Министерство образования и науки Российской Федерации

Нижегородский государственный университет им. Н.И. Лобачевского

**Институт информационных технологий, математики и механики**

**Отчет по лабораторной работе**

**«Умножение плотных матриц. Элементы типа double. Алгоритм Штрассена»**

**Выполнил**:

Касмазюк Никита (381606-1)

**Проверил**:

Доцент кафедры МОСТ ИИТММ

Кустикова В.Д.

Нижний Новгород

2018

Оглавление

* Введение
* Постановка задачи
* Метод решения
* Схема распараллеливания
* Описание программной реализации
* Подтверждение корректности
* Результаты экспериментов
* Заключение
* Приложение

**Введение:**

Фо́лькер Штра́ссен (нем. *Volker Strassen*; род. 29 апреля 1936, Дюссельдорф, Германия)  — немецкий математик, почетный профессор кафедры математики и статистики Констанцского университета.

Пусть даны две матрицы A и B размерности n x n. По определению результатом умножения двух матриц является матрица C размерности n x n, где каждый элемент матрицы вычисляется по формуле:

, где 0 ≤ i < n, 0 ≤ j < n

Этот алгоритм предполагает выполнение n3 операций умножения и n2(n-1) операций сложения элементов исходных матриц (O(n3) операций). Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью. В частности, к таким алгоритмам относится алгоритм Штрассена. В данном алгоритме каждое восьмое умножение заменяется сложениями, что приводит к оценке сложности () = ().

Тем не менее, самый асимптотически быстрый из известных на сегодняшний день способ умножения матриц был опубликован Копперсмитом и Виноградом в 1987-м году и позволяет перемножать квадратные матрицы за время Θ(n2.38). На практике же метод Штрассена гораздо проще программируется и имеет меньшую константу в оценке трудоемкости.

Бло́чная (кле́точная) ма́трица — представление матрицы, при котором она рассекается вертикальными и горизонтальными линиями на прямоугольные части — блоки (клетки)

Блочная матрица называется блочно-диагональной, если она по своему виду совпадает с обычной диагональной матрицей, то есть, как если бы блоки  Ai j  представляли собой обычные матричные элементы.   
  
Блочная матрица называется блочно-треугольной, если она по своему виду совпадает с обычной треугольной матрицей.

**Постановка задачи**

Разработать и реализовать программу для перемножения двух матриц, в частности алгоритмом Штрассена.

Данная программа должна поддерживать следующие возможности:

* Ввод размера матрицы.
* Генерация матрицы.
* Подсчет времени исполнения последовательной и параллельной части.
* Проверка корректности результата.

**Метод решения**



Вначале мы проверяем, является ли размер умножаемых матриц *n* натуральной степенью двойки. Если нет, мы дополняем исходные матрицы дополнительными нулевыми строками и столбцами, оставляя на главной диагонали единицы. При этом мы получаем удобные для рекурсивного умножения размеры, но теряем в эффективности за счет дополнительных ненужных умножений. Алгоритм Штрассена умножает две (*n*×*n*)-матрицы *A* и *B* следующим образом:

1. Каждая из матриц *A* и *B* разбивается на 4 блока.
2. Строятся 14 матриц *A*1, *B*1, *A*2, *B*2, …, *A*7, *B*7 размера (*n*/2×*n*⁄2) (для чего нужно Θ(*n*2) операций сложения/вычитания чисел).
3. Рекурсивно вычисляются 7 произведений матриц меньшего размера *Pi* = *AiBi* (*i* = 1, …, 7).
4. Вычисляются части *r*, *s*, *t*, *u* искомой матрицы *C*. Они являются линейными комбинациями матриц *Pi* с коэффициентами из множества {−1, 0, 1}, и вычисление их требует Θ(*n*2) операций сложения/вычитания чисел.

**Схема распараллеливания**

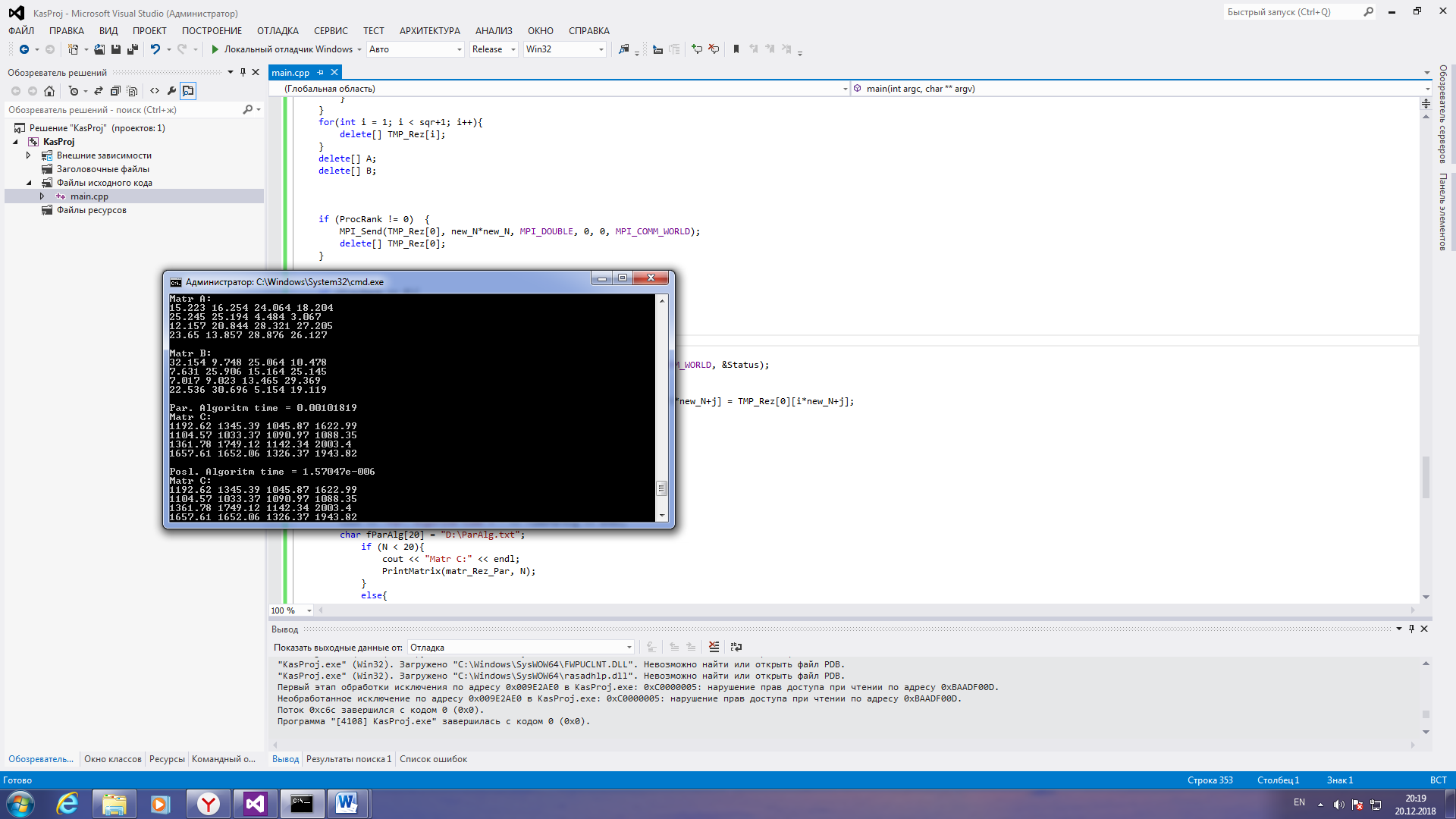
* Мы разбиваем наши матрицы А и В на маленькие матрицы (блоки).
* Далее мы рассылаем эти данные другим процессам.
* Вычисление каждым процессом своего куска матрицы. (Перемножение двух маленьких блоков по последовательному алгоритму Штрассена)
* Отправляем эти результаты на нулевой процесс.
* Теперь мы записываем результат своей работы и принимаем и записываем результат работы других процессов.
* Освобождаем память всех дополнительных задействованных переменных и матриц.

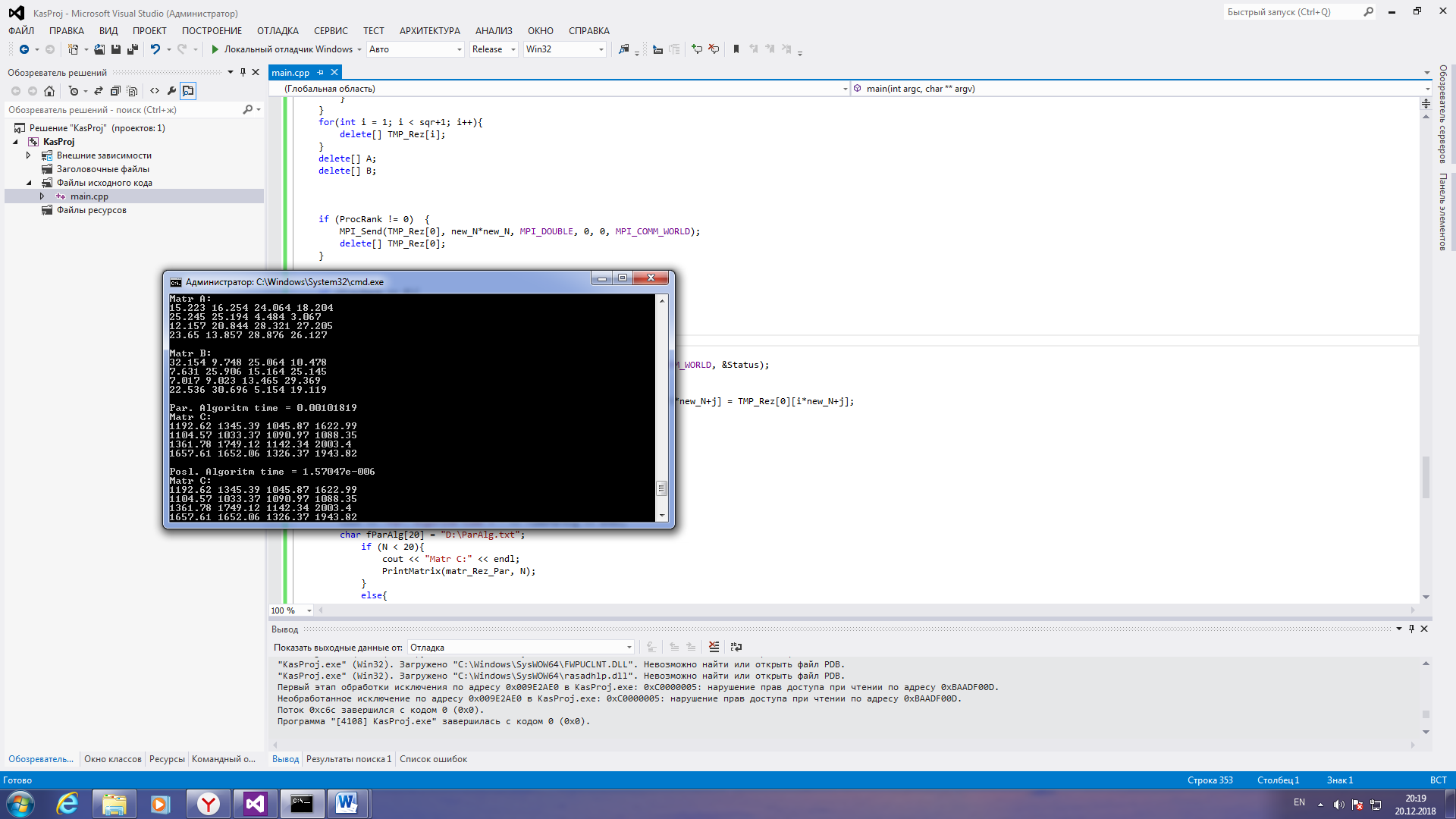
**Описание программной реализации**

Запуск программы осуществляется через командную строку с помощью следующей команды: *mpiexec –n X Program.exe N*, где:

* X – кол-во процессов.
* N – Степень в которую мы возводим число 2. Результат возведения - это размер матриц A и B.
* Program.exe – Название исполняемого файла.

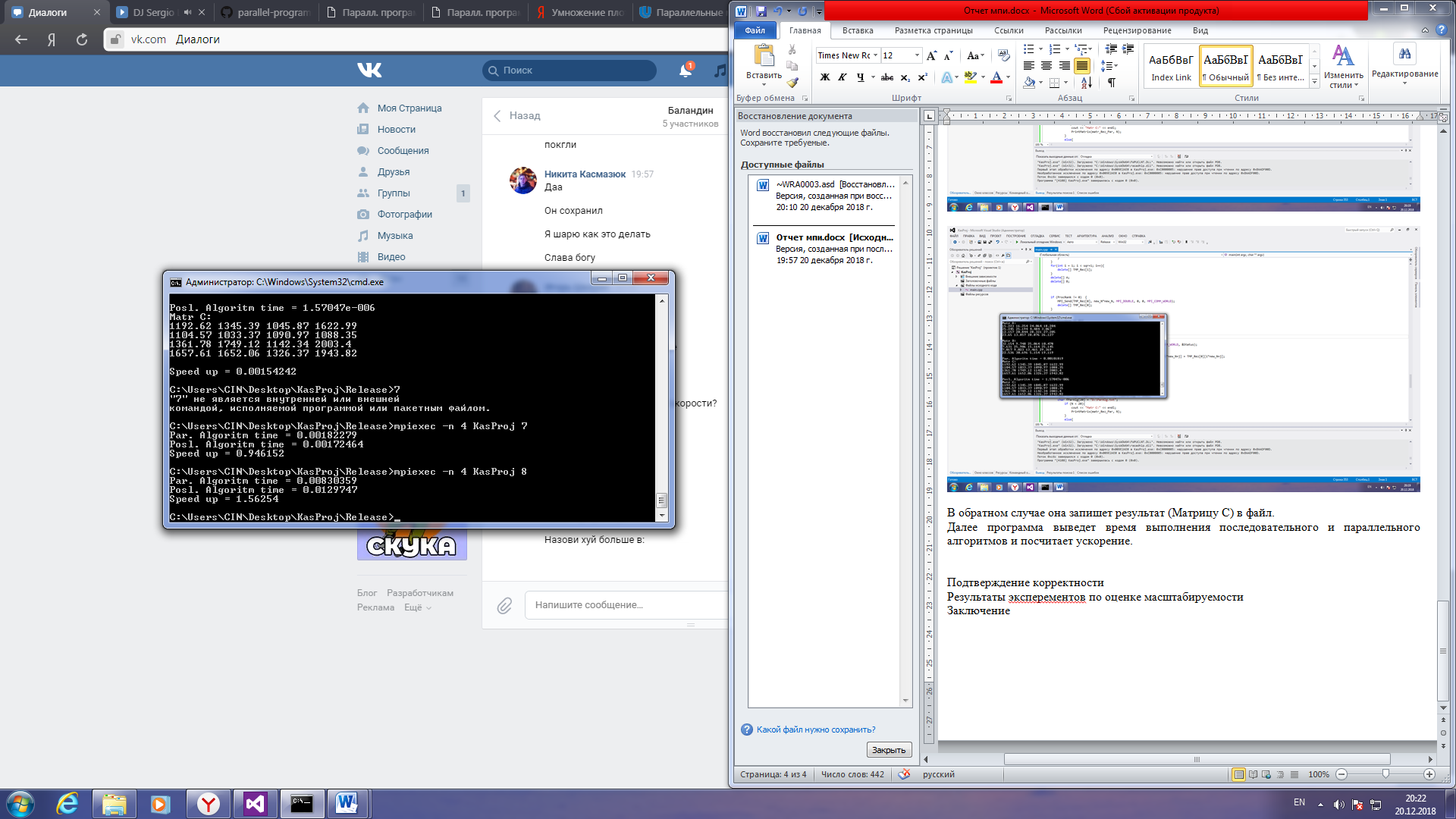
Если матрицы A, B и C размера меньше чем 10, то программа выведет их на экран.





В обратном случае она запишет результат (Матрицу С) в файл.

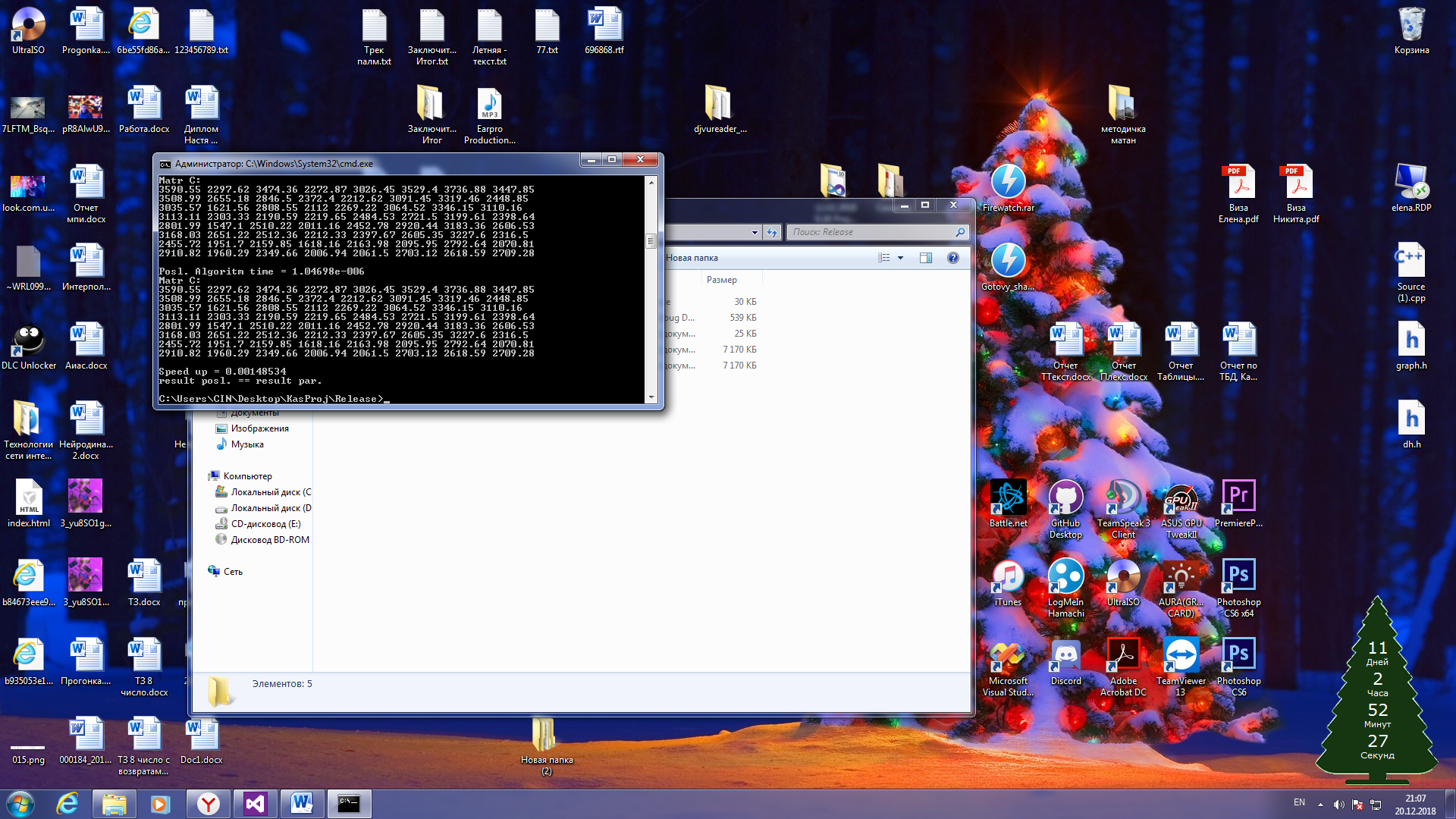
Далее программа выведет время выполнения последовательного и параллельного алгоритмов и посчитает ускорение.



**Подтверждение корректности**

Подтверждение корректности можно получить, визуально сравнив выведенные матрицы на экран. Для удобства добавлена функция, которая сравнивает результат последовательного алгоритма и параллельного и выводит на экран, равны ли результаты.

В случае верного решения, когда они совпадают, то она выдаст такой результат:



Иначе программа сообщит об ошибке, что наши результаты не совпадают. А так же она выдаст те два элемента, которые оказались не равны.

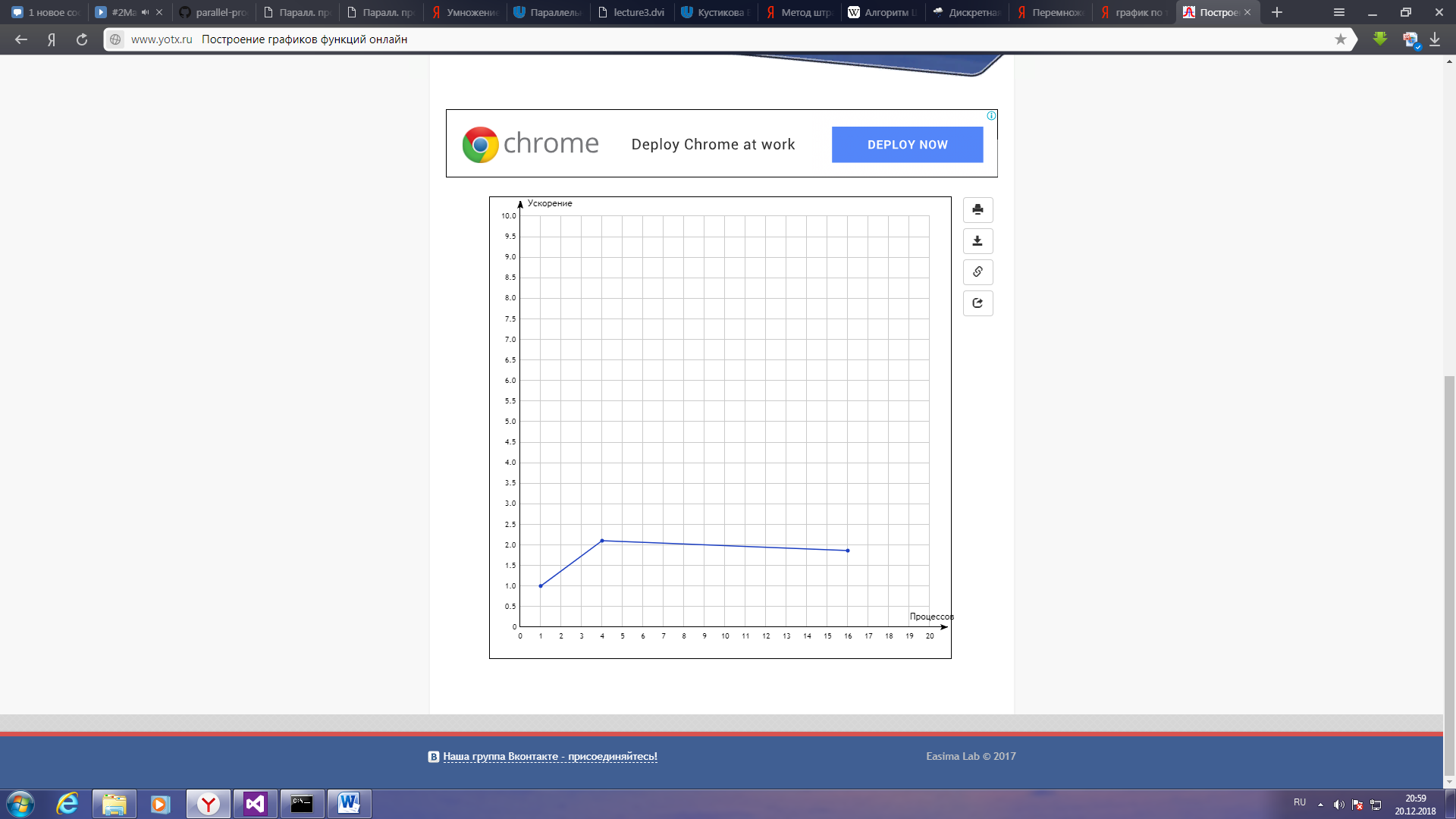
**Результаты экспериментов**

Для данного теста использовались матрицы размера 2048x2048.

Результаты экспериментов:

|  |  |  |  |
| --- | --- | --- | --- |
| Кол-во процессов | Последовательный алгоритм | 4 процесса | 16 процессов |
| Время работы | 5.086 | 2.418 | 2.720 |
| Ускорение | 1 | 2.103 | 1.862 |

На 4 процессах быстрее, чем на 16, т.к. процессор, на котором проводились тесты – четырехядерный.



**Заключение**

В ходе выполнения лабораторной работы мы ознакомились с алгоритмами перемножения матриц. В частности, изучили алгоритм Штрассена. Как последовательный, так и параллельный. По результатам экспериментов видно, что распараллеливание принесло значимое ускорение работы программы.

В результате все поставленные задачи были выполнены, цели достигнуты. Мы приобрели важные навыки в применении параллельного программирования с помощью MPI.

**Приложение**

#include <iostream>

#include <math.h>

#include <time.h>

#include <mpi.h>

#include <fstream>

using namespace std;

double\* CreateVectorMatrix(int N){

double \*matrix = new double[N\*N];

return matrix;

}

void PrintMatrix (double\* matrix,int N){

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

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

cout << matrix[i\*N+j] << " ";

cout << endl;

}

cout << endl;

}

void fPrintMatrix(double\* matrix, int N, char name[20]){

ofstream fout(name, ios\_base::trunc);

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

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

fout << matrix[i\*N+j] << " ";

fout << endl;

}

fout.close();

}

void RandMatrix(double\* matrix1, double\* matrix2, int N){

srand (time (0));

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

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

matrix1[i\*N+j] = (double)rand() / 1000.0;

matrix2[i\*N+j] = (double)rand() / 1000.0;

}

}

double\* Trivial\_alghorithm(double\* matrix1, double\* matrix2, int N){

double\* Rez = CreateVectorMatrix(N);

double sum;

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

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

sum = 0;

for(int k = 0; k < N; k++)

sum += matrix1[i\*N+k] \* matrix2[k\*N+j];

Rez[i\*N+j]= sum;

}

return Rez;

}

double\* Add(double\* matrix1, double\* matrix2, int N){

double\* Rez = CreateVectorMatrix(N);

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

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

Rez[i\*N+j] = matrix1[i\*N+j] + matrix2[i\*N+j];

return Rez;

}

double\* Add(double\* matrix1, double\* matrix2, double\* matrix3, double\* matrix4, int N){

double\* Rez = CreateVectorMatrix(N);

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

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

Rez[i\*N+j] = matrix1[i\*N+j] + matrix2[i\*N+j] + matrix3[i\*N+j] + matrix4[i\*N+j];

return Rez;

}

double\* Sub(double\* matrix1, double\* matrix2, int N){

double\* Rez = CreateVectorMatrix(N);

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

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

Rez[i\*N+j] = matrix1[i\*N+j] - matrix2[i\*N+j];

return Rez;

}

double\* Sub(double\* matrix1, double\* matrix2, double\* matrix3, double\* matrix4, int N){

double\* Rez = CreateVectorMatrix(N);

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

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

Rez[i\*N+j] = matrix1[i\*N+j] + matrix2[i\*N+j] + matrix3[i\*N+j] - matrix4[i\*N+j];

return Rez;

}

double\* Shtr\_alg(double\* matrix1, double\* matrix2, int N, int threshold){

double\* Rez;

if(N <= threshold){

Rez = Trivial\_alghorithm(matrix1, matrix2, N);

}

else{

Rez = CreateVectorMatrix(N);

N = N/2;

double\* A[4];

double\* B[4];

double\* C[4];

double\* P[7];

for(int i = 0; i < 4; i++){

A[i] = CreateVectorMatrix(N);

B[i] = CreateVectorMatrix(N);

}

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

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

int index\_new = i\*N+j,index\_old = 2\*i\*N+j, N\_N = 2\*N\*N;

A[0][index\_new] = matrix1[index\_old];

A[1][index\_new] = matrix1[index\_old+N];

A[2][index\_new] = matrix1[index\_old+N\_N];

A[3][index\_new] = matrix1[index\_old+N\_N+N];

B[0][index\_new] = matrix2[index\_old];

B[1][index\_new] = matrix2[index\_old+N];

B[2][index\_new] = matrix2[index\_old+N\_N];

B[3][index\_new] = matrix2[index\_old+N\_N+N];

}

double \*TMP = Add(A[0], A[3], N);

double \*\_TMP = Add(B[0], B[3], N);

P[0] = Shtr\_alg(TMP, \_TMP, N,threshold);

delete[] TMP;

delete[] \_TMP;

TMP = Add(A[2], A[3], N);

P[1] = Shtr\_alg(TMP, B[0], N, threshold);

delete[] TMP;

TMP = Sub(B[1], B[3], N);

P[2] = Shtr\_alg(A[0], TMP, N, threshold);

delete[] TMP;

TMP = Sub(B[2], B[0], N);

P[3] = Shtr\_alg(A[3], TMP, N, threshold);

delete[] TMP;

TMP = Add(A[0], A[1], N);

P[4] = Shtr\_alg(TMP, B[3], N, threshold);

delete[] TMP;

TMP = Sub(A[2], A[0], N);

\_TMP = Add(B[0], B[1], N);

P[5] = Shtr\_alg(TMP, \_TMP, N, threshold);

delete[] TMP;

delete[] \_TMP;

TMP = Sub(A[1], A[3], N);

\_TMP = Add(B[2], B[3], N);

P[6] = Shtr\_alg(TMP, \_TMP, N, threshold);

delete[] TMP;

delete[] \_TMP;

C[0] = Sub(P[0], P[3], P[6], P[4], N);

C[1] = Add(P[2], P[4], N);

C[2] = Add(P[1], P[3], N);

C[3] = Sub(P[0], P[2], P[5], P[1], N);

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

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

Rez[i\*2\*N+j] = C[0][i\*N+j];

Rez[i\*2\*N+j+N] = C[1][i\*N+j];

Rez[i\*2\*N+j+2\*N\*N] = C[2][i\*N+j];

Rez[i\*2\*N+j+2\*N\*N+N] = C[3][i\*N+j];

}

for(int i = 0; i < 4; i++){

delete[] A[i];

delete[] B[i];

delete[] C[i];

}

for(int i = 0; i < 7; i++)

delete[] P[i];

}

return Rez;

}

int main(int argc, char\*\* argv){

double \*matr\_A, \*matr\_B, \*matr\_Rez\_Seq, \*matr\_Rez\_Par, \*\*A, \*\*B, \*\*TMP\_Rez;

int N, thr = 64, sqr, new\_N;

int ProcSize = 1, ProcRank = 0;

MPI\_Status Status;

double EndSeqAlg = 0;

double StartSeqAlg = 0;

double EndParAlg = 0;

double StartParAlg = 0;

double TimeSeqAlg = 0;

double TimeParAlg = 0;

// ----------------------- //

setlocale(LC\_ALL, "Russian");

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcSize);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

if (ProcRank == 0){

int k = 0;

if (argc > 1)

k = atoi (argv[1]);

N = (int)pow(2.0,k);

if(argc > 2)

thr = atoi(argv[2]);

matr\_A=CreateVectorMatrix(N);

matr\_B=CreateVectorMatrix(N);

matr\_Rez\_Par=CreateVectorMatrix(N);

RandMatrix(matr\_A, matr\_B, N);

char fA[20] = "D:\MatrA.txt";

char fB[20] = "D:\MatrB.txt";

if (N < 10){

cout << "Matr A:" << endl;

PrintMatrix(matr\_A, N);

cout << "Matr B:" << endl;

PrintMatrix(matr\_B, N);

}

else{

fPrintMatrix(matr\_A, N, fA);

fPrintMatrix(matr\_B, N, fB);

}

// --------------------- //

StartParAlg = MPI\_Wtime();

sqr = (int)sqrt((double)ProcSize), new\_N = N/sqr;

A = new double\*[ProcSize], B = new double\*[ProcSize];

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

A[i] = CreateVectorMatrix(new\_N);

B[i] = CreateVectorMatrix(new\_N);

}

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

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

A[sqr\*(i/new\_N)+j/new\_N][(i%new\_N)\*new\_N+(j%new\_N)] = matr\_A[i\*N+j];

B[sqr\*(i/new\_N)+j/new\_N][(i%new\_N)\*new\_N+(j%new\_N)] = matr\_B[i\*N+j];

}

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&thr, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

for(int i = 1; i < ProcSize; i++){

int coef\_A = sqr\*(i / sqr), coef\_B = i % sqr;

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

MPI\_Send(A[coef\_A], new\_N\*new\_N, MPI\_DOUBLE, i , 0, MPI\_COMM\_WORLD);

MPI\_Send(B[coef\_B], new\_N\*new\_N, MPI\_DOUBLE, i , 0, MPI\_COMM\_WORLD);

coef\_A++;

coef\_B += sqr;

}

}

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

double\* TMP = B[i];

B[i] = B[i\*sqr];

B[i\*sqr] = TMP;

}

}

if(ProcRank !=0){

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&thr, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

sqr = (int)sqrt((double)ProcSize), new\_N = N/sqr;

A = new double\*[sqr], B = new double\*[sqr];

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

A[i] = CreateVectorMatrix(new\_N);

B[i] = CreateVectorMatrix(new\_N);

}

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

MPI\_Recv(A[i], new\_N\*new\_N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &Status);

MPI\_Recv(B[i], new\_N\*new\_N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &Status);

}

}

TMP\_Rez = new double\*[sqr+1];

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

TMP\_Rez[i+1] = Shtr\_alg(A[i], B[i], new\_N, thr);

}

if(ProcSize == 4)

TMP\_Rez[0] = Add(TMP\_Rez[1], TMP\_Rez[2], new\_N);

if(ProcSize == 16)

TMP\_Rez[0] = Add(TMP\_Rez[1], TMP\_Rez[2], TMP\_Rez[3], TMP\_Rez[4], new\_N);

if(ProcRank == 0){

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

delete[] A[i];

delete[] B[i];

}

}

else{

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

delete[] A[i];

delete[] B[i];

}

}

for(int i = 1; i < sqr+1; i++){

delete[] TMP\_Rez[i];

}

delete[] A;

delete[] B;

if (ProcRank != 0) {

MPI\_Send(TMP\_Rez[0], new\_N\*new\_N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

delete[] TMP\_Rez[0];

}

if (ProcRank == 0){

int coef = sqrt(ProcSize);

for(int i = 0; i < new\_N; i++)

for(int j = 0; j < new\_N; j++)

matr\_Rez\_Par[coef\*i\*new\_N+j] = TMP\_Rez[0][i\*new\_N+j];

for(int k = 1; k < ProcSize; k++){

MPI\_Recv(TMP\_Rez[0], new\_N\*new\_N, MPI\_DOUBLE, k, 0, MPI\_COMM\_WORLD, &Status);

for(int i = 0; i < new\_N; i++)

for(int j = 0; j < new\_N; j++)

matr\_Rez\_Par[(k/coef)\*new\_N\*N+(k%coef)\*new\_N+coef\*i\*new\_N+j] = TMP\_Rez[0][i\*new\_N+j];

}

// ----------------- //

EndParAlg = MPI\_Wtime();

TimeParAlg = EndParAlg - StartParAlg;

cout << "Par. Algoritm time = " << TimeParAlg << endl;

char fParAlg[20] = "D:\ParAlg.txt";

if (N < 20){

cout << "Matr C:" << endl;

PrintMatrix(matr\_Rez\_Par, N);

}

else{

fPrintMatrix(matr\_Rez\_Par, N, fParAlg);

}

delete[] TMP\_Rez[0];

StartSeqAlg = MPI\_Wtime();

matr\_Rez\_Seq = Shtr\_alg(matr\_A, matr\_B, N, thr);

EndSeqAlg = MPI\_Wtime();

TimeSeqAlg = EndSeqAlg - StartSeqAlg;

cout << "Posl. Algoritm time = " << TimeSeqAlg << endl;

char fSeqAlg[20] = "D:\SeqAlg.txt";

if (N < 10){

cout << "Matr C:" << endl;

PrintMatrix(matr\_Rez\_Seq, N);

}

else{

fPrintMatrix(matr\_Rez\_Seq, N, fSeqAlg);

}

cout<< "Speed up = " << TimeSeqAlg/TimeParAlg << endl;

bool flag = false;

for(int k = 0; k < N; k++)

for(int l = 0; l < N; l++)

if(ceil (matr\_Rez\_Seq[k\*N+l]) != ceil (matr\_Rez\_Par[k\*N+l]))

flag = true;

if(flag)

cout << "result posl. != result par." << endl;

cout << matr\_Rez\_Seq[k\*N+l] << " " << matr\_Rez\_Par[k\*N+l] << endl;

if(!flag)

cout << "result posl. == result par." << endl;

delete[] matr\_Rez\_Par;

delete[] matr\_Rez\_Seq;

delete[] matr\_A;

delete[] matr\_B;

}

delete[] TMP\_Rez;

MPI\_Finalize();

return 0;

}