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

«Параллельные методы матричного умножения. Алгоритм Фокса»

Выполнил:
студент группы 381706-1
Максимова И.И.

Проверил:
Доцент кафедры МОСТ, кандидат технических наук,
Сысоев А.В.

Нижний Новгород
2019

Содержание

Введение

Матричное умножение является одной из существенных проблем в матричных вычислениях. Оно определяется соотношением

$$c_{ij} = \sum_{k=0}^{n-1} a_{i,k} \times b_{k,j}\,, \qquad 0 \le i,j < n \tag{1}$$

Нетрудно заметить, что умножение матриц требует выполнения большого количества операций - n^3 скалярных умножений и сложений. Что в свою очередь сильно сказывается на эффективности программ, использующих данную операцию. Встает необходимость распараллеливания вычислений.

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется *блочное представление* матриц. При таком подходе матрицы-операнды A, B и результирующая матрица C рассматриваются как набор блоков.

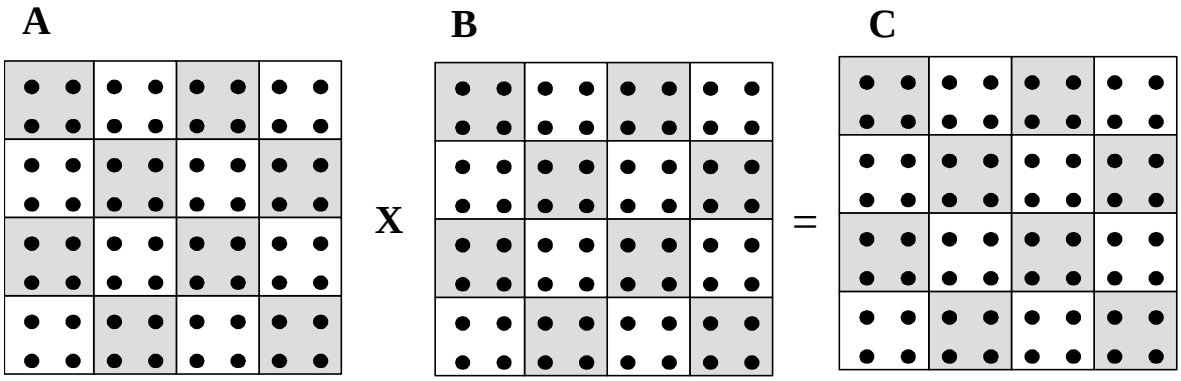

Рисунок 1. Блочная схема распределения данных.

Разбиение матриц на блоки значительно упрощает проблему выбора эффективных способов распараллеливания вычислений.

Одним из основных методов параллельного умножения матриц, использующий блочное распределение данных между процессами, является алгоритм Фокса. Он и будет подробно рассмотрен в настоящей работе.

Цель данной работы – реализовать параллельный алгоритм матричного умножения - алгоритм Фокса, и сравнить его с последовательным алгоритмом.

Постановка задачи

Даны две квадратные матрицы A и B размера $n \times n$. Элементы матриц – вещественные числа типа double. Результатом умножения матриц A и B является матрица C размера $n \times n$, каждый элемент которой определяется в соответствии с выражением (1)

Требуется перемножить матрицы A и B задействовав P вычислительных узлов. В качестве параллельного способа выполнения матричного умножения необходимо использовать алгоритм Фокса при блочном разделении данных.

Считается, что число вычислительных улов является квадратом некоторого натурального числа p и размер матриц кратен этому числу:

$$P = p^2, \qquad p \in \mathbb{N}^* \tag{2}$$

$$\frac{\mathrm{n}}{p} = z, \qquad z \in \mathbb{N}^* \tag{3}$$

Метод решения

В алгоритме Фокса используется блочная схема разбиения матриц - исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Количество блоков по горизонтали и вертикали одинаково и равно q (т.е. размер всех блоков равен $k \times k$, $k = n/q$). При таком представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена так:

$$\begin{pmatrix} A_{0,0} & \cdots & A_{0,q-1} \\ \vdots & \ddots & \vdots \\ A_{q-1,0} & \cdots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & \cdots & B_{0,q-1} \\ \vdots & \ddots & \vdots \\ B_{q-1,0} & \cdots & B_{q-1,q-1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & \cdots & C_{0,q-1} \\ \vdots & \ddots & \vdots \\ C_{q-1,0} & \cdots & C_{q-1,q-1} \end{pmatrix}, \quad (4)$$

где каждый блок результирующей матрицы C определяется по формуле

$$C_{ij} = \sum_{s=0}^{q-1} A_{i,s} \times B_{s,j} \quad (5)$$

Определим *базовую подзадачу* (i, j) как процедуру вычисления всех элементов блока $C_{i,j}$. Тем самым, все подзадачи образуют прямоугольную решетку размером $q \times q$. При этом, в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц A и B.

Выполнение алгоритма Фокса включает:

1. Этап инициализации.

   Каждой подзадаче (i, j) передаются блоки $A_{i,j}$, $B_{i,j}$ и обнуляются блоки $C_{i,j}$.

2. Этап вычислений.

   На каждой итерации I: $0 \le I < q$, осуществляются следующие операции:

   2.1.1. Для каждой строки i: $0 \le i < q$, блок $A_{i,j}$ подзадачи (i, j) пересылается на все подзадачи той же строки i решетки. Индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

   $$j = (i + I) \; mod \; q \quad (6)$$

   2.1.2. Полученные в результаты пересылок блоки $A'_{i,j}$, $B'_{i,j}$ каждой подзадачи (i, j) перемножаются и прибавляются к блоку $C_{i,j}$.

   2.1.3. Блоки $B'_{i,j}$ 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

Рассмотрим основные моменты в организации параллельных вычислений для выполнения алгоритма Фокса.

1. Построение топологии вычислительной системы.

   Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки размера $q \times q$, множество имеющихся процессов P также представляется в виде квадратной решетки размера $p \times p$, это возможно (см. (3)).

   Так как число процессов в данной работе является полным квадратом, можно выбрать количество блоков в матрицах по вертикали и горизонтали равным p (т.е. $q = p$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и тем самым достигается полная балансировка вычислительной нагрузки между процессами.

   Используя топологию вычислительной системы в виде квадратной решетки размера $p \times p$, производим отображение набора подзадач на множество процессов: базовая подзадача (i, j) располагается на процессе $P_{i,j}$.

2. Распределение памяти на процессах.

   В соответствии с алгоритмом Фокса, вычисления организованы так, что в каждый текущий момент времени каждая подзадача (i, j) содержат лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивается при помощи передачи данных между процессами. Таким образом, в ходе вычислений на каждой базовой подзадаче (i, j) располагается четыре матричных блока:

   - Блок $C_{i,j}$ матрицы C, вычисляемый подзадачей.
   - Блок $A_{i,j}$ матрицы A, размещаемый в подзадаче перед началом вычислений.
   - Блоки $A'_{i,j}$, $B'_{i,j}$ матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Описание программной реализации

Рассмотрим подробно основные моменты программной реализации параллельного алгоритма Фокса.

Описание глобальных переменных:

- *int procNum* - число задействованных вычислительных узлов;
- *int cartSize* - размер решетки процессов;
- *int blockSize* - размер матричных блоков;
- *int rank* - ранк процесса;
- *int coords[2]* - координаты процесса в решетке;
- *MPI_Comm comm_cart* - коммуникатор решетки;
- *MPI_Comm comm_row* - коммуникаторы строк;
- *MPI_Comm comm_col* - коммуникаторы столбцов.

Описание основных методов:

void Topology() - создает коммуникатор в виде двумерной квадратной решетки (используя функцию *MPI_Cart_create*), определяет координаты каждого процесса в этой решетке (*MPI_Cart_coords*), а также создает коммуникаторы отдельно для каждой строки и каждого столбца решетки(*MPI_Cart_sub*).

void Scatter(double Matrix, double* block, int size)* - обеспечивает выполнение блочного распределения матрицы *Matrix* размера $size \times size$ между процессами, организованными в двумерную квадратную решетку. Данные рассылаются в два этапа. На первом этапе матрица *Matrix*, находящаяся в «корневом» процессе с координатами (0,0), разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец решетки процессов (*MPI_Scatter* для коммуникатора нулевого столбца). Далее каждая полоса разделяется на блоки между процессами, составляющими строки решетки процессов (*MPI_Scatter* для каждого коммуникатора строк). Полученные на процессах блоки записываются в буфер *block*. В программе этот метод реализует распределение матриц-операндов A и B между процессами.

void ABlockTransfer(int it, double Ablock, double* AMatrixblock)* - выполняет рассылку блоков матрицы A по строкам решетки процессов в буфер приема *Ablock*. Для этого в каждой строке решетки определяется ведущий процесс (см. (6)), осуществляющий рассылку. Для рассылки используется блок *AMatrixblock*, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции *MPI_Bcast*. Следует отметить, что данная операция является коллективной и ее

локализация пределами отдельных строк решетки обеспечивается за счет использования коммуникаторов *comm_row*, определенных для набора процессов каждой строки решетки в отдельности.

void BlockMultiplication(double Ablock, double* Bblock, double* Cblock, int _BlockSize)* - обеспечивает перемножение блоков *Ablock* и *Bblock*, размерами *_BlockSize* в соответствии с формулой (1). Результат умножения записывается в *Cblock*.

void BblockTransfer(double Bblock)*_- выполняет циклический сдвиг блоков матрицы B по столбцам процессорной решетки вверх. Каждый процесс передает свой блок следующему процессу, расположенному выше в столбце решетки процессов, и получает блок, переданный из процесса, расположенного ниже. Выполнение операций передачи данных осуществляется при помощи функции *MPI_SendRecv_replace*, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти *Bblock*.

void FoxAlg(double Ablock, double* AMatrixblock, double* Bblock, double* Cblock)* – реализует логику работы параллельного алгоритма Фокса, описанную в главе «Методы решения»: В цикле от 0 до *GridSize* происходит рассылка блоков *AMatrixblock,* по строкам процессной решетки в буферы *Ablock* (функция *ABlockTransfer*), затем выполняется умножение блоков (*BlockMultiplication*). Результат умножения записывается в *Cblock***.** После чего осуществляется циклический сдвиг блоков *Bblock* в столбцах процессной решетки (*BblockTransfer*).

void ImportResult(double C, double* Cblock, int size)* - обеспечивает сбор результирующей матрицы C размера $size \times size$ из блоков *Cblock*. Сбор данных также выполняется при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы *Scatter*.

void Fox(double A, double* B, double* C, int size)* - главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы, описанные выше. В качестве аргументов принимает матрицы A и B участвующие в умножении, матрицу C для записи результата и их размер $size$.

Подтверждение корректности

Для подтверждения корректности в программе реализован набор тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework:

Test_on_Matrix_size_4 - проверка корректности реализации алгоритма Фокса и последовательного алгоритма умножения матриц. Размер матриц равен 4x4. Проверка выполнения программы, при условии, что на каждом процессе будет находиться по одному элементу каждой матрицы, участвующей в умножении.

Test_on_Matrix_size_16 - проверка корректности выполнения алгоритма Фокса, при условии что размер блоков равен 8x8;

Test_on_Matrix_size_64 - проверка корректности выполнения алгоритма Фокса, при условии что размер блоков равен 15x15;

Throw_Matrix_size_0 – проверка наличия исключения, при задании размера матрицы равным 0;

Throw_Matrix_size_11 - проверка наличия исключения при задании размера матрицы, не соответствующего условию (3);

Предполагается что тестирование ведется на четырех процессах.

Результаты экспериментов

Эксперименты проводились на ПК со следующими параметрами:

- Операционная система: Windows 10 Домашняя
- Процессор: Intel(R) Core™ i5-7200U CPU @ 2.70 GHz
- Версия Visual Studio: 2019
- Версия MPI: Microsoft MPI v10.0

В рамках эксперимента было вычислено время работы параллельного (алгоритм Фокса) и последовательно алгоритмов умножения квадратных матриц A и B. Размер матриц изменяется в диапазоне от [500; 2500] с шагом 500. Элементами матриц являются вещественные числа типа double. Эксперименты проводятся на 4, 9 и 49 процессах. Время выражено в секундах. Ускорение рассчитывается по формуле:

$$\text{Ускорение} = \frac{\text{время последовательного алгоритма}}{\text{время параллельного алгоритма}} \quad (7)$$

Размер Матриц	Время послед. алгоритма.	Параллельный алгоритм					
		4 процесса		9 процессов		49 процессов	
		время	ускорение	время	Ускорение	время	ускорение
500	0.637	0.236	2.699	0.309	2.061	1.951	0.326
1000	8.132	3.707	2.194	2.681	3.033	6.773	1.201
1500	23.189	12.065	1.922	10.426	2.224	20.947	1.107
2000	62.404	32.366	1.928	25.778	2.421	32.814	1.901
2500	159.031	72.379	2.197	56.994	2.790	59.324	2.68

Таблица 1. Результаты вычислительных экспериментов по исследованию параллельного алгоритма Фокса.

На графиках ниже изображены полученные результаты.

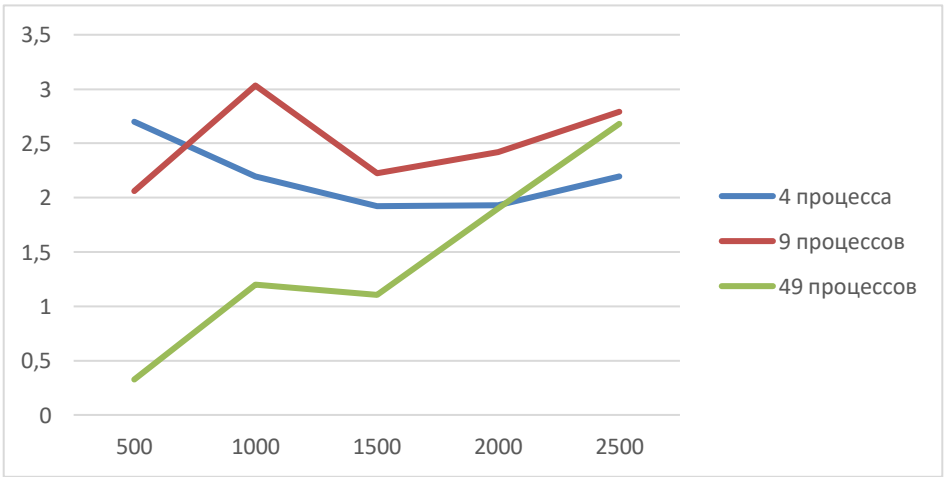

График 1. Зависимость ускорения от размера матриц.

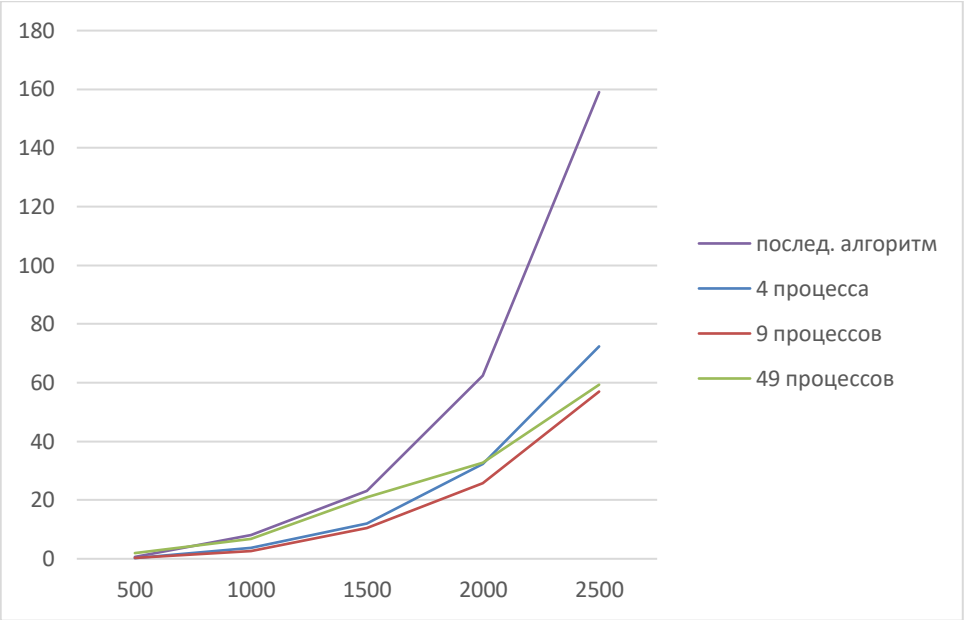

График 2. Зависимость времени работы алгоритмов умножения матриц от их размера.

По данным экспериментов видно, что алгоритм Фокса работает намного быстрее, чем классический алгоритм умножения матриц. Наибольшая эффективность достигается при достаточно большом размере матриц - размер матриц должен превышать 1000 элементов.

Также заметим, что размер матриц и число процессов должны быть прямо пропорциональны друг другу, для поддержки наилучшей эффективности. Так, из графика ускорения видно, что при малом размере матриц 49 процессов дают низкую эффективность из-за больших латентностей, возникающих при передаче данных между процессами. В то время как операции умножения блоков выполняются быстро.

Таким образом, можно сделать вывод: чем меньше матрица, тем меньшее число процессов нужно использовать. И наоборот, чем больше матрица, тем большее число процессов необходимо для эффективного умножения.

Заключение

В результате выполнения лабораторной работы была разработана библиотека, реализующая параллельный метод матричного умножения – алгоритм Фокса, используя технологию MPI.

Для подтверждения корректности работы программы разработан и доведен до успешного выполнения набор тестов с использованием библиотеки модульного тестирования Google C++ Testing Framework.

По данным экспериментов удалось сравнить время работы параллельного и последовательного алгоритмов умножения матриц. Выявлено, что параллельный алгоритм Фокса показывает высокую эффективность на достаточно большом объеме данных. И число задействованных вычислительных узлов должно быть прямо пропорционально размеру матриц, участвующих в умножении.

Литература

Книги:

- Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. 184 с. ISBN 5-85746-602-4.

Internet-ресурсы:

- Национальный Открытый Университет «ИНТУИТ». Академия: Интернет-Университет Суперкомпьютерных Технологий. Курс: Теория и практика параллельных вычислений. Автор: Виктор Гергель. ISBN: 978-5-9556-0096-3. URL: https://www.intuit.ru/studies/courses/1156/190/info
- Центр суперкомпьютерных технологий, официальный сайт. Нижегородский государственный университет им. Н.И. Лобачевского. URL: http://www.hpcc.unn.ru/?dir=1034

Приложение

maximova_i_fox.h

```
// Copyright 2019 Maximova Irina
#ifndef MODULES_TASK_3_MAXIMOVA_I_FOX_FOX_H_
#define MODULES_TASK_3_MAXIMOVA_I_FOX_FOX_H_

void Fox(double* A, double* B, double* C, int size);
void SequentialAlgorithm(double* A, double* B, double* C, int size);
void RandomOperandMatrix(double* A, double* B, int size);

#endif  // MODULES_TASK_3_MAXIMOVA_I_FOX_FOX_H_
```

maximova_i_fox.cpp

```
// Copyright 2019 Maximova Irina
#include <mpi.h>
#include <cmath>
#include <ctime>
#include <random>
#include <stdexcept>
#include "../../../modules/task_3/maximova_i_fox/fox.h"

#define ndims 2
#define reoder true

int procNum;
int rank;
int cartSize;
int blockSize;
MPI_Comm comm_cart;
MPI_Comm comm_row;
MPI_Comm comm_col;
int coords[2];

void ImportResult(double* C, double* Cblock, int size) {
  double* rowbuff = new double[size * blockSize];
  for (int i = 0; i < blockSize; i++) {
    MPI_Gather(&Cblock[i * blockSize], blockSize, MPI_DOUBLE,
               &rowbuff[i * size], blockSize, MPI_DOUBLE, 0, comm_row);
  }
  if (coords[1] == 0) {
    MPI_Gather(rowbuff, blockSize * size, MPI_DOUBLE, C, blockSize * size,
               MPI_DOUBLE, 0, comm_col);
  }
  delete[] rowbuff;
}

void Scatter(double* Matrix, double* block, int size) {
  double* rowbuff = new double[blockSize * size];
  if (coords[1] == 0) {
    MPI_Scatter(Matrix, blockSize * size, MPI_DOUBLE, rowbuff, blockSize * size,
                MPI_DOUBLE, 0, comm_col);
  }
  for (int i = 0; i < blockSize; i++) {
    MPI_Scatter(&rowbuff[i * size], blockSize, MPI_DOUBLE,
                &(block[i * blockSize]), blockSize, MPI_DOUBLE, 0, comm_row);
  }
  delete[] rowbuff;
}
```

```
void BlockExport(double* A, double* B, double* AMatrixblock, double* Bblock,
                 int size) {
  Scatter(A, AMatrixblock, size);
  Scatter(B, Bblock, size);
}

void ABlockTransfer(int it, double* Ablock, double* AMatrixblock) {
  int main = (coords[0] + it) % cartSize;
  if (coords[1] == main) {
    for (int i = 0; i < blockSize * blockSize; ++i) Ablock[i] = AMatrixblock[i];
  }
  MPI_Bcast(Ablock, blockSize * blockSize, MPI_DOUBLE, main, comm_row);
}

void BlockMultiplication(double* Ablock, double* Bblock, double* Cblock,
                         int _blockSize) {
  double temp;
  for (int i = 0; i < _blockSize; i++)
    for (int j = 0; j < _blockSize; j++) {
      temp = 0;
      for (int k = 0; k < _blockSize; k++)
        temp += Ablock[i * _blockSize + k] * Bblock[k * _blockSize + j];
      Cblock[i * _blockSize + j] += temp;
    }
}

void BblockTransfer(double* Bblock) {
  MPI_Status status;
  int dest = coords[0] - 1;
  if (coords[0] == 0) dest = cartSize - 1;
  int source = coords[0] + 1;
  if (coords[0] == cartSize - 1) source = 0;
  MPI_Sendrecv_replace(Bblock, blockSize * blockSize, MPI_DOUBLE, dest, 0,
                       source, 0, comm_col, &status);
}

void FoxAlg(double* Ablock, double* AMatrixblock, double* Bblock,
            double* Cblock) {
  for (int it = 0; it < cartSize; ++it) {
    ABlockTransfer(it, Ablock, AMatrixblock);
    BlockMultiplication(Ablock, Bblock, Cblock, blockSize);
    BblockTransfer(Bblock);
  }
}

void Topology() {
  int dimSize[2] = {cartSize, cartSize};
  int periods[2] = {false, false};
  MPI_Cart_create(MPI_COMM_WORLD, ndims, dimSize, periods, reoder, &comm_cart);
  MPI_Cart_coords(comm_cart, rank, ndims, coords);

  int subdims[2];
  subdims[0] = false;
  subdims[1] = true;
  MPI_Cart_sub(comm_cart, subdims, &comm_row);

  subdims[0] = true;
  subdims[1] = false;
  MPI_Cart_sub(comm_cart, subdims, &comm_col);
}

void Fox(double* A, double* B, double* C, int size) {
  if (size <= 0) throw std::runtime_error("Wrong size matrix");

  MPI_Comm_size(MPI_COMM_WORLD, &procNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  if (procNum == 1) {
    SequentialAlgorithm(A, B, C, size);
  } else {
    cartSize = static_cast<int>(std::sqrt(procNum));
    if (cartSize * cartSize != procNum)
      throw std::runtime_error("Wrong number of processes");
    if (size % cartSize != 0) throw std::runtime_error("Wrong size matrix");
    blockSize = size / cartSize;

    int numElemBlock = blockSize * blockSize;
    double* Ablock = new double[numElemBlock];
    double* Bblock = new double[numElemBlock];
    double* Cblock = new double[numElemBlock];
    double* AMatrixblock = new double[numElemBlock];
    for (int i = 0; i < numElemBlock; i++) Cblock[i] = 0;

    Topology();

    BlockExport(A, B, AMatrixblock, Bblock, size);

    FoxAlg(Ablock, AMatrixblock, Bblock, Cblock);

    ImportResult(C, Cblock, size);

    delete[] Ablock;
    delete[] Bblock;
    delete[] Cblock;
    delete[] AMatrixblock;
  }
}

void SequentialAlgorithm(double* A, double* B, double* C, int size) {
  if (size <= 0) throw std::runtime_error("Wrong size matrix");
  for (int i = 0; i < size * size; ++i) C[i] = 0;
  BlockMultiplication(A, B, C, size);
}

void RandomOperandMatrix(double* A, double* B, int size) {
  std::mt19937 gen;
  gen.seed((unsigned)time(0));
  for (int i = 0; i < size * size; ++i) {
    A[i] = gen() % 5 + static_cast<float>(gen() % 10) / 10;
    B[i] = gen() % 5 + static_cast<float>(gen() % 10) / 10;
  }
}
```

main.cpp

```
// Copyright 2019 Maximova Irina
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <cmath>
#include "./fox.h"

TEST(Fox, Test_on_Matrix_size_4) {
  int size = 4;
  double tmp;
  double* A = &tmp;
  double* B = &tmp;
  double* C = &tmp;
  double* CFox = &tmp;
  int rank, procNum;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procNum);
```

```
  int cartSize = static_cast<int>(std::sqrt(procNum));
  if (cartSize * cartSize == procNum && size % cartSize == 0) {
    if (rank == 0) {
      A = new double[size * size];
      B = new double[size * size];
      RandomOperandMatrix(A, B, size);
      C = new double[size * size];
      CFox = new double[size * size];
      SequentialAlgorithm(A, B, C, size);
    }
    Fox(A, B, CFox, size);

    if (rank == 0) {
      for (int i = 0; i < size * size; ++i) ASSERT_NEAR(CFox[i], C[i], 0.0001);
    }

    if (rank == 0) {
      delete[] A;
      delete[] B;
      delete[] C;
      delete[] CFox;
    }
  }
}

TEST(Fox, Test_on_Matrix_size_16) {
  int size = 16;
  double tmp;
  double* A = &tmp;
  double* B = &tmp;
  double* C = &tmp;
  double* CFox = &tmp;
  int rank, procNum;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procNum);

  int cartSize = static_cast<int>(std::sqrt(procNum));
  if (cartSize * cartSize == procNum && size % cartSize == 0) {
    if (rank == 0) {
      A = new double[size * size];
      B = new double[size * size];
      RandomOperandMatrix(A, B, size);
      C = new double[size * size];
      CFox = new double[size * size];
      SequentialAlgorithm(A, B, C, size);
    }
    Fox(A, B, CFox, size);

    if (rank == 0) {
      for (int i = 0; i < size * size; ++i) ASSERT_NEAR(CFox[i], C[i], 0.0001);;
    }

    if (rank == 0) {
      delete[] A;
      delete[] B;
      delete[] C;
      delete[] CFox;
    }
  }
}

TEST(Fox, Test_on_Matrix_size_64) {
  int size = 64;
  double temp;
  double* A = &temp;
```

```
  double* B = &temp;
  double* C = &temp;
  double* CFox = &temp;
  int rank, procNum;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procNum);

  int cartSize = static_cast<int>(std::sqrt(procNum));
  if (cartSize * cartSize == procNum && size % cartSize == 0) {
    if (rank == 0) {
      A = new double[size * size];
      B = new double[size * size];
      RandomOperandMatrix(A, B, size);
      C = new double[size * size];
      CFox = new double[size * size];
      SequentialAlgorithm(A, B, C, size);
    }
    Fox(A, B, CFox, size);

    if (rank == 0) {
      for (int i = 0; i < size * size; ++i) ASSERT_NEAR(CFox[i], C[i], 0.0001);
    }

    if (rank == 0) {
      delete[] A;
      delete[] B;
      delete[] C;
      delete[] CFox;
    }
  }
}

TEST(Fox, Throw_Matrix_size_0) {
  int size = 0;
  double tmp;
  double* M = &tmp;
  ASSERT_ANY_THROW(Fox(M, M, M, size));
  ASSERT_ANY_THROW(SequentialAlgorithm(M, M, M, size));
}

TEST(Fox, Throw_Matrix_size_11) {
  int size = 13;
  double tmp;
  double* M = &tmp;
  int procNum;
  MPI_Comm_size(MPI_COMM_WORLD, &procNum);

  if (procNum > 1) {
    ASSERT_ANY_THROW(Fox(M, M, M, size));
  }
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  MPI_Init(&argc, &argv);

  ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
  ::testing::TestEventListeners& listeners =
      ::testing::UnitTest::GetInstance()->listeners();

  listeners.Release(listeners.default_result_printer());
  listeners.Release(listeners.default_xml_generator());

  listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
  return RUN_ALL_TESTS();
}
```