МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №6

по дисциплине «Параллельные алгоритмы»

Тема: Умножение матриц

Студент гр. 8303 _____ Почаев Н.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

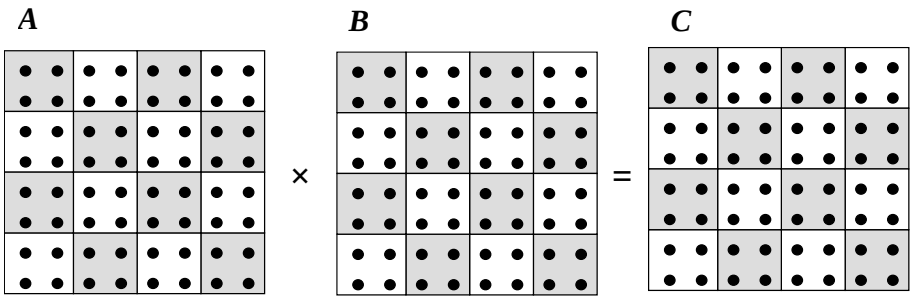
2020

Задание.

Выполнить задачу умножения двух квадратных матриц A и B размера $m \times m$, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания - **№3 Блочный алгоритм Кэннона**. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Теоретические положения.

Алгоритм Кэннона — это распределенный алгоритм умножения матриц для двумерных сеток. Данный алгоритм является блочным, т.е. для его решения используется представление матрицы, при котором она рассекается вертикальными и горизонтальными линиями на прямоугольные части — блоки (клетки).



Базовой подзадачей является процедура вычисления всех элементов одного из блоков матрицы C.

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix}$$

$$= \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \vdots & \vdots & \vdots & \vdots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}$$

$$C_{ij} = \sum_{s=1}^{q} A_{is} B_{sj}$$

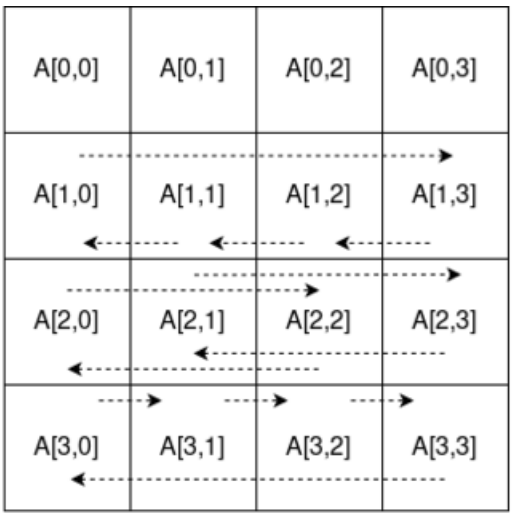
Выделение информационных зависимостей.

- Подзадача (i, j) отвечает за вычисление блока C_{ij}, все подзадачи образуют прямоугольную решетку размером $q \times q$,
- Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки в подзадачах могли бы быть перемножены без каких-либо дополнительных передач данных:
  - в каждую подзадачу (i, j) передаются блоки $\boldsymbol{A_{ij}}$, $\boldsymbol{B_{ij}}$,
  - для каждой строки i решетки подзадач блоки матрицы $\boldsymbol{A}$ сдвигаются на $(i - 1)$ позиций влево,
  - для каждой строки j решетки подзадач блоки матрицы B сдвигаются на $(j - 1)$ позиций вверх,
- процедуры передачи данных являются примером операции *циклического сдвига*.

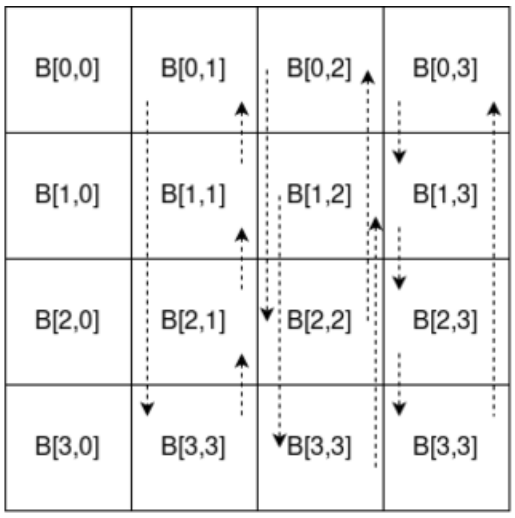
- В результате начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных,
- Для получения всех последующих блоков после выполнения операции блочного умножения:
  - каждый блок матрицы A передается предшествующей подзадаче влево по строкам решетки подзадач,
  - каждый блок матрицы B передается предшествующей подзадаче вверх по столбцам решетки.

Масштабирование и распределение подзадач по процессорам.

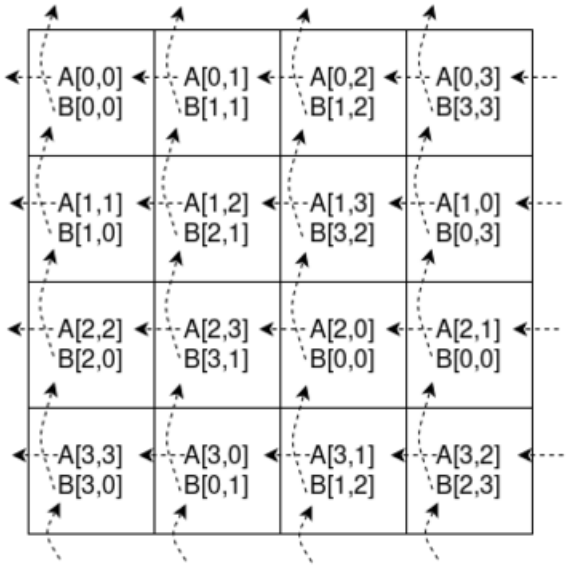
- Размер блоков может быть подобран таким образом, чтобы количество базовых подзадач совпадало с числом имеющихся процессоров,
- Множество имеющихся процессоров представляется в виде квадратной решетки и размещение базовых подзадач (i, j) осуществляется на процессорах $\boldsymbol{p_{i,j}}$ (соответствующих узлов процессорной решетки).



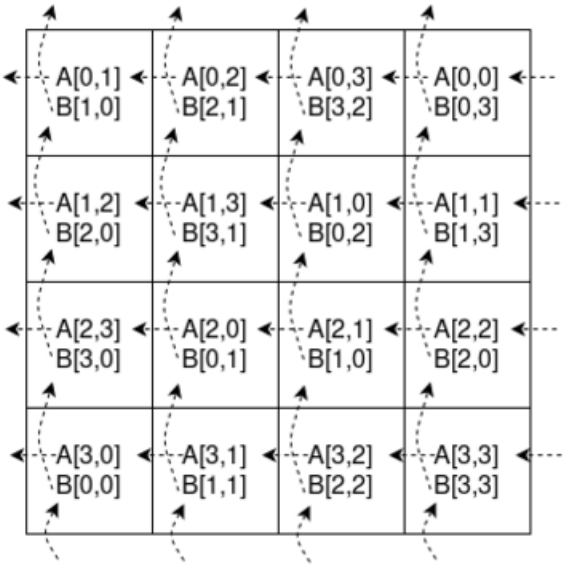
(а) первичное выравнивание A



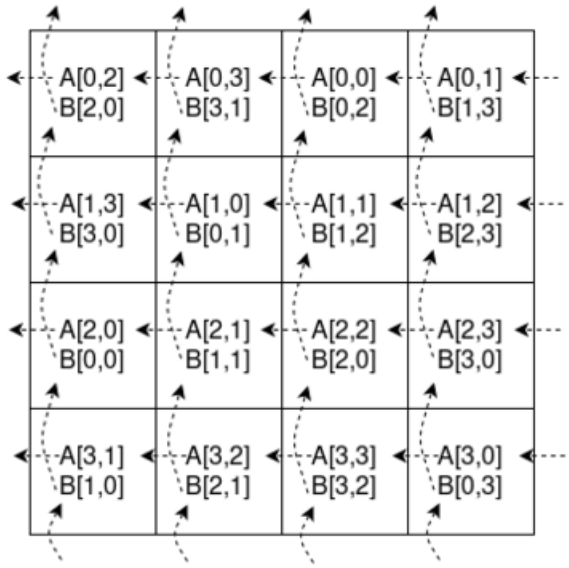
(б) первичное выравнивание B



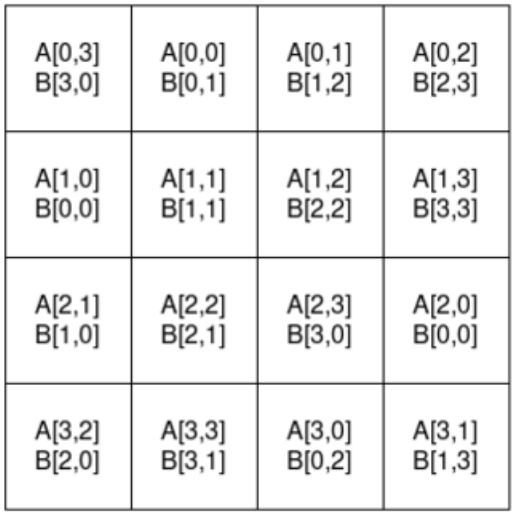
(в) A и B поле первичного выравнивания



(г) Расположение подматриц первого сдвига



(д) Расположение подматриц второго сдвига



(е) Расположение подматриц третьего сдвига

Анализ эффективности.

- Общая оценка показателей ускорения и эффективности:

$$S_p = \frac{n^2}{n^2/p} = p \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1$$

- Сложность:

$T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}$, где $t_s -$ стоимость старта, t_w

– время передачи блок или обратная полоса пропускания

Программная реализация.

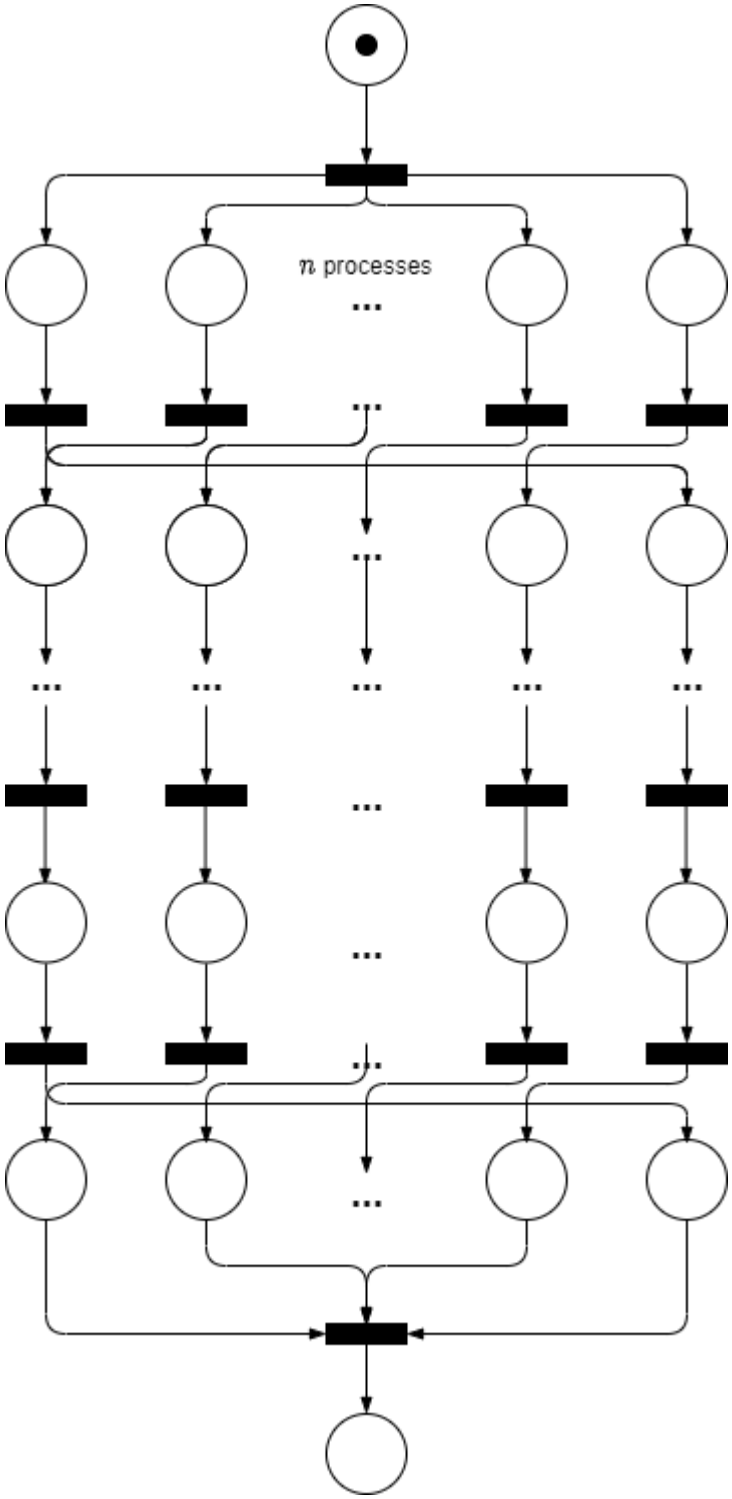
При помощи операции широковещательного обмена `MPI_Bcast` происходит создание 2D сетки процессов алгоритма Кэннона. Далее для создания 2-мерной декартовой топологии используется операция `MPI_Cart_create`.

После рутинных операция над двумерными массивами (матрицами) используется операция `MPI_Scatterv`, разбивающая сообщение из буфера посылки процесса root на равные части размером sendcount и посылающая *i*-ю часть в буфер приема процесса с номером *i* (в том числе и самому себе).

Далее используется функция `MPI_Cart_shift`, которая указывает признаки для кольцевого сдвига или для сдвига без переноса.

Для сбора блоков данных от всех процессов группы используется операция `MPI_Gatherv` (в корневом процессе).

Сеть Петри.



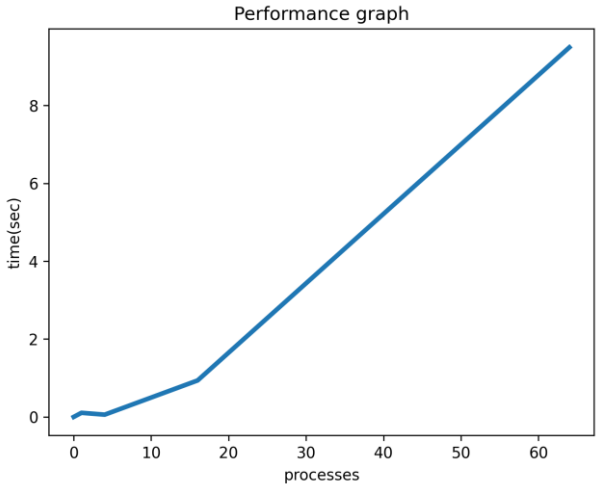
Технические параметры.

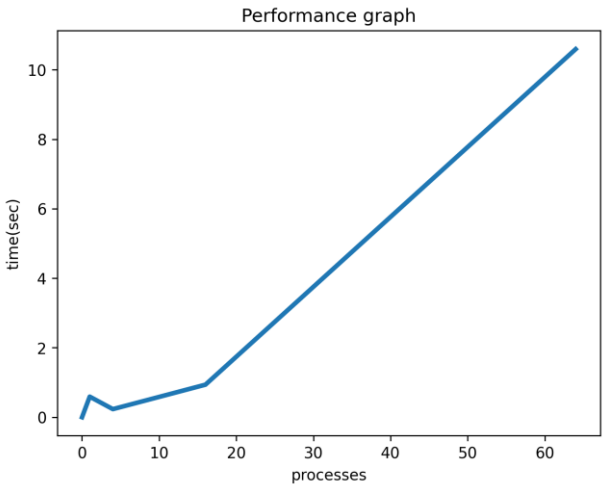
Запуск программ производился на ОС Linux Manjaro и процессоре AMD Ryzen™ 7 3700U (базовая частота 2.3GHz) с 4 ядрами CPU и 8 потоками.

Анализ эффективности.

В силе того, что на запускаемой машине 4 ядра, анализ производительности в зависимости от их количества провести затруднительно. Учитывая, что для корректной работы алгоритма, число ядер должно быть полным квадратом.

Для демонстрации были произведены измерения для матриц размером 256×256 и 512×512. Соответствующие графики приведены ниже.





Измерения приведены в таблице ниже:

Таблица 256×256		**Таблица 512×512**	
Процессы	**Время (сек.)**	**Процессы**	**Время (сек.)**
1	0.1055	1	0.5921
4	0.0581	4	0.2332
16	0.9368	16	0.9367
64	9.4912	64	10.5931

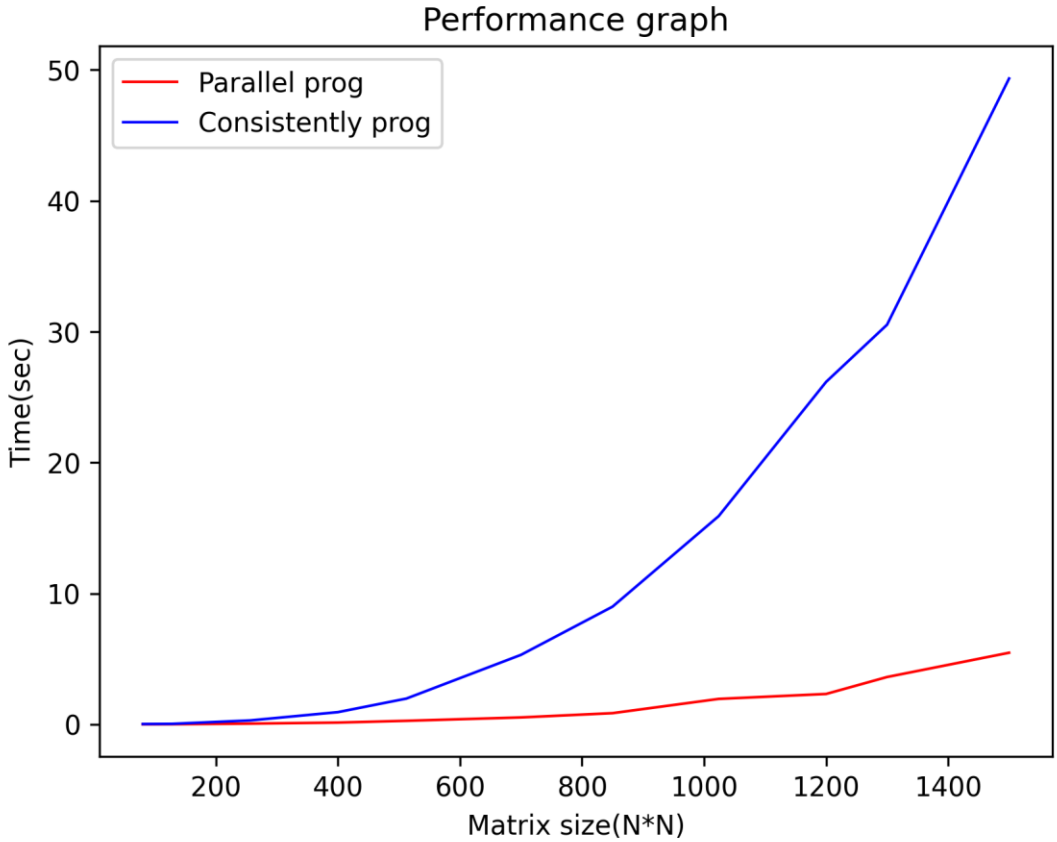
По полученным данным отчётливо видно, что при увеличении кол-ва процессов до 4 (макс. значение) мы получаем прирост производительности,

далее же расходы на менеджмент процессов является более высоким, чем на вычисления.

Далее производилось сравнение между параллельной реализацией данной задачи и параллельной. Стоит отметить, что последовательная программа выполнена с использованием самых быстрых доступных технологий языка C++, таких как std::vector с реверсированием памяти до использования, что даёт высокий прирост производительности. Параллельная программа выполняется на 4 процессах. Результаты замеров приведены в таблице ниже.

Размер матрицы	**Последовательная программа (время в сек.)**	**Параллельная программа (время в сек.)**
80	0.0252	0.0042
128	0.0406	0.0094
256	0.3003	0.0565
400	0.9339	0.1359
512	1.9687	0.2709
700	5.3115	0.5299
850	8.9921	0.8556
1024	15.9087	1.9448
1200	26.1579	2.3215
1300	30.5316	3.6169
1500	49.34017	5.4714

График полученной зависимости приведён ниже:



На основе полученных данных можно сделать однозначный вывод, что грамотное использование процессов даёт большой прирост производительности в вычисления по сравнению с последовательной программой, а также меньший прирост времени благодаря равномерному распределению нагрузки между процессами в алгоритме Кэннона.

Выводы.

В ходе выполнения лабораторной работы были закреплены навыки работы с библиотекой MPI, были на практике применены знания по построению виртуальных топологий, а также групповым операциям. В итоге был реализован алгоритм Кэннона, который на экспериментальных данных показал рост производительности при грамотном распараллеливании задачи.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ. ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <chrono>

#define SIZE 1300

size_t split(const std::string &txt, std::vector<std::string> &strs,
char ch)
{
    size_t pos = txt.find( ch );
    size_t initialPos = 0;
    strs.clear();

    // Decompose statement
    while( pos != std::string::npos ) {
        strs.push_back( txt.substr( initialPos, pos - initialPos )
);
        initialPos = pos + 1;

        pos = txt.find( ch, initialPos );
    }

    // Add the last one
    strs.push_back(   txt.substr(   initialPos,   std::min(   pos,
txt.size() ) - initialPos + 1 ) );

    return strs.size();
}

std::vector<std::vector<int>> read_matrix(std::string name)
{
    std::ifstream matrix_stream;
    matrix_stream.open("./" + name + "_" + std::to_string(SIZE) +
".txt" /*,ios_base::app*/);

    std::vector<std::vector<int>>                    matrix(SIZE,
std::vector<int>(SIZE, 0));

    std::string line;
    int i = 0;
    while (!matrix_stream.eof())
    {
        getline(matrix_stream, line);
        if (line.length() == 0) {
```

```
            break;
        }
        std::vector<std::string> numbers;
        split(line, numbers, ' ');
        for (int j = 0; j < SIZE; ++j)
        {
            int numb = std::stoi(numbers[j]);
            matrix[i][j] = std::stoi(numbers[j]);
        }
        i++;
    }

    matrix_stream.close();

    return matrix;
}

int main()
{
    std::vector<std::vector<int>> A, B;
    std::vector<std::vector<int>>  C(SIZE,  std::vector<int>(SIZE,
0));

    auto start = std::chrono::high_resolution_clock::now();

    A = read_matrix("A");
    B = read_matrix("B");

    for (int i = 0; i < SIZE; ++i)
    {
        for (int j = 0; j < SIZE; ++j)
        {
            C[i][j] = 0;
            for (int k = 0; k < SIZE; ++k)
                C[i][j] += A[i][k] * A[k][j];
        }
    }

    auto stop = std::chrono::high_resolution_clock::now();

    auto                          duration                          =
std::chrono::duration_cast<std::chrono::microseconds>(stop       -
start);

    std::cout << "Time: " + std::to_string(duration.count() * 1e-6)
+ "s" << std::endl;

    return 0;
}
```

ПРИЛОЖЕНИЕ Б.

ИСХОДНЫЙ КОД ПРОГРАММЫ. ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ.

```
#include <cmath>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// #define PRINT

std::string file_postfix = "_1300";

int allocMatrix(int ***mat, int rows, int cols)
{
    // Allocate rows * cols contiguous items
    int *p = (int *)malloc(sizeof(int *) * rows * cols);
    if (!p)
    {
        return -1;
    }

    // Allocate row pointers
    *mat = (int **)malloc(rows * sizeof(int *));
    if (!mat)
    {
        free(p);
        return -1;
    }

    // Set up the pointers into the contiguous memory
    for (int i = 0; i < rows; i++)
    {
        (*mat)[i] = &(p[i * cols]);
    }

    return 0;
}

int freeMatrix(int ***mat)
{
    free(&((*mat)[0][0]));
    free(*mat);

    return 0;
}

void matrixMultiply(int **a, int **b, int rows, int cols, int ***c)
{
    for (int i = 0; i < rows; i++)
    {
```

```
        for (int j = 0; j < cols; j++)
        {
            int val = 0;
            for (int k = 0; k < rows; k++)
            {
                val += a[i][k] * b[k][j];
            }
            (*c)[i][j] = val;
        }
    }
}

void printMatrix(int **mat, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

void printMatrixFile(int **mat, int size, FILE *fp)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            fprintf(fp, "%d ", mat[i][j]);
        }
        fprintf(fp, "\n");
    }
}

int main(int argc, char *argv[])
{
    MPI_Comm cartComm;
    int dim[2], period[2], reorder;
    int coord[2], id;
    FILE *fp;
    int **A = NULL, **B = NULL, **C = NULL;
    int **localA = NULL, **localB = NULL, **localC = NULL;
    int **localARec = NULL, **localBRec = NULL;
    int rows = 0;
    int columns;
    int count = 0;
    int worldSize;
    int procDim;
    int blockDim;
    int left, right, up, down;
```

```
    int bCastData[4];

    // For time measuring
    double start, end;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // World size
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);

    start = MPI_Wtime();

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        int n;
        char ch;

        // Determine matrix dimensions
        fp = fopen(("A" + file_postfix + ".txt").c_str(), "r");
        if (fp == NULL)
        {
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        while (fscanf(fp, "%d", &n) != EOF)
        {
            ch = fgetc(fp);
            if (ch == '\n')
            {
                rows = rows + 1;
            }
            count++;
        }
        columns = count / rows;

        // Check matrix and world size
        if (columns != rows)
        {
            printf("[ERROR] Matrix must be square!\n");
            MPI_Abort(MPI_COMM_WORLD, 2);
        }
        double sqroot = sqrt(worldSize);
        if ((sqroot - floor(sqroot)) != 0)
        {
            printf("[ERROR] Number of processes must be a perfect
square!\n");
            MPI_Abort(MPI_COMM_WORLD, 2);
        }
        int intRoot = (int)sqroot;
```

```
        if (columns % intRoot != 0 || rows % intRoot != 0)
        {
            printf("[ERROR] Number of rows/columns not divisible by
%d!\n", intRoot);
            MPI_Abort(MPI_COMM_WORLD, 3);
        }
        procDim = intRoot;
        blockDim = columns / intRoot;

        fseek(fp, 0, SEEK_SET);

        if (allocMatrix(&A, rows, columns) != 0)
        {
            printf("[ERROR] Matrix alloc for A failed!\n");
            MPI_Abort(MPI_COMM_WORLD, 4);
        }
        if (allocMatrix(&B, rows, columns) != 0)
        {
            printf("[ERROR] Matrix alloc for B failed!\n");
            MPI_Abort(MPI_COMM_WORLD, 5);
        }

        // Read matrix A
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                fscanf(fp, "%d", &n);
                A[i][j] = n;
            }
        }

#ifdef PRINT
        printf("A matrix:\n");
        printMatrix(A, rows);
#endif /* PRINT */
        fclose(fp);

        // Read matrix B
        fp = fopen(("B" + file_postfix + ".txt").c_str(), "r");
        if (fp == NULL)
        {
            return 1;
        }
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                fscanf(fp, "%d", &n);
                B[i][j] = n;
            }
        }
```

```
#ifdef PRINT
        printf("B matrix:\n");
        printMatrix(B, rows);
#endif /* PRINT */
        fclose(fp);

        if (allocMatrix(&C, rows, columns) != 0)
        {
            printf("[ERROR] Matrix alloc for C failed!\n");
            MPI_Abort(MPI_COMM_WORLD, 6);
        }

        bCastData[0] = procDim;
        bCastData[1] = blockDim;
        bCastData[2] = rows;
        bCastData[3] = columns;
    }

    // Create 2D Cartesian grid of processes
    MPI_Bcast(&bCastData, 4, MPI_INT, 0, MPI_COMM_WORLD);
    procDim = bCastData[0];
    blockDim = bCastData[1];
    rows = bCastData[2];
    columns = bCastData[3];

    dim[0] = procDim;
    dim[1] = procDim;
    period[0] = 1;
    period[1] = 1;
    reorder = 1;
    MPI_Cart_create(MPI_COMM_WORLD,  2,  dim,  period,  reorder,
&cartComm);

    // Allocate local blocks for A and B
    allocMatrix(&localA, blockDim, blockDim);
    allocMatrix(&localB, blockDim, blockDim);

    // Create datatype to describe the subarrays of the global array
    int globalSize[2] = {rows, columns};
    int localSize[2] = {blockDim, blockDim};
    int starts[2] = {0, 0};
    MPI_Datatype type, subarrtype;
    MPI_Type_create_subarray(2,  globalSize,  localSize,  starts,
MPI_ORDER_C, MPI_INT, &type);
    MPI_Type_create_resized(type,  0,  blockDim  *  sizeof(int),
&subarrtype);
    MPI_Type_commit(&subarrtype);

    int *globalptrA = NULL;
    int *globalptrB = NULL;
    int *globalptrC = NULL;
```

```
    if (rank == 0)
    {
        globalptrA = &(A[0][0]);
        globalptrB = &(B[0][0]);
        globalptrC = &(C[0][0]);
    }

    // Scatter the array to all processors
    int *sendCounts = (int *)malloc(sizeof(int) * worldSize);
    int *displacements = (int *)malloc(sizeof(int) * worldSize);

    if (rank == 0)
    {
        for (int i = 0; i < worldSize; i++)
        {
            sendCounts[i] = 1;
        }
        int disp = 0;
        for (int i = 0; i < procDim; i++)
        {
            for (int j = 0; j < procDim; j++)
            {
                displacements[i * procDim + j] = disp;
                disp += 1;
            }
            disp += (blockDim - 1) * procDim;
        }
    }

    MPI_Scatterv(globalptrA, sendCounts, displacements, subarrtype,
&(localA[0][0]),
                 rows * columns / (worldSize), MPI_INT,
                 0, MPI_COMM_WORLD);
    MPI_Scatterv(globalptrB, sendCounts, displacements, subarrtype,
&(localB[0][0]),
                 rows * columns / (worldSize), MPI_INT,
                 0, MPI_COMM_WORLD);

    if (allocMatrix(&localC, blockDim, blockDim) != 0)
    {
        printf("[ERROR] Matrix alloc for localC in rank %d
failed!\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 7);
    }

    // Initial skew
    MPI_Cart_coords(cartComm, rank, 2, coord);
    MPI_Cart_shift(cartComm, 1, coord[0], &left, &right);
    MPI_Sendrecv_replace(&(localA[0][0]), blockDim * blockDim,
MPI_INT, left, 1, right, 1, cartComm, MPI_STATUS_IGNORE);
    MPI_Cart_shift(cartComm, 0, coord[1], &up, &down);
```

```
    MPI_Sendrecv_replace(&(localB[0][0]), blockDim * blockDim,
MPI_INT, up, 1, down, 1, cartComm, MPI_STATUS_IGNORE);

    // Init C
    for (int i = 0; i < blockDim; i++)
    {
        for (int j = 0; j < blockDim; j++)
        {
            localC[i][j] = 0;
        }
    }

    int **multiplyRes = NULL;
    if (allocMatrix(&multiplyRes, blockDim, blockDim) != 0)
    {
        printf("[ERROR] Matrix alloc for multiplyRes in rank %d
failed!\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 8);
    }
    for (int k = 0; k < procDim; k++)
    {
        matrixMultiply(localA, localB, blockDim, blockDim,
&multiplyRes);

        for (int i = 0; i < blockDim; i++)
        {
            for (int j = 0; j < blockDim; j++)
            {
                localC[i][j] += multiplyRes[i][j];
            }
        }
        // Shift A once (left) and B once (up)
        MPI_Cart_shift(cartComm, 1, 1, &left, &right);
        MPI_Cart_shift(cartComm, 0, 1, &up, &down);
        MPI_Sendrecv_replace(&(localA[0][0]), blockDim * blockDim,
MPI_INT, left, 1, right, 1, cartComm, MPI_STATUS_IGNORE);
        MPI_Sendrecv_replace(&(localB[0][0]), blockDim * blockDim,
MPI_INT, up, 1, down, 1, cartComm, MPI_STATUS_IGNORE);
    }

    // Gather results
    MPI_Gatherv(&(localC[0][0]), rows * columns / worldSize,
MPI_INT,
                globalptrC, sendCounts, displacements, subarrtype,
                0, MPI_COMM_WORLD);

    freeMatrix(&localC);
    freeMatrix(&multiplyRes);

    if (rank == 0)
    {
#ifdef PRINT
```

```
            printf("C is:\n");
            printMatrix(C, rows);
    #endif /* PRINT */

            end = MPI_Wtime();
            if (rank == 0)
                printf("Time: %.4fs\n", end - start);
        }

        // Finalize the MPI environment
        MPI_Finalize();

        return 0;
}
```