МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

*«Умножение матрицы на матрицу в MPI 2D решетка*»

студента 2 курса, группы 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

ассистент,

А.А.Ажбаков

Новосибирск 2023

# СОДЕРЖАНИЕ

[ЦЕЛИ 3](#_Toc165623241)

[ЗАДАНИЕ 3](#_Toc165623242)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc165623243)

[ЗАКЛЮЧЕНИЕ 8](#_Toc165623244)

[ПРИЛОЖЕНИЕ 1 9](#_Toc165623245)

[ПРИЛОЖЕНИЕ 2 10](#_Toc165623246)

# ЦЕЛИ

Исследовать производительность параллельной программы в зависимости от размера решетки.

# ЗАДАНИЕ

Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке.

Исследовать производительность параллельной программы в зависимости от размера матрицы и размера решетки.

Выполнить профилирование программы с помощью MPE при использовании 16-и ядер.

# ОПИСАНИЕ РАБОТЫ

Реализованная программа работает по алгоритму:

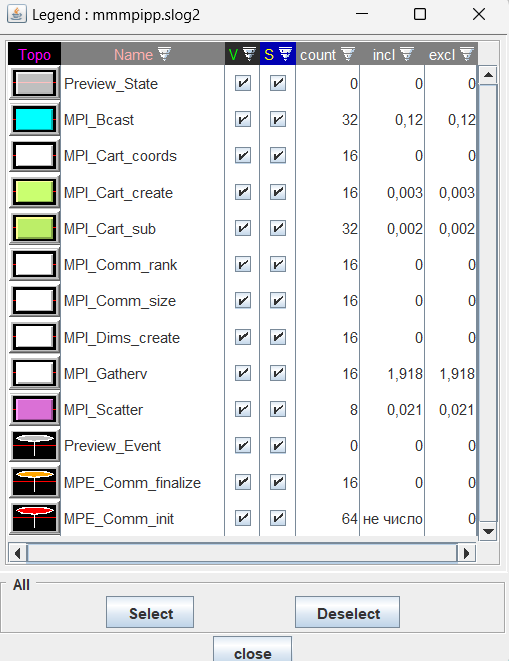
Вычисляется произведение С = А × В, где А – матрица размера 𝑛1 × 𝑛2 и В –матрица 𝑛2 × 𝑛3. Матрица результатов С имеет размер 𝑛1 × 𝑛3. Исходные матрицы первоначально доступны на нулевом процессе, и матрица результатов возвращена в нулевой процесс.

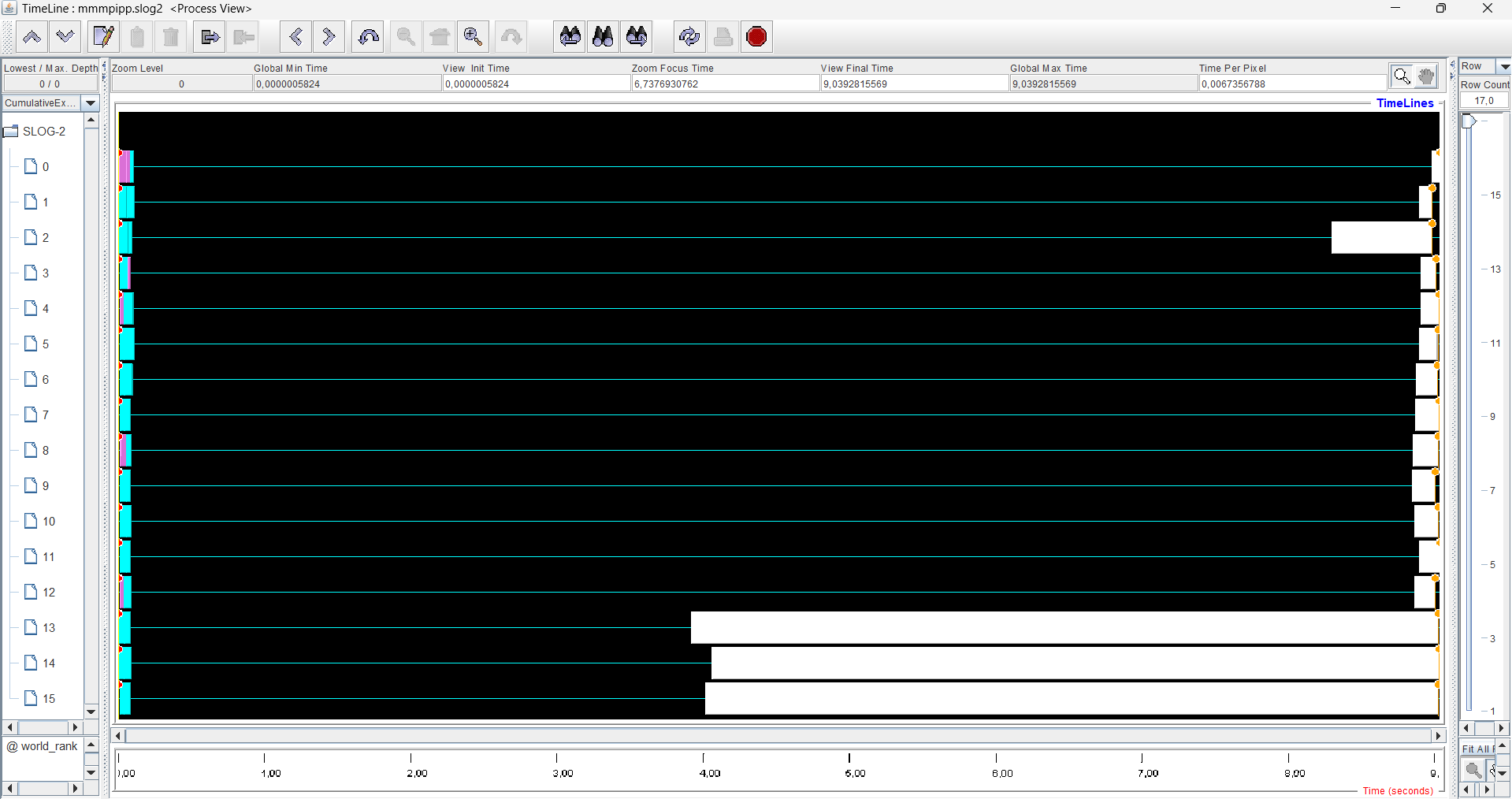
Параллельное выполнение алгоритма осуществляется на двумерной (2D) решетке компьютеров размером 𝑝1 × 𝑝2. Матрица А разрезана на 𝑝1 горизонтальных полос, матрица В разрезана на 𝑝2 вертикальных полос, и матрица результата C разрезана на 𝑝1 × 𝑝2 подматрицы.

Выполнение программы производилось на вычислительном кластере НОЦ «Газпромнефть-НГУ» с помощью скрипта системы пакетной обработки SLURM (Приложение 1).   
 Для измерения времени использовалась функция MPI\_Wtime().

Результаты измерений представлены на диаграммах:

**Профилирование:**

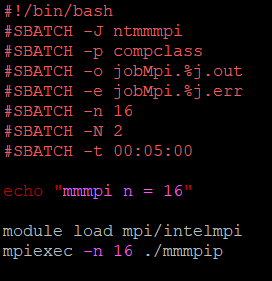




# ЗАКЛЮЧЕНИЕ

В ходе работы были построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых потоков. Также выполнено профилирование. Эффективность и ускорение распараллеливания программы сравнимы с такими же характеристиками ранее написанных программ с помощью MPI.

# ПРИЛОЖЕНИЕ 1



# ПРИЛОЖЕНИЕ 2

#include <iostream>

#include "mpi.h"

const double N1 = 2048;

const double N2 = 1024;

const double N3 = 4096;

enum {

NUMBER\_DIMS = 2,

X = 0,

Y = 1,

ROOT = 0

};

void printVector(double\* vector, int size, int rank)

{

printf("rank:%d\n", rank);

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

printf("%1.0f ", vector[i]);

}

printf("\n");

}

double\* generateMatrixA(int nLines, int nColumns) {

double\* matrix = new double[nLines \* nColumns];

int addPart = 0;

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

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

matrix[i \* nColumns + j] = j + 1 + i \* nColumns;

}

if (i != 0 && (i - 1) % 2 == 0) {

addPart += nColumns;

}

}

return matrix;

}

double\* generateMatrixB(int nLines, int nColumns) {

double\* matrix = new double[nLines \* nColumns];

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

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

matrix[i \* nColumns + j] = j + 1 + 24 + i \* nColumns;

}

}

return matrix;

}

double\* generateZeroVector(int vectorSize)

{

double\* vector = new double[vectorSize];

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

vector[i] = 0;

}

return vector;

}

void createComms(int coords[], int dims[], int rank, int nProcesses, MPI\_Comm\* comm2d, MPI\_Comm\* commColumn, MPI\_Comm\* commLine) {

MPI\_Dims\_create(nProcesses, NUMBER\_DIMS, dims);

int periods[NUMBER\_DIMS] = { 0, 0 };

int reorder = 1;

MPI\_Cart\_create(MPI\_COMM\_WORLD, NUMBER\_DIMS, dims, periods, reorder, comm2d);

MPI\_Cart\_coords(\*comm2d, rank, NUMBER\_DIMS, coords);

int coordsLine[NUMBER\_DIMS] = { 0, 1 };

int coordsColumn[NUMBER\_DIMS] = { 1, 0 };

MPI\_Cart\_sub(\*comm2d, coordsLine, commLine);

MPI\_Cart\_sub(\*comm2d, coordsColumn, commColumn);

}

void calculateSizes(int\* partASize, int\* partBSize, int n1, int n2, int n3, int dims[]) {

\*partASize = n1 \* n2 / dims[X];

\*partBSize = n2 \* n3 / dims[Y];

}

void distributeMatrixA(double\* A, double\* partA, int partASize, int coords[], MPI\_Comm commColumn, MPI\_Comm commLine) {

if (coords[Y] == 0) {

MPI\_Scatter(A, partASize, MPI\_DOUBLE, partA, partASize, MPI\_DOUBLE, ROOT, commColumn);

}

MPI\_Bcast(partA, partASize, MPI\_DOUBLE, ROOT, commLine);

}

void distributeMatrixB(double\* B, double\* partB, int partBSize, int coords[], int n2, int n3,int rank, MPI\_Comm commColumn, MPI\_Comm commLine) {

if (coords[X] == 0) {

MPI\_Datatype vectorType;

MPI\_Type\_vector(n2, partBSize / n2, n3, MPI\_DOUBLE, &vectorType);

MPI\_Type\_create\_resized(vectorType, 0, sizeof(double) \* partBSize / n2, &vectorType);

MPI\_Type\_commit(&vectorType);

MPI\_Scatter(B, 1, vectorType, partB, partBSize, MPI\_DOUBLE, ROOT, commLine);

}

MPI\_Bcast(partB, partBSize, MPI\_DOUBLE, ROOT, commColumn);

}

double\* multiplyPartMatrices(double\* partA, double\* partB, int partASize, int partBSize, int n1, int n2, int n3, int coords[]) {

double\* result = generateZeroVector(n1 \* n3);

for (int i = 0; i < n1; i++) {

for (int k = 0; k < n3; k++) {

for (int j = 0; j < n2; j++) {

result[i \* n3 + k] += partA[i \* n2 + j] \* partB[j \* n3 + k];

}

}

}

return result;

}

void CreateGathervOptions(int recvCounts[], int displs[], int dims[], int n1) {

int sizeShift = n1 \* dims[Y] / dims[X];

for (int i = 0; i < dims[X]; ++i) {

for (int j = 0; j < dims[Y]; ++j) {

recvCounts[i \* dims[Y] + j] = 1;

displs[i \* dims[Y] + j] = i \* sizeShift + j;

}

}

}

void unitResultParts(double\* result, double\* partResult, int partResultSize, int nLinesPart, int nColumnsPart, int n1, int n3,

int dims[], int nProcesses, MPI\_Comm\* comm2d) {

MPI\_Datatype matrixPart;

MPI\_Type\_vector(nLinesPart, nColumnsPart, n3, MPI\_DOUBLE, &matrixPart);

MPI\_Type\_commit(&matrixPart);

MPI\_Type\_create\_resized(matrixPart, 0, sizeof(double) \* nColumnsPart, &matrixPart);

MPI\_Type\_commit(&matrixPart);

int\* receiveCounts = new int[nProcesses];

int\* displays = new int[nProcesses];

CreateGathervOptions(receiveCounts, displays, dims, n1);

MPI\_Gatherv(partResult, partResultSize, MPI\_DOUBLE, result, receiveCounts, displays, matrixPart, ROOT, \*comm2d);

delete[] receiveCounts;

delete[] displays;

}

double\* multiplyMatrices(double\* A, double\* B, int n1, int n2, int n3, int rank, int nProcesses) {

int coords[NUMBER\_DIMS] = { 0, 0 };

int dims[NUMBER\_DIMS] = { 0, 0 };

MPI\_Comm comm2d, commColumn, commLine;

createComms(coords, dims, rank, nProcesses, &comm2d, &commColumn, &commLine);

int partASize, partBsize;

calculateSizes(&partASize, &partBsize, n1, n2, n3, dims);

double\* partA = generateZeroVector(partASize);

double\* partB = generateZeroVector(partBsize);

distributeMatrixA(A, partA, partASize, coords, commColumn, commLine);

distributeMatrixB(B, partB, partBsize, coords, n2, n3, rank, commColumn, commLine);

double\* partResult = multiplyPartMatrices(partA, partB, partASize, partBsize, partASize / n2, n2, partBsize / n2, coords);

int partResultSize = (partASize / n2) \* (partBsize / n2);

double\* result = generateZeroVector(n1 \* n3);

unitResultParts(result, partResult, partResultSize, partASize / n2, partBsize / n2, n1, n3, dims, nProcesses, &comm2d);

delete[] partA;

delete[] partB;

delete[] partResult;

return result;

}

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

{

double\* A = generateMatrixA(N1, N2);

double\* B = generateMatrixB(N2, N3);

MPI\_Init(&argc, &argv);

double start = MPI\_Wtime();

int rank, nProcesses;

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

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

double\* solution = multiplyMatrices(A, B, N1, N2, N3, rank, nProcesses);

double finish = MPI\_Wtime();

if (rank == 0) {

printf("Time: %f\n", finish - start);

//printVector(solution, N1 \* N3, rank);

}

delete[] A;

delete[] B;

delete[] solution;

MPI\_Finalize();

return 0;

}