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

Кафедра алгебры, геометрии и дискретной математики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

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

Выполнил: студент группы 381706-02
Окмянский Андрей Владимирович

_____ Подпись

Научный руководитель:
Доцент кафедры МОСТ
Сысоев Александр Владимирович

_____ Подпись

Нижний Новгород
2019

Оглавление

Введение

В современном мире многие вычисления уже не производятся вручную – для этого существует специальная вычислительная техника, ведь проводимые эксперименты становятся сложнее: объём выборки и трудоёмкость алгоритмов уже не позволяют производить расчёты на бумаге – такой объём работы может оказаться просто не под силу даже группе людей, к тому же в таком случае нельзя гарантировать правильность подсчётов наверняка, так как имеет место быть человеческих фактор. Для больших объёмов вычислений уже давно используют специальную технику и имеющиеся алгоритмы, однако даже при использовании производящих подсчёты устройств может пройти достаточно много времени до получения результата. А если это не научный эксперимент, результаты которого в дальнейшем станут почвой для размышлений больших умов, а программа, работающая в реальном времени, то вопрос скорости получения ответа встаёт более остро. Здесь нам на помощь приходит понятие оптимизации.

Оптимизация — процесс максимизации выгодных характеристик, соотношений (например, оптимизация производственных процессов и производства), и минимизации расходов.

В нашем случае речь идёт о минимизации времени вычислений путём распараллеливания программы на некоторое количество процессов, так как время в контексте поставленной задачи – самый ценный ресурс.

Реализация блочного алгоритма Кэннона для умножения плотных матриц основывается на распараллеливании программы, что позволяет сократить время работы алгоритма, в отличие от стандартного алгоритма умножения плотных матриц.

Постановка задачи

Одной из основных операций над матрицами является их перемножение. Эта операция используется практически везде: от компьютерной графики до расчета поверхностей самолетов.

Цель данной работы - реализация параллельного и последовательного алгоритмов, вычисляющих произведение квадратных матриц $n * n$. В результате работы программы, мы должны получить еще одну матрицу порядка $n * n$, являющуюся произведением двух исходных, тестирование работоспособности написанных алгоритмов и анализ полученных результатов и эффективности на основе времени работы программы. Реализация параллельного алгоритма будет проводиться средствами MPI.

Описание алгоритма

Суть алгоритма Кэннона заключается в разбиении матриц A и B на блоки одинакового размера. в качестве базовой подзадачи выберем вычисления, связанные с определением одного из блоков результирующей матрицы C. Для вычисления элементов этого блока подзадача должна иметь доступ к элементам горизонтальной полосы матрицы A и элементам вертикальной полосы матрицы B.

Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы блоки в подзадачах могли бы быть перемножены без каких-либо дополнительных передач данных. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между подзадачами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

Этап инициализации алгоритма Кэннона (рис. 1) включает выполнение следующих операций передач данных:

- в каждую подзадачу (i,j) передаются блоки A_{ij}, B_{ij};
- для каждой строки i решетки подзадач блоки матрицы A сдвигаются на (i-1) позиций влево;
- для каждого столбца j решетки подзадач блоки матрицы B сдвигаются на (j-1) позиций вверх.

Результат подобного перераспределения блоков в случае, когда количество блоков равно 2, приведен на рисунке 1.

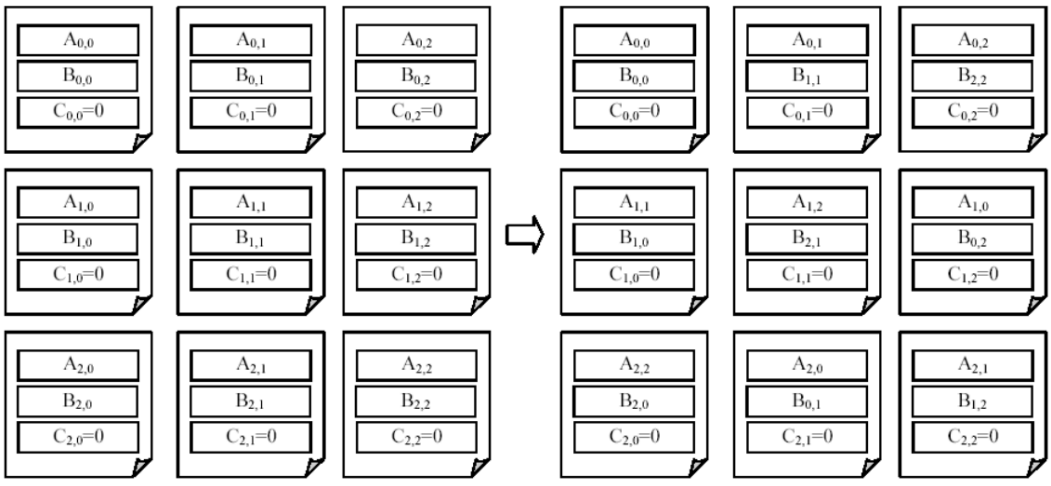

Рисунок 1.

В результате такого начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных. Кроме того, получение всех последующих блоков для подзадач может быть обеспечено при помощи простых коммуникационных действий — после выполнения операции блочного умножения каждый блок матрицы A должен быть передан предшествующей подзадаче влево по строкам решетки подзадач, а каждый блок матрицы B – предшествующей подзадаче вверх по столбцам решетки. Как можно показать, последовательность таких циклических сдвигов и умножение получаемых блоков исходных матриц A и B приведет к получению в базовых подзадачах соответствующих блоков результирующей матрицы C.

Схема распараллеливания

В первую очередь создается некоторое число p исполняющих процессов. Это число должно быть полным квадратом. Процессы организуются в виртуальную декартову топологию. Исходные матрицы должны иметь размерность, кратную $\sqrt{p}$. Матрицы разбиваются на равное количество квадратных блоков. Блоки исходных матриц распределяются по исполняющим процессам. Затем для каждой строки i декартовой решетки блоки матрицы A сдвигаются на (i-1) позиций влево, для каждого столбца j декартовой решетки блоки матрицы B сдвигаются на (j-1) позиций вверх.

Далее проводится $\sqrt{p}$ итераций, во время которых сначала происходит перемножение блоков по методу трех вложенных циклов, и произведение складывается с текущим значением результирующего блока. Затем выполняется циклический сдвиг блоков матрицы A вдоль строк решетки и циклический сдвиг блоков матрицы B вверх по столбцам виртуальной решетки. После выполнения всех итераций результирующая матрица собирается из полученных на каждом процессе результирующих блоков.

Описание программной реализации

Для решения поставленной задачи используется три основные функции и одна вспомогательная:

- std::vector<double> getRandomMatrix(int size) – генерирование случайной матрицы, размером size;
- std::vector<double> Multyplication(const std::vector<double> A, const std::vector<double> B) – последовательное умножение матриц;
- std::vector<double> getParallelMult(const std::vector<double> A, const std::vector<double> B, const int matrSize) – параллельное умножение матриц;
- std::vector<double> Add(std::vector<double> A, std::vector<double> B, int Size) – сложение матриц A и B.

Подтверждение корректности

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

Test_Equals_Parallel_And_Sequintial_Size_Number_x_10 – проверка корректности реализации алгоритма Кэннона и последовательного алгоритма умножения матриц. Размер матриц равен 10*количество процессов. Проверка выполнения программы, при условии, что на каждом процессе будет находиться по 10 элементов каждой матрицы, участвующей в умножении.

Test_Equals_Parallel_And_Sequintial_Size_Number_x_50 – проверка корректности выполнения алгоритма Кэннона, при условии, что размер матриц равен 50*количество процессов;

Test_Equals_Parallel_And_Sequintial_Size_Number_x_100 – проверка корректности выполнения алгоритма Кэннона, при условии, что размер матриц равен 100*количество процессов;

Test_Different_Size_Sequintial – проверка наличия исключения, при разных размерах матриц A и B при последовательном умножении;

Test_Different_Size_Parallel – проверка наличия исключения, при разных размерах матриц A и B при параллельном умножении;

Test_Matrix_Size_Zero – проверка наличия исключения, при задании размера матрицы равного 0;

Результаты экспериментов

Эксперименты проводились на ПК со следующими параметрами:

- Операционная система: Windows 10 Домашняя
- Процессор: Intel(R) Core™ i3-4030U CPU @ 1.90 GHz
- Версия Visual Studio: 2017
- Версия MPI: Microsoft MPI v10.0

В рамках эксперимента было вычислено время работы параллельного (алгоритм Кэннона) и последовательно алгоритмов умножения квадратных матриц A и B. Размер матриц изменяется в диапазоне [500; 2500]. Элементами матриц являются вещественные числа типа double. Эксперименты проводятся на 4, 9 и 16 процессах. Время выражено в секундах. Ускорение рассчитывается по формуле:

$$\text{Ускорение} = \frac{\text{время последовательного алгоритма}}{\text{время параллельного алгоритма}}$$

Размер матрицы	Время послед. алгоритма	Параллельный алгоритм					
		4 процесса		9 процессов		16 процессов	
		время	ускорение	время	ускорение	время	ускорение
576	1.549	0.641	2.416	0.505	3.067	1.502	1.031
1008	16.483	2.669	6.288	3.946	4.253	3.657	4.589
1440	63.296	15.807	4.004	11.593	5.459	8.751	7.233
2016	175.396	70.915	2.473	37.345	4.696	26.647	6.582
2448	336.194	133.589	2.516	66.397	5.063	60.686	5.539

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

Полученные данные демонстрируют разность во времени работы при последовательном и параллельном вычислениях. По результатам можно сделать вывод, что параллельное выполнение программы выигрывает во времени у последовательного во всех случаях уравнений. Причём чем больше процессов, тем быстрее работает параллельная программа.

Вывод

В результате выполнения лабораторной работы была разработана библиотека, реализующая параллельный метод матричного умножения – алгоритм Кэннона, 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 Кэннона показывает высокую эффективность на достаточно большом объеме данных.

Список используемой литературы

1. 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.
2. http://www.hpcc.unn.ru/?dir=883
3. http://edu.mmcs.sfedu.ru/file.php/74/MPIMatr.pdf

Приложение

cannon_algorithm.h:

```
// Copyright 2019 Okmyanskiy Andrey

#ifndef MODULES_TASK_3_OKMYANSKIY_A_CANNON_ALGORITHM_CANNON_ALGORITHM_H_
#define MODULES_TASK_3_OKMYANSKIY_A_CANNON_ALGORITHM_CANNON_ALGORITHM_H_

#include <mpi.h>
#include <vector>

std::vector<double> getRandomMatrix(int size);
std::vector<double> Multyplication(const std::vector<double> A, const std::vector<double> B);
std::vector<double> Add(std::vector<double> A, std::vector<double> B, int Size);
std::vector<double> getParallelMult(const std::vector<double> A,
    const std::vector<double> B, const int matrSize);

#endif  // MODULES_TASK_3_OKMYANSKIY_A_CANNON_ALGORITHM_CANNON_ALGORITHM_H_
```

cannon_algorithm.cpp:

```
// Copyright 2019 Okmyanskiy Andrey
#include <iostream>
#include <random>
#include <vector>
#include <ctime>
#include <stdexcept>
#include "../../../modules/task_3/okmyanskiy_a_cannon_algorithm/cannon_algorithm.h"

double eps = 1E-10;
std::vector<double> getRandomMatrix(int Size) {
    if (Size <= 0) {
        throw std::runtime_error("The size of the matrix <= 0");
    }
    static int s_count = 0;
    ++s_count;
    std::mt19937 gen;
    gen.seed(static_cast<unsigned int>(time(0) + s_count));
    std::vector<double> vec(Size);
    for (int i = 0; i < Size; i++) {
        vec[i] = gen() % 100;
    }
    return vec;
}
std::vector<double> Add(std::vector<double> A, const std::vector<double> B, int Size) {
    std::vector<double> C(Size);
    for (int i = 0; i < Size; i++) {
        C[i] = A[i] + B[i];
    }
    return C;
}
```

```
std::vector<double> Multyplication(const std::vector<double> A, const std::vector<double> B) {
    if (A.size() != B.size()) {
        throw std::runtime_error("Matrixes have different sizes");
    }
    std::vector<double> C(A.size());
    int root = static_cast<int>(sqrt(A.size()));
    for (int i = 0; i < root; i++) {
        for (int j = 0; j < root; j++) {
            C[i*root + j] = 0;
            for (int k = 0; k < root; k++) {
                C[i*root + j] += A[i*root + k] * B[k*root + j];
            }
        }
    }
    return C;
}

std::vector<double> getParallelMult(const std::vector<double> A,
    const std::vector<double> B, const int MatrSize) {
    if (A.size() != B.size()) {
        throw std::runtime_error("Matrixes have different sizes");
    }
    int ProcRank, ProcNum;
    MPI_Status Status;
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    if ((MatrSize*MatrSize) % ProcNum != 0) {
        throw std::runtime_error("Matrix size does not match");
    }
    int Size = static_cast<int>(MatrSize*MatrSize / ProcNum);
    int root = static_cast<int>(sqrt(Size));
    double rootTemp = sqrt(ProcNum);
    int rootProcNum = static_cast<int>(sqrt(ProcNum));
    if (fabs(rootTemp - rootProcNum) > eps) {
        throw std::runtime_error("The square root of a number processes is not an integer");
    }
    std::vector<double> BlockA(Size);
    std::vector<double> BlockB(Size);
    std::vector<double> BlockC(Size);
    std::vector<double> tempC(Size);
    std::vector<double> C(MatrSize*MatrSize);

    MPI_Datatype type, type2;
    MPI_Type_vector(root, root, root*rootProcNum, MPI_DOUBLE, &type);
    MPI_Type_contiguous(Size, MPI_DOUBLE, &type2);
    MPI_Type_commit(&type);
    MPI_Type_commit(&type2);

    MPI_Comm GridComm;
    int GridCoords[2];
    int dims[2] = { rootProcNum, rootProcNum };
    int periods[2] = { 1, 1 };
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &GridComm);
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

    int left_rank, right_rank, up_rank, down_rank;
    MPI_Cart_shift(GridComm, 1, -1, &right_rank, &left_rank);
    MPI_Cart_shift(GridComm, 0, -1, &down_rank, &up_rank);
```

```
if (ProcRank == 0) {
    for (int i = 0; i < rootProcNum; ++i) {
        for (int j = 0; j < rootProcNum; ++j) {
            if ((i != 0) || (j != 0)) {
                int dest, block_coords[2] = { i, j - i };
                if (block_coords[1] < 0) {
                    block_coords[1] += rootProcNum;
                }
                MPI_Cart_rank(GridComm, block_coords, &dest);
                MPI_Send(&A[i * Size * rootProcNum + j * root], 1, type,
                    dest, 1, MPI_COMM_WORLD);
            }
        }
    }
    for (int i = 0; i < rootProcNum; ++i) {
        for (int j = 0; j < rootProcNum; ++j) {
            if ((i != 0) || (j != 0)) {
                int dest, block_coords[2] = { i - j, j };
                if (block_coords[0] < 0) {
                    block_coords[0] += rootProcNum;
                }
                MPI_Cart_rank(GridComm, block_coords, &dest);
                MPI_Send(&B[i * Size * rootProcNum + j * root], 1, type,
                    dest, 2, MPI_COMM_WORLD);
            }
        }
    }
    for (int i = 0, j = -1; i < Size; i++) {
        if (i % root == 0) {
            j++;
        }
        BlockA[i] = A[i + j * root*(rootProcNum - 1)];
        BlockB[i] = B[i + j * root*(rootProcNum - 1)];
    }
} else {
    MPI_Recv(&BlockA[0], 1, type2, 0, 1, MPI_COMM_WORLD, &Status);
    MPI_Recv(&BlockB[0], 1, type2, 0, 2, MPI_COMM_WORLD, &Status);
}
BlockC = Multyplication(BlockA, BlockB);
for (int i = 1; i < rootProcNum; i++) {
    MPI_Sendrecv_replace(&BlockA[0], 1, type2, left_rank, 0, right_rank,
        0, MPI_COMM_WORLD, &Status);
    MPI_Sendrecv_replace(&BlockB[0], 1, type2, up_rank, 1, down_rank,
        1, MPI_COMM_WORLD, &Status);
    tempC = Multyplication(BlockA, BlockB);
    BlockC = Add(BlockC, tempC, Size);
}

if (ProcRank == 0) {
    for (int i = 0; i < root; ++i) {
        for (int j = 0; j < root; ++j) {
            C[i * root*rootProcNum + j] = BlockC[i * root + j];
        }
    }
    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(GridComm, block_coords, &source);
                MPI_Recv(&C[i * root*rootProcNum * root + j * root], 1, type,
                    source, 4, MPI_COMM_WORLD, &Status);
            }
```

```
            }
        }
    } else {
        MPI_Send(&BlockC[0], 1, type2, 0, 4, MPI_COMM_WORLD);
    }
    MPI_Type_free(&type);
    MPI_Type_free(&type2);
    MPI_Comm_free(&GridComm);

    return C;
}
```

main.cpp:

```
// Copyright 2019 Okmyanskiy Andrey
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <stdio.h>
#include <mpi.h>
#include <iostream>
#include <random>
#include <vector>
#include <ctime>
#include "../../../modules/task_3/okmyanskiy_a_cannon_algorithm/cannon_algorithm.h"

double epsilon = 1E-5;
int range1 = 10;
int range2 = 50;
int range3 = 500;
TEST(Parallel_, Test_Equals_Parallel_And_Sequintial_Size_Number_x_10) {
    int ProcRank, ProcNum;
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    int Size = ProcNum * range1;
    int root = static_cast<int> (sqrt(ProcNum));
    double root2 = sqrt(ProcNum);
    if (fabs(root2 - root) <= epsilon) {
        if ((Size*Size) % ProcNum == 0) {
            std::vector<double> A = getRandomMatrix(Size*Size);
            std::vector<double> B = getRandomMatrix(Size*Size);
            std::vector<double> C_parallel(Size*Size);

            double TotalTimeL = 0;
            double startL = MPI_Wtime();
            C_parallel = getParallelMult(A, B, Size);
            double endL = MPI_Wtime() - startL;

            if (ProcRank == 0) {
                std::cout << "ProcNum: " << ProcNum;
                std::cout << "\nSize: " << Size << "*" << Size << "\n";
                double startW = MPI_Wtime();
                std::vector<double> C_sequintial = Multyplication(A, B);
                double endW = MPI_Wtime() - startW;
                std::cout << "Parallel: " << endL;
                std::cout << "\nSequintial: " << endW << "\n";
                ASSERT_EQ(C_parallel, C_sequintial);
            }
        }
    }
}
```

```
TEST(Parallel_, Test_Equals_Parallel_And_Sequintial_Size_Number_x_50) {
    int ProcRank, ProcNum;
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    int Size = ProcNum * range2;
    int root = static_cast<int> (sqrt(ProcNum));
    double root2 = sqrt(ProcNum);
    if (fabs(root2 - root) <= epsilon) {
        if ((Size*Size) % ProcNum == 0) {
            std::vector<double> A = getRandomMatrix(Size*Size);
            std::vector<double> B = getRandomMatrix(Size*Size);
            std::vector<double> C_parallel(Size*Size);

            double TotalTimeL = 0;
            double startL = MPI_Wtime();
            C_parallel = getParallelMult(A, B, Size);
            double endL = MPI_Wtime() - startL;

            if (ProcRank == 0) {
                std::cout << "ProcNum: " << ProcNum;
                std::cout << "\nSize: " << Size << "*" << Size << "\n";
                double startW = MPI_Wtime();
                std::vector<double> C_sequintial = Multyplication(A, B);
                double endW = MPI_Wtime() - startW;
                std::cout << "Parallel: " << endL;
                std::cout << "\nSequintial: " << endW << "\n";
                ASSERT_EQ(C_parallel, C_sequintial);
            }
        }
    }
}

TEST(Parallel_, Test_Equals_Parallel_And_Sequintial_Size_Number_x_100) {
    int ProcRank, ProcNum;
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    int Size = ProcNum * range3;
    int root = static_cast<int> (sqrt(ProcNum));
    double root2 = sqrt(ProcNum);
    if (fabs(root2 - root) <= epsilon) {
        if ((Size*Size) % ProcNum == 0) {
            std::vector<double> A = getRandomMatrix(Size*Size);
            std::vector<double> B = getRandomMatrix(Size*Size);
            std::vector<double> C_parallel(Size*Size);
            double TotalTimeL = 0;
            double startL = MPI_Wtime();
            C_parallel = getParallelMult(A, B, Size);
            double endL = MPI_Wtime() - startL;

            if (ProcRank == 0) {
                std::cout << "ProcNum: " << ProcNum;
                std::cout << "\nSize: " << Size << "*" << Size << "\n";
                double startW = MPI_Wtime();
                std::vector<double> C_sequintial = Multyplication(A, B);
                double endW = MPI_Wtime() - startW;
                std::cout << "Parallel: " << endL;
                std::cout << "\nSequintial: " << endW << "\n";
                ASSERT_EQ(C_parallel, C_sequintial);
            }
        }
    }
}
```

```
TEST(Parallel_, Test_Different_Size_Sequintial) {
    int ProcRank, ProcNum;
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    int Size = ProcNum * range1;
    std::vector<double> A = getRandomMatrix(Size*Size);
    std::vector<double> B = getRandomMatrix(Size*Size + 1);
    if (ProcRank == 0) {
        ASSERT_ANY_THROW(Multyplication(A, B));
    }
}

TEST(Parallel, Test_Different_Size_Parallel) {
    int ProcRank, ProcNum;
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    int Size = ProcNum * range1;
    std::vector<double> A = getRandomMatrix(Size*Size);
    std::vector<double> B = getRandomMatrix(Size*Size + 1);
    if (ProcRank == 0) {
        ASSERT_ANY_THROW(getParallelMult(A, B, Size));
    }
}

TEST(Parallel, Test_Matrix_Size_Zero) {
    int ProcRank;
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    if (ProcRank == 0) {
        ASSERT_ANY_THROW(getRandomMatrix(0));
    }
}

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();
}
```