###### 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 решетке»

студента 2 курса, 20204 группы

**Дронова Дениса Юрьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc101966097)

[ЗАДАНИЕ 3](#_Toc101966098)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc101966099)

[ЗАКЛЮЧЕНИЕ 5](#_Toc101966100)

[Приложение 1. Таблицы и графики 6](#_Toc101966101)

[Приложение 2. Профилирование 8](#_Toc101966102)

[Приложение 3. Код программы 9](#_Toc101966103)

# ЦЕЛЬ

Освоить концепции MPI-коммуникаторов и декартовых топологий, а также концепции производных типов данных.

# ЗАДАНИЕ

1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке процессов с соблюдением требований:
2. Использование декартовой топологии процессов и создание новых коммуникаторов.
3. Использование производных типов данных.
4. Программа должна работать для матрицы B любого содержания - предполагать, что матрица B будет симметричной нельзя.
5. Транспонировать матрицу B нельзя.
6. При раздаче матрицы B по полосам каждый из процессов с координатами (0; x) должен получить **по одному элементу** производного типа, а процесс (0;0), соответственно, высылает процессам по одному элементу производного типа (не обязательно того же самого).
7. Исследовать производительность параллельной программы при фиксированном размере матрицы в зависимости от и размера решетки: 2x12, 3x8, 4x6, 6x4, 8x3, 12x2. Размер подобрать таким образом, чтобы худшее из времен данного набора было не менее 30 сек.
8. Выполнить профилирование программы при использовании 8-и ядер с решетками 2x4, 4x2.

# ОПИСАНИЕ РАБОТЫ

1. Был реализован параллельный алгоритм умножения матрицы на матрицу с использованием 2D решетки процессов. Корректность алгоритма проверена путем тестирования и анализа полученных ответов.
2. Программа была запущена на кластере одном числе процессов с различными формами 2D решеток и впоследствии составлена таблица зависимости времени исполнения от формы. Также были сформированы таблицы и графики ускорения и эффективности в зависимости от числа процессов (см. Приложение 1).
3. Выполнено профилирование программы на 8 процессах с решеткой 4x2 (см. Приложение 2).
4. На основе анализа полученных таблиц и графиков был сформулирован вывод.

# **ЗАКЛЮЧЕНИЕ**

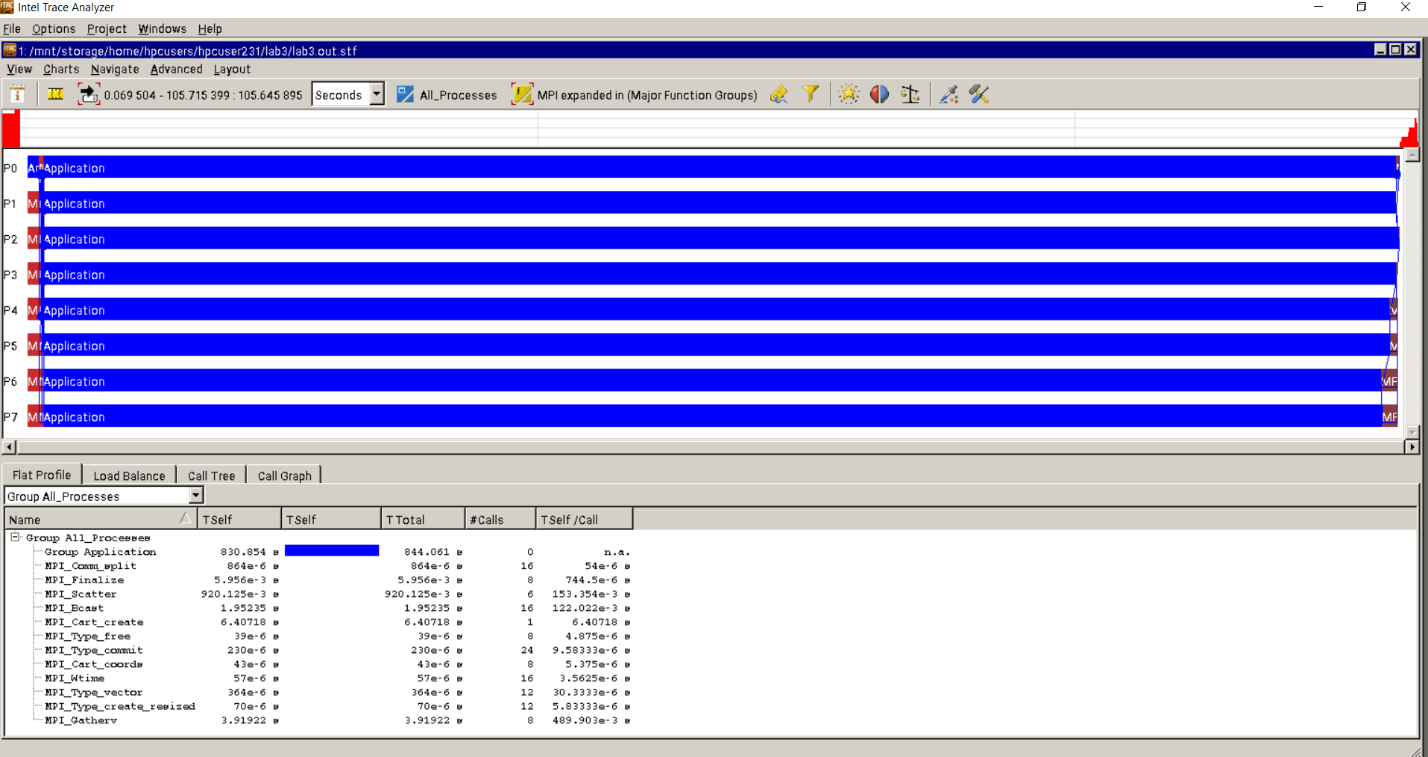
Анализ первой таблицы позволяет сделать вывод, что решетки 1на 2 и 2 на 4 обладают наибольшей эффективностью. Также можно прийти к заключению о том, что увеличение числа процессов приводит к снижению эффективности из-за большего взаимодействия(общения) процессов. После рассмотрения второй таблицы становится понятно, что наилучшим видом 2D решетки будет более квадратная, т. к. в таком случае одновременно обе матрицы будут разрезаться на меньшие куски, что благоприятно скажется на времени подсчета итоговых частей. После анализа профилирования можно заключить, что на распределение данных между процессами и сбор тратится незначительное количество времени по сравнению с временем, затрачиваемым на вычисления.

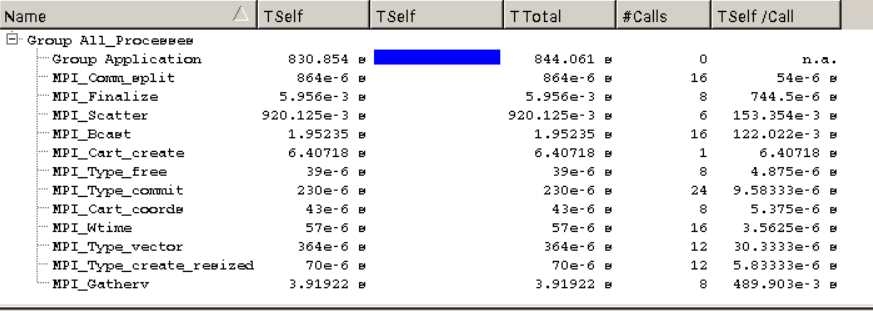
# Приложение 1. Таблицы и графики

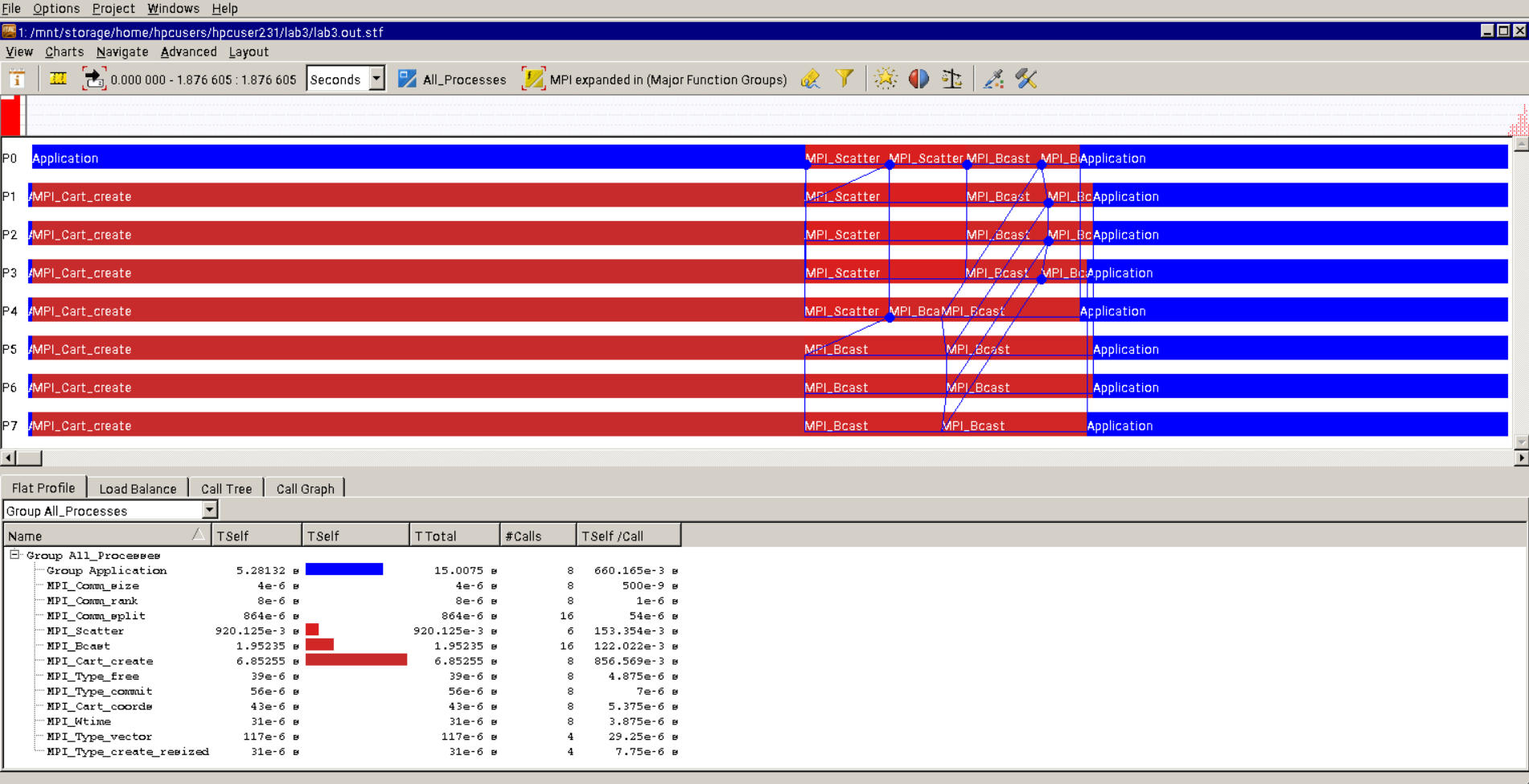
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Число процессов | P1 | P2 | N1 | N2 | N3 | Время исполнения | Эффективность | Ускорение |
| 1 | 1 | 1 | 2200 | | | 80,168 | 100,00% | 1 |
| 2 | 1 | 2 | 2200 | | | 40,1035 | 99,95% | 1,999027516 |
| 4 | 2 | 2 | 2200 | | | 20,1163 | 99,63% | 3,985225911 |
| 8 | 2 | 4 | 2200 | | | 10,183 | 98,41% | 7,872729058 |
| 12 | 4 | 3 | 2200 | | | 6,93116 | 96,39% | 11,5663179 |
| 16 | 4 | 4 | 2200 | | | 5,25529 | 95,34% | 15,25472429 |
| 24 | 4 | 6 | 2200 | | | 3,58799 | 93,10% | 22,34342905 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Число процессов | P1 | P2 | N1 | N2 | N3 | Время исполнения |
| 24 | 2 | 12 | 4800 | | | 35,4633 |
| 24 | 3 | 8 | 4800 | | | 35,4597 |
| 24 | 4 | 6 | 4800 | | | 35,4199 |
| 24 | 6 | 4 | 4800 | | | 35,3691 |
| 24 | 8 | 3 | 4800 | | | 35,3868 |
| 24 | 12 | 2 | 4800 | | | 35,4725 |

# Приложение 2. Профилирование







# Приложение 3. Код программы

#include "mpi.h"

#include <iostream>

#include <stdlib.h>

#define HEIGHT1 4800

#define WIDTH\_HEIGHT 4800

#define WIDTH2 4800

#define P1 2

#define P2 4

using namespace std;

void showMatrix(double \*matrix, size\_t height, size\_t width)

{

    size\_t i, j;

    for (i = 0; i < height; ++i)

    {

        for (j = 0; j < width; ++j)

        {

            cout << matrix[i \* width + j] << " ";

        }

        cout << endl;

    }

    cout << "\n"

         << "\n"

         << "\n";

}

void multiplicateMatrix(double \*M1, double \*M2, double \*result, size\_t n1, size\_t n2, size\_t n3)

{

    size\_t i, k, j;

    for (i = 0; i < n1; i++)

    {

        for (k = 0; k < n2; k++)

        {

            for (j = 0; j < n3; j++)

            {

                result[i \* n3 + j] += M1[i \* n2 + k] \* M2[k \* n3 + j];

            }

        }

    }

}

double \*generateMatrix(size\_t height, size\_t width)

{

    double \*tmp = (double \*)calloc(height \* width, sizeof(double));

    for (size\_t i = 0; i < height \* width; i++)

    {

        tmp[i] = rand() % 100;

    }

    return tmp;

}

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

{

    int procRank, new\_rank, procSize;

    MPI\_Comm vert\_comm, hor\_comm, comm2d;

    MPI\_Init(&argc, &argv);

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

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

    double \*A\_full = NULL;

    double \*B\_full = NULL;

    double \*result = NULL;

    if (procRank == 0)

    {

        A\_full = generateMatrix(HEIGHT1, WIDTH\_HEIGHT);

        B\_full = generateMatrix(WIDTH\_HEIGHT, WIDTH2);

        result = (double \*)calloc(HEIGHT1 \* WIDTH2, sizeof(double));

    }

    int dims[2];

    dims[0] = P1; // width of communicator grid

    dims[1] = P2; // height of communicator grid

    int periods[2] = {0, 0};

    MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, 1, &comm2d);

    int coords[2];

    MPI\_Cart\_coords(comm2d, procRank, 2, coords);

    // coords[0] = y

    // coords[1] = x

    // MPI\_Comm\_split makes new communicator for each split\_key(second parameter -- coords[...])

    MPI\_Comm\_split(comm2d, coords[1], coords[0], &vert\_comm); // vertical communicators

    MPI\_Comm\_split(comm2d, coords[0], coords[1], &hor\_comm);  // horizontal communicators

    int rows\_in\_proc = HEIGHT1 / P1;

    int columns\_in\_proc = WIDTH2 / P2;

    // filling the recvcounts and displs arrays

    int \*recvcounts = (int \*)calloc(procSize, sizeof(int));

    int \*displs = (int \*)calloc(procSize, sizeof(int));

    int temp = 0;

    int shift = 0;

    for (int i = 1; i < procSize; ++i)

    {

        if (i % dims[1] != 0)

        {

            temp = shift + (i % dims[1]) \* columns\_in\_proc;

            displs[i] = temp;

        }

        else

        {

            temp = WIDTH2 \* rows\_in\_proc \* (i / dims[1]);

            shift = temp;

            displs[i] = temp;

        }

        recvcounts[i] = 1;

    }

    displs[0] = 0;

    recvcounts[0] = 1;

    double \*A\_part = (double \*)calloc(WIDTH\_HEIGHT \* rows\_in\_proc, sizeof(double));

    double \*B\_part = (double \*)calloc(WIDTH\_HEIGHT \* columns\_in\_proc, sizeof(double));

    double \*res\_part = (double \*)calloc(rows\_in\_proc \* columns\_in\_proc, sizeof(double));

    double start = MPI\_Wtime();

    if (coords[1] == 0) // zero process by x coordinate

    {

        // matrix A\_full scattering from zero (root) process by vertical communicator

        MPI\_Scatter(A\_full, rows\_in\_proc \* WIDTH\_HEIGHT, MPI\_DOUBLE, A\_part, rows\_in\_proc \* WIDTH\_HEIGHT, MPI\_DOUBLE, 0, vert\_comm);

    }

    if (coords[0] == 0) // zero process by y coordinate

    {

        // matrix B\_full scattering from zero (root) process by horizontal communicator

        // with usage of custom datatype

        MPI\_Datatype column\_type; // new type for storing custom columns

        MPI\_Datatype column\_resized\_type;

        // WIDTH\_HEIGHT - is a B\_full height - number of blocks in vector

        // columns\_in\_proc - number of elements in block

        // WIDTH2 - width of B\_full matrix - elements between blocks beginnings

        MPI\_Type\_vector(WIDTH\_HEIGHT, columns\_in\_proc, WIDTH2, MPI\_DOUBLE, &column\_type);

        MPI\_Type\_commit(&column\_type); // for fixing new type (creating)

        // resized type for correct shift while scattering

        MPI\_Type\_create\_resized(column\_type, 0, columns\_in\_proc \* sizeof(double), &column\_resized\_type);

        MPI\_Type\_commit(&column\_resized\_type);

        // giving only one column\_type(resized) to each process

        MPI\_Scatter(B\_full, 1, column\_resized\_type, B\_part, WIDTH\_HEIGHT \* columns\_in\_proc, MPI\_DOUBLE, 0, hor\_comm);

        MPI\_Type\_free(&column\_resized\_type);

        MPI\_Type\_free(&column\_type);

    }

    // Broadcasting...

    MPI\_Bcast(A\_part, rows\_in\_proc \* WIDTH\_HEIGHT, MPI\_DOUBLE, 0, hor\_comm);

    MPI\_Bcast(B\_part, columns\_in\_proc \* WIDTH\_HEIGHT, MPI\_DOUBLE, 0, vert\_comm);

    multiplicateMatrix(A\_part, B\_part, res\_part, rows\_in\_proc, WIDTH\_HEIGHT, columns\_in\_proc);

    // Traversing the comm2d grid and sending each C\_part to definite place in zero process

    MPI\_Datatype res\_block;

    MPI\_Datatype res\_block\_resized;

    // rows\_in\_proc - number of blocks in vector

    // columns\_in\_proc - number of elements in block

    // WIDTH2 - elements between blocks beginnings

    MPI\_Type\_vector(rows\_in\_proc, columns\_in\_proc, WIDTH2, MPI\_DOUBLE, &comm2d);

    MPI\_Type\_Commit(&res\_block);

    MPI\_Type\_create\_resized(res\_block, 0, sizeof(double), &res\_block\_resized);

    MPI\_Type\_Commit(&res\_block\_resized);

    MPI\_Gatherv(res\_part, rows\_in\_proc \* columns\_in\_proc, MPI\_DOUBLE, result, recvcounts, displs, res\_block\_resized, 0, comm2d);

    double finish = MPI\_Wtime();

    double total\_time;

    if (procRank == 0)

    {

        // showMatrix(result, HEIGHT1, WIDTH2);

        total\_time = finish - start;

        cout << "Total time: " << total\_time << " sec." << endl;

    }

    free(A\_full);

    free(B\_full);

    free(result);

    free(A\_part);

    free(B\_part);

    free(res\_part);

    MPI\_Finalize();

    return 0;

}