Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

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

**Выполнил**:

студент группы 381708-2

Соколюк Н. А.

**Проверил**:

Волокитин В.Д.

Нижний Новгород

2019

**Содержание**

[Введение 3](#_Toc33118001)

[Постановка задачи 4](#_Toc33118002)

[Метод решения 5](#_Toc33118003)

[Схема распараллеливания 7](#_Toc33118004)

[Описание программной реализации 7](#_Toc33118005)

[Результаты экспериментов 9](#_Toc33118006)

[Заключение 9](#_Toc33118007)

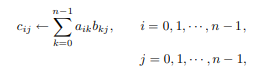
[Список литературы 10](#_Toc33118008)

# Введение

Заданы квадратные матрицы A = (aij) и B = (bij)

порядка n. Произведение A × B = C = (cij) также является квадратной матрицей того же порядка, n.

Обычное матричное умножение:



будет 3-го порядка, что чрезмерно затратно с точки зрения времени выполнения (и памяти тоже!), особенно при n → ∞. Потому что умножение включает в себя n^2 независимых вычислений, одно очевидное решение

к экономии времени- это распараллелить первые

две петли, тем самым уменьшая порядок до 2-го.

Реализация вышеупомянутого алгоритма работает правильно, производя те же результаты, если матрицы A, B и C

являются блочными матрицами, в которых каждый блок имеет порядок n и элемент, скажем, aij, в алгоритме

теперь это блочная матрица n × n.

Последовательный алгоритм Фокса для умножения A и B выполняется в n этапов, где n, опять же, порядок матриц:

https://skrinshoter.ru/i/230420/uq0a4SBW.png?download=1

где k = (i + k) mod n.

На этапе 0: Cij рассчитывается как произведение Aii и Bij,

и на этапе k: Cij рассчитывается как произведение b ik и b kj , где k = (i + k) mod n.

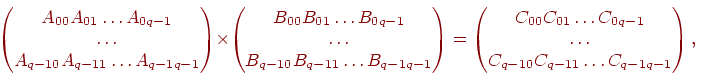
# Постановка задачи

1. Реализовать параллельный алгоритм умножения плотных матриц. Элементы типа double. Блочная схема, алгоритм Фокса.
2. Провести вычислительные эксперименты.

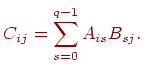
# Метод решения

Алгоритм FOX имеет некоторые требования: входная матрица должна быть квадратной, а процесс должен быть квадратным числом(4,9,16…), а квадратная матрица должна быть равномерно разделена для каждого процесса. На самом деле, условия квадратной матрицы необязательно необходимы, но это увеличит сложность программирования.

Используется блочная схема разбиения матрицы. При таком способе разделения данных исходные матрицы А, В и результирующая матрица С представляются в виде наборов блоков. Далее предполагается что все матрицы являются квадратными размера n×n, количество блоков по горизонтали и вертикали одинаково и равно q (т.е. размер всех блоков равен k×k, k=n/q). При таком представлении данных операция матричного умножения матриц А и B в блочном виде может быть представлена так:



,где каждый блок результирующей матрицы С определяется по формуле:



За основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока Cij – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (i,j) располагается четыре матричных блока:

1. Блок Cij матрицы C, вычисляемый подзадачей;

2. Блок Aij матрицы A, размещаемый в подзадаче перед началом вычислений;

3. Блоки A'ij , B'ij матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

1. Этап инициализации, на котором каждой подзадаче (i,j) передаются блоки Aij, Bij и обнуляются блоки Cij на всех подзадачах;

2. Этап вычислений, в рамках которого на каждой итерации l, l от нуля (включительно) до q, осуществляются следующие операции:

a. Для каждой строки i (i от нуля включительно и строго до q) блок Aij подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением j=(i+l)mod q, где mod есть операция получения остатка от целочисленного деления;

b. Полученные в результаты пересылок блоки A'ij, B'ij каждой подзадачи (i,j) перемножаются и прибавляются к блоку Cij

c. Блоки B'ij каждой подзадачи (i,j) пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

# Схема распараллеливания

У нас имеется четыре переменных A B C T. Первая переменная предоставит некоторую информацию о rank и id процесса. Вторая переменная делает процесс mpi и параллельную строку и столбец. Третья переменная дает нам информацию о потребляемом втором завершенном процессе, и четвертая переменная делает связь между MPI\_Bcast MPI\_Isend MPI\_Wait.

# Описание программной реализации

#include "mpi.h"

#include <algorithm>

#include <fstream>

#include <cmath>

const int root\_id = 0;

const int max\_procs\_size = 16;

int main(int argc, char \*argv[])

{

double start\_time, end\_time, time;

int procs\_id, procs\_size;

MPI\_Status status;

MPI\_Request reqSend, reqRecv;

MPI\_Init(&argc, &argv);

start\_time = MPI\_Wtime();

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

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procs\_id);

int N = 0;

{

for (int i = 1; i < argc; ++i) {

char \* pos = strstr(argv[i], "-N=");

if (pos != NULL) {

scanf\_s(pos, "-N=%d", &N);

break;

}

}

}

const int procs\_size\_sqrt = floor(sqrt(static\_cast<double>(procs\_size)));

const int n = N / procs\_size\_sqrt;

const int n\_sqr = n \* n;

if (procs\_size<4 || procs\_size> max\_procs\_size) {

printf("The fox algorithm requires at least 4 processors and at most %d processors. ",

max\_procs\_size);

MPI\_Finalize();

return 0;

}

if (procs\_size\_sqrt\*procs\_size\_sqrt != procs\_size) {

printf("The number of process must be a square. ");

MPI\_Finalize();

return 0;

}

if (N % procs\_size\_sqrt != 0) {

printf("N mod procs\_size\_sqrt !=0 ");

MPI\_Finalize();

return 0;

}

int \* A = new int[n\_sqr];

int \* B = new int[n\_sqr];

int \* C = new int[n\_sqr];

int \* T = new int[n\_sqr];

for (int i = 0; i < n; ++i)

for (int j = 0; j < n; ++j) {

A[i\*n + j] = (i + j)\*procs\_id;

B[i\*n + j] = (i - j)\*procs\_id;

C[i\*n + j] = 0;

}

printf("A on procs %d : ", procs\_id);

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

printf("%5d", A[i\*n + j]);

}

printf(" ");

}

printf("B on procs %d : ", procs\_id);

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

printf("%5d", B[i\*n + j]);

}

printf(" ");

}

MPI\_Comm cart\_all, cart\_row, cart\_col;

int dims[2], periods[2];

int procs\_cart\_rank, procs\_coords[2];

dims[0] = dims[1] = procs\_size\_sqrt;

periods[0] = periods[1] = true;

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, false, &cart\_all);

MPI\_Comm\_rank(cart\_all, &procs\_cart\_rank);

MPI\_Cart\_coords(cart\_all, procs\_cart\_rank, 2, procs\_coords);

MPI\_Comm\_split(cart\_all, procs\_coords[0], procs\_coords[1], &cart\_row);

MPI\_Comm\_split(cart\_all, procs\_coords[1], procs\_coords[0], &cart\_col);

int rank\_cart\_row, rank\_cart\_col;

MPI\_Comm\_rank(cart\_row, &rank\_cart\_row);

MPI\_Comm\_rank(cart\_col, &rank\_cart\_col);

for (int round = 0; round < procs\_size\_sqrt; ++round) {

MPI\_Isend(B, n\_sqr, MPI\_INT, (procs\_coords[0] - 1 + procs\_size\_sqrt) % procs\_size\_sqrt,

1, cart\_col, &reqSend);

int broader = (round + procs\_coords[0]) % procs\_size\_sqrt;

if (broader == procs\_coords[1]) std::copy(A, A + n\_sqr, T);

MPI\_Bcast(T, n\_sqr, MPI\_INT, broader, cart\_row);

for (int row = 0; row < n; ++row)

for (int col = 0; col < n; ++col)

for (int k = 0; k < n; ++k) {

C[row\*n + col] += T[row\*n + k] \* B[k\*n + col];

}

MPI\_Wait(&reqSend, &status);

MPI\_Recv(T, n\_sqr, MPI\_INT, (procs\_coords[0] + 1) % procs\_size\_sqrt

, 1, cart\_col, &status);

std::copy(T, T + n\_sqr, B);

}

printf("C on procs %d : ", procs\_id);

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

printf("%5d", C[i\*n + j]);

}

printf(" ");

}

MPI\_Comm\_free(&cart\_col);

MPI\_Comm\_free(&cart\_row);

MPI\_Comm\_free(&cart\_all);

delete[]A;

delete[]B;

delete[]C;

delete[]T;

end\_time = MPI\_Wtime();

MPI\_Finalize();

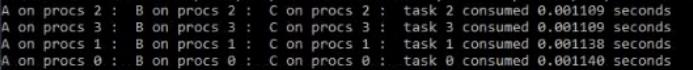
printf("task %d consumed %lf seconds ", procs\_id, end\_time - start\_time);

return 0;

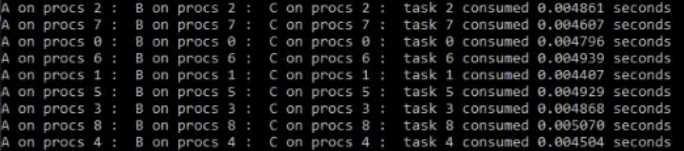
}

# Результаты экспериментов

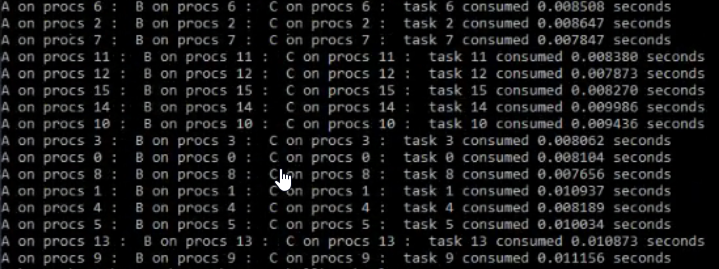
4 процесса:



9 процессов:



16 процессов:



# 

# Заключение

В ходе работы была реализован алгоритм умножения плотных матриц алгоритмом Фокса.

Алгоритм Фокса требует для своего выполнения q итераций, в ходе которых каждый процессор перемножает свои текущие блоки матриц А и В и прибавляет результаты умножения к текущему значению блока матрицы C.

Общий анализ сложности дает идеальные показатели эффективности параллельных вычислений.

# Список литературы

1. <http://www.hpcc.unn.ru/?dir=1034>

2. Хабр, сообщество IT-специалистов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/261777/