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:

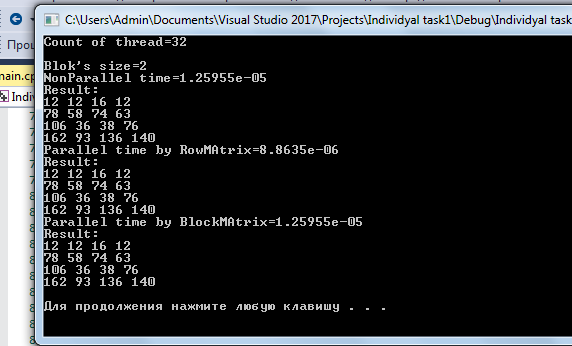
*Ассистент Баглий Антон Павлович*

**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 оптимальный 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 двух вариантов реализации:

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

*Таблица 1.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
| Параллельная реализация | 5,357 | 2,9 | 3,13 | 3,21 | 2,99 | 3,43 | 5,23 |
| Непараллельная реализация | 27,2 | 24,6 | 18,86 | 16,199 | 16,5 | 17,825 | 19,52 |

В таблице 1 представлена зависимость времени выполнения расчёта (в секундах) от размера блока (в количествах элементов в строке блока ). Теперь построим графики зависимости времени выполнения от размера блока: (размеры блоков изменяются по степеням двойки):

Видно, что вариант, когда идёт хранение всех элементов, работает медленнее, чем вариант, когда не храним нули и симметричные элементы.

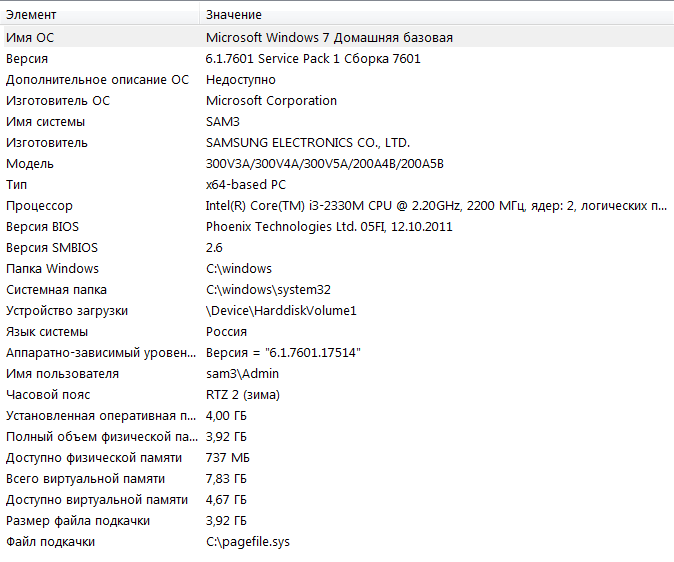
Ряд3 – параллельная реализация, когда матрица храниться по блочным строкам.

**Выводы:**

Построенные графики показывают, что время работы параллельного алгоритма на порядок быстрее непараллельной реализации, что особенно заметно при малых размерах блоков.

Из построенного графика параллельного алгоритма видно, что точкой минимума является , т.е наиболее оптимальный размер блока для реализации блочного умножения матриц.

**Характеристики компьютера:**



**Код программы:**

#include <time.h>

#include <iostream>

#include <omp.h>

#include "blockmatrix.h"

using namespace std;

struct matrix\_block{ int\* value;};

struct matrix\_row{matrix\_block\* blocks;};

struct trmat

{

//запись матрицы в строке с учётом блоков

matrix\_row\* row;

int size;

int blocksize;

};

typedef struct trmat block\_matrix;

typedef struct trmat symmetric\_matrix;

//параллельное умножение

void multParallel(trmat& left,

trmat& right, block\_matrix& result);

void multParallelBlock(trmat& left,

trmat& right, block\_matrix& result);

void multTwoBlockParallel(int \*A, int \*B, int \*C, int n);

//непараллельное умножение

void multNonParallel(trmat& left,

trmat& right, block\_matrix& result);

void multNonParallelBlock(trmat& left,

trmat& right, block\_matrix& result);

void multTwoBlockNonParallel(int \*A, int \*B, int \*C, int n);

int main() {

setlocale(LC\_ALL, "Rus");

int matrix\_size = 1024;

cout << "Matrix size = " << matrix\_size << endl;

int block\_size = 4;

while (block\_size<1024){

cout << "Block's size = " << block\_size<< endl;

int size = matrix\_size / block\_size;

trmat a;

a.size = size;

a.blocksize = block\_size;

a.row = new matrix\_row[size];

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

{

a.row[i].blocks = new matrix\_block[size - i];

for (int j = 0; j < size - i; j++)

{

int s = block\_size\*block\_size;

a.row[i].blocks[j].value = new int[s];

for (int k = 0; k < s; ++k) a.row[i].blocks[j].value[k] = 0;

}

}

int i, j, k;

for (i = 0; i < a.size; i++)

for (j = 0; j < a.size - i; j++)

for (k = 0; k < a.blocksize\*a.blocksize; k++) {

a.row[i].blocks[j].value[k] = rand() % 10 + 1;

}

for (i = 0; i < a.size; i++)

for (j = 1; j < a.blocksize; j++)

for (k = 0; k < j; k++)

a.row[i].blocks[0].value[j\*a.blocksize + k] = 0;

trmat b;

b.size = size;

b.blocksize = block\_size;

b.row = new matrix\_row[size];

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

{

b.row[i].blocks = new matrix\_block[size - i];

for (int j = 0; j < size - i; j++)

{

int s = block\_size\*block\_size;

b.row[i].blocks[j].value = new int[s];

for (int k = 0; k < s; ++k) b.row[i].blocks[j].value[k] = 0;

}

}

for (i = 0; i < b.size; i++)

for (j = 0; j < b.size - i; j++)

for (k = 0; k < b.blocksize\*b.blocksize; k++) {

b.row[i].blocks[j].value[k] = rand() % 10 + 1;

}

for (i = 0; i < b.size; i++)

for (j = 1; j < b.blocksize; j++)

for (k = 0; k < j; k++)

b.row[i].blocks[0].value[j\*b.blocksize + k] = 0;

block\_matrix c;

c.size = size;

c.blocksize = block\_size;

c.row = new matrix\_row[size];

for (i = 0; i < size; i++)

{

c.row[i].blocks = new matrix\_block[size];

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

{

int s = block\_size\*block\_size;

c.row[i].blocks[j].value = new int[s];

for (int k = 0; k < s; ++k) c.row[i].blocks[j].value[k] = 0;

}

}

block\_matrix c1;

c1.size = size;

c1.blocksize = block\_size;

c1.row = new matrix\_row[size];

for (i = 0; i < size; i++)

{

c1.row[i].blocks = new matrix\_block[size];

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

{

int s = block\_size\*block\_size;

c1.row[i].blocks[j].value = new int[s];

for (int k = 0; k < s; ++k) c1.row[i].blocks[j].value[k] = 0;

}

}

double start\_time, end\_time;

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

multParallel(a, b, c);

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

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

multNonParallel(a, b, c1);

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

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

block\_size \*= 2;

}

system("pause");

return 0;

}

void multParallel(trmat& left,trmat& right, block\_matrix& result)

{

//перенумируем элементы матрицы для удобства

for (int k = 0; k< left.size; k++)

for (int i = 1; i < left.blocksize; i++)

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

left.row[k].blocks[0].value[i\*left.blocksize + j] = left.row[k].blocks[0].value[j\*left.blocksize + i];

// параллельное умножение

multParallelBlock(left, right, result);

//перенумируем обратно

for (int k = 0; k < left.size; k++)

for (int i = 1; i < left.blocksize; i++)

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

left.row[k].blocks[0].value[i\*left.blocksize + j] = 0;

}

void multParallelBlock(trmat& left,trmat& right, block\_matrix& result)

{

#pragma omp parallel for shared(left, right, result) num\_threads(4)

for (int i1 = 0; i1 < left.size; i1++)

{

for (int j = 0, j1 = left.size - 1; j < left.size; j++, j1--)

{

for (int k = 0, k1 = j; k <= j; k++, k1--)

{

if (i1 > k)

{

int \*LT = new int[left.blocksize\*left.blocksize];

for (int i = 0; i < left.blocksize; i++)

for (int j = 0; j < left.blocksize; j++)

LT[i\*left.blocksize + j] = left.row[k].blocks[i1 - k].value[i + j\*left.blocksize];

multTwoBlockParallel(LT, right.row[k].blocks[j - k].value, result.row[i1].blocks[j].value, left.blocksize);

}

else

multTwoBlockParallel(left.row[i1].blocks[k - i1].value, right.row[k].blocks[j - k].value, result.row[i1].blocks[j].value, left.blocksize);

}

}

}

}

void multTwoBlockParallel(int \*A, int \*B, int \*C, int n)

{

int \*BT = new int[n\*n];

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

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

BT[i\*n + j] = B[i + j\*n];

#pragma omp parallel num\_threads(4), shared(A, B,n,C)

{

int i, j, k;

#pragma omp for

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

{

int dot = 0;

for (k = 0; k < n; k++)

dot += A[i\*n + k] \* BT[j\*n + k];

C[i\*n + j] += dot;

}

}

delete[] BT;

}

void multNonParallel(trmat& left,trmat& right, block\_matrix& result)

{

//перенумируем элементы матрицы для удобства

for (int k = 0; k< left.size; k++)

for (int i = 1; i < left.blocksize; i++)

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

left.row[k].blocks[0].value[i\*left.blocksize + j] = left.row[k].blocks[0].value[j\*left.blocksize + i];

// непараллельное умножение

multNonParallelBlock(left, right, result);

//перенумируем обратно

for (int k = 0; k < left.size; k++)

for (int i = 1; i < left.blocksize; i++)

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

left.row[k].blocks[0].value[i\*left.blocksize + j] = 0;

}

void multNonParallelBlock(trmat& left,trmat& right, block\_matrix& result)

{

for (int i1 = 0; i1 < left.size; i1++)

{

for (int j = 0, j1 = left.size - 1; j < left.size; j++, j1--)

for (int k = 0, k1 = j; k <= j; k++, k1--)

{

if (i1 > k)

{

int \*LT = new int[left.blocksize\*left.blocksize];

for (int i = 0; i < left.blocksize; i++)

for (int j = 0; j < left.blocksize; j++)

LT[i\*left.blocksize + j] = left.row[k].blocks[i1 - k].value[i + j\*left.blocksize];

multTwoBlockParallel(LT, right.row[k].blocks[j - k].value, result.row[i1].blocks[j].value, left.blocksize);

}

else

multTwoBlockParallel(left.row[i1].blocks[k - i1].value, right.row[k].blocks[j - k].value, result.row[i1].blocks[j].value, left.blocksize);

}

}

}

void multTwoBlockNonParallel(int \*A, int \*B, int \*C, int n)

{

int \*BT = new int[n\*n];

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

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

BT[i\*n + j] = B[i + j\*n];

int i, j, k;

for (i = 0; i < n; i++)

for (j = 0; j < n; j++){

int dot = 0;

for (k = 0; k < n; k++)

dot += A[i\*n + k] \* BT[j\*n + k];

C[i\*n + j] += dot;

}

delete[] BT;

}