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

Многопоточная реализация вычислительно сложного алгоритма с применением библиотеки OpenMP

Отчет по лабораторной работе №3 дисциплины
«Параллельное программирование»

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

Киров 2023

1. Цель лабораторной работы

Знакомство со стандартом OpenMP, получение навыков реализации многопоточных SPMD-приложений с применением библиотеки OpenMP.

2. Задание

1) Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора
2) Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер
3) Реализовать многопоточную версию алгоритм с помощью языка C++ и библиотеки OpenMP, 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 OpenMP-реализации алгоритма.

3. Выделение распараллеливаемых фрагментов

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:

$Q_1 = P_1 + P_4$

$Q_2 = P_2 + P_4$

$Q_3 = P_3 + P_6$

$Q_4 = P_7 - P_5$

$Q_5 = P_3 + P_5$

$Q_6 = P_1 - P_2$

Данные преобразования сводятся к каскадной схеме вычислений, представленной на рисунке 1.

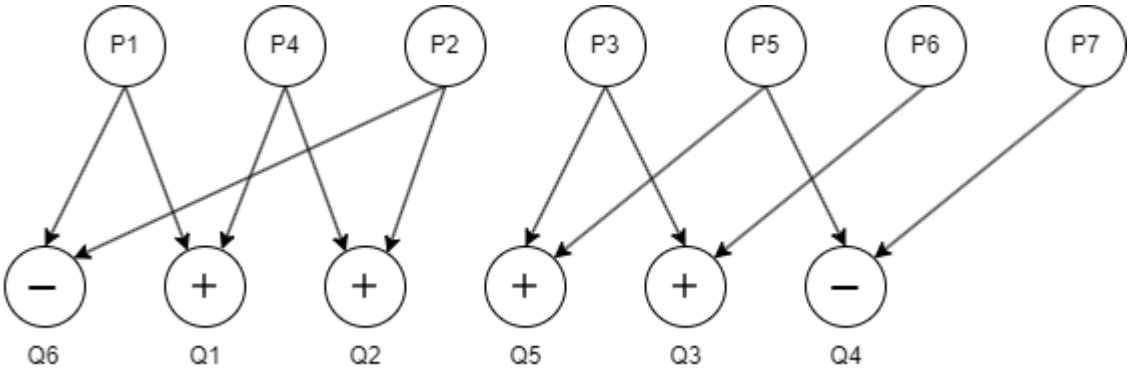

Рисунок 1 – Каскадная схема

4. Программная реализация

Листинг программной реализации приведен в приложении А.

5. 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	OpenMP, с
1	2048	3.562	0.998	1.01
2	4096	24.81	6.745	6.7
3	8192	176.352	46.397	46.0

6. Вывод

В ходе выполнения лабораторной работы были изучены принципы создания приложений с использованием библиотеки OpenMP, рассмотрены его директивы.

На основе программы, разработанной в ходе первой лабораторной работы был разработан параллельный алгоритм умножения матриц методом Штрассена с использованием библиотеки OpenMP.

Реализованный с помощью OpenMP алгоритм показал равную скорость выполнения по сравнению с алгоритмом, выполненном с использованием потоков стандартной библиотеки и почти 4-кратное ускорение по сравнению с линейном реализацией.

Приложение А.

Листинг программной реализации

main.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <cstring>
#include <omp.h>

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) {
    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;
#pragma omp task shared(p1)
{
    // A11 + A22
    int** t1 = addMatrix(a11, a22, n);
    // B11 + B22
    int** t2 = addMatrix(b11, b22, n);
    // P1 = t1 * t2
    p1 = strassen(t1, t2, n);
    deleteMatrix(t1, n);
    deleteMatrix(t2, n);
}

int** p2;
#pragma omp task shared(p2)
{
    // A21 + A22
    int** t1 = addMatrix(a21, a22, n);
    // P2 = t1 * B11
    p2 = strassen(t1, b11, n);
    deleteMatrix(t1, n);
}

int** p3;
#pragma omp task shared(p3)
{
    // B12 - B22
    int** t1 = subMatrix(b12, b22, n);
    // P3 = A11 * t1
    p3 = strassen(a11, t1, n);
    deleteMatrix(t1, n);
}

int** p4;
#pragma omp task shared(p4)
{
    // B21 - B11
    int** t1 = subMatrix(b21, b11, n);
    // P4 = A22 * t1
    p4 = strassen(a22, t1, n);
    deleteMatrix(t1, n);
}

int** p5;
#pragma omp task shared(p5)
{
    // A11 + A12
    int** t1 = addMatrix(a11, a12, n);
    // P5 = t1 * B22
    p5 = strassen(t1, b22, n);
    deleteMatrix(t1, n);
}

int** p6;
#pragma omp task shared(p6)
{
    // A21 - A11
    int** t1 = subMatrix(a21, a11, n);
    // B11 + B12
    int** t2 = addMatrix(b11, b12, n);
    // P6 = t1 * t2
```

```
    p6 = strassen(t1, t2, n);
    deleteMatrix(t1, n);
    deleteMatrix(t2, n);
}

int** p7;
#pragma omp task shared(p7)
{
    // A12 - A22
    int** t1 = subMatrix(a12, a22, n);
    // B21 + B22
    int** t2 = addMatrix(b21, b22, n);
    // P7 = t1 * t2
    p7 = strassen(t1, t2, n);
    deleteMatrix(t1, n);
    deleteMatrix(t2, n);
}

#pragma omp taskwait

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


int** q1;
#pragma omp task shared(q1)
{
    q1 = addMatrix(p1, p4, n);
}

int** q2;
#pragma omp task shared(q2)
{
    q2 = addMatrix(p2, p4, n);
}

int** q3;
#pragma omp task shared(q3)
{
    q3 = addMatrix(p3, p6, n);
}

int** q4;
#pragma omp task shared(q4)
{
    q4 = subMatrix(p7, p5, n);
}

int** q5;
#pragma omp task shared(q5)
{
    q5 = addMatrix(p3, p5, n);
}

int** q6;
#pragma omp task shared(q6)
{
    q6 = subMatrix(p1, p2, n);
}

#pragma omp taskwait

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;
    #pragma omp parallel
    {
        #pragma omp single
        {
           result = strassen(a, b, n);
        }

    }


    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;
}
```