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 /Жеребцов К. А/ Проверил /Долженкова М. Л./

Киров 2023

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

Для вычисления результата Штрассен предложил алгоритм с семью умножениями:

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

Получение матрицы результата:

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

1. Программная реализация

Листинг программной реализации приведен в приложении А.

1. 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 |

1. 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;

};

}