Федеральное государственное автономное образовательное учреждение высшего образования

«Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И.И.Воровича

Кафедра прикладной математики и программирования

Направление 01.03.02 - Прикладная математики и информатика

**ОТЧЁТ**

по индивидуальному заданию

Студентка 3 курса:

*Кораблина Элла Викторовна*

Преподаватель:

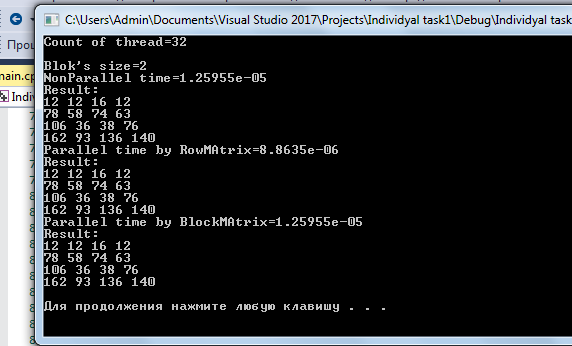
*Ассистент Баглий Антон Павлович*

**Постановка задачи:**

Необходимо реализовать непараллельное и параллельное умножение матриц. Опытным путём получить оптимальный размер блока матрицы. Сравнить время расчёта в обоих случаях.

**Ход работы:**

1. Проверим корректность алгоритма на малых значениях параметров. А именно возьмём матрицы размера и размер блока равный . Выполним умножение матриц в каждом из указанных случаев и сравним результаты:

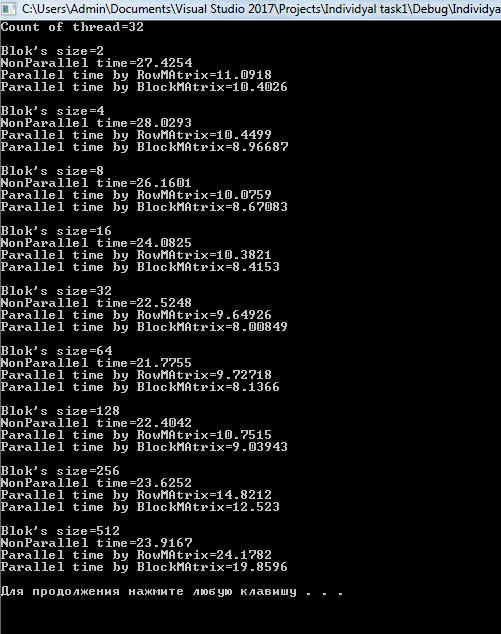


Видно, что все реализации поставленной задачи дают один и тот же результат.

1. Опытным путём вычислим оптимальное количество потоков и размер блоков:

Зададим матрицы размеров и количество потоков равное . Размеры блоков будем изменять в цикле по степеням двойки, начиная с и заканчивая размеров в элементов. Проведём сравнение трёх вариантов реализации:

* Непараллельный алгоритм, в случае, когда матрица записана в массив построчно
* Параллельный алгоритм, в случае, когда матрица записана в массив построчно
* Непараллельный алгоритм, в случае, когда матрица записана в массив по блочным строкам



Построим графики зависимости времени выполнения от размера блока: (размеры блоков изменяются по степеням двойки):

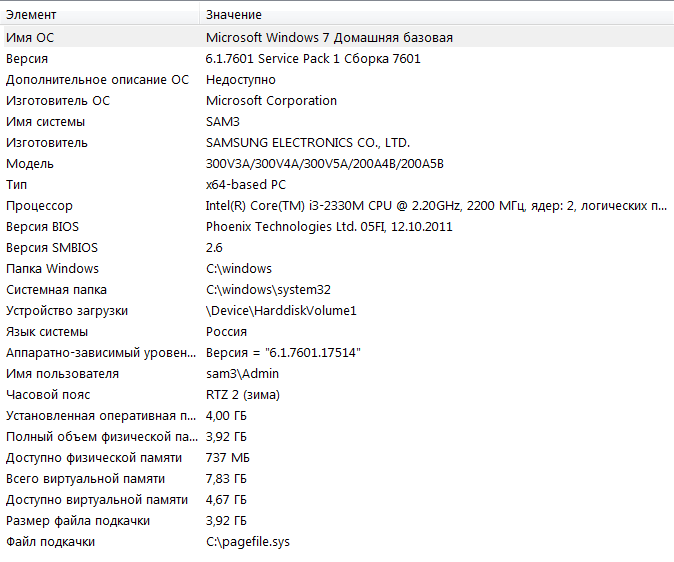
**Выводы:**

Построенные графики показывают, что время работы параллельных алгоритмов на порядки быстрее непараллельной реализации, что особенно заметно при малых размерах блоков и больших размерах матриц.

Если проанализировать время работы параллельных алгоритмов, в случае, когда матрица записана построчно и по блочным строкам, видно, что второй вариант является более эффективным. Однако разница времени выполнения двух алгоритмов .

Из построенных графиков видно, что при время выполнения тем меньше, чем больше размер блока. У обоих параллельных алгоритмов точкой минимума является , т.е наиболее оптимальный размер блока для реализации блочного умножения матриц.

**Характеристики компьютера:**



**Код программы:**

#include <iostream>

#include <omp.h>

using namespace std;

double\* rowBlock(double \* m, size\_t size\_m, size\_t size\_block);

double\* rowMatrixA(double \* m, size\_t size\_m);

double\* rowMatrixB(double \* m, size\_t size\_m);

double\* multRowMAtrixNonParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block);

double\* multRowMAtrixParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block);

double\* multBlockMAtrixParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block);

int main() {

setlocale(LC\_ALL, "Rus");

size\_t size\_m = 1024 , size\_block;

double \*A1 = new double[size\_m\*size\_m];

double \*A2 = new double[size\_m\*size\_m];

double \*B1 = new double[size\_m\*size\_m];

double \*B2 = new double[size\_m\*size\_m];

size\_t pos = 0;

int size\_arr = size\_m \* size\_m - ((size\_m \* size\_m - size\_m) / 2);

double \*A = new double[size\_arr];

for (int i = 0; i < size\_m; ++i)

for (int j = 0; j < size\_m; ++j) {

if (i >= j) {

A[pos] = rand() % 10 + 1;

pos++;

}

}

pos = 0;

double \*B = new double[size\_arr];

for (int i = 0; i < size\_m; ++i)

for (int j = 0; j < size\_m; ++j) {

if (i >= j) {

B[pos] = rand() % 10 + 1;

pos++;

}

}

cout << "Count of thread=32" << endl;

cout << endl;

size\_block = 2;

while (size\_block <= 256) {

A1 = rowMatrixA(A, size\_m);

A2 = rowBlock(A1, size\_m, size\_block);

B1 = rowMatrixB(B, size\_m);

B2 = rowBlock(B1, size\_m, size\_block);

cout << "Blok's size=" << size\_block << endl;

double start\_time, end\_time;

start\_time = omp\_get\_wtime();

double\* C1 = multRowMAtrixNonParallel(A1, B1, size\_m, size\_block);

end\_time = omp\_get\_wtime();

cout << "NonParallel time=" << end\_time - start\_time << endl;

start\_time = omp\_get\_wtime();

C1 = multRowMAtrixParallel(A1, B1, size\_m, size\_block);

end\_time = omp\_get\_wtime();

cout << "Parallel time by RowMAtrix=" << end\_time - start\_time << endl;

start\_time = omp\_get\_wtime();

C1 = multBlockMAtrixParallel(A2, B2, size\_m, size\_block);

end\_time = omp\_get\_wtime();

cout << "Parallel time by BlockMAtrix=" << end\_time - start\_time << endl;

size\_block \*= 2;

cout << endl;

}

system("pause");

return 0;

}

double\* rowBlock(double \* m, size\_t size\_m, size\_t size\_block) {

double\* res = new double[size\_m\*size\_m];

int pe = 0;

for (int jk = 0; jk < size\_m; jk += size\_block) {

for (int kk = 0; kk < size\_m; kk += size\_block) {

for (int j = 0; j < size\_block; ++j) {

for (int k = 0; k < size\_block; ++k) {

res[pe] = m[(jk + j)\*size\_m + (kk + k)];

pe++;

}

}

}

}

return res;

}

double\* rowMatrixA(double \* m, size\_t size\_m) {

double \* res = new double[size\_m\*size\_m];

int pos = 0, pe = 0;

for (int i = 0; i < size\_m; ++i) {

for (int j = 0; j < size\_m; ++j) {

if (i < j)

res[pe] = 0;

else {

res[pe] = m[pos];

pos++;

}

pe++;

}

}

return res;

}

double\* rowMatrixB(double \* m, size\_t size\_m) {

double \* res = new double[size\_m\*size\_m];

int pos = 0;

int p = 0, pe = 0;

int length, ll = 2, l = 0;

for (int i = 0; i < size\_m; ++i) {

int pos\_blok = i + 1, pos\_elem = 2 \* i, s = l;

length = ll;

l += ll;

for (int j = 0; j < size\_m; ++j) {

if (i < j) {

int sz = pos\_blok + s;

res[pe] = m[sz];

pe++;

p++;

s += length;

length++;

}

else {

res[pe] = m[pos];

pe++;

pos++;

}

}

ll++;

}

return res;

}

double\* multRowMAtrixNonParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block) {

double\* C = new double[size\_m\*size\_m];

for (int j = 0; j < size\_m; ++j)

for (int i = 0; i < size\_m; ++i)

C[j\*size\_m + i] = 0;

for (int jk = 0; jk < size\_m; jk += size\_block)

for (int kk = 0; kk < size\_m; kk += size\_block)

for (int ik = 0; ik<size\_m; ik += size\_block)

for (int j = 0; j < size\_block; ++j)

for (int k = 0; k < size\_block; ++k)

for (int i = 0; i < size\_block; ++i)

C[(jk + j)\*size\_m + (ik + i)] += A[(jk + j)\*size\_m + (kk + k)] \* B[(kk + k)\*size\_m + (ik + i)];

return C;

}

double\* multRowMAtrixParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block) {

double\* C = new double[size\_m\*size\_m];

for (int j = 0; j < size\_m; ++j)

for (int i = 0; i < size\_m; ++i)

C[j\*size\_m + i] = 0;

#pragma omp parallel num\_threads(32), shared(A, B,size\_block,size\_m,C)

{

#pragma omp for schedule(dynamic,2)

//#pragma omp for

for (int jk = 0; jk < size\_m; jk += size\_block)

for (int kk = 0; kk < size\_m; kk += size\_block)

for (int ik = 0; ik < size\_m; ik += size\_block)

for (int j = 0; j < size\_block; ++j)

for (int k = 0; k < size\_block; ++k)

for (int i = 0; i < size\_block; ++i)

C[(jk + j)\*size\_m + (ik + i)] += A[(jk + j)\*size\_m + (kk + k)] \* B[(kk + k)\*size\_m + (ik + i)];

}

return C;

}

double\* multBlockMAtrixParallel(double\* A, double\*B, size\_t size\_m, size\_t size\_block) {

double\* C = new double[size\_m\*size\_m];

for (int j = 0; j < size\_m; ++j)

for (int i = 0; i < size\_m; ++i)

C[j\*size\_m + i] = 0;

int p1, p2,p3,p4,p;

#pragma omp parallel num\_threads(32), shared(A, B,size\_block,size\_m,C), private(p1,p2,p3,p4,p)

{

p1 = 0, p2 = (-1)\*size\_m;

p3 = 0, p4 = 0,p=0;

#pragma omp for schedule(dynamic,2)

//#pragma omp for

for (int jk = 0; jk < size\_m; jk += size\_block) {

p3 = 0, p4 = 0, p = 0;

p2 += size\_m;

for (int kk = 0; kk < size\_m; kk += size\_block) {

for (int ik = 0; ik < size\_m; ik += size\_block) {

p1 = 0;

for (int j = 0; j < size\_block; ++j) {

p = p3;

for (int k = 0; k < size\_block; ++k) {

for (int i = 0; i < size\_block; ++i) {

C[(jk + j)\*size\_m + (ik + i)] += B[p + p4]\*A[p1 + p2];

p++;

}

p1++;

}

}

p4 += size\_block;

p3 += size\_block;

}

}

}}

return C;

}