Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Отчет по лабораторной работе №2 дисциплины «Параллельное программирование»

Выполнил студент группы ИВТ-31_____/Крючков И. С/
Проверил_____/Долженкова М. Л./

Киров 2023

1. Задание
    1) Выполнить разбиение исследованного в ходе первой лабораторной работы алгоритма на независимо выполняемые фрагменты
    2) Реализовать многопоточную версию алгоритм с помощью языка C++ и потоков операционной системы, используя при этом необходимые примитивы синхронизации
    3) Показать корректность полученной реализации путем осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов
    4) Провести доказательную оценку эффективности многопоточной реализации алгоритма.

2. Метод распараллеливания алгоритма

Для вычисления результата Штрассен предложил алгоритм с семью умножениями:

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$P_2 = (A_{21} + A_{22})B_{11}$

$P_3 = A_{11}(B_{12} - B_{22})$

$P_4 = A_{22}(B_{21} - B_{11})$

$P_5 = (A_{11} + A_{12})B_{22}$

$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$

$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Вычисление значения каждого P_i выполняется независимо, поэтому их вычисление можно ускорить за счет выполнения в несколько потоков.

Получение матрицы результата:

$C_{11} = P_1 + P_4 - P_5 + P_7$

$C_{12} = P_3 + P_5$

$C_{21} = P_2 + P_4$

$C_{22} = P_1 + P_3 - P_2 + P_6$

Вычисления каждой подматрицы результата можно разбить на независимые части и выполнять в отдельных потоках:

$Q_1 = P_1 + P_4$

$Q_2 = P_2 + P_4$

$Q_3 = P_3 + P_6$

$Q_4 = P_7 - P_5$

$Q_5 = P_3 + P_5$

$Q_6 = P_1 - P_2$

3. Программная реализация

Листинг программной реализации приведен в приложении А.

4. Тестирование

При тестировании выполнялось умножение квадратных матриц, сгенерированных случайным образом.

Тестирование выполнялось на ОС Windows 10 x64, с процессором Intel Xeon E5-1620v3 с частотой 3.5 ГГц (4 физических, 8 логических ядер), 16 Гб ОЗУ.

Результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты тестирования.

№	Размер матриц	Линейный алгоритм, с	Параллельный алгоритм, с
1	256	0.01	0.005
2	512	0.072	0.025
3	1024	0.51	0.153
4	2048	3.562	0.998
5	4096	24.81	6.745
6	8192	176.352	46.397

5. Вывод

В ходе выполнения лабораторной работы была разработана многопоточная версия алгоритма умножения матриц методом Штрассена с использованием потоков стандартной библиотеки C++. Многопоточный алгоритм оказался быстрее однопоточного на всех тестовых входных данных. Исходя из этого можно предположить, что многопоточная реализация будет быстрее при любых входных данных.

Приложение А.

Листинг программной реализации

main.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <cstring>
#include "thread_pool.h"

Lab::thread_pool pool;

int** newMatrix(int64_t n) {
    int** arr = new int* [n];

    for (int64_t i = 0; i < n; ++i) {
        arr[i] = new int[n];
    }

    return arr;
}


void deleteMatrix(int** m, int64_t n) {
    for (int64_t i = 0; i < n; ++i) {
        delete[] m[i];
    }

    delete[] m;
}


int** read_matrix(std::ifstream &in, int64_t n, int64_t real_n) {
    int** m = newMatrix(n);

    for (int64_t i = 0; i < real_n; ++i) {
        memset(m[i], 0, n * sizeof *m[i]);
        for (int64_t j = 0; j < real_n; ++j) {
            in >> m[i][j];
        }
    }

    return m;
}


int** matrix_multiply(int** a, int** b, int n) {
    int** result = newMatrix(n);

    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            result[i][j] = 0;
            for (k = 0; k < n; k++)
                result[i][j] += a[i][k] * b[k][j];
        }
    }

    return result;
}

int** addMatrix(int** a, int** b, int64_t n) {
    int** result = newMatrix(n);

    for (int64_t i = 0; i < n; ++i) {
        for (int64_t j = 0; j < n; ++j) {
```

```
            result[i][j] = a[i][j] + b[i][j];
        }
    }

    return result;
}

int** subMatrix(int** a, int** b, int64_t n) {
    int** result = newMatrix(n);
    for (int64_t i = 0; i < n; ++i) {
        for (int64_t j = 0; j < n; ++j) {
            result[i][j] = a[i][j] - b[i][j];
        }
    }

    return result;
}

int** getSlice(int** m, int oi, int oj, int64_t n) {
    int** matrix = newMatrix(n);

    for (int64_t i = 0; i < n; ++i) {
        for (int64_t j = 0; j < n; ++j) {
            matrix[i][j] = m[i+oi][j+oj];
        }
    }

    return matrix;
}


int** combMatrix(int** c11, int** c12, int** c21, int** c22, int64_t n) {
    int64_t m = n*2;

    int** result = newMatrix(m);

    for (int64_t i = 0; i < m; ++i) {
        for (int64_t j = 0; j < m; ++j) {
            if (i < n && j < n) {
                result[i][j] = c11[i][j];
            } else if (i < n) {
                result[i][j] = c12[i][j-n];
            } else if (j < n) {
                result[i][j] = c21[i-n][j];
            } else {
                result[i][j] = c22[i-n][j-n];
            }
        }
    }
    return result;
}

int64_t new_size(int64_t n) {
    int64_t r = 1;
    while((n >>= 1) != 0) {
        r++;
    }
    return 1 << r;
}

bool isPowerOfTwo(int64_t v) {
    return v && !(v & (v - 1));
}

int** strassen(int**a, int**b, int64_t n, int depth) {
    if (n <= 64) {
        return matrix_multiply(a, b, n);
    } else {
```

```
n = n >> 1;

int** a11 = getSlice(a, 0, 0, n);
int** a12 = getSlice(a, 0, n, n);
int** a21 = getSlice(a, n, 0, n);
int** a22 = getSlice(a, n, n, n);
int** b11 = getSlice(b, 0, 0, n);
int** b12 = getSlice(b, 0, n, n);
int** b21 = getSlice(b, n, 0, n);
int** b22 = getSlice(b, n, n, n);

int** p1;
int** p2;
int** p3;
int** p4;
int** p5;
int** p6;
int** p7;

std::future<int**> p1_f;
std::future<int**> p2_f;
std::future<int**> p3_f;
std::future<int**> p4_f;
std::future<int**> p5_f;
std::future<int**> p6_f;
std::future<int**> p7_f;


auto getP1 {
    [&]() {
        // A11 + A22
        int** t1 = addMatrix(a11, a22, n);
        // B11 + B22
        int** t2 = addMatrix(b11, b22, n);
        // P1 = t1 * t2
        int** p1 = strassen(t1, t2, n, 1);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);
        return p1;
    }
};

auto getP2 {
    [&]() {
        // A21 + A22
        int** t1 = addMatrix(a21, a22, n);
        // P2 = t1 * B11
        int** p2 = strassen(t1, b11, n, 1);
        deleteMatrix(t1, n);
        return p2;
    }
};


auto getP3 {
    [&]() {
        // B12 - B22
        int** t1 = subMatrix(b12, b22, n);
        // P3 = A11 * t1
        int** p3 = strassen(a11, t1, n, 1);
        deleteMatrix(t1, n);
        return p3;
    }
};


auto getP4 {
    [&]() {
        // B21 - B11
```

```
        int** t1 = subMatrix(b21, b11, n);
        // P4 = A22 * t1
        int** p4 = strassen(a22, t1, n, 1);
        deleteMatrix(t1, n);
        return p4;
    }
};


auto getP5 {
    [&]() {
        // A11 + A12
        int** t1 = addMatrix(a11, a12, n);
        // P5 = t1 * B22
        int** p5 = strassen(t1, b22, n, 1);
        deleteMatrix(t1, n);
        return p5;
    }
};

auto getP6 {
    [&]() {
        // A21 - A11
        int** t1 = subMatrix(a21, a11, n);
        // B11 + B12
        int** t2 = addMatrix(b11, b12, n);
        // P6 = t1 * t2
        int** p6 = strassen(t1, t2, n, 1);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);
        return p6;
    }
};

auto getP7 {
    [&]() {
        // A12 - A22
        int** t1 = subMatrix(a12, a22, n);
        // B21 + B22
        int** t2 = addMatrix(b21, b22, n);
        // P7 = t1 * t2
        int** p7 = strassen(t1, t2, n, 1);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);
        return p7;
    }
};


if (depth == 0) {
    p1_f = pool.submit(getP1);
    p2_f = pool.submit(getP2);
    p3_f = pool.submit(getP3);
    p4_f = pool.submit(getP4);
    p5_f = pool.submit(getP5);
    p6_f = pool.submit(getP6);
    p7_f = pool.submit(getP7);

    p1 = p1_f.get();
    p2 = p2_f.get();
    p3 = p3_f.get();
    p4 = p4_f.get();
    p5 = p5_f.get();
    p6 = p6_f.get();
    p7 = p7_f.get();
} else {
    p1 = getP1();
    p2 = getP2();
    p3 = getP3();
```

```
    p4 = getP4();
    p5 = getP5();
    p6 = getP6();
    p7 = getP7();
}


int** t1;
int** t2;


deleteMatrix(a11, n);
deleteMatrix(a12, n);
deleteMatrix(a21, n);
deleteMatrix(a22, n);
deleteMatrix(b11, n);
deleteMatrix(b12, n);
deleteMatrix(b21, n);
deleteMatrix(b22, n);


auto getQ1 {
    [&]() {
        return addMatrix(p1, p4, n);
    }
};

auto getQ2 {
    [&]() {
        return addMatrix(p2, p4, n);
    }
};

auto getQ3 {
    [&]() {
        return addMatrix(p3, p6, n);
    }
};

auto getQ4 {
    [&]() {
        return subMatrix(p7, p5, n);
    }
};

auto getQ5 {
    [&]() {
        return addMatrix(p3, p5, n);
    }
};

auto getQ6 {
    [&]() {
        return subMatrix(p1, p2, n);
    }
};


std::future<int**> q1_f;
std::future<int**> q2_f;
std::future<int**> q3_f;
std::future<int**> q4_f;
std::future<int**> q5_f;
std::future<int**> q6_f;

int** q1;
int** q2;
int** q3;
int** q4;
```

```
        int** q5;
        int** q6;

        if (depth == 0) {
            q1_f = pool.submit(getQ1);
            q2_f = pool.submit(getQ2);
            q3_f = pool.submit(getQ3);
            q4_f = pool.submit(getQ4);
            q5_f = pool.submit(getQ5);
            q6_f = pool.submit(getQ6);


            q1 = q1_f.get();
            q2 = q2_f.get();
            q3 = q3_f.get();
            q4 = q4_f.get();
            q5 = q5_f.get();
            q6 = q6_f.get();
        } else {
            q1 = getQ1();
            q2 = getQ2();
            q3 = getQ3();
            q4 = getQ4();
            q5 = getQ5();
            q6 = getQ6();
        }


        deleteMatrix(p1, n);
        deleteMatrix(p2, n);
        deleteMatrix(p3, n);
        deleteMatrix(p4, n);
        deleteMatrix(p5, n);
        deleteMatrix(p6, n);
        deleteMatrix(p7, n);

        int** c11 = addMatrix(q1, q4, n);
        int** c22 = addMatrix(q6, q3, n);


        int** res = combMatrix(c11, q5, q2, c22, n);
        deleteMatrix(c11, n);
        deleteMatrix(c22, n);

        deleteMatrix(q1, n);
        deleteMatrix(q2, n);
        deleteMatrix(q3, n);
        deleteMatrix(q4, n);
        deleteMatrix(q5, n);
        deleteMatrix(q6, n);

        return res;
    }
}

int main() {
    int64_t n, real_n;
    std::ifstream in("matrix.txt");

    if (!in.is_open()) {
        std::cout << "matrix.txt open error";
        return 1;
    }

    in >> real_n;
    n = real_n;

    if (!isPowerOfTwo(real_n) || real_n == 1) {
        n = new_size(real_n);
```

```
    }


    int** a = read_matrix(in, n, real_n);
    int** b = read_matrix(in, n, real_n);

    in.close();

    auto begin = std::chrono::steady_clock::now();

    int** result = strassen(a, b, n, 0);

    auto end = std::chrono::steady_clock::now();
    auto elapsed_ms = std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count();


    std::ofstream out("result.txt");
    if (!out.is_open()) {
        std::cout << "Result file open error";
        return 1;
    }


    for (int64_t i = 0; i < real_n; ++i) {
        for (int64_t j = 0; j < real_n; ++j) {
            out << result[i][j] << " ";
        }
        out << std::endl;
    }

    deleteMatrix(a, n);
    deleteMatrix(b, n);
    deleteMatrix(result, n);


    out.close();
    std::cout << "Ok" << std::endl;
    std::cout << "Time (s): " << (double) elapsed_ms/1000;

    return 0;
}
```

thread_pool.h

```
#pragma once

#include <atomic>
#include <condition_variable>
#include <exception>
#include <functional>
#include <future>
#include <memory>
#include <mutex>
#include <queue>
#include <thread>
#include <type_traits>
#include <utility>

namespace Lab {

using concurrency_t = std::invoke_result_t<decltype(std::thread::hardware_concurrency)>;

class thread_pool {
public:
```

```
    thread_pool(const concurrency_t thread_count_ = 0) :
thread_count(determine_thread_count(thread_count_)),
threads(std::make_unique<std::thread[]>(determine_thread_count(thread_count_))) {
        create_threads();
    }

    ~thread_pool() {
        wait_for_tasks();
        destroy_threads();
    }

    template <typename F, typename... A>
    void push_task(F&& task, A&&... args) {
        std::function<void()> task_function = std::bind(std::forward<F>(task),
std::forward<A>(args)...);

        const std::scoped_lock tasks_lock(tasks_mutex);
        tasks.push(task_function);

        ++tasks_total;
        task_available_cv.notify_one();
    }

    template <typename F, typename... A, typename R = std::invoke_result_t<std::decay_t<F>,
std::decay_t<A>...>>
    std::future<R> submit(F&& task, A&&... args) {
        std::function<R()> task_function = std::bind(std::forward<F>(task),
std::forward<A>(args)...);
        std::shared_ptr<std::promise<R>> task_promise = std::make_shared<std::promise<R>>();
        push_task(
            [task_function, task_promise] {
                try {
                    if constexpr (std::is_void_v<R>) {
                        std::invoke(task_function);
                        task_promise->set_value();
                    }
                    else {
                        task_promise->set_value(std::invoke(task_function));
                    }
                }
                catch (...) {
                    try {
                        task_promise->set_exception(std::current_exception());
                    }
                    catch (...)
                    {
                    }
                }
            });
        return task_promise->get_future();
    }

    void wait_for_tasks() {
        waiting = true;
        std::unique_lock<std::mutex> tasks_lock(tasks_mutex);
        task_done_cv.wait(tasks_lock, [this] { return (tasks_total == 0); });
        waiting = false;
    }

private:

    void create_threads() {
        running = true;
        for (concurrency_t i = 0; i < thread_count; ++i) {
            threads[i] = std::thread(&thread_pool::worker, this);
        }
    }

    void destroy_threads() {
```

```
        running = false;
        task_available_cv.notify_all();
        for (concurrency_t i = 0; i < thread_count; ++i) {
            threads[i].join();
        }
    }

    [[nodiscard]] concurrency_t determine_thread_count(const concurrency_t thread_count_) {
        if (thread_count_ > 0) {
            return thread_count_;
        } else {
            if (std::thread::hardware_concurrency() > 0)
                return std::thread::hardware_concurrency();
            else
                return 1;
        }
    }

    void worker() {
        while (running) {
            std::function<void()> task;
            std::unique_lock<std::mutex> tasks_lock(tasks_mutex);
            task_available_cv.wait(tasks_lock, [this] { return !tasks.empty() || !running; });
            if (running) {
                task = std::move(tasks.front());
                tasks.pop();
                tasks_lock.unlock();
                task();
                tasks_lock.lock();
                --tasks_total;
                if (waiting)
                    task_done_cv.notify_one();
            }
        }
    }

    std::atomic<bool> running = false;

    std::condition_variable task_available_cv = {};

    std::condition_variable task_done_cv = {};

    std::queue<std::function<void()>> tasks = {};

    std::atomic<size_t> tasks_total = 0;

    mutable std::mutex tasks_mutex = {};

    concurrency_t thread_count = 0;

    std::unique_ptr<std::thread[]> threads = nullptr;

    std::atomic<bool> waiting = false;
};

}
```