**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»

студентки 2 курса, 19202 группы

**Хаецкой Дарьи Владимировны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.Ю. Власенко

Новосибирск 2021

# ЗАДАНИЕ

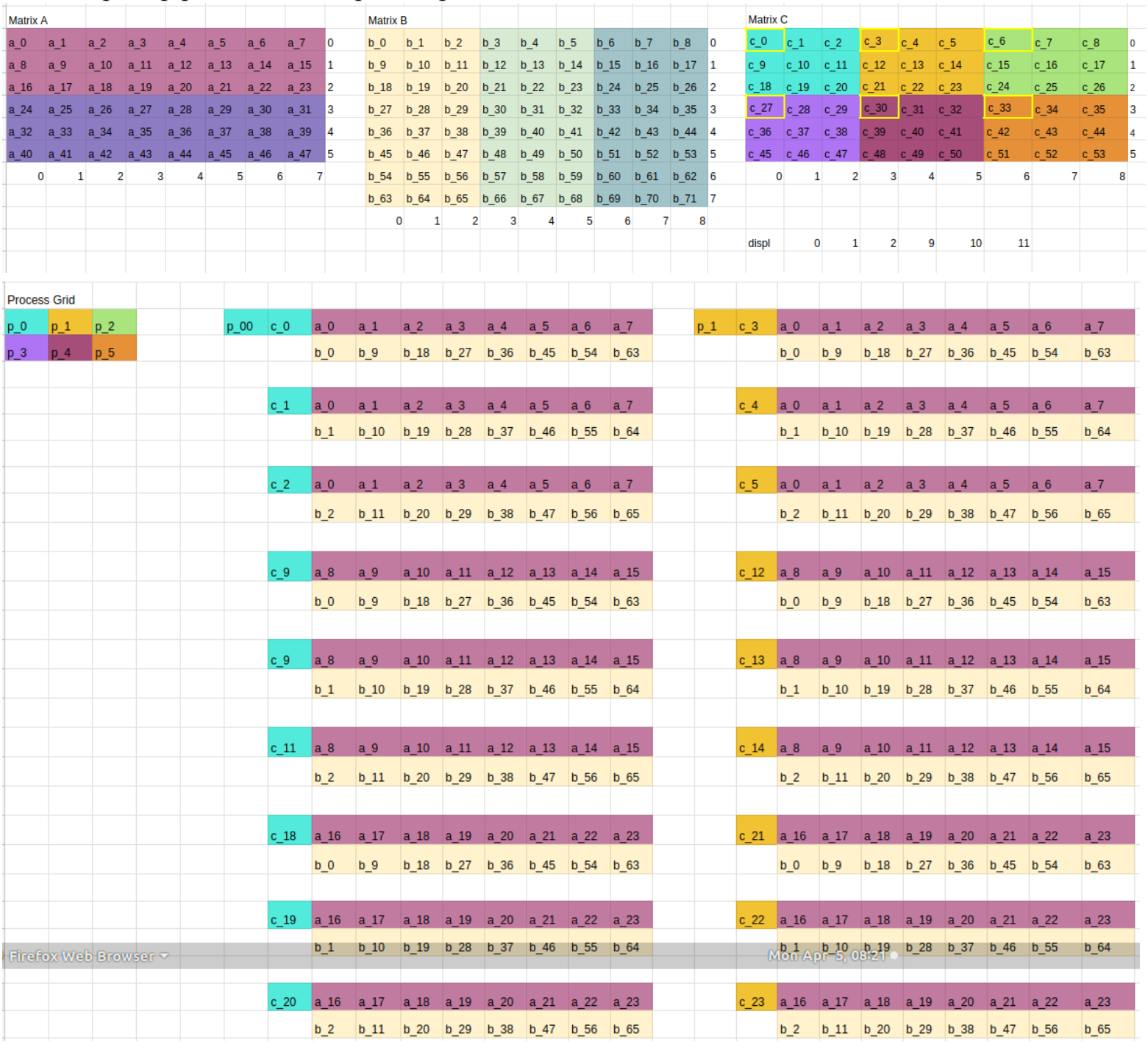
1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке.
2. Исследовать производительность параллельной программы в зависимости от размера матрицы и размера решетки.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер.

# ОПИСАНИЕ РАБОТЫ

Для реализации был выбран следующий алгоритм:

1. Матрица A разбивается на p1 строк.
2. Матрица B разбивается на p2 столбцов
3. Строки и столбцы отправляются первым процессам в решетке и затем распространяются вдоль соответствующей координате сетки процессов.
4. Производится умножение полученных частей матрицы, затем эти части собираются в результирующую матрицу C.

Пример разбиения для p1 = 2, p2 = 3, n1 = 6, n2 = 8, n3 = 9



## Рассмотрим подробнее каждый из этапов алгоритма.

1. Для разбиения матрицы A была использована функция MPI_Scatter, получившиеся части матрицы были отправлены на процессы, лежащие на оси OY. Затем, с помощью функции MPI_Bcast на строчном коммуникаторе RowComm части матрицы были распространены вдоль оси OX по всей сетке процессов.
2. Для разбиения матрицы B был создан производный тип col содержащий столбец матрицы, это было сделано силами функции `MPI_Type_vector`. Затем, в силу особенности функции MPI_Scatter, было необходимо произвести манипуляции с размером типа, это было сделано функцией `MPI_Type_create_resized`. Далее, была использована функция `MPI_Scatter`, но уже для типа “колонка” а не число.Полученные блоки распространены по процессам вдоль оси OY.
3. Для сбора частей матрицы со всех процессов в одну результирующую матрицу, вновь был создан производный тип “блок”. Его размер таким же образом был изменен и далее использована функция `MPI_Gatherv` с предварительным вычислением массива отступов `displ`.

## Анализ времени работы в зависимости от топологии.

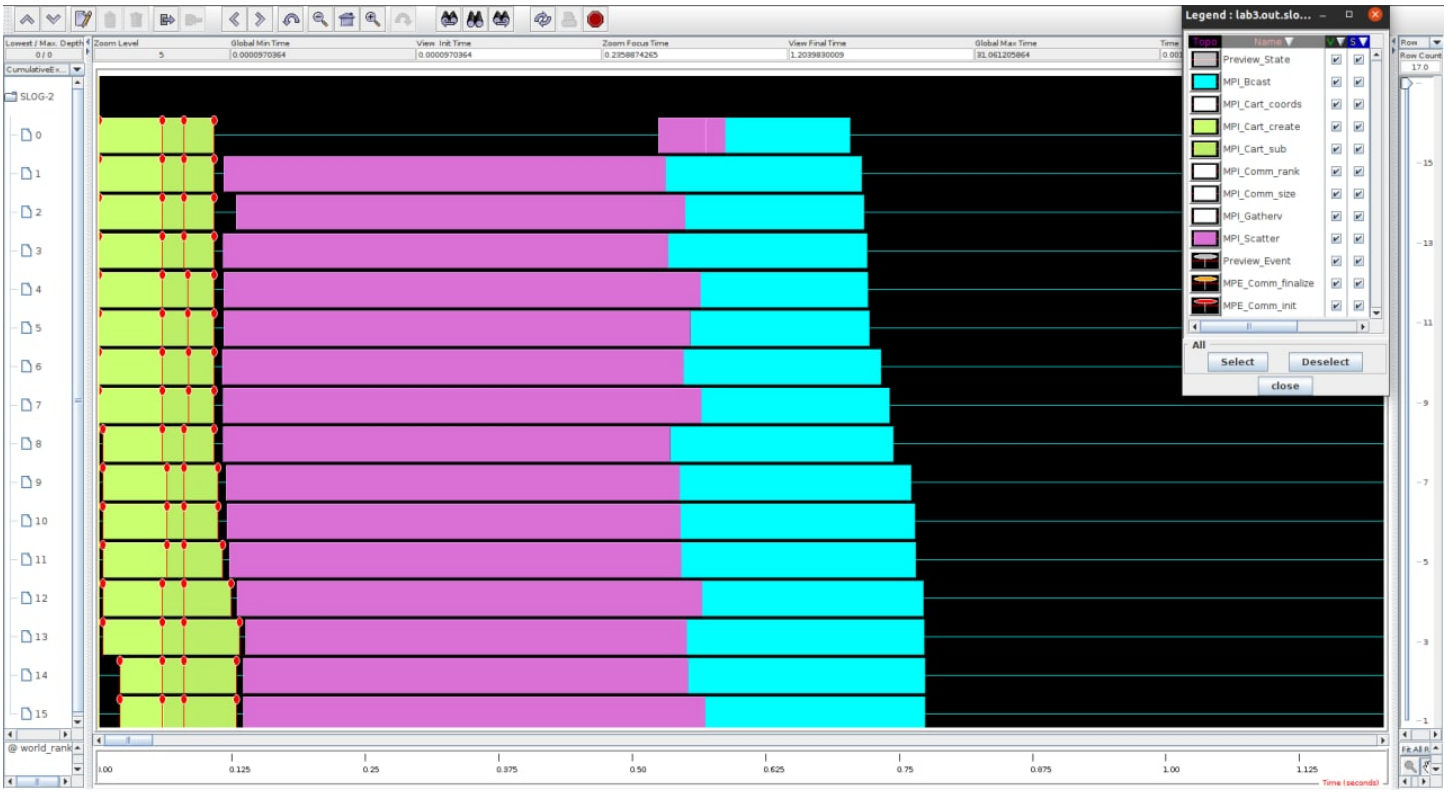
В ходе измерения времени работы программы были получены следующие данные:

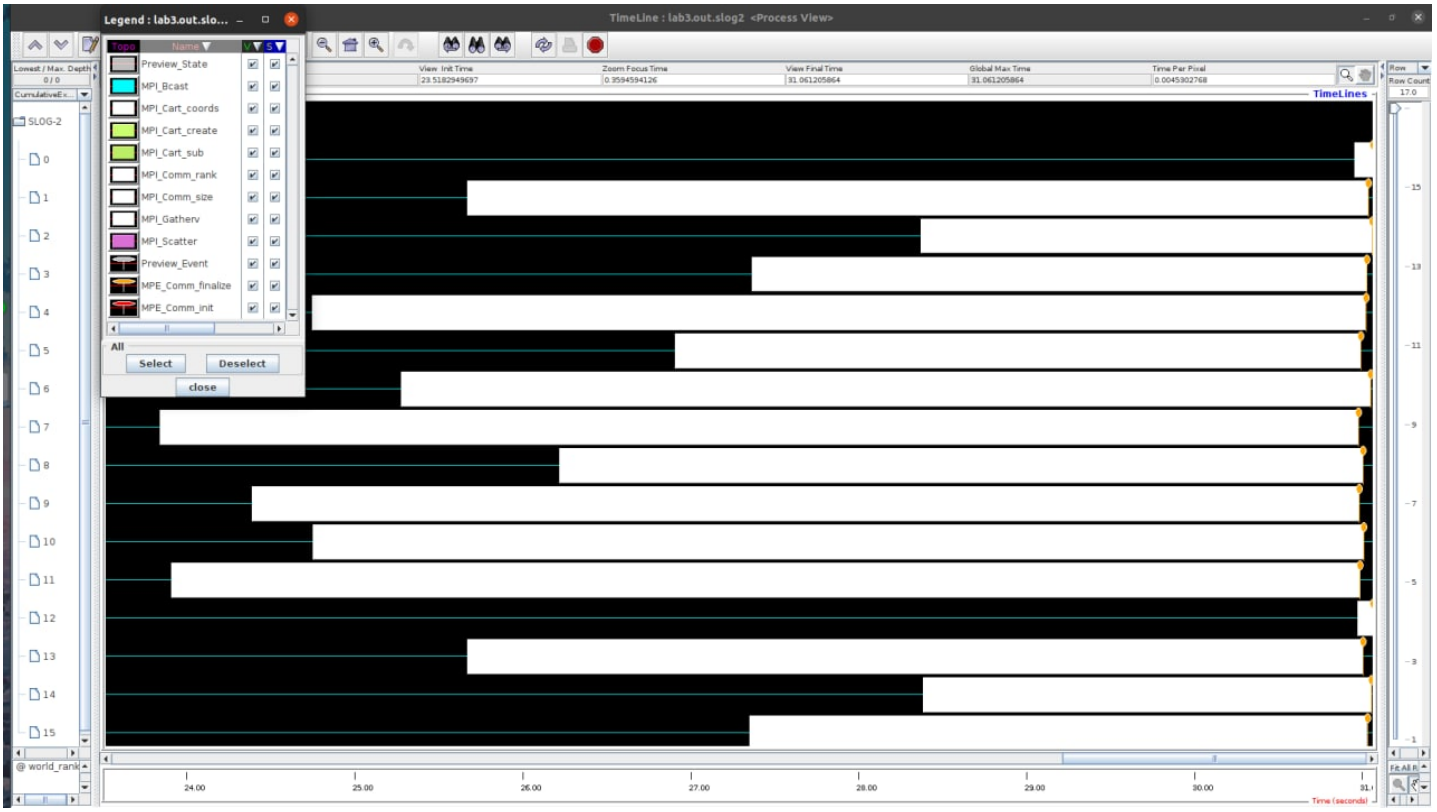
| n1 | n2 | n3 |
|---|---|---|
| 4200 | 1000 | 3600 |

| p1\p2 | 24 | 12 | 8 | 6 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| 1 | 22.4737 | | | | | | | |
| 2 | | 21.8371 | | | | | | |
| 3 | | | 33.4734 | | | | | |
| 4 | | | | 21.4350 | | | | |
| 6 | | | | | 25.410563 | | | |
| 8 | | | | | | 22.262596 | | |
| 12 | | | | | | | 23.019814 | |
| 24 | | | | | | | | 21.792634 |

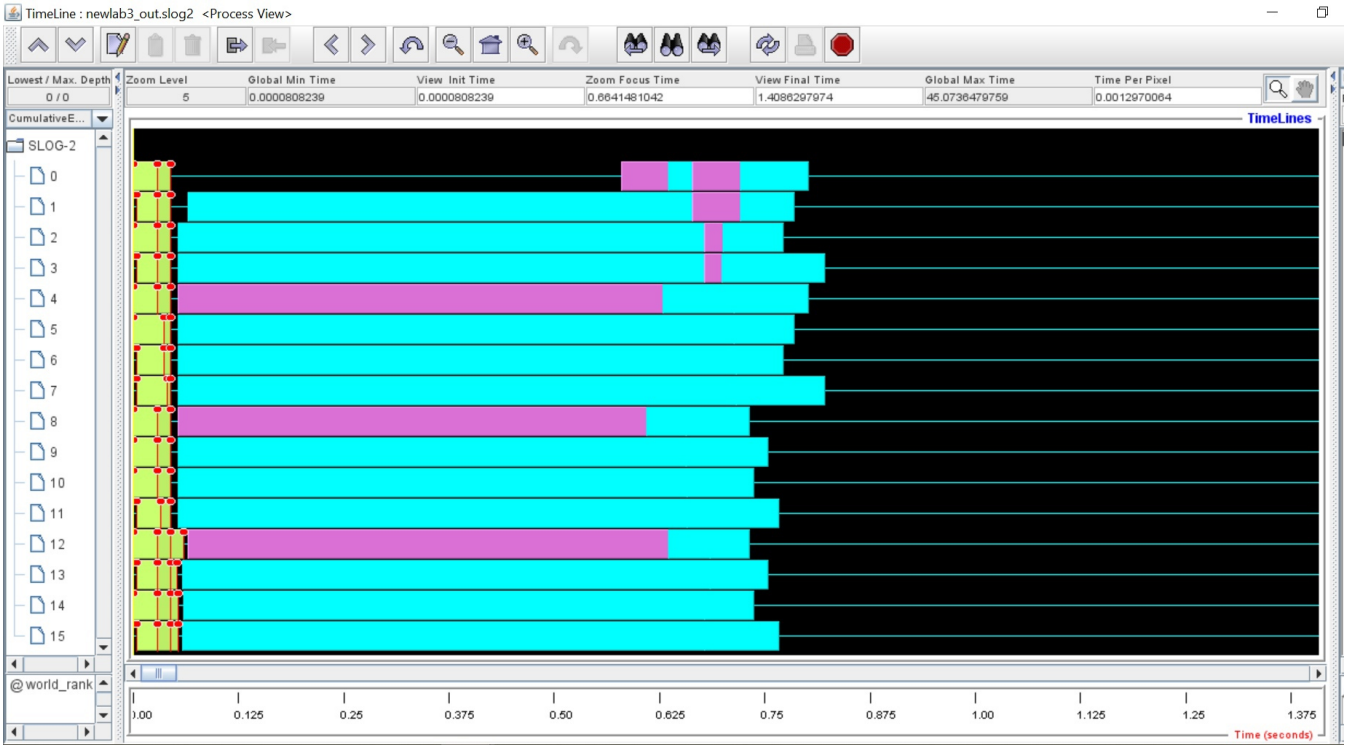
# Результат профилирования

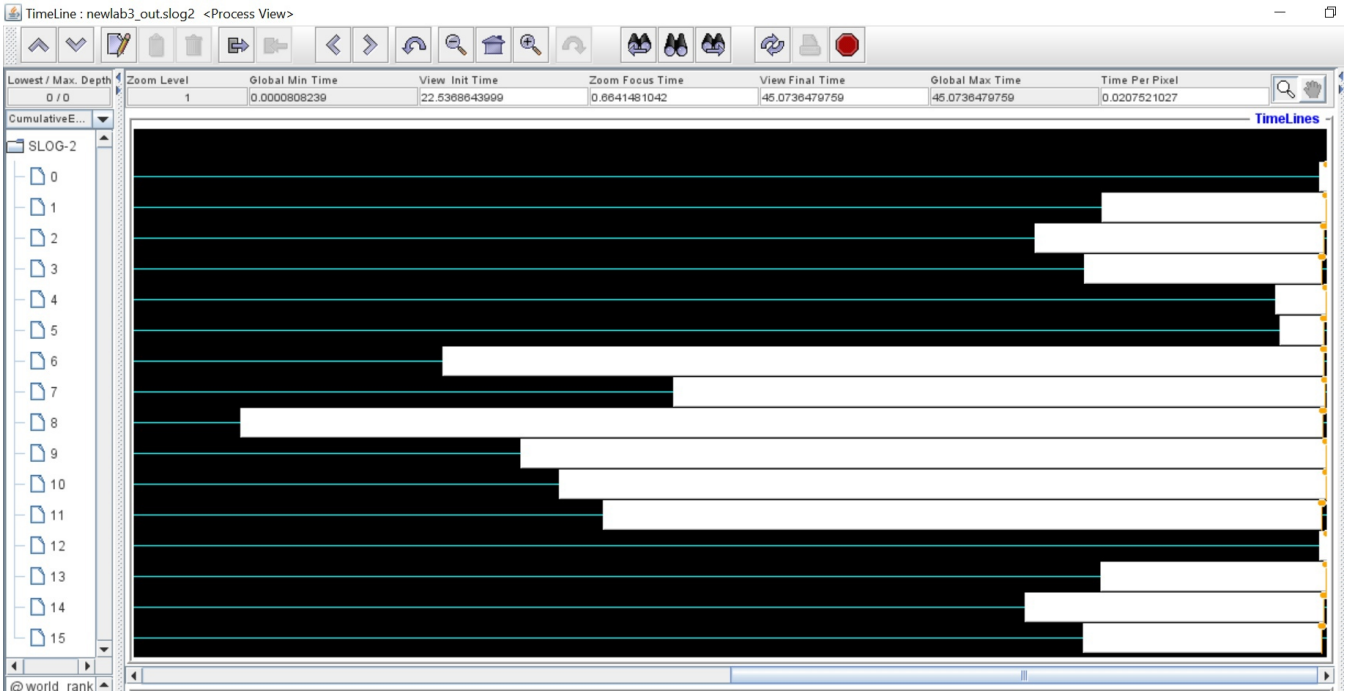
## Для топологии 16:1



# Для топологии 4:4



# **Приложение 1.** *Листинг программы*

```
#include <iostream>
#include <mpi.h>
#include <cmath>
#include <zconf.h>
#include "MatrixOperations.h"

MPI_Comm GridComm;    // Grid communicator
MPI_Comm ColComm;     // Column communicator
MPI_Comm RowComm;     // Row communicator
int GridCoords[2];
int ProcNum = 0;
int ProcRank = 0;
int p1 = 6;
int p2 = 3;
int n1 = 24;
int n2 = 8;
int n3 = 9;

void CreateGridCommunicators(){
   int DimSize[2]; // Number of processes in each dimension of the grid
   int Periodic[2]; // =1, if the grid dimension should be periodic
   int SubDimension[2];  // =1, if the grid dimension should be fixed

   DimSize[0] = p1;
   DimSize[1] = p2;

   Periodic[0] = 1;
   Periodic[1] = 1;
   // Creation of the Cartesian communicator
   MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 0, &GridComm);
   // Determination of the cartesian coordinates for every process
   MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

   // Creating communicators for rows
   SubDimension[0] = 0; // Dimension is fixed
   SubDimension[1] = 1; // Dimension belong to the subgrid
   MPI_Cart_sub(GridComm, SubDimension, &RowComm);

   // Creating communicators for columns
   SubDimension[0] = 1; // Dimension belong to the subgrid
   SubDimension[1] = 0; // Dimension is fixed
   MPI_Cart_sub(GridComm, SubDimension, &ColComm);
}

void InitializeProcess(double* &pAMatrix, double* &pBMatrix, double* &pCMatrix,
                              double* &pAblock, double* &pBblock, double* &pCblock, int
&ABlockSize, int &BBlockSize ) {

   ABlockSize = n1/p1;
   BBlockSize = n3/p2;

   pAblock = new double [n2*ABlockSize];
   pBblock = new double [n2*BBlockSize];
   pCblock = new double [ABlockSize*BBlockSize];

   if (ProcRank == 0){
       pAMatrix = new double [n1*n2];
```

```
        pBMatrix = new double [n2*n3];
        pCMatrix = new double [n1*n3];
        DataInitialization(pAMatrix, n1, n2);
        DataInitialization(pBMatrix, n2, n3);
        SetToZero(pCMatrix, n1, n3);
    }

    SetToZero(pCblock, ABlockSize, BBlockSize);
}

void TerminateProcess (double* AMatrix, double* BMatrix,
                       double* CMatrix, double* Ablock, double* Bblock, double* Cblock){

    if (ProcRank == 0){
        delete [] AMatrix;
        delete [] BMatrix;
        delete [] CMatrix;
    }
    delete [] Ablock;
    delete [] Bblock;
    delete [] Cblock;
}

void DataDistribution(double* AMatrix, double* BMatrix, double* Ablock,
                      double* Bblock, int ABlockSize, int BBlockSize) {

    if (GridCoords[1] == 0) {
        MPI_Scatter(AMatrix, ABlockSize * n2, MPI_DOUBLE, Ablock,
                ABlockSize * n2, MPI_DOUBLE, 0, ColComm);
    }

    MPI_Bcast(Ablock, ABlockSize * n2, MPI_DOUBLE, 0, RowComm);
    MPI_Datatype col, coltype;

    MPI_Type_vector(n2, BBlockSize, n3, MPI_DOUBLE, &col);
    MPI_Type_commit(&col);
    MPI_Type_create_resized(col, 0, BBlockSize * sizeof(double), &coltype);
    MPI_Type_commit(&coltype);

    if (GridCoords[0] == 0) {
        MPI_Scatter(BMatrix, 1, coltype, Bblock, n2 * BBlockSize,
                    MPI_DOUBLE, 0, RowComm);
    }

    MPI_Bcast(Bblock, BBlockSize * n2, MPI_DOUBLE, 0, ColComm);
}

int main(int argc, char* argv[]) {
    double* AMatrix = NULL;      // First argument of matrix multiplication
    double* BMatrix = NULL;      // Second argument of matrix multiplication
    double* CMatrix = NULL;      // Result matrix

    int ABlockSize = 0;
    int BBlockSize = 0;

    double *Ablock = NULL;       // Current block of matrix A
    double *Bblock = NULL;       // Current block of matrix B
    double *Cblock = NULL;       // Block of result matrix C

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
        if ((n1 % p1 != 0) || (n3 % p2 != 0)) {
            if (ProcRank == 0) {
                printf ("invalid grid size\n");
            }
        } else {
            if (ProcRank == 0)
                       printf("Parallel matrix multiplication program, on %d processes\n",
ProcNum);
            CreateGridCommunicators(); // Grid communicator creating
        }

        // Memory allocation and initialization of matrix elements
        InitializeProcess(AMatrix, BMatrix, CMatrix, Ablock, Bblock,
                          Cblock, ABlockSize, BBlockSize);

        double startTime;

        if (ProcRank == 0) {
            startTime = MPI_Wtime();
            printf("Initial matrix A \n");
            PrintMatrix(AMatrix, n1, n2);
            printf("Initial matrix B \n");
            PrintMatrix(BMatrix, n2, n3);
        }
        // distribute data among the processes
        DataDistribution(AMatrix, BMatrix, Ablock, Bblock, ABlockSize, BBlockSize);
        // Multiply matrix blocks of the current process
        MatrixMul(Ablock, Bblock, Cblock, ABlockSize, n2, BBlockSize);

        // Gather all data in one matrix

        // the first step is creating a block type and resizing it
        MPI_Datatype block, blocktype;
        MPI_Type_vector(ABlockSize, BBlockSize, n3, MPI_DOUBLE, &block);
        MPI_Type_commit(&block);

        MPI_Type_create_resized(block, 0, BBlockSize*sizeof(double), &blocktype);
        MPI_Type_commit(&blocktype);

        // calculate displ
        int* displ =  new int[p1*p2];
        int* rcount =  new int[p1*p2];
        int BlockCount = 0;
        int BlockSize = ABlockSize*BBlockSize;
        int NumCount = 0;
        int Written;
        int j = 0;

        while (NumCount < p1*p2*BlockSize) {
            Written = 0;
            for (int i = 0; i < n3; i += BBlockSize) {
                displ[j] = BlockCount;
                rcount[j] = 1;
                j++;
                BlockCount++;

                Written++;
            }
            NumCount += Written * BlockSize;
            BlockCount += Written * (ABlockSize - 1);
```

```
    }

    MPI_Gatherv(Cblock, BlockSize, MPI_DOUBLE, CMatrix,
                rcount, displ, blocktype, 0, MPI_COMM_WORLD);

    if (ProcRank == 0){
        double endTime = MPI_Wtime();
        printf("matrix C \n");

        PrintMatrix(CMatrix, n1, n3);
        printf("That took %lf seconds\n",endTime-startTime);
    }

    TerminateProcess(AMatrix, BMatrix, CMatrix, Ablock, Bblock, Cblock);
    delete [] displ;
    delete [] rcount;
    MPI_Finalize();
    return 0;
}
```

## **Приложение 2.** *Листинг файла MatrixOperations.cpp*

```
#include "MatrixOperations.h"
#include <mpi.h>
#include <cstdlib>

double rand_double(){
   return (double)rand()/RAND_MAX*50.0 - 2.0;
}

void DataInitialization(double* pMatrix, int rowCount, int colCount) {
   for (int i = 0; i < rowCount; i++){
      for (int j = 0; j < colCount; j++){
          pMatrix[i*colCount + j] = rand_double();
      }
  }
}

void SetToZero(double* pMatrix, int rowCount, int colCount) {
   for (int i = 0; i < rowCount; i++){
       for (int j = 0; j < colCount; j++){
           pMatrix[i*colCount + j] = 0;
       }
   }
}

// Function for formatted vector output
void PrintVector(double* pVector, int Size, int ProcNum) {
   printf("proc #%d ", ProcNum);
//    MPI_Barrier(MPI_COMM_WORLD);
   for (int i = 0; i < Size; i++)
       printf("%7.4f ", pVector[i]);

//    MPI_Barrier(MPI_COMM_WORLD);
   printf("\n");
}

// Function for formatted vector output
void PrintVector(int* pVector, int Size, int ProcNum) {
   printf("proc # %d ", ProcNum);
//    MPI_Barrier(MPI_COMM_WORLD);
   for (int i = 0; i < Size; i++)
       printf("%d ", pVector[i]);

//    MPI_Barrier(MPI_COMM_WORLD);
   printf("\n");
}

// Function for formatted matrix output
void PrintMatrix(double* pMatrix, int RowCount, int ColCount) {
   int i, j; // Loop variables
   for (i = 0; i < RowCount; i++) {
       for (j = 0; j < ColCount; j++)
           printf("%7.4f ", pMatrix[i * ColCount + j]);
       printf("\n");
   }
}

// Function for matrix multiplication
```

```
     void MatrixMul(double* pAMatrix, double* pBMatrix, double* pCMatrix, int n1, int n2,
int n3) {
        int i, j, k;  // Loop variables
        for (i = 0; i < n1; i++) {
            for (j = 0; j < n3; j++)
                for (k = 0; k < n2; k++)
                    pCMatrix[i*n3 + j] += pAMatrix[i*n2 + k] * pBMatrix[k*n3 + j];
        }
     }
```

# Вывод

В ходе выполнения данной лабораторной работы мной были освоены концепции коммуникаторов и декартовых топологий. Приобретены навыки работы с производными типами и методы работы с ними при помощи MPI функций.