**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. 8304 |  | Сергеев А.Д. |
| Преподаватель |  | Татаринов Ю.С. |

Санкт-Петербург

2020

**Задание.**

Вариант 4. Реализовать блочный алгоритм Фокса.

Выполнить задачу умножения двух квадратных матриц A и B размера m × 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. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

**Алгоритм решения задачи.**

Исходные матрицы и результирующая матрица представляются в виде набора блоков – подматриц, количество которых равно количеству процессов минус один. Сторона этих подматриц подбирается соответствующим образом. Каждый процесс отвечает за вычисление определенного блока матрицы C. Поскольку для большого количества процессов и матриц большого размера хранить исходные матрицы целиком в каждом процессе невыгодно, в каждом процессе одновременно располагается по одному обрабатываемому ими блоку матриц A и B.

В процессе работы алгоритма, после того как корневой процесс рассылает блоки исходных матриц всем процессам (включая себя самого), в цикле от 0 до количества блоков для каждой строки блок матрицы A рассылается процессам той же строки, полученные блоки матриц A и B перемножаются и прибавляются к блоку матрицы C, после чего блоки матрицы B в каждом процессе пересылаются вверх по столбцу, причем из первой строки они пересылаются в последнюю. В конце работы алгоритма блоки матрицы C собираются из всех процессов обратно в корневой и из них строится матрица.

Для обеспечения простой рассылки данных между процессами, все они группируются в двухмерную декартову топологию – матрицу, зацикленную по вертикали. Также создаются отдельные коммуникаторы для строк и столбцов этой матрицы процессов.

Для удобной группировки и пересылки данных о матрице используется структура *matrix*, хранящая размер матрицы в целочисленном поле, а элементы матрицы – в одномерном целочисленном массиве, строки по порядку, а также пользовательский тип *MPI* – *MPI\_BLOCK*, использующий функциональность типа *MPI\_Vector* и настроенный для работы с массивом, размер которого равен количеству элементов в блоке.

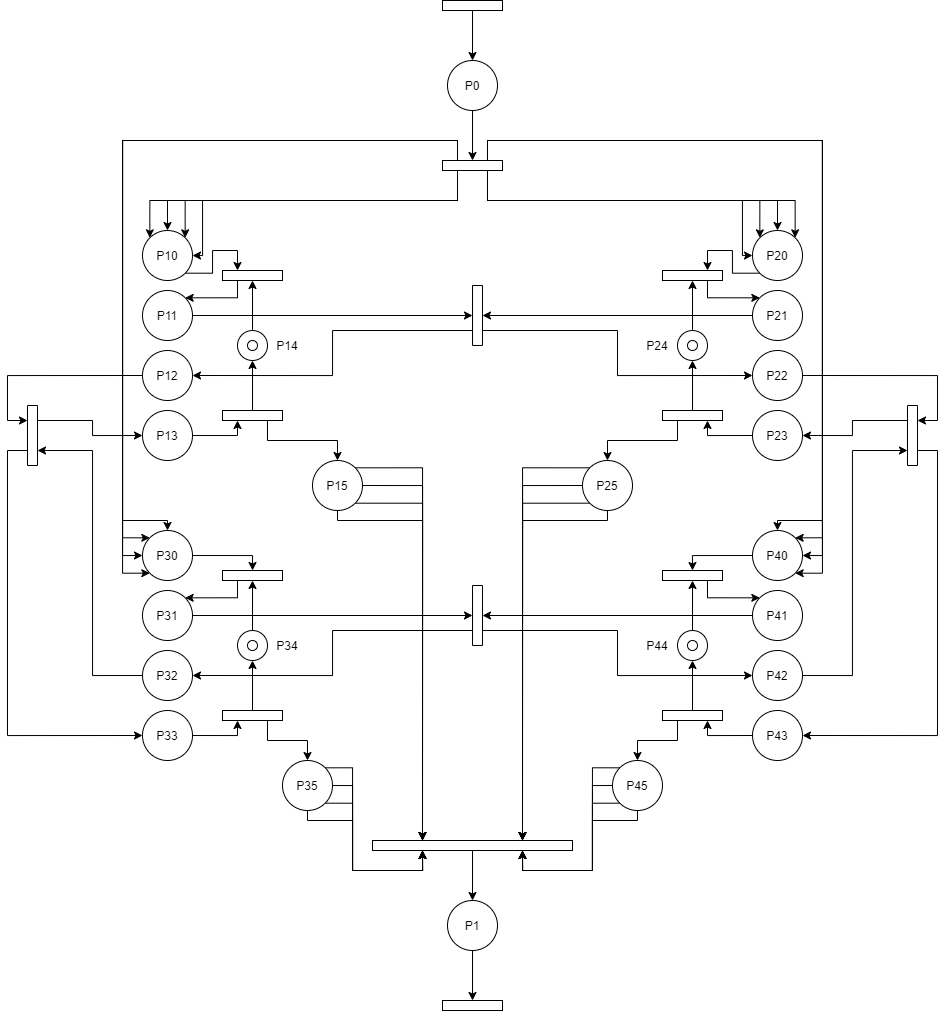
**Сеть Петри.**

Схема работы алгоритма для четырех процессов. P0 и P1 – состояния корневого процесса, P10 – P15 – первого, P20 – P25 – второго, P30 – P35 – третьего, P40 – P45 – четвертого.

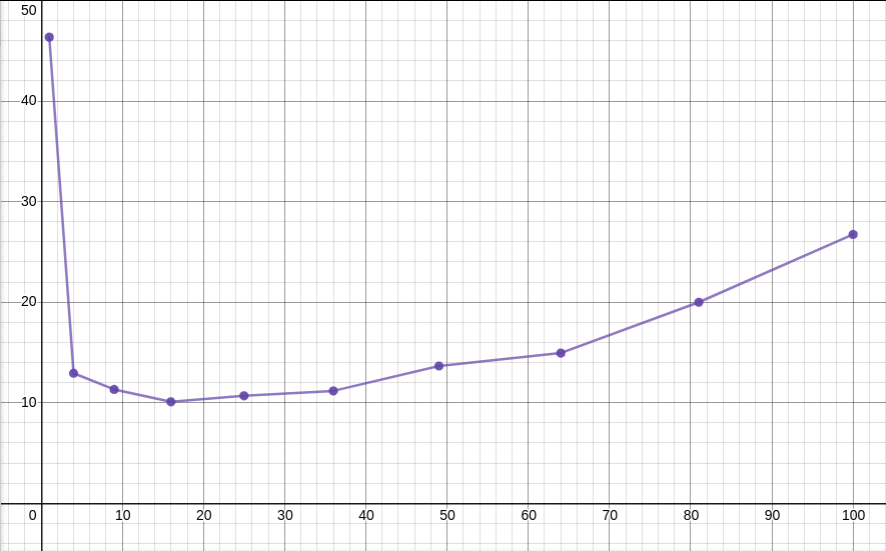
Расположение на схеме групп состояний, относящихся к определенному процессу, демонстрирует его позицию в декартовой топологии.

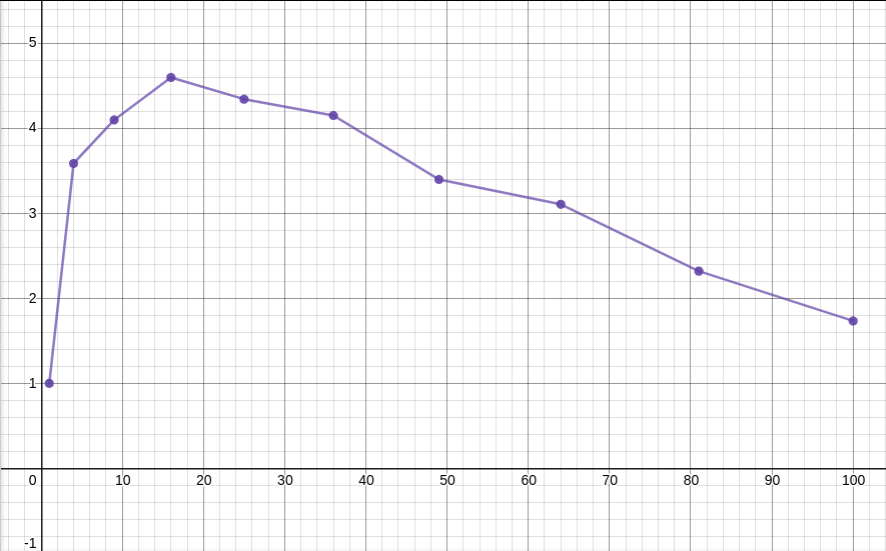
**Результаты вычислительных экспериментов.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность матриц (m) | Последовательный алгоритм | 4 процесса | | 16 процессов | |
|  | время | время | ускорение | время | ускорение |
| 10 | 0.000032 | 0.000889 | 0.036000 | 0.480071 | 0.000067 |
| 50 | 0.002479 | 0.001576 | 1.572970 | 0.478945 | 0.005176 |
| 100 | 0.015191 | 0.003038 | 5.000330 | 0.477437 | 0.031818 |
| 200 | 0.064582 | 0.029891 | 2.160583 | 0.454178 | 0.142195 |
| 500 | 0.940888 | 0.264747 | 3.553914 | 0.643422 | 1.462319 |
| 1000 | 11.136799 | 3.562549 | 3.126076 | 3.491894 | 3.189329 |

**Графики времени и ускорения.**

Значения получены для матриц размера 1500.

График зависимости врмемени выполнения от количества процессов

График зависимости ускорения от количества процессов

**Выводы.**

В ходе выполнения лабораторной был изучен блочный алгоритм Фокса по перемножению квадратных матриц. Изучены способы разбиения задачи на подзадачи.

**Приложение А. Исходный код программы**

#include <stdlib.h>

#include <stdio.h>

#include <mpi.h>

#include <string.h>

#include <math.h>

#include <time.h>

#define boolean int

#define true 1

#define false 0

#define gen\_matrix true

#define out\_matrix false

typedef struct {

int dimension;

int\* elements;

} matrix;

matrix\* create\_matrix (int d) {

matrix\* m = malloc(sizeof(matrix));

m->dimension = d;

m->elements = calloc(d \* d, sizeof(int));

return m;

}

matrix\* input\_matrix () {

char\* line, \* end;

size\_t len = 0;

printf("Введите размерность матрицы: ");

getline(&line, &len, stdin);

int dimension = (int) strtol(line, &end, 10);

matrix\* m = create\_matrix(dimension);

printf("Введите строки матрицы, разделяя элементы пробелами!\n");

for (int i = 0; i < m->dimension; ++i) {

printf("Введите строку №%d: ", i + 1);

getline(&line, &len, stdin);

char\* str = strtok(line, " ");

int j = 0;

while (str != NULL) {

m->elements[i \* m->dimension + j] = (int) strtol(str, &end, 10);

str = strtok(NULL, " ");

j++;

}

if (j != m->dimension) {

printf("Повторите ввод! ");

i--;

}

}

return m;

}

matrix\* generate\_matrix (char name) {

char\* line, \* end;

size\_t len = 0;

printf("Введите размерность матрицы %c: ", name);

getline(&line, &len, stdin);

int dimension = (int) strtol(line, &end, 10);

matrix\* m = create\_matrix(dimension);

for (int i = 0; i < m->dimension; ++i)

for (int j = 0; j < m->dimension; ++j)

m->elements[i \* m->dimension + j] = ((int) rand()) % 10;

return m;

}

void print\_matrix(matrix\* m) {

printf("Матрица %dx%d элементов:\n", m->dimension, m->dimension);

for (int i = 0; i < m->dimension; ++i) {

printf("[");

for (int j = 0; j < m->dimension; ++j) {

printf(" %d", m->elements[i \* m->dimension + j]);

if (j < m->dimension - 1) printf(",");

}

printf(" ]\n");

}

}

void remove\_matrix (matrix\* m) {

if (m == NULL) return;

free(m->elements);

free(m);

}

int proc\_coords [2];

int proc\_num, proc\_rank;

int block\_size, grid\_size, matrix\_dim;

matrix\* A = NULL, \* B = NULL, \* C = NULL, \* A\_part = NULL, \* B\_part = NULL, \* C\_part = NULL, \* temp\_part = NULL;

MPI\_Comm grid\_comm, col\_comm, row\_comm;

MPI\_Datatype MPI\_BLOCK;

MPI\_Status status;

MPI\_Request request;

boolean is\_root () {

return proc\_rank == 0;

}

void solve\_parallel() {

if (is\_root()) printf("Исходя из количества процессов выбран параллельный способ решения\n");

grid\_size = (int) sqrt(proc\_num);

if (proc\_num != grid\_size \* grid\_size) {

if (is\_root()) printf("Невозможно построить сеть для такого количества процессов!\n");

return;

}

double start\_time = MPI\_Wtime();

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, (int[]) {grid\_size, grid\_size},

(boolean[]) {true, false}, false, &grid\_comm);

MPI\_Cart\_coords(grid\_comm, proc\_rank, 2, proc\_coords);

MPI\_Cart\_sub(grid\_comm, (int[]) {true, false}, &col\_comm);

MPI\_Cart\_sub(grid\_comm, (int[]) {false, true}, &row\_comm);

MPI\_Bcast(&matrix\_dim, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

block\_size = matrix\_dim / grid\_size;

A\_part = create\_matrix(block\_size);

B\_part = create\_matrix(block\_size);

C\_part = create\_matrix(block\_size);

temp\_part= create\_matrix(block\_size);

MPI\_Type\_vector(block\_size, block\_size, matrix\_dim, MPI\_INT, &MPI\_BLOCK);

MPI\_Type\_commit(&MPI\_BLOCK);

if (is\_root()) for (int i = 0; i < proc\_num; ++i) {

int coords [2];

MPI\_Cart\_coords(grid\_comm, i, 2, coords);

MPI\_Isend(A->elements + coords[0] \* matrix\_dim \* block\_size + coords[1] \* block\_size,

1, MPI\_BLOCK, i, 0, grid\_comm, &request);

MPI\_Isend(B->elements + coords[0] \* matrix\_dim \* block\_size + coords[1] \* block\_size,

1, MPI\_BLOCK, i, 0, grid\_comm, &request);

}

MPI\_Recv(A\_part->elements, block\_size \* block\_size, MPI\_INT, 0, 0, grid\_comm, &status);

MPI\_Recv(B\_part->elements, block\_size \* block\_size, MPI\_INT, 0, 0, grid\_comm, &status);

for (int i = 0; i < grid\_size; ++i) {

int p = (proc\_coords[0] + i) % grid\_size;

if (proc\_coords[1] == p) for (int j = 0; j < block\_size \* block\_size; j++)

temp\_part->elements[j] = A\_part->elements[j];

MPI\_Bcast(temp\_part->elements, block\_size \* block\_size, MPI\_INT, p, row\_comm);

for (int j = 0; j < block\_size; j++) for (int k = 0; k<block\_size; k++) {

int t = 0;

for (int l = 0; l < block\_size; l++)

t += temp\_part->elements[j \* block\_size + l] \* B\_part->elements[l \* block\_size + k];

C\_part->elements[j \* block\_size + k] += t;

}

int giver, getter;

MPI\_Cart\_shift(col\_comm, 0, 1, &giver, &getter);

MPI\_Sendrecv\_replace(B\_part->elements, block\_size \* block\_size, MPI\_INT, getter, proc\_rank,

giver, MPI\_ANY\_TAG, col\_comm, &status);

}

if (!is\_root()) MPI\_Send(C\_part->elements, block\_size \* block\_size, MPI\_INT, 0, 0, grid\_comm);

else {

MPI\_Isend(C\_part->elements, block\_size \* block\_size, MPI\_INT, 0, 0, grid\_comm, &request);

for (int i = 0; i < proc\_num; ++i) {

int coords[2];

MPI\_Cart\_coords(grid\_comm, i, 2, coords);

MPI\_Recv(C->elements + coords[0] \* matrix\_dim \* block\_size + coords[1] \* block\_size,

1, MPI\_BLOCK, i, 0, grid\_comm, &status);

}

double end\_time = MPI\_Wtime();

printf("Времени прошло: %lf\n", end\_time - start\_time);

if (out\_matrix) {

printf("Матрица C:\n");

print\_matrix(C);

}

}

remove\_matrix(A\_part);

remove\_matrix(B\_part);

remove\_matrix(C\_part);

remove\_matrix(temp\_part);

}

void solve\_stepped() {

printf("Исходя из количества процессов выбран последовательный способ решения\n");

double start\_time = MPI\_Wtime();

for (int i = 0; i < C->dimension; i++) for (int j = 0; j < C->dimension; j++) for (int k = 0; k < C->dimension; k++)

C->elements[i \* C->dimension + j] += A->elements[i \* A->dimension + k] \* B->elements[k \* B->dimension + j];

double end\_time = MPI\_Wtime();

printf("Времени прошло: %lf\n", end\_time - start\_time);

if (out\_matrix) {

printf("Матрица C:\n");

print\_matrix(C);

}

}

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

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

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

if (is\_root()) {

if (gen\_matrix) {

srand(time(NULL));

A = generate\_matrix('A');

B = generate\_matrix('B');

} else {

printf("Введите информацию о матрице А:\n");

A = input\_matrix();

printf("Введите информацию о матрице B:\n");

B = input\_matrix();

}

if (A->dimension != B->dimension) {

printf("Размерности матриц не совпадают!\n");

return 0;

} else matrix\_dim = A->dimension;

if (out\_matrix) {

printf("Матрица А:\n");

print\_matrix(A);

printf("Матрица B:\n");

print\_matrix(B);

}

C = create\_matrix(matrix\_dim);

}

if (proc\_num > 1) solve\_parallel();

else solve\_stepped();

remove\_matrix(A);

remove\_matrix(B);

remove\_matrix(C);

MPI\_Finalize();

return 0;

}