Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Пермский национальный исследовательский**

**политехнический университет»**

**(ПНИПУ)**

Факультет: Электротехнический

Кафедра: Информационные технологии и автоматизированные системы.

# Направление подготовки: Разработка программно-информационных систем.

**Лабораторная работа № 4**

Технология MPI. Умножение матриц.

Выполнил студент гр. РИС-18-1б

Гилязов Р.А.

(Фамилия, имя, отчество)

###### Проверил:

Старший преподаватель кафедры ИТАС, Щапов В.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(должность, Ф.И.О. руководителя от кафедры)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата)

**Пермь, 2020**

**Постановка задачи:**

1. Подключить MPI для локального компьютера.
2. Реализовать перемножение матриц с использованием технологии MPI
3. Протестировать программу с различным количеством ядер.

**Анализ задачи**

Для реализации проекта использовалась библиотека MSMPI v10.1.2 для Windows. Код, использующий данную библиотеку, легко импортировать для использования библиотеки MPICH. Для равномерной загрузки всех используемых процессов, используется формула: E = n\*p / count\_proc, где n – количество строк в новой матрице, p – количество столбиков в новой матрице, count\_proc – количество процессов, E - нагрузка. В случае, если количество процессов больше, чем размер новой матрицы, то остальные процессы простаивают. Если после распределения вычислений по процессам имеется остаток вычислений, то они дополнительно ложатся на выполнение последнему процессу. В программе используется определение начальной позиции вычисления по следующим формулам:

curr\_str = (current\_proc \* E) / p;

curr\_col = (current\_proc \* E) % p, где

curr\_str, curr\_col – положение начала вычислений для текущего процесса,

current\_proc – номер текущего процесса,

E – распределение вычислений по процессам,

p – количество элементов в колонке.

**Код программы**

#include <iostream>

#include <mutex>

#include <vector>

#include <random>

#include <mpi.h>

void multyply(

std::vector<std::vector<double>>& res, // Результат манипулирования данными

std::vector<std::vector<double>> const\* a, // Первая матрица

std::vector<std::vector<double>> const\* b, // Вторая матрица

int curr\_str, // Текущее начальное положение в строке

int curr\_col, // Текущее начальное положение в столбике

int end\_str, // Текущее конечное положение в строке

int end\_col) // Текущее конечное положение в столбике

{

for (int i = curr\_str; i <= end\_str; ++i) {

for (int j = curr\_col; j <= end\_col; ++j){

for (int k = 0; k < b->size(); ++k) {

res[i][j] += a->operator[](i)[k] \* b->operator[](k)[j];

}

}

}

}

void init\_vector(std::vector<std::vector<double>>& vec, int const& str, int const& col) {

std::default\_random\_engine generator;

std::normal\_distribution<double> randomize(-10, 100);

for (int i = 0; i < str; i++) {

std::vector<double> tmp;

for (int j = 0; j < col; j++) {

tmp.push\_back(randomize(generator));

}

vec.push\_back(tmp);

}

}

void resize\_vector(std::vector<std::vector<double>>& vec, int str, int col) {

for (int i = 0; i < vec.size(); i++) {

vec[i].resize(col);

}

vec.resize(str);

}

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

{

int const str\_a = 3;

int const col\_a = 2;

int const str\_b = 2;

int const col\_b = 3;

std::vector<std::vector<double>> a;

std::vector<std::vector<double>> b;

init\_vector(a, str\_a, col\_a);

init\_vector(b, str\_b, col\_b);

std::vector<std::vector<double>> res;

for (int i = 0; i < str\_a; ++i) {

std::vector<double> vec;

for (int j = 0; j < col\_b; ++j) {

vec.push\_back(0);

}

res.push\_back(vec);

}

int mpi\_count\_proc, // количество процессов

mpi\_init\_status, // статус инициализации процессов

mpi\_current\_rank; // номер текущего процесса

int E, // нагрузка

n = str\_a, // количество новых строк

p = col\_b, // количество новых столбиков

curr\_str = -1, // положение текущего процесса в матрице (№строки)

curr\_col = -1, // положение текущего процесса в матрице (№колонки)

end\_str = -1 , // конечное положение по номеру строки

end\_col = -1 ; // конечное положение по номеру колонки

// Инициализация MPI

if ((mpi\_init\_status = MPI\_Init(&argc, &argv)) != 0) {

return EXIT\_FAILURE;

}

MPI\_Comm\_size(MPI\_COMM\_WORLD, &mpi\_count\_proc); // получение числа процессов

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &mpi\_current\_rank); // получение текущего процесса

// выделение памяти под каждую матрицу в каждом процессе.

resize\_vector(a, a.size(), a[0].size());

resize\_vector(b, b.size(), b[0].size());

resize\_vector(res, res.size(), res[0].size());

// Передача векторов другим процессам

for (int i = 0; i < a.size(); i++) {

MPI\_Bcast(a[i].data(), a[i].size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

for (int i = 0; i < b.size(); i++) {

MPI\_Bcast(b[i].data(), b[i].size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

for (int i = 0; i < res.size(); i++) {

MPI\_Bcast(res[i].data(), res[i].size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

// определение начала и конца рассчета для текущего процесса

E = n \* p / mpi\_count\_proc;

if (E == 0) {

if (mpi\_current\_rank < n \* p) {

curr\_str = mpi\_current\_rank / p;

curr\_col = mpi\_current\_rank % p;

end\_str = (mpi\_current\_rank + 1) / p;

end\_col = (mpi\_current\_rank + 1) % p;

}

}

else {

curr\_str = (mpi\_current\_rank \* E) / p;

curr\_col = (mpi\_current\_rank \* E) % p;

end\_str = (mpi\_current\_rank \* E + E - 1) / p;

end\_col = (mpi\_current\_rank \* E + E - 1) % p;

}

if (mpi\_current\_rank == mpi\_count\_proc - 1) {

end\_str = n - 1;

end\_col = p - 1;

}

// начало измерения потоконезависимого таймера

double beg = MPI\_Wtime();

try {

// выполнение алгоритма

multyply(res, &a, &b, curr\_str, curr\_col, end\_str, end\_col);

}

catch (std::exception err) {

std::cerr << "Процесс №" << mpi\_current\_rank << " незадействуется!";

}

// синхронизация потоков

MPI\_Barrier(MPI\_COMM\_WORLD);

// сбор результата

double\*\* result = new double\* [res.size()];

for (int i = 0; i < res.size(); i++) {

result[i] = new double[res[i].size()];

}

for (int i = 0; i < res.size(); i++) {

MPI\_Reduce(res[i].data(), result[i], res[i].size(), MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

}

double end = MPI\_Wtime() - beg;

if (mpi\_current\_rank == 0) {

std::cout << "Runtime: " << end << " sec. \n";

}

MPI\_Finalize();

return EXIT\_SUCCESS;

}

**Тесты**

Таблица 1 – Замеры времени для разного количества процессов

|  |  |  |  |
| --- | --- | --- | --- |
| Размеры матрицы | 1 ядро, сек | 4 ядра, сек | 8 ядер, сек |
| 100х50\*50х100 | 0.0007341 | 0.0008727 | 0.001579 |
| 500х1000\*1000х400 | 0.283061 | 0.0940158 | 0.0522094 |
| 10000х500\*500х10000 | 83.6075 | 24.8048 | 22.8615 |

**Вывод:** после замеров времени от размеров перемножаемой матрицы и количеством ядер используемого процессора получились ожидаемые результаты. При малых размерах результирующей матрицы много времени выделяется на создание и синхронизацию потоков. Вследствие этого, время исполнения на 1 ядро выше. Однако при больших размерах матрицы время при 1 ядре в разы было больше, чем на 4 ядрах.