**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 \*\* 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 = 1024;

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;

}

}

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

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

double 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++)

{

int i, j, block\_column, br;

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

{

#pragma omp for schedule(static)

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

{

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

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

{

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

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

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

}

}

}

}

}

}

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

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

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

cout << "no parallel: ";

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

{

int i, j, block\_column, br;

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

{

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

}

}

}

}

}

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

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

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

}

}

return 0;

}

3. Результаты.

График зависимости времени перемножения и размерности блока матрицы для матриц размерности 1024 для программы с параллельным, не параллельным выполнением и параллельным с резервированием памяти для каждого отдельного блока.

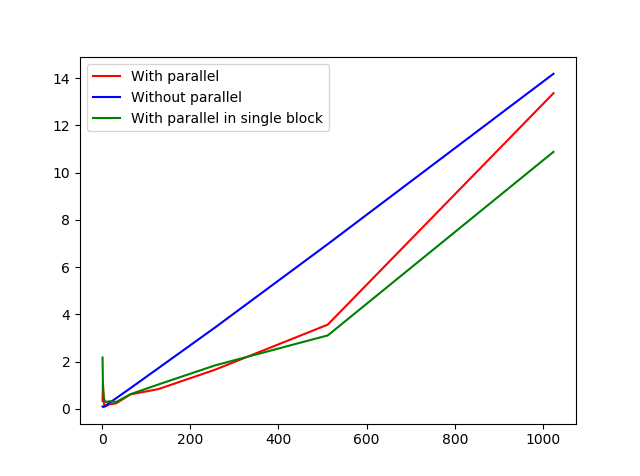


Таблица зависимости времени перемножения и размерности блока матрицы для матриц размерности 1024

|  |  |  |  |
| --- | --- | --- | --- |
| Block size | With parallel | Without parallel | With parallel in single block |
| 1 | 0.336292 | 0.0963924 | 2.16937 |
| 2 | 1.08158 | 0.104115 | 0.865159 |
| 4 | 0.216352 | 0.0749196 | 0.421688 |
| 8 | 0.101358 | 0.13213 | 0.284986 |
| 16 | 0.18101 | 0.243115 | 0.316266 |
| 32 | 0.230402 | 0.447297 | 0.285191 |
| 64 | 0.604674 | 0.867962 | 0.613729 |
| 128 | 0.833712 | 1.72363 | 1.03038 |
| 256 | 1.65159 | 3.43776 | 1.83476 |
| 512 | 3.5629 | 6.97251 | 3.10599 |
| 1024 | 13.3618 | 14.185 | 10.8782 |

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: 8….1024

Наилучший результат: 512