**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №6**

**по дисциплине «Системы параллельной обработки данных»**

**Тема: «Умножение матриц»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9303 |  | Королёв С.Ю. |
| Преподаватель |  | Татаринов Ю.С. |

Санкт-Петербург

2023

**Цель работы**

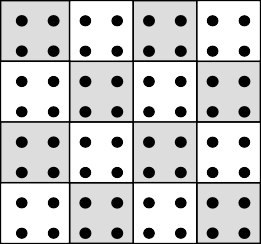
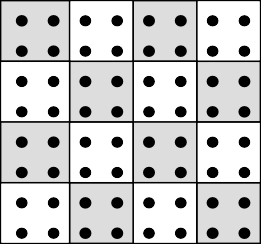
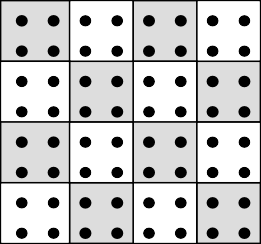
Базовое освоение и изучение алгоритмов умножения матриц в рамках библиотеки MPI.

**Формулировка задания**

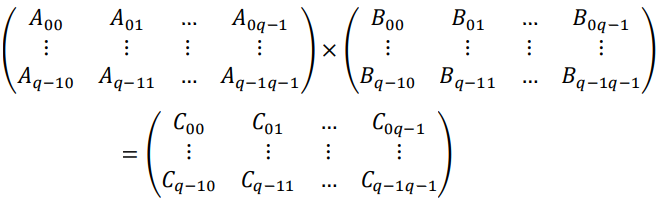
Выполнить задачу умножения двух квадратных матриц 𝐴 и 𝐵 размера 𝑚 × 𝑚, результат записать в матрицу 𝐶. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером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

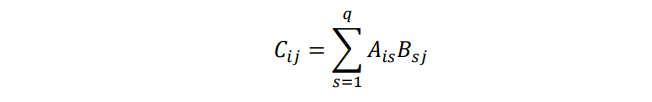
**Теоретические положения**

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).

***A B C***

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





**Выделение информационных зависимостей.**

* Подзадача (𝑖, 𝑗) отвечает за вычисление блока 𝐶𝑖𝑗, все подзадачи образуют прямоугольную решетку размером 𝑞 × 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 𝑨𝒊𝒋, 𝑩𝒊𝒋,
  + для каждой строки 𝑖 решетки подзадач блоки матрицы 𝑨 сдвигаются на

(𝑖 − 1) позиций влево,

* + для каждой строки 𝑗 решетки подзадач блоки матрицы 𝐵 сдвигаются на

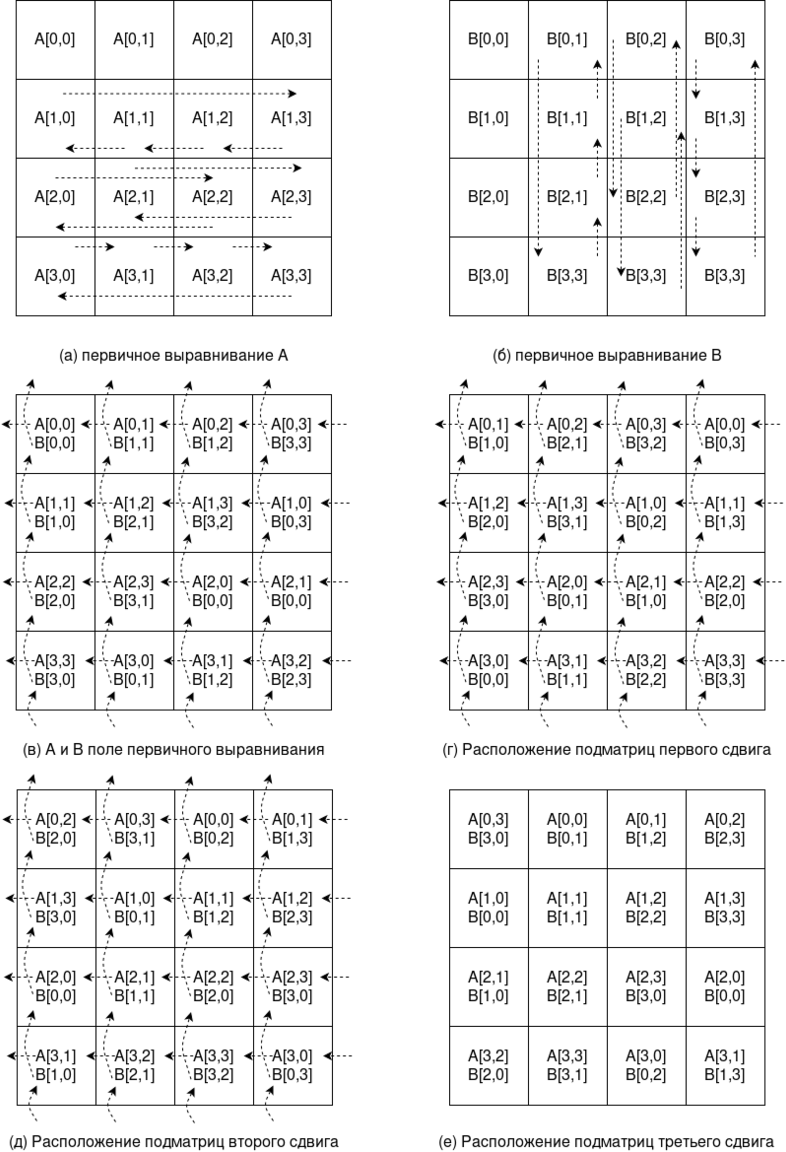
(𝑗 − 1) позиций вверх,

* 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.

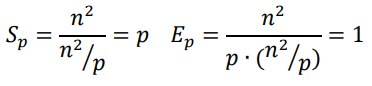
* Размер блоков может быть подобран таким образом, чтобы количество базовых подзадач совпадало с числом имеющихся процессоров,
* Множество имеющихся процессоров представляется в виде квадратной решетки и размещение базовых подзадач (𝑖, 𝑗) осуществляется на процессорах

𝒑𝒊,𝒋 (соответствующих узлов процессорной решетки).

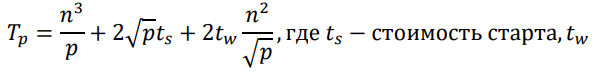


## Анализ эффективности.

* Общая оценка показателей ускорения и эффективности:



* Сложность:



− время передачи блок или обратная полоса пропускания

**Ход работы**

Для изучения как последовательной, так и параллельной реализации блочного алгоритма Кэннона были созданы скрипты на языке программирования С++ для запуска и обработки результатов запусков процессов, а также скрипт на языке программирования JavaScript для генерирования и записи в текстовый файл квадратных матриц заданной размерности.

Генерация квадратных матриц в рамках JavaScript-скрипта предполагает заполнение их ячеек случайно выбранными числами от 1 до 9. Скрипт приведен в приложении А.

**Листинг итоговой программы**

**Последовательная реализация**:

#include <iostream>

#include <fstream>

#include <vector>

#include <chrono>

using namespace std;

using namespace chrono;

// Функция для чтения матрицы из файла

vector<vector<int>> readMatrixFromFile(const string& fileName) {

ifstream file(fileName);

if (!file.is\_open()) {

cerr << "Unable to open file: " << fileName << endl;

exit(EXIT\_FAILURE);

}

vector<vector<int>> matrix;

int value;

while (file >> value) {

matrix.push\_back({ value });

char c;

while (file.get(c) && c != '\n' && c != '\r') {

file >> value;

matrix.back().push\_back(value);

}

}

file.close();

return matrix;

}

// Функция для записи матрицы в файл

void writeMatrixToFile(const string& fileName, const vector<vector<int>>& matrix) {

ofstream file(fileName);

if (!file.is\_open()) {

cerr << "Unable to open file: " << fileName << endl;

exit(EXIT\_FAILURE);

}

for (const auto& row : matrix) {

for (int value : row)

file << value << " ";

file << endl;

}

file.close();

}

// Функция для вывода матрицы в консоль

void printMatrix(const vector<vector<int>>& matrix) {

for (const auto& row : matrix) {

for (int value : row)

cout << value << " ";

cout << endl;

}

}

// Функция для перемножения двух матриц

vector<vector<int>> multiplyMatrices(const vector<vector<int>>& matrixA,

const vector<vector<int>>& matrixB) {

int size = matrixA.size();

vector<vector<int>> result(size, vector<int>(size, 0));

for (int i = 0; i < size; ++i)

for (int j = 0; j < size; ++j)

for (int k = 0; k < size; ++k)

result[i][j] += matrixA[i][k] \* matrixB[k][j];

return result;

}

int main() {

// Чтение матриц из файлов

vector<vector<int>> matrixA = readMatrixFromFile("A.txt");

vector<vector<int>> matrixB = readMatrixFromFile("B.txt");

// Начало отсчета времени

auto start = high\_resolution\_clock::now();

// Перемножение матриц

vector<vector<int>> resultMatrix = multiplyMatrices(matrixA, matrixB);

// Запись результата в файл

// writeMatrixToFile("C.txt", resultMatrix);

// Вывод результата в консоль

cout << "Matrix A:" << endl;

printMatrix(matrixA);

cout << "\nMatrix B:" << endl;

printMatrix(matrixB);

cout << "\nResult Matrix C:" << endl;

printMatrix(resultMatrix);

// Конец отсчета времени и расчёт длительности

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

// Вывод времени выполнения

cout << "Time taken by function: " << duration.count() / 1000000.0 << " seconds" << endl;

return 0;

}

**Параллельная реализация**:

// parallel\_lab.cpp : Defines the entry point for the console application.

#include <iostream>

#include <math.h>

#include "mpi.h"

// выделение памяти под матрицу

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;

}

// освобождение памяти от матрицы

int freeMatrix(int\*\*\* mat) {

free(&((\*mat)[0][0]));

free(\*mat);

return 0;

}

// умножение матрицы

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;

}

}

}

// вывод матрицы

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");

}

}

// запись матрицы в файл

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];

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 rank;

int worldSize;

int procDim;

int blockDim;

int left, right, up, down;

int bCastData[4];

double start, end; // для измерения времени

MPI\_Init(&argc, &argv);

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

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

start = MPI\_Wtime();

if (rank == 0) {

int n;

char ch;

// определение размерности матрицы

fp = fopen("A.txt", "r"); // открытие .txt-файла с матрицей на чтение

if (fp == NULL) {

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

while (fscanf(fp, "%d", &n) != EOF) { // подсчёт числа строк

ch = fgetc(fp);

if (ch == '\n') {

rows = rows + 1;

}

count++;

}

columns = count / rows; // подсчёт числа столбцов

// проверка матрицы и числа процессов

if (columns != rows) { // проверка на квадратную матрицу

printf("[ERROR] Matrix must be square!\n");

MPI\_Abort(MPI\_COMM\_WORLD, 2);

}

double sqroot = sqrt(worldSize); // проверка, является ли число процессов квадратом целого числа

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; // проверка на делимость строк и столбцов матрицы на корень числа процессов

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; // размерность сети процессов

blockDim = columns / intRoot; // размер блока

fseek(fp, 0, SEEK\_SET); // переустановка указателя на файл в начало, чтобы повторно прочитать содержимое

// выделение памяти под матрицы А и В

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);

}

// чтение матрицы А и заполнение ячеек

for (int i = 0; i < rows; i++) {

for (int j = 0; j < columns; j++) {

fscanf(fp, "%d", &n);

A[i][j] = n;

}

}

// printf("A matrix:\n"); // вывод в консоль матрицы А

// printMatrix(A, rows);

fclose(fp); // закрытие файла

// чтение матрицы В и заполнение ячеек

fp = fopen("B.txt", "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;

}

}

// printf("B matrix:\n"); // вывод в консоль матрицы В

// printMatrix(B, rows);

fclose(fp); // закрытие файла

// выделение памяти под матрицу С

if (allocMatrix(&C, rows, columns) != 0) {

printf("[ERROR] Matrix alloc for C failed!\n");

MPI\_Abort(MPI\_COMM\_WORLD, 6);

}

// заполняем массив bCastData

bCastData[0] = procDim; // размерность сети процессов

bCastData[1] = blockDim; // размер блока

bCastData[2] = rows; // строки

bCastData[3] = columns; // столбцы

}

// создаём двумерную решетку процессов

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);

// выделяем памяти под локальные блоки матриц А и В

allocMatrix(&localA, blockDim, blockDim);

allocMatrix(&localB, blockDim, blockDim);

// создаём тип данных для описания подмассивов в глобальном массиве

int globalSize[2] = { rows, columns }; // массив с размерами всей матрицы

int localSize[2] = { blockDim, blockDim }; // массив с размерами блока

int starts[2] = { 0,0 }; // начальные индексы блока

MPI\_Datatype type, subarrtype;

MPI\_Type\_create\_subarray(2, globalSize, localSize, starts, MPI\_ORDER\_C, MPI\_INT, &type); // тип данных блока матрицы

MPI\_Type\_create\_resized(type, 0, blockDim \* sizeof(int), &subarrtype); // тип данных чтобы учесть смещение блока

MPI\_Type\_commit(&subarrtype); // фиксируем созданный тип данных

// создаем указатели для доступа к глобальным матрицам (только для процесса с рангом 0)

int\* globalptrA = NULL;

int\* globalptrB = NULL;

int\* globalptrC = NULL;

if (rank == 0) {

globalptrA = &(A[0][0]);

globalptrB = &(B[0][0]);

globalptrC = &(C[0][0]);

}

// распределение данных глобальных матриц A и B по локальным блокам на каждом процессе

// выделение памяти для массивов

int\* sendCounts = (int\*)malloc(sizeof(int) \* worldSize); // число элементов, отправляемых каждому процессу

int\* displacements = (int\*)malloc(sizeof(int) \* worldSize); // смещение данных, начиная с которого отправляются данные каждому процессу

if (rank == 0) {

for (int i = 0; i < worldSize; i++) {

sendCounts[i] = 1; // sendCounts = 1 для каждого процесса

}

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;

}

}

// рассылаем данные матриц A и B по локальным блокам localA и localB каждого процесса

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);

// выделение памяти для локальной матрицы C

if (allocMatrix(&localC, blockDim, blockDim) != 0) {

printf("[ERROR] Matrix alloc for localC in rank %d failed!\n", rank);

MPI\_Abort(MPI\_COMM\_WORLD, 7);

}

// изначальный сдвиг

MPI\_Cart\_coords(cartComm, rank, 2, coord); // получение координат текущего процесса в массив coords

MPI\_Cart\_shift(cartComm, 1, coord[0], &left, &right); // вычисление соседей по оси Х (слева и справа)

MPI\_Sendrecv\_replace(&(localA[0][0]), blockDim \* blockDim, MPI\_INT, left, 1, right, 1, cartComm, MPI\_STATUS\_IGNORE); // обмен данными между процессами вдоль оси x

MPI\_Cart\_shift(cartComm, 0, coord[1], &up, &down); // вычисление соседей по оси Y (сверху и снизу)

MPI\_Sendrecv\_replace(&(localB[0][0]), blockDim \* blockDim, MPI\_INT, up, 1, down, 1, cartComm, MPI\_STATUS\_IGNORE); // обмен данными между процессами вдоль оси y:

// инициализация блока в матрице С

for (int i = 0; i < blockDim; i++) {

for (int j = 0; j < blockDim; j++) {

localC[i][j] = 0;

}

}

// выделение памяти для временной матрицы multiplyRes:

int\*\* multiplyRes = NULL; // multiplyRes - временное хранение промежуточных результатов умножения

if (allocMatrix(&multiplyRes, blockDim, blockDim) != 0) {

printf("[ERROR] Matrix alloc for multiplyRes in rank %d failed!\n", rank);

MPI\_Abort(MPI\_COMM\_WORLD, 8);

}

// перемножение блоков localA и localB и обновление localC

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];

}

}

// 1 раз: сдвиг блока матрицы А (влево) и блока матрицы В (вверх)

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);

}

// сбор результатов

MPI\_Gatherv(&(localC[0][0]), rows \* columns / worldSize, MPI\_INT,

globalptrC, sendCounts, displacements, subarrtype,

0, MPI\_COMM\_WORLD);

// освобождение памяти локальных матриц

freeMatrix(&localC);

freeMatrix(&multiplyRes);

// работа с результатами

if (rank == 0) {

// вывести результаты в консоли

// printf("C is:\n");

// printMatrix(C, rows);

// сохранить результаты в файл output.txt

FILE\* output\_file = fopen("output.txt", "w");

if (output\_file == NULL) {

fprintf(stderr, "Error opening output file.\n");

MPI\_Abort(MPI\_COMM\_WORLD, 9);

}

printMatrixFile(C, rows, output\_file);

fclose(output\_file);

// вывод времени выполнения в консоль

end = MPI\_Wtime();

printf("Time: %.4fs\n", end - start); // точность времени: до 4 чисел после запятой

}

MPI\_Finalize();

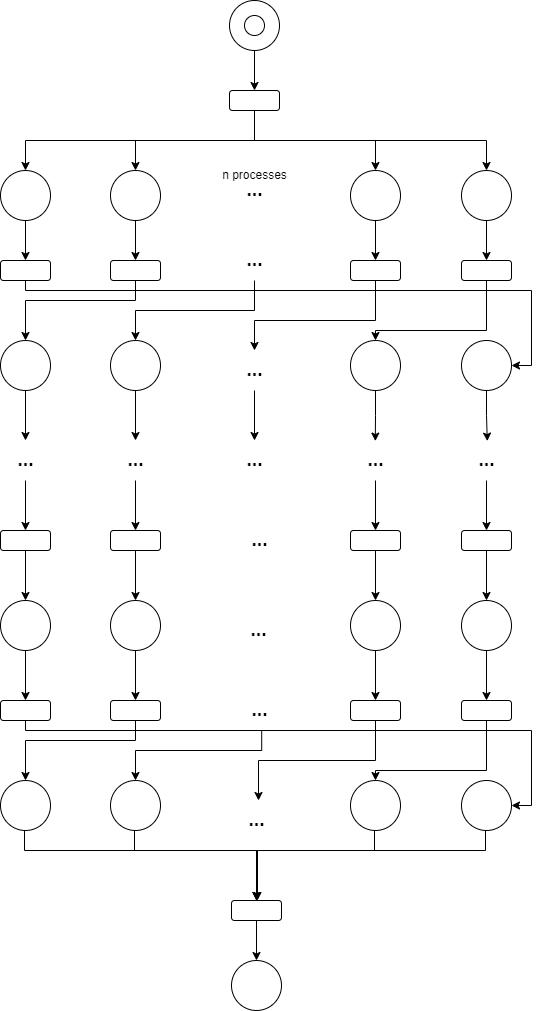
return 0;

}

**Краткое описание выбранного алгоритма решения**

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 0 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).

**Формальное описание выбранного алгоритма решения с использованием аппарата сетей Петри**



**Технические параметры**

Запуск программ производился на ОС Windows 11 и процессоре 12th Gen Intel(R) Core(TM) i7-12650H (базовая тактовая частота 2.7 GHz) c 10 ядрами CPU и 16 потоками.

**Анализ эффективности**

Во время тестирования программы были проведены эксперименты по измерению времени выполнения программы в зависимости от количества инициализированных процессов, а также вычислено ускорение.

В реализованном алгоритме присутствуют некоторые ограничения, связанные с техническими параметрами ПК и самой реализацией параллельного алгоритма. Таким образом, ограничения следующие:

- число процессов должно быть квадратом целого числа;

- количество строк и столбцов матрицы должно нацело делиться на корень числа процессов.

Ограничения были обработаны в программной реализации параллельного алгоритма. Для удобства тестирования и измерения результатов экспериментах в рамках указанных ограничений для выбора размерности матриц и числа процессов использовались степени двойки.

Таблица 1. Результаты вычислительных экспериментов.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Размерность матриц  (m) | Последова-тельный алгоритм | 4 процесса | | 16 процессов | | 64 процесса | |
|  | время (с.) | время | ускор. | время | ускор. | время | ускор. |
| 4 | 0.00031 | 0.0012 | 0.2583 | 0.0043 | 0.0721 | - | - |
| 16 | 0.00163 | 0.0016 | 1.0187 | 0.0050 | 0.3260 | 0.0592 | 0.0275 |
| 64 | 0.01068 | 0.0080 | 1.3350 | 0.0125 | 0.8544 | 0.0681 | 0.1568 |
| 256 | 0.36860 | 0.0954 | 3.8637 | 0.1074 | 3.4320 | 0.2249 | 1.6389 |
| 1024 | 22.3107 | 1.8970 | 11.7610 | 1.8501 | 12.0591 | 1.9554 | 11.4097 |
| 4096 | 1420.52 | 95.546 | 14.8673 | 105.474 | 13.4679 | 70.773 | 20.0714 |

Замечание к таблице 1: для матрицы размерности 4х4 запуск с 64 процессорами выведет ошибку из-за ограничения о том, что количество строк и столбцов матрицы не делится нацело на корень числа процессов.

Обобщенные результаты представлены на графике ниже.

Таблица 2. Результаты сравнения экспериментальных и теоретических оценок времени выполнения на 4 процессах.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размерность матриц (m)** | **Последовательный алгоритм** | **4 процесса** | | |
|  | **время (сек.)** | **время теор. (сек.)** | **время (сек.)** | **эффективность** |
| 4 | 0.00031 | 0.0000775 | 0.0012 | 0.06458 |
| 16 | 0.00163 | 0.0004075 | 0.0016 | 0.25469 |
| 64 | 0.01068 | 0.00267 | 0.0080 | 0.33375 |
| 256 | 0.36860 | 0.09215 | 0.0954 | 0.96593 |
| 1024 | 22.3107 | 5.577675 | 1.8970 | 2.94026 |
| 4096 | 1420.52 | 355.13 | 95.546 | 3.71685 |

**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. Был реализован блочный алгоритм Кэннона, который на экспериментальных данных показал рост производительности при грамотном распараллеливании задачи. Кроме того, были сделаны следующие выводы:

1. Эффективность:

- Параллельный алгоритм эффективен для больших размеров исходных данных при достаточном числе процессов. При прочих равных условиях увеличение числа процессов должно приводить к ускорению выполнения.

- Сложность последовательного алгоритма по времени – O(n3). Он малоэффективен на большом объёме исходных данных.

1. Масштабируемость:

- Параллельный алгоритм хорошо масштабируются, потому что разделяет вычисления между всеми запущенными процессами. Это означает, что они могут эффективно использовать ресурсы при увеличении числа процессоров.

- Последовательный алгоритм слабо масштабируется, так как он не может использовать множество процессоров.

1. Объем памяти:

- Параллельный алгоритм может требовать дополнительного объема памяти для хранения блоков данных, что может быть проблемой для больших матриц.

- Последовательный алгоритм требует меньшего объема памяти, потому что работает над элементами поочередно.

1. Время:

- Параллельный алгоритм из-за разделения задачи на подзадачи и использования множества числа процессов существенно сокращает время вычислений; так, было затрачено всего 70 секунд для умножения матриц 4096х4096.

- Последовательный алгоритм из-за ограниченного объема памяти работает значительно дольше на большом количестве данных; так, было затрачено 24 минуты для умножения матриц размерности 4096х4096.

Нужно заметить, что для маленького объёма исходных данных и большого числа процессов параллельный алгоритм уступает по времени выполнения последовательному; видимо, это связано с тем, что время затрачивается в том числе на создание и завершение самих процессов внутри программы. Впрочем, в практическом смысле разница по времени зачастую не такая существенная, всего около 0.001 с.

**ПРИЛОЖЕНИЕ А.**

**ИСХОДНЫЙ JAVASCRIPT-КОД ПРОГРАММЫ ДЛЯ СОЗДАНИЯ .TXT-ФАЙЛА С КВАДРАТНОЙ МАТРИЦЕЙ.**

import fs from 'fs/promises';

const generateSquareMatrix = () => {

const matrix = [];

const maxNumber = 9; // Максимальное значение (включительно)

const size = 10; // Размерность матрицы

for (let i = 0; i < size; i++) {

const row = [];

for (let j = 0; j < size; j++) {

const randomNum = Math.floor(Math.random() \* maxNumber) + 1; // Генерируем случайное целое число от 1 до maxNumber

row.push(randomNum);

}

matrix.push(row);

}

return matrix;

};

const matrixToString = (matrix) => {

return matrix.map((row) => row.join(' ')).join('\n');

};

const randomMatrix = generateSquareMatrix();

const matrixString = matrixToString(randomMatrix);

// console.log(matrixString);

fs.writeFile(`A\_10.txt`, matrixString);