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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Системы параллельной обработки данных»

Тема: Умножение матриц

Студент гр. 5304 _____ Лянгузов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2019

ФОРМУЛИРОВКА ЗАДАНИЯ

1. Реализовать алгоритм умножения двух квадратных матриц;
    a. последовательный алгоритм;
    b. параллельный алгоритм (2-варианта):
        i. Матрица A разбивается на горизонтальные полосы, матрица B – делится на вертикальные полосы;
        ii. Матрица A и B – разбивается на горизонтальные полосы;
2. Определить подзадачи (для решения задачи масштабируемости-размерность матрицы превышает число процессоров);
3. Определить информационные связи (простое решение этой проблемы – дублирование матрицы B во всех подзадачах – является, как правило, неприемлемым в силу больших затрат памяти для хранения данных.);
4. Реализовать масштабирование и распределение подзадач по процессам;
5. Провести анализ эффективности (анализ эффективности алгоритмов – до проведения эксперимента)
6. Выполнить программную реализацию;
7. Привести результаты вычислительных экспериментов в виде таблицы и графика ускорения в зависимости от кол-ва процессоров;

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Последовательный алгоритм умножения матриц имеет сложность $O(n^3)$. Данный алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы. Вычисление результирующий матрицы способствует распараллеливанию.

В работе был реализован параллельный алгоритм ленточного умножения матриц. Алгоритм использует сдвиг в кольце по модулю (топология кольцо).

Каждый процесс считывает соответствующие ему строки матрицы A. Нулевая строка должна считываться нулевым процессом, первая строка - первым процессом и т.д. (по модулю).

Каждый процесс считывает соответствующий ему столбец матрицы B.

Каждый процесс вычисляет одну подматрицу произведения. Столбцы матрицы B сдвигаются вдоль кольца процессов.

Матрица C собирается в нулевом процессе.

ОПИСАНИЕ АЛГОРИТМА С ПОМОЩЬЮ СЕТЕЙ ПЕТРИ

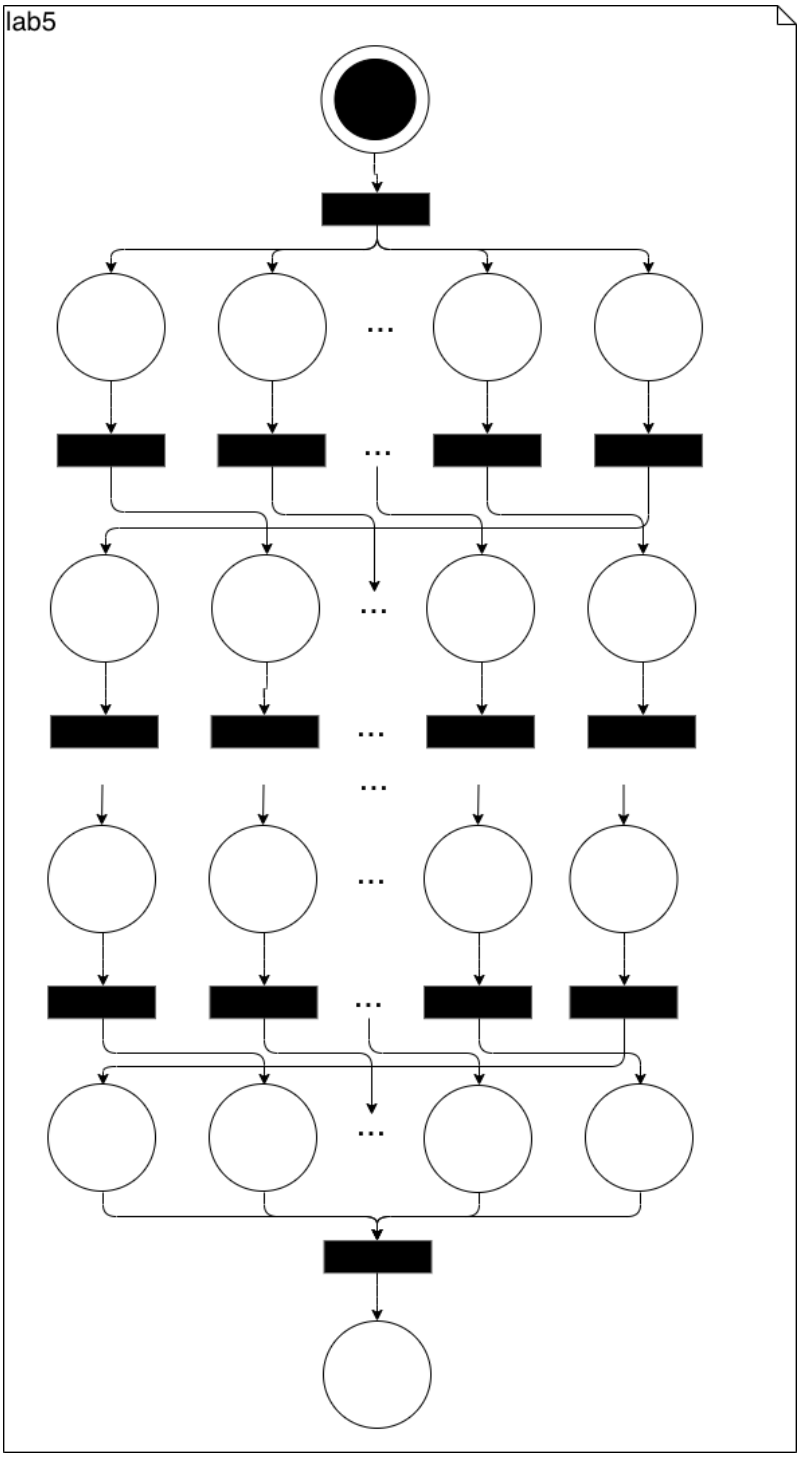

Рисунок 1 – Сеть Петри для алгоритма параллельного ленточного перемножения матриц.

РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

```
(base) xtail@~/Projects/LETI/Parallel/ParallelLabs$ ./assembly++.sh launch lab5 2

Parallel latency = 0.000358

Sequential latency = 0.000001

Process 0: MATRIX A
 1680.700000  330237.489000  1203733.775000  1857601.685000  594259.709000
 1923970.613000  1512819.812000  1903683.451000  1996387.951000  1007791.729000
 752065.414000  2022150.503000  207306.499000  981853.259000  743380.465000
 2083100.656000  246828.351000  1653172.900000  733505.414000  1469359.318000
 1607600.773000  1454428.904000  1927719.374000  93736.529000  1324329.652000

Process 0: MATRIX B
 1680.700000  330237.489000  1203733.775000  1857601.685000  594259.709000
 1923970.613000  1512819.812000  1903683.451000  1996387.951000  1007791.729000
 752065.414000  2022150.503000  207306.499000  981853.259000  743380.465000
 2083100.656000  246828.351000  1653172.900000  733505.414000  1469359.318000
 1607600.773000  1454428.904000  1927719.374000  93736.529000  1324329.652000

Process 0: MATRIX C
 11775788122252.681641  7649877428231.837891  9047904668875.656250  6469413918676.648438  8703247555009.193359
 18628578785606.031250  15998342623111.037109  19724537359621.582031  19949757835194.867188  15367684145292.705078
 13381117720347.982422  9019321434441.431641  14274969268467.031250  12785230149434.478516  9147743917384.355469
 8861445838605.193359  11307756847652.814453  11897925268589.892578  13184812295192.320312  9532663742798.945312
 6575007431009.195312  8578597762658.165039  7811422954061.430664  7975518110404.503906  5745704381544.087891
(base) xtail@~/Projects/LETI/Parallel/ParallelLabs$
```

Рисунок 2 – Результат работы программы.

Вычислительные эксперименты:

Размер матриц	*1*	*2*		*3*		*4*		*5*		*7*		*8*		*10*	
10	0,000014	0,000 305	0,045 902	0,000 712	0,019 663	0,000 543	0,026 783	0,019 961	0,000 701	0,092 715	0,000 151	0,061 631	0,000 227	0,113 337	0,000 124
50	0,00095	0,002 65	0,358 491	0,001 856	0,511 853	0,001 746	0,544 101	0,034 893	0,027 226	0,101 194	0,009 388	0,131 031	0,007 250	0,283 741	0,003 348
100	0,004589	0,003 189	1,439 009	0,004 543	1,010 125	0,004 029	1,138 992	0,041 306	0,111 098	0,183 767	0,024 972	0,368 081	0,012 467	0,794 891	0,005 773
200	0,043388	0,017 636	2,460 195	0,023 356	1,857 681	0,023 073	1,880 466	0,068 679	0,631 751	0,219 577	0,197 598	0,299 882	0,144 684	0,828 234	0,052 386
500	0,822593	0,307 855	2,672 014	0,371 269	2,215 625	0,309 884	2,654 519	0,443 617	1,854 286	0,588 051	1,398 846	0,650 593	1,264 374	1,578 6920	5,210 598
1000	7,564862	3,346 669	2,260 415	3,191 255	2,370 497	3,499 129	2,161 927	3,031 73	2,495 230	3,057 000	2,474 603	3,780 413	2,001 068	4,793 643	1,578 102

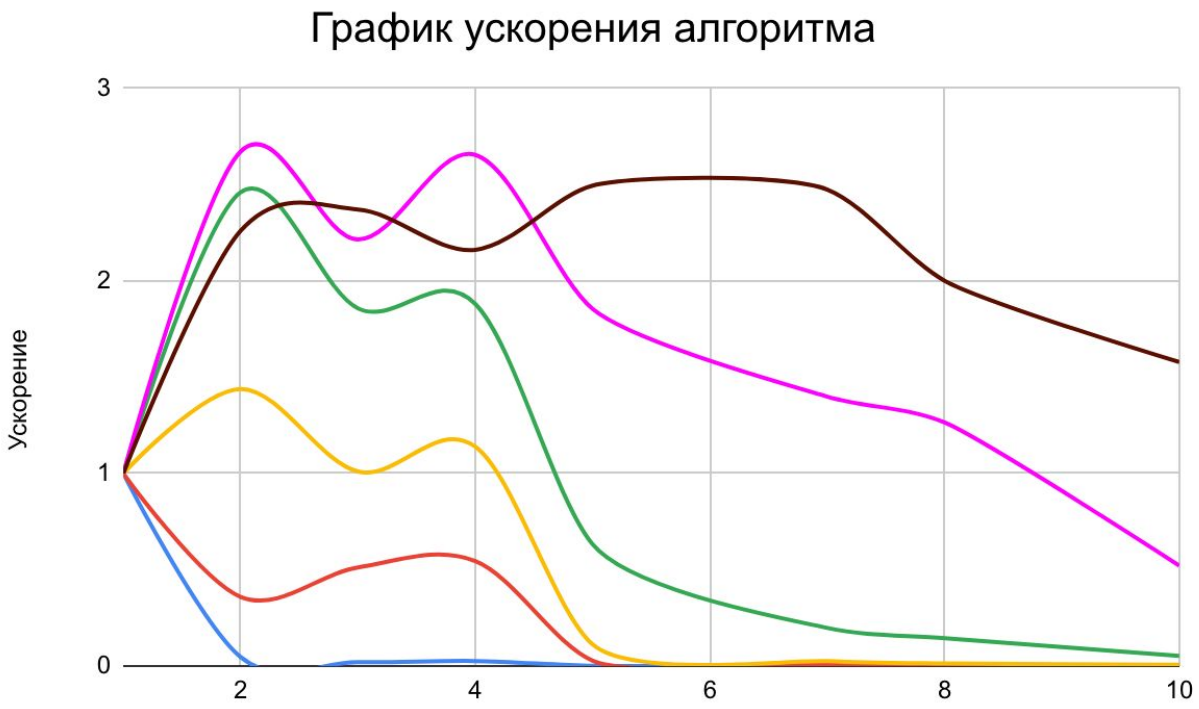

Рисунок 3 – Графики ускорений алгоритма.

x – количество процессов, y – ускорение.

синий - размерность матриц **10**, **красный** - размерность матриц **50**,

желтый - размерность матриц **100**, **зеленый** - размерность матриц **200**,

розовый - размерность матриц **500**, **коричневый** - размерность матриц **1000**.

ВЫВОДЫ

В ходе лабораторной работы был реализован алгоритм параллельного перемножения матриц.

В виду невозможности измерить время выполнения параллельной программы на кластере, были проведены измерения на локальной машине имеющей 4Гб оперативной памяти и двухъядерный процессор intel core i5 5250U с частотой каждого ядра в 1.6ГГц. Полученные данные сведены в таблицу времени и ускорения на различных размерах матриц и количестве процессов.

Анализируя полученные результаты, можно заключить, что при малых размерностях матриц (10-50) параллельное выполнение не является рациональным. Для матриц большего размера, оптимальным количеством является 2-4 процесса. При количестве процессов большем чем 6 скорость вычисления снижается, из-за накладных расходов на коммуникацию и передачу данных между процессами.

ЛИСТИНГ ПРОГРАММЫ

```
#include <stdio.h>

void fillMatrixWithValue(int *M, int matrixSize, int value) {
    for (int i = 0; i < matrixSize * matrixSize; i++) {
        M[i] = value;
    }
}

void fillMatrixWithIncrementValue(int *M, int matrixSize, int value, int
step) {
    for (int i = 0; i < matrixSize * matrixSize; i++) {
        M[i] = value;
        value += step;
    }
}

void fillData(int *A, int *B, int *C, int blockSize, int step) {
    int counter = 0;
    for (int i = 0; i < blockSize; i++) {
        for (int j = 0; j < blockSize; j++) {
            counter += step;
            A[i * blockSize + j] = counter;
            B[i * blockSize + j] = 1;
            C[i * blockSize + j] = 0;
        }
    }
}

void RandInit (double* pMatrix, int Size) {
    srand(100);
    for (int i=0; i<Size; i++) {
        for (int j=0;j<Size;j++)  pMatrix[i*Size+j]=rand()/double(1000);
    }
}

void printMatrix(int *M, int matrixSize) {
```

```
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            printf("%d\t", M[i * matrixSize + j]);
        }
        printf("\n");
    }
}

double matrixMultiplicationSequential(double *a, double *b, double *c, int
n) {
    double time_start = MPI_Wtime();
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                c[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
    double time_finish = MPI_Wtime();
    return time_finish - time_start;
}

void printMatrices(double *A, double *B, double *C, int blockSize) {
    printf("\nProcess 0: MATRIX A\n");
    for (int i = 0; i < blockSize; i++) {
        for (int j = 0; j < blockSize; j++) {
            printf(" %f ", A[i * blockSize + j]);
        }
        printf("\n");
    }
    printf("\nProcess 0: MATRIX B\n");
    for (int i = 0; i < blockSize; i++) {
        for (int j = 0; j < blockSize; j++) {
            printf(" %f ", B[i * blockSize + j]);
        }
        printf("\n");
    }
```

```
    printf("\nProcess 0: MATRIX C\n");
    for (int i = 0; i < blockSize; i++) {
        for (int j = 0; j < blockSize; j++) {
            printf(" %f ", C[i * blockSize + j]);
        }
        printf("\n");
    }
}
#include <mpi.h>
#include <iostream>

#include "matrix_functions.h"

int ProcNum;
int ProcRank;
int debug_info=1;
int Size = 5;
double *A;
double *B;
double *C;

double matrixMultiplicationParallelRibbon(double *A, double *B,  double
*C, int size){
    int i = 0, j = 0;

    double *bufA, *bufB, *bufC;
    int dim = size;

    MPI_Status Status;

    int ProcPartsize = dim/ProcNum;
    int ProcPartElem = ProcPartsize*dim;

    bufA = new double[ProcPartElem];
    bufB = new double[ProcPartElem];
    bufC = new double[ProcPartElem];

    for (i = 0; i < ProcPartElem; i++)
```

```
    {
        bufC[i] = 0;
    }

    double time_start = MPI_Wtime();
    MPI_Scatter(A, ProcPartElem, MPI_DOUBLE, bufA, ProcPartElem,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(B, ProcPartElem, MPI_DOUBLE, bufB, ProcPartElem,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

    int k = 0 ;
    int NextProc = ProcRank + 1;
    if ( ProcRank == ProcNum - 1 ) NextProc = 0;

    int PrevProc = ProcRank - 1;
    if ( ProcRank == 0 ) PrevProc = ProcNum - 1;

    int PrevDataNum = ProcRank;
    for (int p = 0; p < ProcNum; p++)
    {
        for (i = 0; i < ProcPartsize; i++)
        {
            for (j = 0; j < size; j++)
            {
                double tmp = 0;
                for (k = 0; k < ProcPartsize; k++)
                    tmp += bufA[PrevDataNum * ProcPartsize + i * size + k]
* bufB[k * size + j];
                bufC[i * size + j] += tmp;
            }
        }

        PrevDataNum -= 1;

        if (PrevDataNum < 0)
            PrevDataNum = ProcNum - 1;
```

```
        MPI_Sendrecv_replace(bufB, ProcPartElem, MPI_DOUBLE, NextProc, 0,
PrevProc, 0, MPI_COMM_WORLD, &Status);
    }

    MPI_Gather(bufC, ProcPartElem, MPI_DOUBLE, C, ProcPartElem, MPI_DOUBLE,
0, MPI_COMM_WORLD);

    double time_finish = MPI_Wtime();
    return time_finish - time_start;
}

void InitProcess (double* &A,double* &B,double* &C ,int &Size) {
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (ProcRank == 0) {
        A = new double [Size*Size];
        B = new double [Size*Size];
        C = new double [Size*Size];
        RandInit (A, Size); RandInit (B, Size);
    }
}


int main(int argc, char **argv) {
    double beg, end, serial;

    MPI_Init ( &argc, &argv );
    InitProcess (A,B,C,Size);

    double parallel_latency =
matrixMultiplicationParallelRibbon(A,B,C,Size);

    if (ProcRank == 0) {
        std::cout << std::fixed << std::endl << "Parallel latency = " <<
parallel_latency << std::endl;
```

```
        double sequential_latency = matrixMultiplicationSequential(A, B, C,
Size);
        std::cout << std::fixed << std::endl << "Sequential latency = " <<
sequential_latency << std::endl;

        if (debug_info) {
            printMatrices(A, B, C, Size);
        }
    }
    MPI_Finalize();
    delete [] A;
    delete [] B;
    delete [] C;
}
```