**МИНОБРНАУКИ РОССИИ**

Федеральное государственное автономное образовательное

учреждение высшего профессионального образования

**«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

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

Кафедра алгебры и дискретной математики

**Отчет**

Тема: Блочные вычисления. Модели времени выполнения программ. Блочные размещения массивов, дополняющие блочные вычисления.

Выполнил студент 4 курса Ласковец А.Д.

Содержание.

1.Условие задачи.

2.Алгоритм решения.

3.Результаты.

1.Условие задачи.

Задание 14.

Написать программу блочного умножения двух матриц C=A\*B.

Перемножение каждых двух блоков выполнить параллельно.

Матрица A верхне-треугольная. Хранится в виде одномерного массива по блочным строкам.

Матрица B верхне-треугольная. Хранится в виде одномерного массива по блочным столбцам.

2.Алгоритм решения.

**A B C**

**1 2 3 1 2 3 1\*1 1\*2+2\*4 1\*3+2\*5+3\*6**

**4 5 \* 4 5 = 4\*4 4\*5+5\*6**

**6 6 6\*6**

1...6 - блоки (внутренние матрицы) пронумерованные по порядку хранения в массиве

Блочное умножение - это умножение строки блоков матрицы A на столбец блоков матрицы B.

1) т.к. по условию блоки матрицы A хранятся по столбцам, то для перехода к следующему блоку в строке нужно сделать шаг равный количеству блоков в столбце.

2) т.к. мы умножаем на верхне-треугольную матрицу, то в матрице A нужно, для каждого последующего столбца матрицы B, пропускать на 1 блок меньше.

3) переход к следующему столбцу матрицы B происходит сразу после текущего столбца, т.к. в памяти они находятся по порядку.

4) для перехода к новой строке матрицы A, нужно от начала перейти на количество блоков равное номеру нужной строки.

5) при переходе к новой строке матрицы A, положение указателя в матрице B возвращается в начало.

6) на каждом шаге итерации происходит умножение блоков и прибавление его к соответствующему блоку в результирующей матрице C.

7) указатель на блок в результирующей матрице C сдвигается при проходе одного столбца в матрице B.

2.Алгоритм решения.

Код программы на C++

#include "pch.h"

#include <iostream>

#include <time.h>

#include <omp.h>

#include <Windows.h>

using namespace std;

int \*\* get\_A\_block(int \*\*A, int position1, int position2, int k)

{

int \*\*D = new int\*[k];

int x = position1 \* k;

int y = position2 \* k;

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

{

D[i] = new int[k];

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

{

D[i][j] = A[i + x][j + y];

}

}

return D;

}

int \*\* get\_B\_block(int \*\*B, int position1, int position2, int k)

{

int \*\*D = new int\*[k];

int x = position1 \* k;

int y = position2 \* k;

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

{

D[i] = new int[k];

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

{

D[i][j] = B[i + y][j + x];

}

}

return D;

}

int \*\* create\_rect\_matrix(int \*\*A, int n, int \*sub\_A, int count)

{

int chet = n;

int k = 0;

int sum = 0;

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

{

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

{

A[i][j] = 0;

}

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

{

A[i][j] = sub\_A[k];

k++;

}

}

return A;

}

int main()

{

const int n = 256;

int count = 0;

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

count += i;

}

int \*sub\_A = new int[count];

int \*sub\_B = new int[count];

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

sub\_A[i] = rand() % 9;

sub\_B[i] = rand() % 9;

}

/\*int A[n][n];

int B[n][n];

int D[n][n];\*/

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

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

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

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

{

A[i] = new int[n];

B[i] = new int[n];

D[i] = new int[n];

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

{

A[i][j] = rand() % 9;

B[i][j] = rand() % 9;

A[i][j] = 0;

B[i][j] = 0;

D[i][j] = 0;

}

}

/\*int A[6][6] = {

{1,2,3,4,5,6},

{0,2,3,4,5,6} ,

{0,0,3,4,5,6} ,

{0,0,0,4,5,6} ,

{0,0,0,0,5,6} ,

{0,0,0,0,0,6} ,

};

int B[6][6] = {

{1,2,3,4,5,6},

{0,2,3,4,5,6} ,

{0,0,3,4,5,6} ,

{0,0,0,4,5,6} ,

{0,0,0,0,5,6} ,

{0,0,0,0,0,6} ,

};

int D[6][6] = {

{0,0,0,0,0,0},

{0,0,0,0,0,0} ,

{0,0,0,0,0,0} ,

{0,0,0,0,0,0} ,

{0,0,0,0,0,0} ,

{0,0,0,0,0,0} ,

};\*/

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

// cout << endl;

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

// cout << A[i][j] << " ";

// //D[i][j] = 0;

// }

//}

//cout << endl;

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

// cout << endl;

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

// cout << B[i][j] << " ";

// //D[i][j] = 0;

// }

//}

//cout << endl;

//A = create\_rect\_matrix(A, n, sub\_A, count);

//B = create\_rect\_matrix(B, n, sub\_B, count);

double start\_time, end\_time;

//DWORD start\_time, end\_time;

for (int k = 1; k < n + 1; k++)

{

if (n%k == 0)

{

cout << "block " << k << "\n";

cout << "with parallel: ";

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

for (int block\_row = 0; block\_row < int(n/k); block\_row++)

{

for (int block\_column = 0; block\_column < int(n/k); block\_column++)

{

for (int block\_column = 0; block\_column < n / k; block\_column++)

{

#pragma omp parallel num\_threads(8) //private(block\_column, i, j)

{

#pragma omp for schedule(static)

for (int block\_column2 = 0; block\_column2 < n / k; block\_column2++)

{

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

{

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

{

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

D[block\_row\*k + i][block\_column2\*k + j] += A[block\_row\*k + i][block\_column\*k + f] \* B[block\_column\*k + f][block\_column2\*k + j];

}

}

}

}

}

}

}

}

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

cout << end\_time - start\_time << "\n";

cout << "with parallel in single block: ";

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

for (int block\_row = 0; block\_row < int(n / k); block\_row++)

{

for (int block\_column = 0; block\_column < int(n / k); block\_column++)

{

#pragma omp parallel num\_threads(8) //private(block\_column, i, j)

{

#pragma omp for schedule(static)

for (int block\_column2 = 0; block\_column2 < n / k; block\_column2++)

{

int \*\* sub\_A = get\_A\_block(A, block\_row, block\_column, k);

int \*\* sub\_B = get\_B\_block(A, block\_column, block\_column2, k);

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

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

{

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

D[block\_row\*k + i][block\_column2\*k + j] += sub\_A[i][j] \* sub\_B[j][f];

}

}

}

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

delete[] sub\_A[i];

delete[] sub\_B[i];

}

delete[] sub\_A;

delete[] sub\_B;

}

}

}

}

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

cout << end\_time - start\_time << "\n";

cout << "no parralel ";

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

for (int block\_row = 0; block\_row < n / k; block\_row++)

{

for (int block\_column = 0; block\_column < n / k; block\_column++)

{

for (int block\_column2 = 0; block\_column2 < n / k; block\_column2++)

{

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

{

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

{

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

D[block\_row\*k + i][block\_column2\*k + j] += A[block\_row\*k + i][block\_column\*k + f] \* B[block\_column\*k + f][block\_column2\*k + j];

}

}

}

}

}

}

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

cout << " " <<end\_time - start\_time << "\n";

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

// cout << endl;

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

// cout << D[i][j] << " ";

// //D[i][j] = 0;

// }

//}

//cout << endl;

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

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

// //cout << D[i][j] << " ";

// D[i][j] = 0;

// }

//}

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

cout << "no parallel in single block: ";

for (int block\_row = 0; block\_row < int(n / k); block\_row++)

{

for (int block\_column = 0; block\_column < int(n / k); block\_column++)

{

for (int block\_column2 = 0; block\_column2 < n / k; block\_column2++)

{

int \*\* sub\_A = get\_A\_block(A, block\_row, block\_column, k);

int \*\* sub\_B = get\_B\_block(A, block\_column, block\_column2, k);

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

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

{

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

D[block\_row\*k + i][block\_column2\*k + j] += sub\_A[i][j] \* sub\_B[j][f];

}

}

}

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

delete[] sub\_A[i];

delete[] sub\_B[i];

}

delete[] sub\_A;

delete[] sub\_B;

}

}

}

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

cout << " " << end\_time - start\_time << "\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\n";

}

}

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

delete[] A[i];

delete[] B[i];

delete[] D[i];

}

delete[] A;

delete[] B;

delete[] D;

return 0;

}

3. Результаты.

Для начала продемонстрируем результат перемножения матриц размера 6х6, для доказательства работоспособности алгоритма:

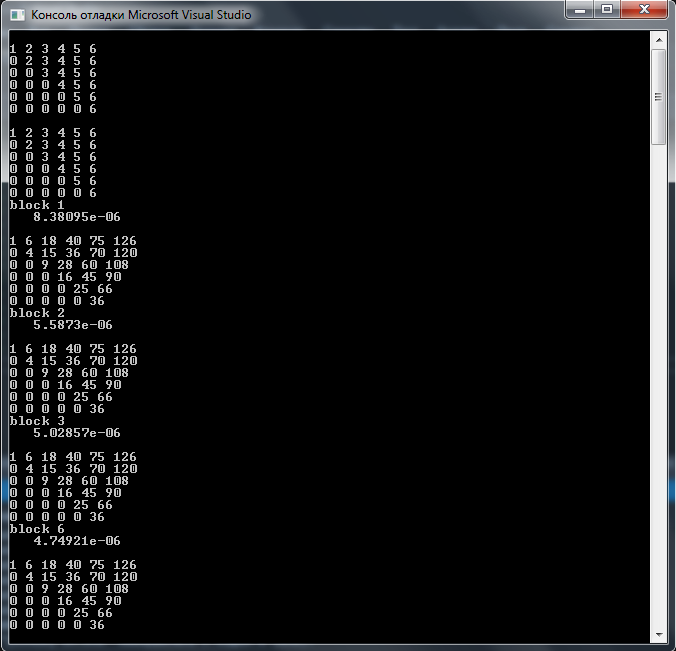


График зависимости времени перемножения и размерности блока матрицы для матриц размерности 256 для программы с параллельным, не параллельным выполнением и параллельным с резервированием памяти для каждого отдельного блока.

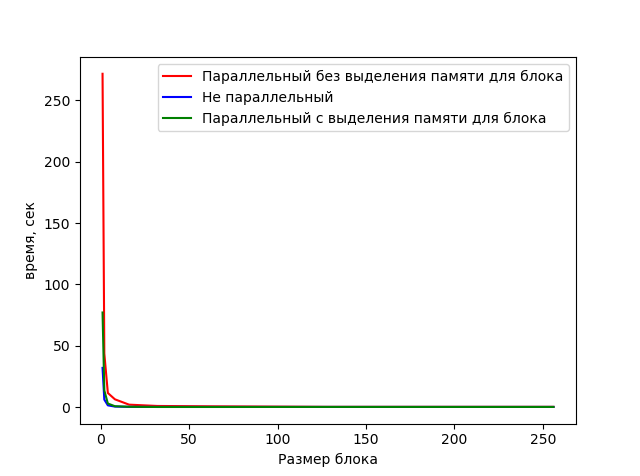
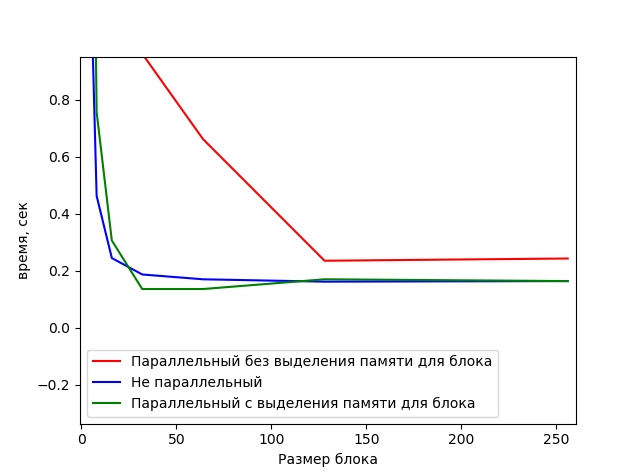
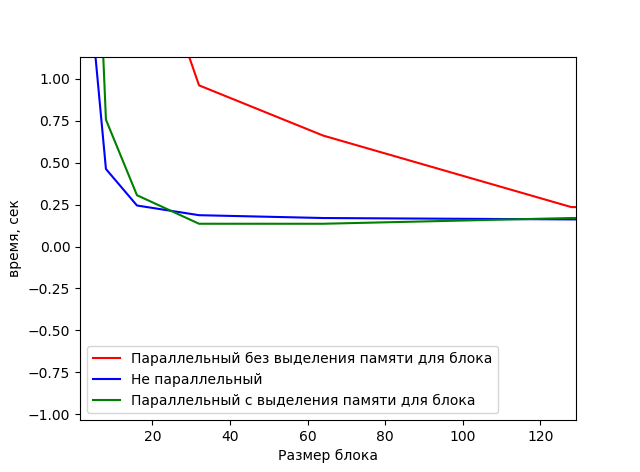


Таблица зависимости времени перемножения и размерности блока матрицы для матриц размерности 256

Тот же график, только проясняющий ситуацию на начальных данных



|  |  |  |  |
| --- | --- | --- | --- |
| Размер блока | Параллельный без выделения памяти для блока | Не параллельный | Параллельный с выделения памяти для блока |
| 1 | 271.754 | 31.9983 | 77.165 |
| 2 | 43.6841 | 6.16305 | 13.683 |
| 4 | 11.5456 | 1.45609 | 2.93498 |
| 8 | 6.37924 | 0.462744 | 0.756752 |
| 16 | 2.03132 | 0.244991 | 0.306357 |
| 32 | 0.961676 | 0.18719 | 0.13617 |
| 64 | 0.662105 | 0.170333 | 0.13617 |
| 128 | 0.235432 | 0.162026 | 0.170313 |
| 256 | 0.243287 | 0.164061 | 0.164061 |

Вычисления проводились на компьютере с характеристиками:

Процессор: IntelCore i3-3240U CPU 3.40 GHz

Кэш: 1 Гб

Оперативная память: 4 Гб

Соотношение памяти и системной шины 1:4

Выводы:

Самым плохим вариантом разбития на блоки является случай, когда размерность блока равна единице.

Самыми оптимальными: 128….256

Наилучший результат: 32