Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н. И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Умножение плотных матриц. Элементы типа double. Блочная схема, алгоритм Кэннона »**

**Выполнил:**

студентка группы 381608

Новикова А. А.

**Проверила:**

Кустикова В. Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 2](#_Toc531367741)

[Метод решения 3](#_Toc531367742)

[Схема распараллеливания 4](#_Toc531367743)

[Описание программной реализации 5](#_Toc531367744)

[Подтверждение корректности 6](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 8](#_Toc531367747)

[Приложение 9](#_Toc531367748)

# Постановка задачи

В данной работе необходимо реализовать умножение матриц с помощью блочного алгоритма Кэннона. На вход поступает две матрицы А и В со значениями типа double, на выходе – матрица С = А\*В.

Таким образом, для достижения поставленной задачи необходимо: реализовать параллельный алгоритм Кэннона, а также последовательное умножение матриц для проверки корректности работы параллельного алгоритма и оценки эффективности.

# Метод решения

Последовательный алгоритм перемножения двух плотных матриц *A* и *B* размерности реализуется по формуле:

Для работы алгоритма Кэннона процессы выстраиваются в квадратную решетку. Задача

каждого процесса – вычислить соответствующий блок результирующей матрицы с помощью операций сложения и перемножения блоков исходных матриц.

Для перемножения блоков исходных матриц может использоваться тот же последовательный алгоритм перемножения матриц.

# Схема распараллеливания

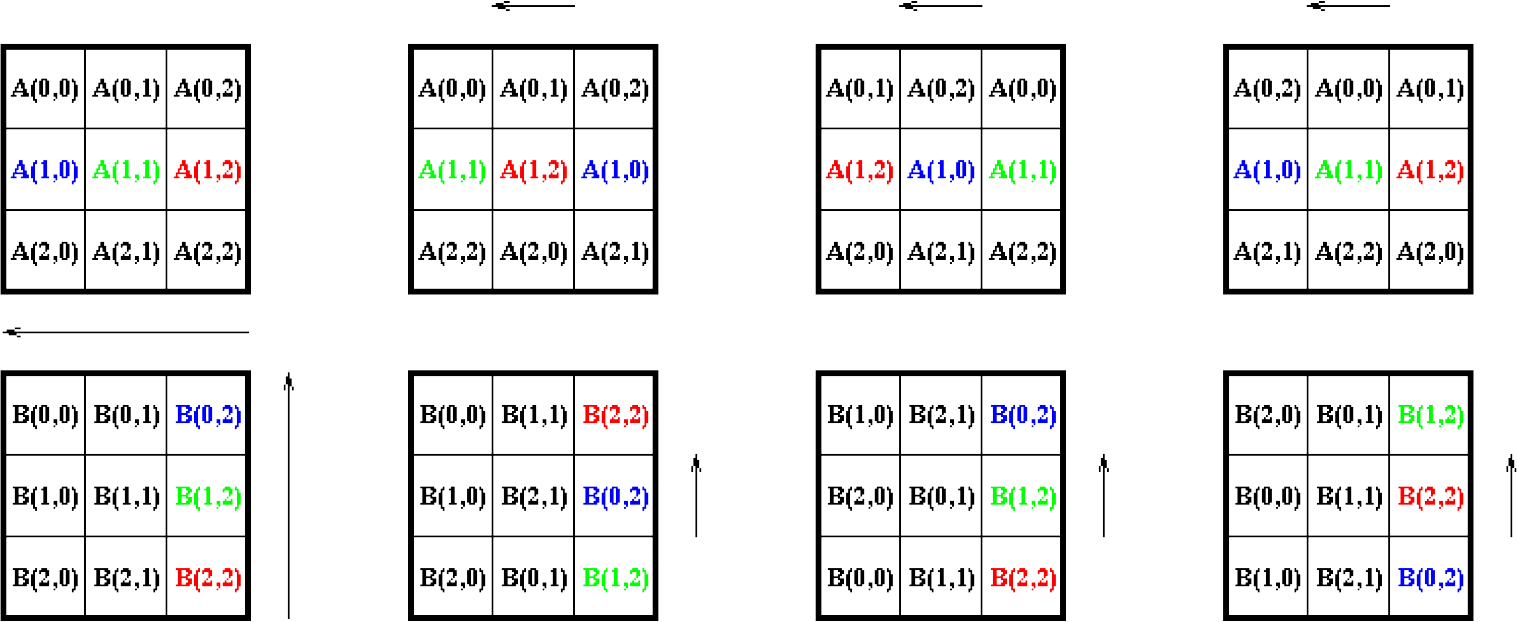
Алгоритм Кэннона – блочный алгоритм перемножения матриц. Для корректной работы необходимо, чтобы число процессов *p* было полным квадратом.

Разобьём каждую из матриц и на квадратных блоков размером и зададим нумерацию процессов , где , а умножение блоков и привяжем к процессу соответственно.

Алгоритм выполняется в два этапа:

1. блоки матриц циклически сдвигаются влево на некоторое кол-во позиций вдоль строки процессов, а блоки матриц циклически сдвигаются вверх на некоторое кол-во позиций вдоль столбцов процессов, так что процесс принимает блоки и соответственно;
2. происходит последовательность операций умножений-сложений-сдвига. Во время каждого шага блоки матриц циклически смещаются на одну позицию влево, а блоки матриц – циклически на одну позицию вверх, данный процесс происходит раз; каждый процесс перемножает полученные блоки матриц и и добавляет результат в свой блок (не стоит забывать, что перемножаются в том числе и блоки, полученные при первоначальной рассылке).

Для каждого процесса требуется машинных слов для хранения блоков матриц , и . Основным преимуществом этого алгоритма является то, что он использует постоянную память, независимо от кол-ва процессов.



# Описание программной реализации

**Последовательный алгоритм**

1. Создается три массива (**A, B, C**) размера **N\*N**.

2. **A** и **B** заполняются случайными значениями.

3. Заполняем ***C*** на основе следующей формулы

**Параллельный алгоритм**

* + - 1. Корневой процесс выделяет память для исходных матриц и , заполняет их случайными числами типа double, выделяет память под результирующую матрицу .
      2. Происходит создание новой виртуальной топологии – декартовой (квадратной) решетки. Процесс с координатами является корневым.
      3. Все процессы выделяют память под блоки матриц , и .
      4. Создается производный тип данных.
      5. Начинается отсчет времени.
      6. Корневой процесс высчитывает необходимые смещения для правильной рассылки блоков матриц.
      7. Выполняется коллективная операция рассылки блоков матрицы и для каждого процесса.
      8. Все процессы выполняют перемножение соответствующих блоков, затем передают свой блок матрицы влево по решетке, а блок матрицы – вверх.
      9. Все процессы выполняют коллективную операцию – отправку блоков матрицы на корневой процесс.
      10. Отсчет времени останавливается. Время выводится на экран. Программа завершает свою работу.

Код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

# Подтверждение корректности

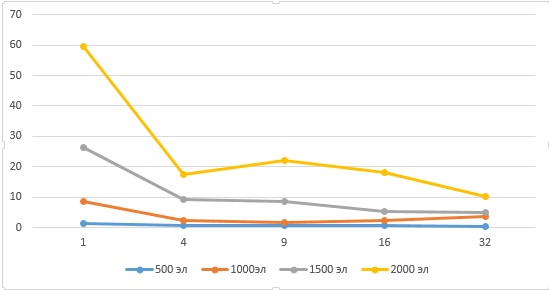
Для подтверждения корректности в программе осуществляется сравнение результатов, полученных параллельным путем и последовательным. После выполнения выводится соответствующее сообщение: в случае совпадения «Matrices are equal», в случае ошибки «Matrices are not equal».

# Результаты экспериментов

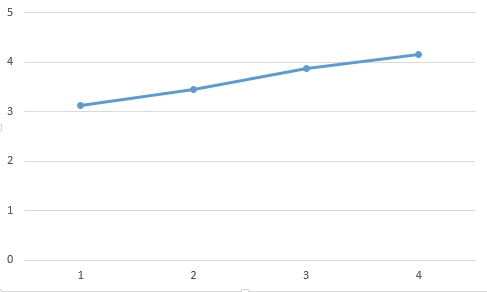
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во элементов | Время работы (секунд) | | | | |
| 1 процесс | 4 процесса | 9 процессов | 16 процессов | 36 процессов |
| 500 | 1,25 | 0,59 | 0,67 | 0,76 | 0,51 |
| 1000 | 8,48 | 2,36 | 1,79 | 2,17 | 3,64 |
| 1500 | 26,3 | 9,1 | 8,6 | 5,3 | 5,1 |
| 2000 | 59,37 | 17,3 | 22,14 | 18,1 | 10,19 |

Соотношение времени работы алгоритма при различном количестве доступных процессов:

Время работы (секунд)



Зависимость ускорения от числа процессов:



# Заключение

По результатам проведенных экспериментов можно сделать следующие выводы:

1. Алгоритм Кэннона на больших матрицах, в среднем, показывает ускорение, равное числу участвующих процессов, что соответствует теоретическим оценкам сложности алгоритма.
2. Алгоритм Кэннона может работать как в несколько раз медленнее, так и в несколько раз быстрее теоретической оценки. На маленьких матрицах Алгоритм Кэннона показывает ускорение ниже теоритического, так как большое время тратится на пересылку данных между процессами, а не на сами вычисления.
3. При большом количестве доступных процессов матрицы делятся на небольшие, по отношению к исходному размеру матриц, блоки, которые могут быть закешированы процессором, в результате чего операции над ними выполняются быстро. Для получения одного элемента результирующей матрицы последовательному алгоритму необходима одна строчка первой матрицы и один столбец второй, элементы которого находятся далеко друг от друга в оперативной памяти. В результате, во время выполнения последовательного алгоритма тратится большое количество времени на обмен данными с оперативной памятью.

# Приложение

#include <iostream>

#include <time.h>

#include <mpi.h>

#include <cmath>

using namespace std;

int ShowMatrix(double\* A, int N){

cout << endl;

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

{

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

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

cout << endl;

}

cout << endl;

return 1;

}

double\* GetRandMatrix(int N){

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

srand(time(0));

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

result[i] = (rand() % 10000) / 1000.0f;

return result;

}

int CheckMatrixForEqual(double \*A, double \*C, int N){

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

if (abs(A[i] - C[i]) > 0.000001)

return 0;

return 1;

}

void MultiplayMatrix(double\* A, double\* B, double\* C, const int n)

{

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

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

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

C[i \* n + j] += A[i \* n + k] \* B[k \* n + j];

}

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

double \*A = nullptr, \*B = nullptr, \*C = nullptr;

double timeP, timeL;

int N, size, root;

int procNum, rank;

int left\_rank, right\_rank, up\_rank, down\_rank;

MPI\_Status status;

MPI\_Datatype blockType;

MPI\_Comm grid;

if (argc>1)

N = atoi(argv[1]);

else

{

cout << "Try again.Usage:Task1.exe n(n is size of matrices)" << endl;

return 2;

}

MPI\_Init(&argc, &argv);

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

//Create a grid

int dims[2];

dims[0] = dims[1] = (int)(sqrt(procNum) + 0.5);

if ((dims[0] \* dims[0] != procNum) || (N % dims[0] != 0))

{

MPI\_Finalize();

printf("Number of processes must be a perfect square\n");

printf("Matrices must be fully devisible to processes\n");

exit(-1);

}

int periods[2] = { 1, 1 };

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, 1, &grid);

MPI\_Comm\_rank(grid, &rank);

// Get root process

int coords[2] = { 0, 0 };

MPI\_Cart\_rank(grid, coords, &root);

// Get neighbors

MPI\_Cart\_shift(grid, 1, -1, &right\_rank, &left\_rank);

MPI\_Cart\_shift(grid, 0, -1, &down\_rank, &up\_rank);

//Give memory to blocks

size = N / dims[0];

double \*blockA = new double[size\*size];

double \*blockB = new double[size\*size];

double \*blockC = new double[size\*size] ();

/\*C = new double[N\*N];

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

blockC[i] = 0;

\*/

// Create vector type to represent a block

MPI\_Datatype temptype;

MPI\_Type\_vector(size, size, N, MPI\_DOUBLE, &temptype);

MPI\_Type\_create\_resized(temptype, 0, size\*size\*sizeof(double), &blockType);

MPI\_Type\_commit(&blockType);

if (rank == root){

A = GetRandMatrix(N);

B = GetRandMatrix(N);

C = new double[N \* N];

ShowMatrix(A, N);

ShowMatrix(B, N);

//Start parallel work

timeP = MPI\_Wtime();

// Copy the first blocks of matrices to block arrays

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

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

{

blockA[i \* size + j] = A[i \* N + j];

blockB[i \* size + j] = B[i \* N + j];

}

// Initial Cannon's alignment (shifted rows in A)

for (int i = 0; i < dims[0]; ++i)

for (int j = 0; j < dims[1]; ++j)

if ((i != 0) || (j != 0))

{

// Send block of A matrix with initial shift to left

int dest, block\_coords[2] = { i, j - i };

if (block\_coords[1] < 0) block\_coords[1] += dims[1];

MPI\_Cart\_rank(grid, block\_coords, &dest);

MPI\_Send(&A[i \* N \* size + j \* size], 1, blockType, dest, 0, grid);

}

// Initial Cannon's alignment (shifted columns in B)

for (int i = 0; i < dims[0]; ++i)

for (int j = 0; j < dims[1]; ++j)

if ((i != 0) || (j != 0))

{

// Send block of B matrix with initial shift to up

int dest, block\_coords[2] = { i - j, j };

if (block\_coords[0] < 0) block\_coords[0] += dims[0];

MPI\_Cart\_rank(grid, block\_coords, &dest);

MPI\_Send(&B[i \* N \* size + j \* size], 1, blockType, dest, 1, grid);

}

}

else

{

// Recieve initial blocks from root process

MPI\_Recv(blockA, size \* size, MPI\_DOUBLE, root, 0, grid, &status);

MPI\_Recv(blockB, size \* size, MPI\_DOUBLE, root, 1, grid, &status);

}

// Main loop

MultiplayMatrix(blockA, blockB, blockC, size);

for (int i = 1; i < dims[0]; ++i)

{

MPI\_Sendrecv\_replace(blockA, size \* size, MPI\_DOUBLE, left\_rank, 0, right\_rank, 0, grid, &status);

MPI\_Sendrecv\_replace(blockB, size \* size, MPI\_DOUBLE, up\_rank, 1, down\_rank, 1, grid, &status);

MultiplayMatrix(blockA, blockB, blockC, size);

}

if (rank == root){

// Copy first BlockC to matrix C

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

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

C[i \* N + j] = blockC[i \* size + j];

// Collect blocks from processes

for (int i = 0; i < dims[0]; ++i)

for (int j = 0; j < dims[1]; ++j)

if ((i != 0) || (j != 0))

{

int source, block\_coords[2] = { i, j };

MPI\_Cart\_rank(grid, block\_coords, &source);

MPI\_Recv(&C[i \* N \* size + j \* size], 1, blockType, source, 4, grid, &status);

}

printf("Time parallel = %.10f\n", MPI\_Wtime() - timeP);

ShowMatrix(C, N);

double \*C\_Lin = new double[N\*N];

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

C\_Lin[i] = 0;

timeL = MPI\_Wtime();

MultiplayMatrix(A, B, C\_Lin, N);

printf("Time linear = %.10f\n", MPI\_Wtime() - timeL);

ShowMatrix(C\_Lin, N);

// Check if results are equal

if (CheckMatrixForEqual(C, C\_Lin, N) == 1)

cout << "Matrices are equal" << endl;

else

cout << "Matrices are not equal" << endl;

delete C\_Lin;

}

else

// Send blocks of C matrix to root process

MPI\_Send(blockC, size \* size, MPI\_DOUBLE, root, 4, grid);

MPI\_Type\_free(&blockType);

MPI\_Comm\_free(&grid);

MPI\_Finalize();

delete A;

delete B;

delete C;

delete blockA;

delete blockB;

delete blockC;

return 0;

}