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

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Параллельные алгоритмы»
Тема: Основы работы с процессами и потоками

Студентка гр. 0303 _____ Костебелова Е. К.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2023

Цель работы.

Изучение и практическое применение работы с процессами и потоками на языке C++.

Задание.

Выполнить умножение 2х матриц:

1)Входные матрицы вводятся из файла (или генерируются).

2)Результат записывается в файл.

1.1. Выполнить задачу, разбив её на 3 процесса. Выбрать механизм обмена данными между процессами.

1) Процесс 1: заполняет данными входные матрицы (читает из файла или генерирует их некоторым образом).

2) Процесс 2: выполняет умножение

3) Процесс 3: выводит результат

1.2.1 Аналогично 1.1, используя потоки (std::threads)

1.2.2 Разбить умножение на P потоков (“наивным” способом по строкам-столбцам).

Исследовать зависимость между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы. Сформулировать ограничения.

Выполнение работы.

Создание матрицы происходит с помощью функции:

void generateMatrices () – Функция, которая реализовывает генерацию двух матриц размера MATRIX_SIZE случайными числами от 0 до 10. Входные параметры меняются в зависимости от рассматриваемой задачи.

Умножение матриц происходит с помощью функции:

void multiplyMatrices () – Функция, которая реализовывает умножение двух матриц размера MATRIX_SIZE, а также записывает результат умножения в

новую результирующую матрицу. Входные параметры меняются в зависимости от рассматриваемой задачи.

Умножение матриц для задачи 1.2.2 происходит с помощью функции:

void multiplyMatrices(const std::vector<std::vector<int>>& matrix1, const std::vector<std::vector<int>>& matrix2, std::vector<std::vector<int>>& result, int start, int end) – на вход подаются ссылки на 3 матрицы: matrix1 и matrix2 – матрицы, которые перемножаются, result – результирующая матрица, а также индексы start – начала и end – конца строк, которые будут перемножаться во время выполнения одного потока.

Запись результирующей матрицы осуществляется при помощи функции:

void writeToFile () – в функцию передаётся результирующая матрица, после чего происходит её запись в файл (имя файла задаётся в виде константы name)

1.1 Реализация с помощью процессов.

Умножение двух матриц при помощи процессов реализовано в файле main.cpp в папке Processes.

В качестве механизма обмена данными между процессами был выбран способ общения через файлы.

Разбиение на 3 процесса происходит при помощи функции fork() из библиотеки unistd.h, которая создаёт процессы. Первый процесс реализовывает генерацию случайных чисел в файл для соответствующей матрицы. Второй процесс считывает значения матриц из файлов, производит умножение, а затем записывает результирующую матрицу в файл. Третий процесс считывает значения результирующей матрицы и выводит их в консоль.

1.2.1 Реализация с помощью потоков.

Умножение двух матриц при помощи потоков реализовано в файле main.cpp в папке Threads.

Разбиение на потоки происходит при помощи создания конструктора класса thread из библиотеки thread, которая принимает необходимую функцию и её аргументы. Ожидание исполнения потока происходит при помощи метода

join(). Первый поток генерирует матрицы и передаёт их с помощью std::promise и std::future. Второй поток получает на вход две сгенерированные матрицы, производит умножение, а затем записывает результирующую матрицу и передаёт её с помощью std::promise и std::future. Третий поток получает на вход результирующую матрицу и записывает её в файл (имя файла задаётся в виде константы name).

1.2.1 Реализация умножения матриц с помощью p потоков.

Умножение двух матриц при помощи p потоков реализовано в файле main.cpp в папке PThreads.

Создаётся P потоков, затем идёт распределение вычислений по строкам матрицы между потоками и выполняется умножение элементов матрицы в каждом потоке. Каждый поток работает с определенными строками матрицы, это реализовано с помощью переменных start, end и step. В конце мы дожидаемся завершения работы всех потоков и выводим результат.

1.3 Исследовать зависимость между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы.

Исследуем зависимость времени работы программы от количества потоков. Для этого возьмём постоянный размер матрицы равный 1000x1000. Результат зависимости времени умножения матриц от количества потоков представлен в табл. 1.

Таблица 1 — Зависимость времени работы программы от количества потоков

Количество потоков P	Время работы программы, ms
1	11046
5	3251
25	2460
100	2511
500	2567
1000	2860

Исходя из результатов таблицы, можно сделать вывод, что увеличение количества потоков ускоряет выполнение работы, однако с определенного момента увеличение количества потоков мало влияет на время выполнения программы.

Исследуем зависимость времени работы программы от размера входных данных, при количестве потоков равное 1. Результат зависимости времени умножения матриц от количества входных данных представлен в табл. 2.

Таблица 2 — Зависимость работы программы от входных данных

Размер входной матрицы	Время работы программы, ms
100x100	14
200x200	76
500x500	1064
1000x1000	11046

Исходя из результатов таблицы, можно сделать вывод, что увеличение размерности матрицы замедляет выполнение работы программы.

Выводы.

В процессе выполнения лабораторной работы были изучены и практически применены работы с процессами и потоками на языке C++. Были реализованы программы, соответствующие поставленным задачам, а именно:

1) Разбиение чтения, умножения и вывода матриц на 3 процесса

2) Разбиение чтения, умножения и вывода матриц на 3 потока

3) Разбиение чтения, умножения и вывода матриц на p потоков

При исследовании зависимостей времени работы от количества потоков, размера входных данных и параметров целевой вычислительной системы, было выявлено:

1) Увеличение количества потоков уменьшает время работы программы, но с определенного момента происходит "насыщение" и время работы программы не сильно уменьшается.

2) Увеличение количества элементов матрицы приводит к увеличению времени работы программы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: *main.cpp (Processes)*

```
#include <iostream>
#include <random>
#include <string>
#include <sys/wait.h>
#include <unistd.h>
#include <fstream>

const int MATRIX_SIZE = 4;

// Функция для генерации случайных входных матриц
void generateMatrices(const char* filename_matrix1, const char*
filename_matrix2) {
    std::ofstream file1(filename_matrix1);
    if (!file1.is_open()) {
        std::cerr << "Error opening file " << filename_matrix1 <<
std::endl;
        return;
    }

    // Генерация случайных значений для матрицы A
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            int valueA = rand() % 10;    // Значение матрицы A от 0 до 9
            file1 << valueA <<  " ";
        }
        file1 << std::endl;
    }

    file1.close();

    std::ofstream file2(filename_matrix2);
    if (!file2.is_open()) {
        std::cerr << "Error opening file " << filename_matrix2 <<
std::endl;
        return;
    }

    // Генерация случайных значений для матрицы B
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            int valueB = rand() % 10;    // Значение матрицы B от 0 до 9
            file2 << valueB << " ";
        }
        file2 << std::endl;
    }

    file2.close();

    std::cout << "Input matrices are generated and written to a file" <<
std::endl;
}
```

```
// Функция для умножения матриц
void multiplyMatrices(const char* filename_matrix1, const char*
filename_matrix2, const char* filename_result) {

    int matrixA[MATRIX_SIZE][MATRIX_SIZE];
    int matrixB[MATRIX_SIZE][MATRIX_SIZE];

    std::ifstream inputFile_matrix1(filename_matrix1, std::ios::in);
    if (!inputFile_matrix1.is_open()) {
        std::cerr << "Error opening file " << filename_matrix1 <<
std::endl;
        return;
    }
    std::ifstream inputFile_matrix2(filename_matrix2, std::ios::in);
    if (!inputFile_matrix2.is_open()) {
        std::cerr << "Error opening file " << filename_matrix2 <<
std::endl;
        return;
    }


    // Загрузка данных из файлов в матрицы A и B
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            inputFile_matrix1 >> matrixA[i][j];
            inputFile_matrix2 >> matrixB[i][j];
        }
    }

    inputFile_matrix1.close();
    inputFile_matrix2.close();

    int matrixC[MATRIX_SIZE][MATRIX_SIZE];

    // Умножение матриц A и B
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            matrixC[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; ++k) {
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }

    std::ofstream outputFile(filename_result);
    if (!outputFile.is_open()) {
        std::cerr << "Error opening file " << filename_result <<
std::endl;
        return;
    }

    // Запись результирующей матрицы C в файл
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            outputFile << matrixC[i][j] << " ";
        }
        outputFile << std::endl;
    }
```

```
    outputFile.close();
    std::cout << "Matrices multiplied and written to file" << std::endl;
}

void writeToConsole(const char* file) {
    std::ifstream inputFile(file);
    if (!inputFile.is_open()) {
        std::cerr << "Error opening file " << file << std::endl;
    }
    else {
        std::string line;
    while (std::getline(inputFile, line)) {
        std::cout << line << std::endl;
    }

    inputFile.close();
    }
}

int main() {
    const char* filename_matrix1 = "matrix_A.txt";
    const char* filename_matrix2 = "matrix_B.txt";
    const char* filename_result = "result_matrix.txt";

    pid_t gen_pid = fork();
    switch(gen_pid){
        case 0:
            generateMatrices(filename_matrix1, filename_matrix2);
            exit(0);
        default:
            wait(&gen_pid);
    }

    pid_t mult_pid = fork();
    switch(mult_pid){
        case 0:
            multiplyMatrices(filename_matrix1, filename_matrix2,
filename_result);
            exit(0);
        default:
            wait(&mult_pid);
    }

    pid_t write_pid = fork();
    switch(write_pid){
        case 0:
            writeToConsole(filename_result);
            exit(0);
        default:
            wait(&write_pid);
    }

    return 0;
}
```

Название файла: *main.cpp (Threads)*

```
#include <iostream>
#include <vector>
#include <thread>
#include <future>
#include <numeric>
#include <chrono>
#include <random>
#include <fstream>
#include <string>
const int MATRIX_SIZE = 10;
const char* name =
"C:/Users/liza/Desktop/PA_Lab1_Kostebelova_0303_Threads/result.txt";
//const char* name = "result.txt";
//using namespace std;

void generateMatrices(std::promise<int>
matrixA_promise[MATRIX_SIZE][MATRIX_SIZE], std::promise<int>
matrixB_promise[MATRIX_SIZE][MATRIX_SIZE])
{
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            //random_number = firs_value + rand() % last_value;
            matrixA_promise[i][j].set_value(1 + rand() % 10);
        }
    }
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixB_promise[i][j].set_value(1 + rand() % 10);
        }
    }
}

void multiplyMatrices(int matrixA[MATRIX_SIZE][MATRIX_SIZE], int
matrixB[MATRIX_SIZE][MATRIX_SIZE], std::promise<int>
matrixC_promise[MATRIX_SIZE][MATRIX_SIZE])
{
    int count = 0;
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            count = 0;
            for (int k = 0; k < MATRIX_SIZE; k++) {
                count += matrixA[i][k] * matrixB[k][j];
            }
            matrixC_promise[i][j].set_value(count);
        }
    }

}

void writeToFile(int res[MATRIX_SIZE][MATRIX_SIZE]) {
//    std::cout << "NOW" << std::endl;
    std::ofstream myfile;
    myfile.open(name);
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
```

```
            myfile << res[i][j] << ' ';
        }
        myfile << std::endl;
    }
    myfile.close();
}

int main()
{
    int matrixA[MATRIX_SIZE][MATRIX_SIZE];
    int matrixB[MATRIX_SIZE][MATRIX_SIZE];
    int matrixC[MATRIX_SIZE][MATRIX_SIZE];

    std::promise<int> matrixA_promise[MATRIX_SIZE][MATRIX_SIZE];
    std::future<int> matrixA_future[MATRIX_SIZE][MATRIX_SIZE];
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixA_future[i][j] = matrixA_promise[i][j].get_future();
        }
    }

    std::promise<int> matrixB_promise[MATRIX_SIZE][MATRIX_SIZE];
    std::future<int> matrixB_future[MATRIX_SIZE][MATRIX_SIZE];
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixB_future[i][j] = matrixB_promise[i][j].get_future();
        }
    }

    std::thread work_thread(generateMatrices, std::move(matrixA_promise),
std::move(matrixB_promise));

    std::cout << "First Matrix" << std::endl;
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixA[i][j] = matrixA_future[i][j].get();
            std::cout << matrixA[i][j] << ' ';
        }
        std::cout << std::endl;
    }
    std::cout << "Second Matrix" << std::endl;
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixB[i][j] = matrixB_future[i][j].get();
            std::cout << matrixB[i][j] << ' ';
        }
        std::cout << std::endl;
    }

    std::promise<int> matrixC_promise[MATRIX_SIZE][MATRIX_SIZE];
    std::future<int> matrixC_future[MATRIX_SIZE][MATRIX_SIZE];
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixC_future[i][j] = matrixC_promise[i][j].get_future();
        }
    }
    std::thread work_thread2(multiplyMatrices, matrixA, matrixB,
std::move(matrixC_promise));
```

```
    std::cout << "RESULT Matrix in file result.txt" << std::endl;
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            matrixC[i][j] = matrixC_future[i][j].get();
            std::cout << matrixC[i][j] << ' ';
        }
        std::cout << std::endl;
    }


    std::thread work_thread3(writeToFile, matrixC);


    work_thread.join();
    work_thread2.join();
    work_thread3.join();
}
```

Название файла: *main.cpp (PThreads)*

```
#include <iostream>
#include <vector>
#include <thread>
#include <random>
#include <fstream>
#include <string>
#include <chrono>
//#define printToConsole
const char* name =
"C:/Users/liza/Desktop/PA_Lab1_Kostebelova_0303_P_Threads/result.txt";
//const char* name = "result.txt";
const int MATRIX_SIZE = 1000;
const int THREADS_NUM = 1;

void printMatrix(std::vector<std::vector<int>>& matrix, int row, int col)
{
    for (int i=0; i<row; i++) {
        for (int j=0; j<col; j++) {
            std::cout << matrix.at(i).at(j) << " ";
        }
        std::cout << std::endl;
    }
}

void generateMatrices(std::vector<std::vector<int>>& matrix1,
std::vector<std::vector<int>>& matrix2)
{
    //fill vecM1
    std::vector<int> vec_for_matrix;
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            vec_for_matrix.push_back(1 + rand() % 10);
        }
        matrix1.push_back(vec_for_matrix);
        vec_for_matrix.clear();
    }
```

```
    //fill copy vecM2
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            vec_for_matrix.push_back(1 + rand() % 10);
        }
        matrix2.push_back(vec_for_matrix);
        vec_for_matrix.clear();
    }
#ifdef printToConsole
    std::cout << "First Matrix" << std::endl;
    printMatrix(matrix1, MATRIX_SIZE, MATRIX_SIZE);
    std::cout << "Second Matrix" << std::endl;
    printMatrix(matrix2, MATRIX_SIZE, MATRIX_SIZE);
#endif

}


// Функция, которую будут выполнять потоки для вычисления умножения
элементов матрицы
void multiplyMatrices(const std::vector<std::vector<int>>& matrix1, const
std::vector<std::vector<int>>& matrix2,
                      std::vector<std::vector<int>>& result, int start,
int end) {
    for (int i = start; i < end; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            int sum = 0;
            for (int k = 0; k < MATRIX_SIZE; ++k) {
                sum += matrix1[i][k] * matrix2[k][j];
            }
            result[i][j] = sum;
        }
    }
}

void writeToFile(std::vector<std::vector<int>>& matrix) {
  std::ofstream file(name);

  if (file.is_open()) {
    for (const auto& row : matrix) {
      for (const auto& element : row) {
        file << element << " ";
      }
      file << std::endl;
    }

    file.close();
    std::cout << "Done recording" << std::endl;
  } else {
    std::cout << "Error while recording" << std::endl;
  }
}

int main() {

    // Измерение времени начала выполнения программы
    auto start_time = std::chrono::high_resolution_clock::now();
```

```
    std::vector<std::vector<int>> matrix1;
    std::vector<std::vector<int>> matrix2;
    std::vector<std::vector<int>> result(MATRIX_SIZE,
std::vector<int>(MATRIX_SIZE, 0));
    std::thread thread_fill_matrix(generateMatrices, std::ref(matrix1),
std::ref(matrix2));

    thread_fill_matrix.join();

    std::vector<std::thread> threads;

    int step = MATRIX_SIZE / THREADS_NUM;
    int start = 0;
    int end = step;

    // Создаем и запускаем потоки для вычислений
    for (int i = 0; i < THREADS_NUM; ++i) {
        if (i == THREADS_NUM - 1) {
            // Последний поток берет оставшуюся часть
            end = MATRIX_SIZE;
        }
        threads.emplace_back(multiplyMatrices, std::ref(matrix1),
std::ref(matrix2), std::ref(result), start, end);
        start = end;
        end += step;
    }

    // Дожидаемся окончания работы всех потоков
    for (auto& thread : threads) {
        thread.join();
    }

    // Выводим результат
#ifdef printToConsole
    std::cout << "Result Matrix" << std::endl;
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            std::cout << result[i][j] << " ";
        }
        std::cout << std::endl;
    }
#endif

    writeToFile(result);

    // Измерение времени окончания выполнения программы
    auto end_time = std::chrono::high_resolution_clock::now();

    // Вычисление общего времени выполнения программы в миллисекундах
    auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
start_time).count();

    std::cout << "Program duration " << duration << " ms" << std::endl;

    return 0;
}
```