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 Фокса»**

**Выполнил:**

студентка группы 1608

Поздеева В. В.

**Проверил:**

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 Фокса. 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 Фокса, а также последовательное умножение матриц для проверки корректности работы параллельного алгоритма и оценки эффективности.

# Метод решения

Алгоритм Фокса предполагает разбиение исходных матриц (**А** и **В)** и результирующую матрицу (**С**) на блоки. Будем предполагать, что матрицы являются квадратными размера **N\*N**. Тогда количество блоков(**q**), по вертикали и горизонтали равно. Размер блока (**k)** равен **N/q.**

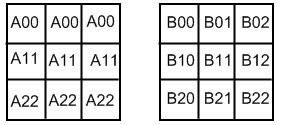
Разделим решение на подзадачи. Каждая подзадача будет отвечать за вычисление блока **Сij**. В качестве нумерации подзадач будем использовать индексы размещаемых в ней блоков. То есть в ходе вычислений на каждой подзадаче (**i**, **j**) располагается 4 блока:

1. **Сij –** блок результата
2. **A`ij -**  блок, получаемый в ходе вычислений.
3. **B`ij -**  блок, получаемый в ходе вычислений (в нем же размещается блок матрицы В перед началом вычислений)
4. **Aij –** блок, получаемый до начала вычислений, остается неизменным.

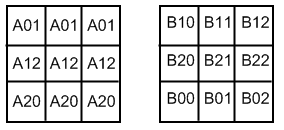
В соответствии с алгоритмом:

1. Блоки **Aij** и **Bij** рассылаются по подзадачам (**i**, **j**), блоки **Cij** обнуляются.
2. На каждой итерации **i=1, 2, …, q** выполняется:
   1. Блок **Alm** пересылается всем подзадачам строки **l**, где **m = (i + l ) mod q**
   2. К блоку **Сij** добавляется произведение блоков **A`ij** и **B`ij**
   3. Блок **B`ij** пересылается подзадаче, расположенной выше (блоки подзадач из первой строки посылаются блокам подзадач из последней строки).

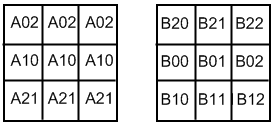
Предположим есть матрицы А и В, количество блоков равно 3 и количество процессов равно 9, тогда после первой итерации цикла процессы будут иметь следующие блоки (каждая клеточка таблицы отвечает за один процесс):



После выполнения второй итерации цикла:



После выполнения третьей итерации:



# Схема распараллеливания

Алгоритм Фокса предполагает, что каждый процесс должен иметь доступ ко всем соответствующим наборам строк матрицы А и столбцов матрицы В. Таким образом за схему распараллеливания удобно взять двумерную сетку, при этом количество процессов должно быть равно **p=q2**.

То есть общение между процессами будет происходить следующим образом, каждый процесс будет иметь возможность:

* отправить данные всем процессам своей строки (отправление блока матрица А, первый шаг итерации цикла).
* отправить данные соседу сверху и получить данные от соседа снизу (обмен блоками матрицы В, заключительный шаг итерации).

# 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. Создается три массива (**A, B, C**) размера **N\*N,** и четыре массива (**bА, bА`, bC, bВ**) размера **q\*q.**
2. На нулевом процессе **A** и **B** заполняются случайными значениями.
3. В соответствии со схемой распараллеливания рассылаются данные по процессам с помощью функции **MPI\_Send()**.
4. На каждом процессе организуется цикл от **i =0** до **q** (количество блоков).

* для каждой строчки определяется блок матрицы **А**, и рассылается всем процессам строки с помощью функции **MPI\_Bcast()**.
* Выполняется умножение блоков **bА`** и **bВ** в соответствии с формулой
* Соседние процессы обмениваются блоками матрицы **В** с помощью функции **MPI\_Sendrecv\_replace()**.

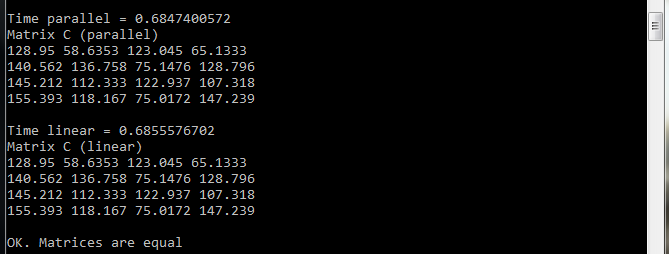
1. На нулевом процессе собираются все блоки матрицы **С** с помощью функции **MPI\_Recv()** на нулевом и **MPI\_Send()** на остальных процессах.

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 «OK. Matrices are equal», в случае ошибки «ERROR. Matrices are not equal».

Запустим программу для матрицы 4 на 4, для 4 процессов.



Видим, что результаты одинаковы, а это значит, что алгоритм работает корректно.

# Результаты экспериментов

Данные результаты получены при умножении двух квадратных матриц четырех размеров: 500, 1000, 1500 и 2000 элементов. Эксперименты проводились на 1, 4, 9, 16 и 36 процессах. Время, представленное в таблице, среднее за 10 экспериментов.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Последовательно, сек | 4 процесса, сек | 9 процессов, сек | 16 процессов, сек | 36 процессов, сек |
| 500 элементов | 1,32 | 0,63 | 0,69 | 0,8 | 0,48 |
| 1000 элементов | 9,35 | 2,47 | 1,74 | 2,11 | 3,73 |
| 1500 элементов | 27,7 | 9,29 | 8,5 | 5,9 | 5,6 |
| 2000 элементов | 66,62 | 17,13 | 21,79 | 18,05 | 10,19 |

По данным экспериментов видно, что при малом порядке матриц последовательная версия работает примерно так же. Когда порядок матриц увеличивается, возрастает и разница во времени, параллельный алгоритм работает быстрее.

По графику ускорения видно, что чем больше процессов задействовано, тем ускорение выше, однако зависимость не линейная, что можно объяснить накладными расходами (затраты на пересылку данных между процессами).

# Заключение

В результате работы была написана программа, позволяющая вычислять произведение матриц двумя способами: последовательным и параллельным (алгоритм Фокса). Из проведенных экспериментов можно сделать вывод, что алгоритм имеет хорошо распараллеливаемую вычислительную часть. Использование параллельного метода существенно сокращает время при перемножении матриц больших размеров.

# Приложение

#include <iostream>

#include <time.h>

#include <mpi.h>

#include <cmath>

using namespace std;

double\* CreateRandMatrix(int size)

{

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

srand(time(0));

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

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

return tmp;

}

void MultiMatrix(double\* A, double\* B, double\* C, 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 PrintMatrix(double\* A,int N)

{

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;

}

int CheckEqual(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;

}

typedef struct Grid

{

int rank;

int rank\_grid;

MPI\_Comm Comm\_grid;

MPI\_Comm Comm\_Row;

int procNum;

int rank\_row;

int root;

int up;

int down;

int q;

int coord[2];

}GRID;

int CreateGrid(GRID \* grid, int N)

{

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

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &grid->rank);

int const ndims =2;

int dims[ndims], periods[ndims], reorder =1;

grid->q=dims[0]=dims[1] = sqrt((double)grid->procNum);

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

{

MPI\_Finalize();

cout << " Error num of proc. It must be a perfect square!" << endl;

return 1;

}

periods[0]=periods[1] = 1;

MPI\_Cart\_create(MPI\_COMM\_WORLD, ndims, dims, periods, reorder, &grid->Comm\_grid);

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

MPI\_Cart\_rank(grid->Comm\_grid, coords, &grid->root);

MPI\_Cart\_shift(grid->Comm\_grid, 0, 1, &grid->up, &grid->down);

MPI\_Comm\_rank(grid->Comm\_grid, &grid->rank\_grid);

MPI\_Cart\_coords(grid->Comm\_grid, grid->rank\_grid, 2, grid->coord);

int remain\_dims[2]={0,1};

MPI\_Cart\_sub(grid->Comm\_grid, remain\_dims, &grid->Comm\_Row);

MPI\_Comm\_rank(grid->Comm\_Row, &grid->rank\_row);

return 0;

}

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

{

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

double \*blokA = nullptr, \*blokA1 = nullptr, \*blokB = nullptr, \*blokC = nullptr;

int N, sizeBlok, time;

GRID \*grid = new GRID;

MPI\_Status status;

MPI\_Datatype TypeBlok;

if(argc>1)

N=atoi(argv[1]);

else

{

cout << "Enter Matrix size. Example: Matrix.exe 4" << endl;

return 2;

}

MPI\_Init(&argc, &argv);

if ( CreateGrid(grid, N) == 1)

return 1;

sizeBlok = N/sqrt((double)grid->procNum);

blokA = new double[sizeBlok\*sizeBlok];

blokA1 = new double[sizeBlok\*sizeBlok];

blokB = new double[sizeBlok\*sizeBlok];

blokC = new double[sizeBlok\*sizeBlok];

C = new double[N\*N];

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

blokC[i]=0;

MPI\_Datatype temptype;

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

MPI\_Type\_create\_resized(temptype, 0, sizeBlok\*sizeBlok\*sizeof(double), &TypeBlok);

MPI\_Type\_commit(&TypeBlok);

if(grid->rank\_grid == grid->root)

{

A = CreateRandMatrix(N);

B = CreateRandMatrix(N);

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

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

{

blokA[i \* sizeBlok + j] = A[i \* N + j];

blokB[i \* sizeBlok + j] = B[i \* N + j];

}

time = MPI\_Wtime();

for(int i=0; i< grid->q; i++ )

for(int j=0; j<grid->q; j++)

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

{

int rank, coord[2] = {i,j};

MPI\_Cart\_rank(grid->Comm\_grid, coord, &rank);

MPI\_Send(&A[i\*sizeBlok\*N + j\* sizeBlok], 1, TypeBlok, rank, 0, grid->Comm\_grid);

MPI\_Send(&B[i\*sizeBlok\*N + j\* sizeBlok], 1, TypeBlok, rank, 1, grid->Comm\_grid);

}

}

else

{

MPI\_Recv(blokA, sizeBlok \* sizeBlok, MPI\_DOUBLE, 0, 0, grid->Comm\_grid, &status);

MPI\_Recv(blokB, sizeBlok \* sizeBlok, MPI\_DOUBLE, 0, 1, grid->Comm\_grid, &status);

}

for(int i=0; i<grid->q; i++)

{

int sourse = (grid->coord[0]+i)%(grid->q);

if(sourse == grid->rank\_row)

for(int j=0; j<sizeBlok\*sizeBlok; j++)

blokA1[j] = blokA[j];

MPI\_Bcast(blokA1, sizeBlok\*sizeBlok , MPI\_DOUBLE, sourse, grid->Comm\_Row);

MultiMatrix(blokA1, blokB, blokC, sizeBlok);

MPI\_Sendrecv\_replace(blokB, sizeBlok \* sizeBlok, MPI\_DOUBLE, grid->up, 1, grid->down, 1, grid->Comm\_grid, &status);

}

if(grid->rank\_grid == grid->root)

{

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

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

C[i \* N + j] = blokC[i \* sizeBlok + j];

for (int i = 0; i < grid->q; ++i)

for (int j = 0; j < grid->q; ++j)

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

{

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

MPI\_Cart\_rank(grid->Comm\_grid, block\_coords, &source);

MPI\_Recv(&C[i \* N \* sizeBlok + j \* sizeBlok], 1, TypeBlok, source, 4, grid->Comm\_grid, &status);

}

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

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

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

res[i]=0;

time = MPI\_Wtime();

MultiMatrix(A, B, res, N);

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

if(CheckEqual(C, res, N) == 1)

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

else

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

delete res;

}

else MPI\_Send(blokC, sizeBlok \* sizeBlok, MPI\_DOUBLE, grid->root, 4, grid->Comm\_grid);

MPI\_Type\_free(&TypeBlok);

MPI\_Comm\_free(&grid->Comm\_grid);

MPI\_Comm\_free(&grid->Comm\_Row);

MPI\_Finalize();

delete A;

delete B;

delete C;

delete blokA;

delete blokB;

delete blokC;

delete blokA1;

delete grid;

return 0;

}