

default:

break;

}

for (int alg = 0; alg < 5; alg++)

{

for (int threads = 1; threads <= 4; threads++)

{

if (threads == 1)

{

if (alg == 0) {

cout << "Thread " << threads << " --------------" << endl;

cout << fNames[alg] << "\t" << times[ND][alg][0] << " ms." << endl;

fout << times[ND][alg][0] << endl;

}

}

else

{

if (alg != 0)

{

cout << "Thread " << threads << " --------------" << endl;

cout << fNames[alg] << "\t" << times[ND][alg][threads - 2] << " ms." << endl;

fout << times[ND][alg][threads - 2] << endl;

}

}

}

}

}

fout.close();

}

int main()

{

void\*\* Functions = new void\* [5] { TestPi\_Posled, TestPi\_Static, TestPi\_dynamic, TestPi\_guided, TestPi\_Section};

vector<string> function\_names = { "Consistent realization","Parallel realization FOR(static)",

"Parallel realization FOR(dinamic)", "Parallel realization FOR(guided)", "Parallel realization Section" };

test\_functions(Functions, function\_names);

return 0;

}

Задание 2.2.

Код программы:

#include <iostream>

#include <omp.h>

#include <Windows.h>

#include <fstream>

#include <iostream>

#include <vector>

typedef double(\*TestFunctTempl)(double\*\*&, double\*\*&, double\*\*&, int&, int&, int&);

using namespace std;

double FillMatr(double\*\*& matrix1, int size1, int size2) {

double t\_start = omp\_get\_wtime();

for (int i = 0; i < size1; i++)

for (int j = 0; j < size2; j++)

matrix1[i][j] = sin(90 - i) + pow(i, 2 / 11);

double t\_end = omp\_get\_wtime();

return t\_end - t\_start;

}

double FillMatrParallel(double\*\*& matrix2, int size1, int size2) {

double t\_start = omp\_get\_wtime();

#pragma omp parallel for schedule(guided)

for (int i = 0; i < size1; i++)

for (int j = 0; j < size2; j++)

matrix2[i][j] = sin(90 - i) + pow(i, 2 / 11);

double t\_end = omp\_get\_wtime();

return t\_end - t\_start;

}

double FillMatrZero(double\*\*& matrix1, int size1, int size2)

{

double time\_start = omp\_get\_wtime();

for (int i = 0; i < size1; i++) {

for (int j = 0; j < size2; j++) {

matrix1[i][j] = 0;

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double FillMatrParallelStatic(double\*\*& matrix, int size1, int size2) {

double t\_start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for (int i = 0; i < size1; i++)

for (int j = 0; j < size2; j++)

matrix[i][j] = sin(i + 0.5) + cos(i / 2);

double t\_end = omp\_get\_wtime();

return t\_end - t\_start;

}

double FillMatrParallelDynamic(double\*\*& matrix, int size1, int size2) {

double t\_start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for (int i = 0; i < size1; i++)

for (int j = 0; j < size2; j++)

matrix[i][j] = sin(i + 0.5) + cos(i / 2);

double t\_end = omp\_get\_wtime();

return t\_end - t\_start;

}

double FillMatrParallelGuided(double\*\*& matrix, int size1, int size2) {

double t\_start = omp\_get\_wtime();

int p = omp\_get\_max\_threads();

#pragma omp parallel for schedule(guided)

for (int i = 0; i < size1; i++)

for (int j = 0; j < size2; j++)

matrix[i][j] = sin(i + 0.5) + cos(i / 2);

double t\_end = omp\_get\_wtime();

return t\_end - t\_start;

}

double MultiplyMatrV4(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC) {

double time\_start = omp\_get\_wtime();

double\*\* mtr = new double\* [sizeC];

for (int i = 0; i < sizeC; i++)

mtr[i] = new double[sizeB];

for (int i = 0; i < sizeB; i++)

for (int j = 0; j < sizeC; j++)

mtr[j][i] = matrix2[i][j];

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

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

double tmp = 0;

for (int k = 0; k < sizeB; k++) {

tmp += matrix1[i][k] \* mtr[j][k];

}

matrix3[i][j] = tmp;

}

}

for (int i = 0; i < sizeC; i++)

{

delete[] mtr[i];

}

delete[] mtr;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double MultiplyMatrV4Parrallelguided(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC) {

double time\_start = omp\_get\_wtime();

double\*\* mtr = new double\* [sizeC];

for (int i = 0; i < sizeC; i++)

mtr[i] = new double[sizeB];

#pragma omp parallel for schedule(guided, sizeB/10)

for (int i = 0; i < sizeB; i++)

for (int j = 0; j < sizeC; j++)

mtr[j][i] = matrix2[i][j];

#pragma omp parallel for schedule(guided, sizeA/10)

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

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

double tmp = 0;

for (int k = 0; k < sizeB; k++) {

tmp += matrix1[i][k] \* mtr[j][k];

}

matrix3[i][j] = tmp;

}

}

for (int i = 0; i < sizeC; i++)

{

delete mtr[i];

}

delete[] mtr;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double MultiplyMatrV4Parrallelstatic(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC) {

double time\_start = omp\_get\_wtime();

double\*\* mtr = new double\* [sizeC];

for (int i = 0; i < sizeC; i++)

mtr[i] = new double[sizeB];

#pragma omp parallel for schedule(static,sizeB/10)

for (int i = 0; i < sizeB; i++)

for (int j = 0; j < sizeC; j++)

mtr[j][i] = matrix2[i][j];

#pragma omp parallel for schedule(static, sizeA/10)

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

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

double tmp = 0;

for (int k = 0; k < sizeB; k++) {

tmp += matrix1[i][k] \* mtr[j][k];

}

matrix3[i][j] = tmp;

}

}

for (int i = 0; i < sizeC; i++)

{

delete mtr[i];

}

delete[] mtr;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double MultiplyMatrV4Parralleldynamic(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC) {

double time\_start = omp\_get\_wtime();

double\*\* mtr = new double\* [sizeC];

for (int i = 0; i < sizeC; i++)

mtr[i] = new double[sizeB];

#pragma omp parallel for schedule(dynamic, sizeB/10)

for (int i = 0; i < sizeB; i++)

for (int j = 0; j < sizeC; j++)

mtr[j][i] = matrix2[i][j];

#pragma omp parallel for schedule(dynamic, sizeA/10)

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

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

double tmp = 0;

for (int k = 0; k < sizeB; k++) {

tmp += matrix1[i][k] \* mtr[j][k];

}

matrix3[i][j] = tmp;

}

}

for (int i = 0; i < sizeC; i++)

{

delete mtr[i];

}

delete[] mtr;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

int ADD(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

for (int i = 0; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

return 0;

}

int ADD\_Guided(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

int n\_t = omp\_get\_max\_threads() \* 10;

#pragma omp parallel for schedule(guided)

for (int i = 0; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

return 0;

}

int ADD\_Sections(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

int n\_t = 4;

#pragma omp parallel

{

n\_t = omp\_get\_max\_threads();

}

int st1 = MatrixSize / n\_t;

int st2 = MatrixSize \* 2 / n\_t;

int st3 = MatrixSize \* 3 / n\_t;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = 0; i < st1; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 1)

for (int i = st1; i < st2; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 2)

for (int i = st2; i < st3; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 3)

for (int i = st3; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] + MatrixB[i][j];

}

}

}

}

return 0;

}

int SUB(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

for (int i = 0; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

return 0;

}

int SUB\_Sections(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

int n\_t = 4;

#pragma omp parallel

{

n\_t = omp\_get\_max\_threads();

}

int st1 = MatrixSize / n\_t;

int st2 = MatrixSize \* 2 / n\_t;

int st3 = MatrixSize \* 3 / n\_t;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = 0; i < st1; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 1)

for (int i = st1; i < st2; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 2)

for (int i = st2; i < st3; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

}

#pragma omp section

{

if (n\_t > 3)

for (int i = st3; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

}

}

return 0;

}

int SUB\_Guided(double\*\*& MatrixA, double\*\*& MatrixB, double\*\*& MatrixResult, int MatrixSize)

{

int p = omp\_get\_max\_threads() \* 10;

#pragma omp parallel for schedule(guided)

for (int i = 0; i < MatrixSize; i++)

{

for (int j = 0; j < MatrixSize; j++)

{

MatrixResult[i][j] = MatrixA[i][j] - MatrixB[i][j];

}

}

return 0;

}

int MUL(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixResult, int MatrixSize)

{

double\*\* mtr = new double\* [MatrixSize];

mtr[0] = new double[MatrixSize \* MatrixSize];

for (int i = 0; i < MatrixSize; i++)

mtr[i] = new double[MatrixSize];

for (int i = 0; i < MatrixSize; i++)

for (int j = 0; j < MatrixSize; j++)

mtr[j][i] = MatrixB[i][j];

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

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

double tmp = 0;

for (int k = 0; k < MatrixSize; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

delete[] mtr[0];

delete[] mtr;

return 0;

}

int MUL\_Section(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixResult, int MatrixSize)

{

double\*\* mtr = new double\* [MatrixSize];

mtr[0] = new double[MatrixSize \* MatrixSize];

for (int i = 1; i < MatrixSize; i++)

mtr[i] = &mtr[0][i \* MatrixSize];

for (int i = 0; i < MatrixSize; i++)

for (int j = 0; j < MatrixSize; j++)

mtr[j][i] = MatrixB[i][j];

int n\_t = 4;

#pragma omp parallel

{

n\_t = omp\_get\_max\_threads();

}

int st1 = MatrixSize / n\_t;

int st2 = MatrixSize \* 2 / n\_t;

int st3 = MatrixSize \* 3 / n\_t;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = 0; i < st1; i++) {

for (int j = 0; j < st1; j++) {

double tmp = 0;

for (int k = 0; k < st1; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

}

#pragma omp section

{

if (n\_t > 1)

{

for (int i = st1; i < st2; i++) {

for (int j = st1; j < st2; j++) {

double tmp = 0;

for (int k = st1; k < st2; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

}

}

#pragma omp section

{

if (n\_t > 2)

{

for (int i = st2; i < st3; i++) {

for (int j = st2; j < st3; j++) {

double tmp = 0;

for (int k = st2; k < st3; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

}

}

#pragma omp section

{

if (n\_t > 3)

{

for (int i = st3; i < MatrixSize; i++) {

for (int j = st3; j < MatrixSize; j++) {

double tmp = 0;

for (int k = st3; k < MatrixSize; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

}

}

}

delete[] mtr[0];

delete[] mtr;

return 0;

}

int MUL\_Guided(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixResult, int MatrixSize)

{

double\*\* mtr = new double\* [MatrixSize];

mtr[0] = new double[MatrixSize \* MatrixSize];

for (int i = 0; i < MatrixSize; i++)

mtr[i] = new double[MatrixSize];

for (int i = 0; i < MatrixSize; i++)

for (int j = 0; j < MatrixSize; j++)

mtr[j][i] = MatrixB[i][j];

#pragma omp parallel for schedule(guided)

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

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

double tmp = 0;

for (int k = 0; k < MatrixSize; k++) {

tmp += MatrixA[i][k] \* mtr[j][k];

}

MatrixResult[i][j] = tmp;

}

}

delete[] mtr[0];

delete[] mtr;

return 0;

}

int Strassen(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixC, int MatrixSize, int linearMultBlockSize)

{

int HalfSize = MatrixSize / 2;

if (MatrixSize <= linearMultBlockSize)

{

MUL(MatrixA, MatrixB, MatrixC, MatrixSize);

}

else

{

double\*\* A11, \*\* A12, \*\* A21, \*\* A22;

double\*\* B11, \*\* B12, \*\* B21, \*\* B22;

double\*\* C11, \*\* C12, \*\* C21, \*\* C22;

double\*\* M1, \*\* M2, \*\* M3, \*\* M4, \*\* M5, \*\* M6, \*\* M7;

double\*\* AResult, \*\* BResult;

A11 = new double\* [HalfSize];

A12 = new double\* [HalfSize];

A21 = new double\* [HalfSize];

A22 = new double\* [HalfSize];

B11 = new double\* [HalfSize];

B12 = new double\* [HalfSize];

B21 = new double\* [HalfSize];

B22 = new double\* [HalfSize];

C11 = new double\* [HalfSize];

C12 = new double\* [HalfSize];

C21 = new double\* [HalfSize];

C22 = new double\* [HalfSize];

M1 = new double\* [HalfSize];

M2 = new double\* [HalfSize];

M3 = new double\* [HalfSize];

M4 = new double\* [HalfSize];

M5 = new double\* [HalfSize];

M6 = new double\* [HalfSize];

M7 = new double\* [HalfSize];

AResult = new double\* [HalfSize];

BResult = new double\* [HalfSize];

A11[0] = new double[HalfSize \* HalfSize];

A12[0] = new double[HalfSize \* HalfSize];

A21[0] = new double[HalfSize \* HalfSize];

A22[0] = new double[HalfSize \* HalfSize];

B11[0] = new double[HalfSize \* HalfSize];

B12[0] = new double[HalfSize \* HalfSize];

B21[0] = new double[HalfSize \* HalfSize];

B22[0] = new double[HalfSize \* HalfSize];

C11[0] = new double[HalfSize \* HalfSize];

C12[0] = new double[HalfSize \* HalfSize];

C21[0] = new double[HalfSize \* HalfSize];

C22[0] = new double[HalfSize \* HalfSize];

M1[0] = new double[HalfSize \* HalfSize];

M2[0] = new double[HalfSize \* HalfSize];

M3[0] = new double[HalfSize \* HalfSize];

M4[0] = new double[HalfSize \* HalfSize];

M5[0] = new double[HalfSize \* HalfSize];

M6[0] = new double[HalfSize \* HalfSize];

M7[0] = new double[HalfSize \* HalfSize];

AResult[0] = new double[HalfSize \* HalfSize];

BResult[0] = new double[HalfSize \* HalfSize];

for (int i = 0; i < HalfSize; i++)

{

A11[i] = &A11[0][i \* HalfSize];

A12[i] = &A12[0][i \* HalfSize];

A21[i] = &A21[0][i \* HalfSize];

A22[i] = &A22[0][i \* HalfSize];

B11[i] = &B11[0][i \* HalfSize];

B12[i] = &B12[0][i \* HalfSize];

B21[i] = &B21[0][i \* HalfSize];

B22[i] = &B22[0][i \* HalfSize];

C11[i] = &C11[0][i \* HalfSize];

C12[i] = &C12[0][i \* HalfSize];

C21[i] = &C21[0][i \* HalfSize];

C22[i] = &C22[0][i \* HalfSize];

M1[i] = &M1[0][i \* HalfSize];

M2[i] = &M2[0][i \* HalfSize];

M3[i] = &M3[0][i \* HalfSize];

M4[i] = &M4[0][i \* HalfSize];

M5[i] = &M5[0][i \* HalfSize];

M6[i] = &M6[0][i \* HalfSize];

M7[i] = &M7[0][i \* HalfSize];

AResult[i] = &AResult[0][i \* HalfSize];

BResult[i] = &BResult[0][i \* HalfSize];

}

/////////////////////////////////////////

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

A11[i][j] = MatrixA[i][j];

A12[i][j] = MatrixA[i][j + HalfSize];

A21[i][j] = MatrixA[i + HalfSize][j];

A22[i][j] = MatrixA[i + HalfSize][j + HalfSize];

B11[i][j] = MatrixB[i][j];

B12[i][j] = MatrixB[i][j + HalfSize];

B21[i][j] = MatrixB[i + HalfSize][j];

B22[i][j] = MatrixB[i + HalfSize][j + HalfSize];

}

}

//P1 == M1[][]

ADD(A11, A22, AResult, HalfSize);

ADD(B11, B22, BResult, HalfSize);

Strassen(AResult, BResult, M1, HalfSize, linearMultBlockSize);

//P2 == M2[][]

ADD(A21, A22, AResult, HalfSize); //M2=(A21+A22)B11

Strassen(AResult, B11, M2, HalfSize, linearMultBlockSize); //Mul(AResult,B11,M2);

//P3 == M3[][]

SUB(B12, B22, BResult, HalfSize); //M3=A11(B12-B22)

Strassen(A11, BResult, M3, HalfSize, linearMultBlockSize); //Mul(A11,BResult,M3);

//P4 == M4[][]

SUB(B21, B11, BResult, HalfSize); //M4=A22(B21-B11)

Strassen(A22, BResult, M4, HalfSize, linearMultBlockSize); //Mul(A22,BResult,M4);

//P5 == M5[][]

ADD(A11, A12, AResult, HalfSize); //M5=(A11+A12)B22

Strassen(AResult, B22, M5, HalfSize, linearMultBlockSize); //Mul(AResult,B22,M5);

//P6 == M6[][]

SUB(A21, A11, AResult, HalfSize);

ADD(B11, B12, BResult, HalfSize); //M6=(A21-A11)(B11+B12)

Strassen(AResult, BResult, M6, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M6);

//P7 == M7[][]

SUB(A12, A22, AResult, HalfSize);

ADD(B21, B22, BResult, HalfSize); //M7=(A12-A22)(B21+B22)

Strassen(AResult, BResult, M7, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M7);

//C11 = M1 + M4 - M5 + M7;

ADD(M1, M4, AResult, HalfSize);

SUB(M7, M5, BResult, HalfSize);

ADD(AResult, BResult, C11, HalfSize);

//C12 = M3 + M5;

ADD(M3, M5, C12, HalfSize);

//C21 = M2 + M4;

ADD(M2, M4, C21, HalfSize);

//C22 = M1 + M3 - M2 + M6;

ADD(M1, M3, AResult, HalfSize);

SUB(M6, M2, BResult, HalfSize);

ADD(AResult, BResult, C22, HalfSize);

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

MatrixC[i][j] = C11[i][j];

MatrixC[i][j + HalfSize] = C12[i][j];

MatrixC[i + HalfSize][j] = C21[i][j];

MatrixC[i + HalfSize][j + HalfSize] = C22[i][j];

}

}

delete[] A11[0]; delete[] A12[0]; delete[] A21[0]; delete[] A22[0];

delete[] B11[0]; delete[] B12[0]; delete[] B21[0]; delete[] B22[0];

delete[] C11[0]; delete[] C12[0]; delete[] C21[0]; delete[] C22[0];

delete[] M1[0]; delete[] M2[0]; delete[] M3[0]; delete[] M4[0]; delete[] M5[0];

delete[] M6[0]; delete[] M7[0];

delete[] AResult[0];

delete[] BResult[0];

delete[] A11; delete[] A12; delete[] A21; delete[] A22;

delete[] B11; delete[] B12; delete[] B21; delete[] B22;

delete[] C11; delete[] C12; delete[] C21; delete[] C22;

delete[] M1; delete[] M2; delete[] M3; delete[] M4; delete[] M5;

delete[] M6; delete[] M7;

delete[] AResult;

delete[] BResult;

}

return 0;

}

int Strassen\_Guided(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixC, int MatrixSize, int linearMultBlockSize)

{

int HalfSize = MatrixSize / 2;

if (MatrixSize <= linearMultBlockSize)

{

MUL\_Guided(MatrixA, MatrixB, MatrixC, MatrixSize);

}

else

{

double\*\* A11, \*\* A12, \*\* A21, \*\* A22;

double\*\* B11, \*\* B12, \*\* B21, \*\* B22;

double\*\* C11, \*\* C12, \*\* C21, \*\* C22;

double\*\* M1, \*\* M2, \*\* M3, \*\* M4, \*\* M5, \*\* M6, \*\* M7;

double\*\* AResult, \*\* BResult;

A11 = new double\* [HalfSize];

A12 = new double\* [HalfSize];

A21 = new double\* [HalfSize];

A22 = new double\* [HalfSize];

B11 = new double\* [HalfSize];

B12 = new double\* [HalfSize];

B21 = new double\* [HalfSize];

B22 = new double\* [HalfSize];

C11 = new double\* [HalfSize];

C12 = new double\* [HalfSize];

C21 = new double\* [HalfSize];

C22 = new double\* [HalfSize];

M1 = new double\* [HalfSize];

M2 = new double\* [HalfSize];

M3 = new double\* [HalfSize];

M4 = new double\* [HalfSize];

M5 = new double\* [HalfSize];

M6 = new double\* [HalfSize];

M7 = new double\* [HalfSize];

AResult = new double\* [HalfSize];

BResult = new double\* [HalfSize];

A11[0] = new double[HalfSize \* HalfSize];

A12[0] = new double[HalfSize \* HalfSize];

A21[0] = new double[HalfSize \* HalfSize];

A22[0] = new double[HalfSize \* HalfSize];

B11[0] = new double[HalfSize \* HalfSize];

B12[0] = new double[HalfSize \* HalfSize];

B21[0] = new double[HalfSize \* HalfSize];

B22[0] = new double[HalfSize \* HalfSize];

C11[0] = new double[HalfSize \* HalfSize];

C12[0] = new double[HalfSize \* HalfSize];

C21[0] = new double[HalfSize \* HalfSize];

C22[0] = new double[HalfSize \* HalfSize];

M1[0] = new double[HalfSize \* HalfSize];

M2[0] = new double[HalfSize \* HalfSize];

M3[0] = new double[HalfSize \* HalfSize];

M4[0] = new double[HalfSize \* HalfSize];

M5[0] = new double[HalfSize \* HalfSize];

M6[0] = new double[HalfSize \* HalfSize];

M7[0] = new double[HalfSize \* HalfSize];

AResult[0] = new double[HalfSize \* HalfSize];

BResult[0] = new double[HalfSize \* HalfSize];

for (int i = 0; i < HalfSize; i++)

{

A11[i] = &A11[0][i \* HalfSize];

A12[i] = &A12[0][i \* HalfSize];

A21[i] = &A21[0][i \* HalfSize];

A22[i] = &A22[0][i \* HalfSize];

B11[i] = &B11[0][i \* HalfSize];

B12[i] = &B12[0][i \* HalfSize];

B21[i] = &B21[0][i \* HalfSize];

B22[i] = &B22[0][i \* HalfSize];

C11[i] = &C11[0][i \* HalfSize];

C12[i] = &C12[0][i \* HalfSize];

C21[i] = &C21[0][i \* HalfSize];

C22[i] = &C22[0][i \* HalfSize];

M1[i] = &M1[0][i \* HalfSize];

M2[i] = &M2[0][i \* HalfSize];

M3[i] = &M3[0][i \* HalfSize];

M4[i] = &M4[0][i \* HalfSize];

M5[i] = &M5[0][i \* HalfSize];

M6[i] = &M6[0][i \* HalfSize];

M7[i] = &M7[0][i \* HalfSize];

AResult[i] = &AResult[0][i \* HalfSize];

BResult[i] = &BResult[0][i \* HalfSize];

}

/////////////////////////////////////////

#pragma omp parallel for schedule(guided, HalfSize / 10)

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

A11[i][j] = MatrixA[i][j];

A12[i][j] = MatrixA[i][j + HalfSize];

A21[i][j] = MatrixA[i + HalfSize][j];

A22[i][j] = MatrixA[i + HalfSize][j + HalfSize];

B11[i][j] = MatrixB[i][j];

B12[i][j] = MatrixB[i][j + HalfSize];

B21[i][j] = MatrixB[i + HalfSize][j];

B22[i][j] = MatrixB[i + HalfSize][j + HalfSize];

}

}

//P1 == M1[][]

ADD\_Guided(A11, A22, AResult, HalfSize);

ADD\_Guided(B11, B22, BResult, HalfSize);

Strassen\_Guided(AResult, BResult, M1, HalfSize, linearMultBlockSize);

//P2 == M2[][]

ADD\_Guided(A21, A22, AResult, HalfSize); //M2=(A21+A22)B11

Strassen\_Guided(AResult, B11, M2, HalfSize, linearMultBlockSize); //Mul(AResult,B11,M2);

//P3 == M3[][]

SUB\_Guided(B12, B22, BResult, HalfSize); //M3=A11(B12-B22)

Strassen\_Guided(A11, BResult, M3, HalfSize, linearMultBlockSize); //Mul(A11,BResult,M3);

//P4 == M4[][]

SUB\_Guided(B21, B11, BResult, HalfSize); //M4=A22(B21-B11)

Strassen\_Guided(A22, BResult, M4, HalfSize, linearMultBlockSize); //Mul(A22,BResult,M4);

//P5 == M5[][]

ADD\_Guided(A11, A12, AResult, HalfSize); //M5=(A11+A12)B22

Strassen\_Guided(AResult, B22, M5, HalfSize, linearMultBlockSize); //Mul(AResult,B22,M5);

//P6 == M6[][]

SUB\_Guided(A21, A11, AResult, HalfSize);

ADD\_Guided(B11, B12, BResult, HalfSize); //M6=(A21-A11)(B11+B12)

Strassen\_Guided(AResult, BResult, M6, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M6);

//P7 == M7[][]

SUB\_Guided(A12, A22, AResult, HalfSize);

ADD\_Guided(B21, B22, BResult, HalfSize); //M7=(A12-A22)(B21+B22)

Strassen\_Guided(AResult, BResult, M7, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M7);

//C11 = M1 + M4 - M5 + M7;

ADD\_Guided(M1, M4, AResult, HalfSize);

SUB\_Guided(M7, M5, BResult, HalfSize);

ADD\_Guided(AResult, BResult, C11, HalfSize);

//C12 = M3 + M5;

ADD\_Guided(M3, M5, C12, HalfSize);

//C21 = M2 + M4;

ADD\_Guided(M2, M4, C21, HalfSize);

//C22 = M1 + M3 - M2 + M6;

ADD\_Guided(M1, M3, AResult, HalfSize);

SUB\_Guided(M6, M2, BResult, HalfSize);

ADD\_Guided(AResult, BResult, C22, HalfSize);

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

MatrixC[i][j] = C11[i][j];

MatrixC[i][j + HalfSize] = C12[i][j];

MatrixC[i + HalfSize][j] = C21[i][j];

MatrixC[i + HalfSize][j + HalfSize] = C22[i][j];

}

}

delete[] A11[0]; delete[] A12[0]; delete[] A21[0]; delete[] A22[0];

delete[] B11[0]; delete[] B12[0]; delete[] B21[0]; delete[] B22[0];

delete[] C11[0]; delete[] C12[0]; delete[] C21[0]; delete[] C22[0];

delete[] M1[0]; delete[] M2[0]; delete[] M3[0]; delete[] M4[0]; delete[] M5[0];

delete[] M6[0]; delete[] M7[0];

delete[] AResult[0];

delete[] BResult[0];

delete[] A11; delete[] A12; delete[] A21; delete[] A22;

delete[] B11; delete[] B12; delete[] B21; delete[] B22;

delete[] C11; delete[] C12; delete[] C21; delete[] C22;

delete[] M1; delete[] M2; delete[] M3; delete[] M4; delete[] M5;

delete[] M6; delete[] M7;

delete[] AResult;

delete[] BResult;

}

return 0;

}

int Strassen\_Section(double\*\* MatrixA, double\*\* MatrixB, double\*\* MatrixC, int MatrixSize, int linearMultBlockSize)

{

int HalfSize = MatrixSize / 2;

if (MatrixSize <= linearMultBlockSize)

{

MUL\_Section(MatrixA, MatrixB, MatrixC, MatrixSize);

}

else

{

double\*\* A11, \*\* A12, \*\* A21, \*\* A22;

double\*\* B11, \*\* B12, \*\* B21, \*\* B22;

double\*\* C11, \*\* C12, \*\* C21, \*\* C22;

double\*\* M1, \*\* M2, \*\* M3, \*\* M4, \*\* M5, \*\* M6, \*\* M7;

double\*\* AResult, \*\* BResult;

A11 = new double\* [HalfSize];

A12 = new double\* [HalfSize];

A21 = new double\* [HalfSize];

A22 = new double\* [HalfSize];

B11 = new double\* [HalfSize];

B12 = new double\* [HalfSize];

B21 = new double\* [HalfSize];

B22 = new double\* [HalfSize];

C11 = new double\* [HalfSize];

C12 = new double\* [HalfSize];

C21 = new double\* [HalfSize];

C22 = new double\* [HalfSize];

M1 = new double\* [HalfSize];

M2 = new double\* [HalfSize];

M3 = new double\* [HalfSize];

M4 = new double\* [HalfSize];

M5 = new double\* [HalfSize];

M6 = new double\* [HalfSize];

M7 = new double\* [HalfSize];

AResult = new double\* [HalfSize];

BResult = new double\* [HalfSize];

A11[0] = new double[HalfSize \* HalfSize];

A12[0] = new double[HalfSize \* HalfSize];

A21[0] = new double[HalfSize \* HalfSize];

A22[0] = new double[HalfSize \* HalfSize];

B11[0] = new double[HalfSize \* HalfSize];

B12[0] = new double[HalfSize \* HalfSize];

B21[0] = new double[HalfSize \* HalfSize];

B22[0] = new double[HalfSize \* HalfSize];

C11[0] = new double[HalfSize \* HalfSize];

C12[0] = new double[HalfSize \* HalfSize];

C21[0] = new double[HalfSize \* HalfSize];

C22[0] = new double[HalfSize \* HalfSize];

M1[0] = new double[HalfSize \* HalfSize];

M2[0] = new double[HalfSize \* HalfSize];

M3[0] = new double[HalfSize \* HalfSize];

M4[0] = new double[HalfSize \* HalfSize];

M5[0] = new double[HalfSize \* HalfSize];

M6[0] = new double[HalfSize \* HalfSize];

M7[0] = new double[HalfSize \* HalfSize];

AResult[0] = new double[HalfSize \* HalfSize];

BResult[0] = new double[HalfSize \* HalfSize];

for (int i = 0; i < HalfSize; i++)

{

A11[i] = &A11[0][i \* HalfSize];

A12[i] = &A12[0][i \* HalfSize];

A21[i] = &A21[0][i \* HalfSize];

A22[i] = &A22[0][i \* HalfSize];

B11[i] = &B11[0][i \* HalfSize];

B12[i] = &B12[0][i \* HalfSize];

B21[i] = &B21[0][i \* HalfSize];

B22[i] = &B22[0][i \* HalfSize];

C11[i] = &C11[0][i \* HalfSize];

C12[i] = &C12[0][i \* HalfSize];

C21[i] = &C21[0][i \* HalfSize];

C22[i] = &C22[0][i \* HalfSize];

M1[i] = &M1[0][i \* HalfSize];

M2[i] = &M2[0][i \* HalfSize];

M3[i] = &M3[0][i \* HalfSize];

M4[i] = &M4[0][i \* HalfSize];

M5[i] = &M5[0][i \* HalfSize];

M6[i] = &M6[0][i \* HalfSize];

M7[i] = &M7[0][i \* HalfSize];

AResult[i] = &AResult[0][i \* HalfSize];

BResult[i] = &BResult[0][i \* HalfSize];

}

/////////////////////////////////////////

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

A11[i][j] = MatrixA[i][j];

A12[i][j] = MatrixA[i][j + HalfSize];

A21[i][j] = MatrixA[i + HalfSize][j];

A22[i][j] = MatrixA[i + HalfSize][j + HalfSize];

B11[i][j] = MatrixB[i][j];

B12[i][j] = MatrixB[i][j + HalfSize];

B21[i][j] = MatrixB[i + HalfSize][j];

B22[i][j] = MatrixB[i + HalfSize][j + HalfSize];

}

}

//P1 == M1[][]

ADD\_Sections(A11, A22, AResult, HalfSize);

ADD\_Sections(B11, B22, BResult, HalfSize);

Strassen\_Section(AResult, BResult, M1, HalfSize, linearMultBlockSize);

//P2 == M2[][]

ADD\_Sections(A21, A22, AResult, HalfSize); //M2=(A21+A22)B11

Strassen\_Section(AResult, B11, M2, HalfSize, linearMultBlockSize); //Mul(AResult,B11,M2);

//P3 == M3[][]

SUB\_Sections(B12, B22, BResult, HalfSize); //M3=A11(B12-B22)

Strassen\_Section(A11, BResult, M3, HalfSize, linearMultBlockSize); //Mul(A11,BResult,M3);

//P4 == M4[][]

SUB\_Sections(B21, B11, BResult, HalfSize); //M4=A22(B21-B11)

Strassen\_Section(A22, BResult, M4, HalfSize, linearMultBlockSize); //Mul(A22,BResult,M4);

//P5 == M5[][]

ADD\_Sections(A11, A12, AResult, HalfSize); //M5=(A11+A12)B22

Strassen\_Section(AResult, B22, M5, HalfSize, linearMultBlockSize); //Mul(AResult,B22,M5);

//P6 == M6[][]

SUB\_Sections(A21, A11, AResult, HalfSize);

ADD\_Sections(B11, B12, BResult, HalfSize); //M6=(A21-A11)(B11+B12)

Strassen\_Section(AResult, BResult, M6, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M6);

//P7 == M7[][]

SUB\_Sections(A12, A22, AResult, HalfSize);

ADD\_Sections(B21, B22, BResult, HalfSize); //M7=(A12-A22)(B21+B22)

Strassen\_Section(AResult, BResult, M7, HalfSize, linearMultBlockSize); //Mul(AResult,BResult,M7);

//C11 = M1 + M4 - M5 + M7;

ADD\_Sections(M1, M4, AResult, HalfSize);

SUB\_Sections(M7, M5, BResult, HalfSize);

ADD\_Sections(AResult, BResult, C11, HalfSize);

//C12 = M3 + M5;

ADD\_Sections(M3, M5, C12, HalfSize);

//C21 = M2 + M4;

ADD\_Sections(M2, M4, C21, HalfSize);

//C22 = M1 + M3 - M2 + M6;

ADD\_Sections(M1, M3, AResult, HalfSize);

SUB\_Sections(M6, M2, BResult, HalfSize);

ADD\_Sections(AResult, BResult, C22, HalfSize);

for (int i = 0; i < HalfSize; i++)

{

for (int j = 0; j < HalfSize; j++)

{

MatrixC[i][j] = C11[i][j];

MatrixC[i][j + HalfSize] = C12[i][j];

MatrixC[i + HalfSize][j] = C21[i][j];

MatrixC[i + HalfSize][j + HalfSize] = C22[i][j];

}

}

delete[] A11[0]; delete[] A12[0]; delete[] A21[0]; delete[] A22[0];

delete[] B11[0]; delete[] B12[0]; delete[] B21[0]; delete[] B22[0];

delete[] C11[0]; delete[] C12[0]; delete[] C21[0]; delete[] C22[0];

delete[] M1[0]; delete[] M2[0]; delete[] M3[0]; delete[] M4[0]; delete[] M5[0];

delete[] M6[0]; delete[] M7[0];

delete[] AResult[0];

delete[] BResult[0];

delete[] A11; delete[] A12; delete[] A21; delete[] A22;

delete[] B11; delete[] B12; delete[] B21; delete[] B22;

delete[] C11; delete[] C12; delete[] C21; delete[] C22;

delete[] M1; delete[] M2; delete[] M3; delete[] M4; delete[] M5;

delete[] M6; delete[] M7;

delete[] AResult;

delete[] BResult;

}

return 0;

}

int Validate\_size(int size, int linearMultBlockSize)

{

int tms = size;

while (tms > linearMultBlockSize)

{

if (tms % 2 == 0)

{

tms /= 2;

}

else

return 0;

}

return 1;

}

int Find\_Valid\_Size(int size, int linearMultBlockSize)

{

int newsize = size;

while (Validate\_size(newsize, linearMultBlockSize) == 0)

{

newsize++;

}

return newsize;

}

double\*\* ExpandMatrixToSize(double\*\* matrix, int sizeA, int sizeB, int NewSize)

{

double\*\* NewMatr = new double\* [NewSize];

for (int i = 0; i < NewSize; i++)

{

NewMatr[i] = new double[NewSize];

for (int j = 0; j < NewSize; j++)

{

NewMatr[i][j] = 0;

}

}

for (int i = 0; i < sizeA; i++)

for (int j = 0; j < sizeB; j++)

{

NewMatr[i][j] = matrix[i][j];

}

return NewMatr;

}

double Shtrassen\_Multiplication(double\*\* A, double\*\* B, double\*\* C, int sizeA, int sizeB, int sizeC)

{

int linearMultBlockSize = 64;

double t\_st = 0, t\_ed = -1;

int size = sizeA;

double time = 0;

if (sizeA == sizeB && sizeA == sizeC && Validate\_size(size, linearMultBlockSize))

{

t\_st = omp\_get\_wtime();

Strassen(A, B, C, size, linearMultBlockSize);

time = omp\_get\_wtime() - t\_st;

return time;

}

else

{

if (size < sizeB) size = sizeB;

if (size < sizeC) size = sizeC;

if (size != Find\_Valid\_Size(size, linearMultBlockSize))

{

size = Find\_Valid\_Size(size, linearMultBlockSize);

}

t\_st = omp\_get\_wtime();

#pragma align(4)

double\*\* TA = ExpandMatrixToSize(A, sizeA, sizeB, size);

#pragma align(4)

double\*\* TB = ExpandMatrixToSize(B, sizeB, sizeC, size);

#pragma align(4)

double\*\* TC = ExpandMatrixToSize(C, 0, 0, size);

t\_st = omp\_get\_wtime();

Strassen(TA, TB, TC, size, linearMultBlockSize);

t\_ed = omp\_get\_wtime() - t\_st;

return t\_ed;

}

}

double Shtrassen\_Multiplication\_Guided(double\*\* A, double\*\* B, double\*\* C, int sizeA, int sizeB, int sizeC)

{

int linearMultBlockSize = 64;

double t\_st = 0, t\_ed = -1;

int size = sizeA;

double time = 0;

if (sizeA == sizeB && sizeA == sizeC && Validate\_size(size, linearMultBlockSize))

{

t\_st = omp\_get\_wtime();

Strassen\_Guided(A, B, C, size, linearMultBlockSize);

time = omp\_get\_wtime() - t\_st;

return time;

}

else

{

if (size < sizeB) size = sizeB;

if (size < sizeC) size = sizeC;

if (size != Find\_Valid\_Size(size, linearMultBlockSize))

{

size = Find\_Valid\_Size(size, linearMultBlockSize);

}

t\_st = omp\_get\_wtime();

#pragma align(4)

double\*\* TA = ExpandMatrixToSize(A, sizeA, sizeB, size);

#pragma align(4)

double\*\* TB = ExpandMatrixToSize(B, sizeB, sizeC, size);

#pragma align(4)

double\*\* TC = ExpandMatrixToSize(C, 0, 0, size);

t\_st = omp\_get\_wtime();

Strassen\_Guided(TA, TB, TC, size, linearMultBlockSize);

t\_ed = omp\_get\_wtime() - t\_st;

return t\_ed;

}

}

double Shtrassen\_Multiplication\_Section(double\*\* A, double\*\* B, double\*\* C, int sizeA, int sizeB, int sizeC)

{

int linearMultBlockSize = 64;

double t\_st = 0, t\_ed = -1;

int size = sizeA;

double time = 0;

if (sizeA == sizeB && sizeA == sizeC && Validate\_size(size, linearMultBlockSize))

{

t\_st = omp\_get\_wtime();

Strassen\_Section(A, B, C, size, linearMultBlockSize);

time = omp\_get\_wtime() - t\_st;

return time;

}

else

{

if (size < sizeB) size = sizeB;

if (size < sizeC) size = sizeC;

if (size != Find\_Valid\_Size(size, linearMultBlockSize))

{

size = Find\_Valid\_Size(size, linearMultBlockSize);

}

t\_st = omp\_get\_wtime();

#pragma align(4)

double\*\* TA = ExpandMatrixToSize(A, sizeA, sizeB, size);

#pragma align(4)

double\*\* TB = ExpandMatrixToSize(B, sizeB, sizeC, size);

#pragma align(4)

double\*\* TC = ExpandMatrixToSize(C, 0, 0, size);

t\_st = omp\_get\_wtime();

Strassen\_Section(TA, TB, TC, size, linearMultBlockSize);

t\_ed = omp\_get\_wtime() - t\_st;

return t\_ed;

}

}

double TestFillMatr(double\*\*& matrix1, double\*\*& empty, double\*\*& empty1, int& sizeA, int& sizeB, int& empty2) {

return(FillMatr(matrix1, sizeA, sizeB));

}

double TestFillMatrParallelguided(double\*\*& empty, double\*\*& matrix2, double\*\*& empty1, int& sizeB, int& sizeA, int& empty2)

{

return FillMatrParallelGuided(matrix2, sizeA, sizeB);

}

double TestFillMatrParallelstatic(double\*\*& matrix1, double\*\*& empty, double\*\*& empty1, int& sizeA, int& sizeB, int& empty2)

{

return FillMatrParallelStatic(matrix1, sizeA, sizeB);

}

double TestFillMatrParalleldynamic(double\*\*& empty, double\*\*& matrix2, double\*\*& empty1, int& sizeB, int& sizeA, int& empty2) {

return FillMatrParallelDynamic(matrix2, sizeA, sizeB);

}

double TestMultiplyMatrV4(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return MultiplyMatrV4(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestMultiplyMatrV4Parrallelstatic(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return MultiplyMatrV4Parrallelstatic(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestMultiplyMatrV4Parralleldynamic(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return MultiplyMatrV4Parralleldynamic(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestMultiplyMatrV4Parrallelguided(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return MultiplyMatrV4Parrallelguided(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestShtrassen\_Multiplication(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return Shtrassen\_Multiplication(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestShtrassen\_Multiplication\_Guided(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return Shtrassen\_Multiplication\_Guided(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double TestShtrassen\_Multiplication\_Section(double\*\*& matrix1, double\*\*& matrix2, double\*\*& matrix3, int& sizeA, int& sizeB, int& sizeC)

{

return Shtrassen\_Multiplication\_Guided(matrix1, matrix2, matrix3, sizeA, sizeB, sizeC);

}

double AvgTrustedInterval(double& avg, vector<double>& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

double TestIter(void\* Funct, double\*\*& a, double\*\*& b, double\*\*& c, int& A, int& B, int& C)

{

double curtime = 0, avgTime = 0, avgTimeT = 0, correctAVG = 0;

int iterations = 4;

vector<double> Times(iterations);

c = new double\* [A];

c[0] = new double[A \* C];

for (int i = 1; i < A; i++)

c[i] = &c[0][i \* C];

for (int i = 0; i < iterations; i++)

{

curtime = (((TestFunctTempl)Funct)(a, b, c, A, B, C)) \* 1000;

Times[i] = curtime;

avgTime += curtime;

cout << "+";

}

cout << endl;

avgTime /= iterations;

cout << "AvgTime:" << avgTime << endl;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

cout << "AvgTimeTrusted:" << avgTimeT << endl;

delete[] c[0];

delete[] c;

return avgTimeT;

}

void test\_functions(void\*\* Functions, vector<string> fNames)

{

int nd = 0;

double\*\* a, \*\* b, \*\* c;

int A = 500, B = 500, C = 500;

double times[4][11][3];

for (int A = 500; A <= 950; A += 150)

{

a = new double\* [A];

b = new double\* [B];

a[0] = new double[A \* B];

b[0] = new double[B \* C];

for (int i = 1; i < A; i++)

a[i] = &a[0][i \* B];

for (int i = 1; i < B; i++)

b[i] = &b[0][i \* C];

for (int threads = 1; threads <= 4; threads++)

{

omp\_set\_num\_threads(threads);

for (int alg = 0; alg <= 10; alg++)

{

if (threads == 1)

{

if (alg == 0 || alg == 4 || alg == 8) {

times[nd][alg][0] = TestIter(Functions[alg], a, b, c, A, B, C);

times[nd][alg][1] = times[nd][alg][0];

times[nd][alg][2] = times[nd][alg][0];

cout << fNames[alg] << endl;

}

}

else

{

if (alg != 0 && alg != 4 && alg != 8)

{

times[nd][alg][threads - 2] = TestIter(Functions[alg], a, b, c, A, B, C);

cout << fNames[alg] << endl;

}

}

}

}

delete[] a[0];

delete[] a;

delete[] b[0];

delete[] b;

nd++;

B += 100;

C += 150;

}

ofstream fout("output.txt");

fout.imbue(locale("Russian"));

for (int ND = 0; ND < 4; ND++)

{

switch (ND)

{

case 0:

cout << "\n----------matrix 800х800----------" << endl;

break;

case 1:

cout << "\n----------matrix 1000х900 ----------" << endl;

break;

case 2:

cout << "\n----------matrix 1200x1000 ----------" << endl;

break;

case 3:

cout << "\n----------matrix 1400x1100 ----------" << endl;

break;

default:

break;

}

for (int alg = 0; alg <= 10; alg++)

{

for (int threads = 1; threads <= 4; threads++)

{

if (threads == 1)

{

if (alg == 0 || alg == 4 || alg == 8) {

cout << "Thread " << threads << " --------------" << endl;

cout << fNames[alg] << "\t" << times[ND][alg][0] << " ms." << endl;

fout << times[ND][alg][0] << endl;

}

}

else

{

if (alg != 0 && alg != 4 && alg != 8)

{

cout << "Thread " << threads << " --------------" << endl;

cout << fNames[alg] << "\t" << times[ND][alg][threads - 2] << " ms." << endl;

fout << times[ND][alg][threads - 2] << endl;

}

}

}

}

}

fout.close();

}

void main() {

void\*\* Functions = new void\* [11] { TestFillMatr, TestFillMatrParallelstatic, TestFillMatrParalleldynamic, TestFillMatrParallelguided, TestMultiplyMatrV4, TestMultiplyMatrV4Parrallelstatic, TestMultiplyMatrV4Parralleldynamic, TestMultiplyMatrV4Parrallelguided, TestShtrassen\_Multiplication, TestShtrassen\_Multiplication\_Guided, TestShtrassen\_Multiplication\_Section };

vector<string> function\_names = { "Consistent filling","Parallel filling schedule(static)","Parallel filling schedule(dynamic)","Parallel filling schedule(guided)","Consistent Multiply",

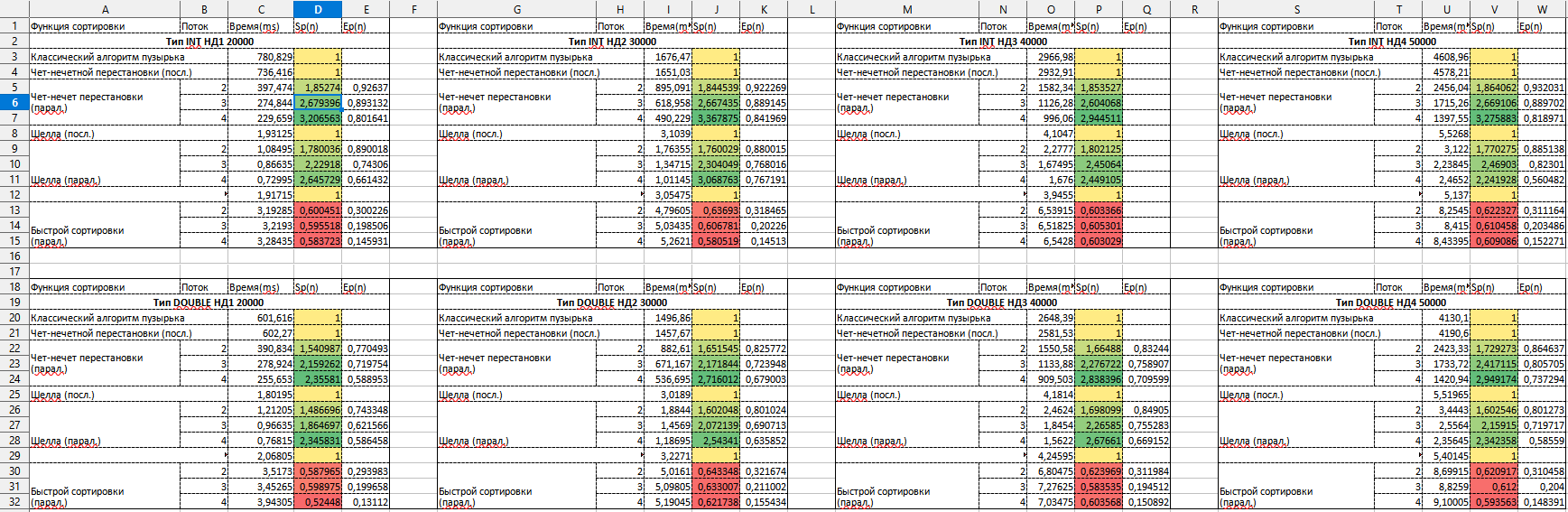
"Parallel Multiply schedule(static)", "Parallel Multiply schedule(dynamic)", "Parallel Multiply schedule(guided)", "Shtrassen Multiply", "Shtrassen Guided", "Shtrassen Section" };

test\_functions(Functions, function\_names);

}

Задание 2.3.

Таблица 1 – результаты замеров времени работы первого задания с разными наборами функций



Вывод: в ходе выполнения работы были получены замеры времени выполнения работы функций в последовательном и многопоточном режимах. В первом задании работа функций в параллельном режиме с использованием алгоритма Odd-Even Sort имеет наибольшее ускорение. Во втором задании работа функций в параллельном режиме с использованием алгоритмов Shell Sort Sections, Shell Sort Parallel For, Quick Sort Sections, Quick Sort Parallel For на 4 потоках имеет наибольшее ускорение.

# ВЫВОДЫ

Были изучены различные алгоритмы сортировки и их параллельные варианты реализации, был использован медианный фильтр для обработки изображений.