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 №1 дисциплины «Параллельное программирование»

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

Киров 2023

1. Задание

   Перемножение матриц с помощью алгоритма Штрассена.

   1) Изучить алгоритм, полученный в соответствии с выданным преподавателем вариантом
   2) Провести доказательную оценку алгоритма по временной сложности и затратам памяти
   3) Реализовать алгоритм с помощью языка C++
   4) Построить набор тестовых примеров (не менее 10) и провести оценку эффективности реализованного алгоритма.

2. Изучение предметной области

   Необходимо вычислить произведение матриц $AB = C$

   Алгоритм Штрассена работает с квадратными матрицами размера 2^n. Если матрицы не соответствуют данному условию, их можно расширить заполнив недостающие строки и столбцы нулями.

   Алгоритм предполагает разбиение каждой из исходных матриц на 4 подматрицы.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

   Матрицы A_{ij} и B_{ij} имеют размер 2^{n-1}.

   Для вычисления матрицы C Штрассен предложил алгоритм с семью умножениями:

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

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

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

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

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

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

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

Получение матрицы C:

$$AB = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{pmatrix}$$

3. Асимптотические оценки временных и ресурсных затрат

Каждое умножение можно совершать рекурсивно по той же процедуре, а сложение – тривиально, складывая $(2^{n-1})^2$ элементов. Тогда время работы алгоритма $T(n)$ оценивается через рекуррентное соотношение: $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$.

Начальные матрицы могут иметь свои размеры, расширенные до следующей степени двойки. Каждая из 7 вспомогательных матриц содержит четверть элементов расширенной. Пространственная сложность алгоритма: $O\left(n^{\log_2 7}\right) = O(n^{2.81})$.

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	Наивный алгоритм, с	Алгоритм Штрассена, с
1	256	0.013	0.01
2	512	0.149	0.072
3	1024	3.51	0.512
4	2048	68.583	3.555
5	4096	726.466	24.806

6. Вывод

В ходе выполнения лабораторной работы был реализован алгоритм перемножения матриц методом Штрассена. Во время тестирования измерялось время работы наивного алгоритма умножения матриц и алгоритма Штрассена. По результатам тестирования алгоритм Штрассена значительно превосходит по времени традиционную реализацию на матрицах размером от 2048.

Приложение А.

Листинг программной реализации

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <cstring>

typedef std::vector<std::vector<int>> matrix;

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** t1;
int** t2;

// A11 + A22
t1 = addMatrix(a11, a22, n);
// B11 + B22
t2 = addMatrix(b11, b22, n);
// P1 = t1 * t2
int** p1 = strassen(t1, t2, n);
deleteMatrix(t1, n);
deleteMatrix(t2, n);


// A21 + A22
t1 = addMatrix(a21, a22, n);
// P2 = t1 * B11
int** p2 = strassen(t1, b11, n);
deleteMatrix(t1, n);

// B12 - B22
t1 = subMatrix(b12, b22, n);
// P3 = A11 * t1
int** p3 = strassen(a11, t1, n);
deleteMatrix(t1, n);

// B21 - B11
t1 = subMatrix(b21, b11, n);
// P4 = A22 * t1
int** p4 = strassen(a22, t1, n);
deleteMatrix(t1, n);


// A11 + A12
t1 = addMatrix(a11, a12, n);
// P5 = t1 * B22
int** p5 = strassen(t1, b22, n);
deleteMatrix(t1, n);


// A21 - A11
t1 = subMatrix(a21, a11, n);
deleteMatrix(a11, n);
deleteMatrix(a21, n);
// B11 + B12
t2 = addMatrix(b11, b12, n);
deleteMatrix(b11, n);
deleteMatrix(b12, n);
// P6 = t1 * t2
int** p6 = strassen(t1, t2, n);
deleteMatrix(t1, n);
deleteMatrix(t2, n);


// A12 - A22
t1 = subMatrix(a12, a22, n);
deleteMatrix(a12, n);
deleteMatrix(a22, n);
// B21 + B22
t2 = addMatrix(b21, b22, n);
deleteMatrix(b21, n);
```

```
        deleteMatrix(b22, n);
        // P7 = t1 * t2
        int** p7 = strassen(t1, t2, n);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);


        t1 = addMatrix(p1, p4, n);
        t2 = subMatrix(p7, p5, n);
        deleteMatrix(p7, n);

        int** c11 = addMatrix(t1, t2, n);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);

        int** c12 = addMatrix(p3, p5, n);
        deleteMatrix(p5, n);
        int** c21 = addMatrix(p2, p4, n);
        deleteMatrix(p4, n);


        t1 = addMatrix(p1, p3, n);
        deleteMatrix(p1, n);
        deleteMatrix(p3, n);

        t2 = subMatrix(p6, p2, n);
        deleteMatrix(p2, n);
        deleteMatrix(p6, n);

        int** c22 = addMatrix(t1, t2, n);
        deleteMatrix(t1, n);
        deleteMatrix(t2, n);



        int** res = combMatrix(c11, c12, c21, c22, n);
        deleteMatrix(c11, n);
        deleteMatrix(c12, n);
        deleteMatrix(c21, n);
        deleteMatrix(c22, 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);

    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;
}
```