МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Вятский государственный университет»
(ФГБОУ ВО «ВятГУ»)
Институт математики и информационных систем
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Отчёт
по лабораторной работе №2 по дисциплине
"Параллельное программирование"

Выполнил студент группы ИВТб-3301-04-00_____/Бикметов И.Р.
Проверил доцент кафедры ЭВМ_____/Исупов К.С.

Киров 2025

1 Цель работы

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

2 Задание

Поиск фрагмента текста в наборе источников с применением хеширования.

- Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора.
- Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер.
- Реализовать многопоточную версию алгоритм с помощью языка C++ и библиотеки OpenMP, используя при этом необходимые примитивы синхронизации.
- Показать корректность полученной реализации путем осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов.
- Провести доказательную оценку эффективности OpenMP-реализации алгоритма.

3 Описание многопоточного алгоритма

Алгоритм умножения полиномов при помощи быстрого преобразования Фурье состоит из 4-ех частей:

1. Преобразование Фурье для первого полинома;
2. Преобразования Фурье для второго полинома;
3. Умножение полиномов;
4. Обратное преобразование получившегося полинома.

Два первых пункта оказались независимы как по данным, так и по управлению, поэтому эти части алгоритма были распараллелены с помощью директив секций OpenMP. Также были распараллелены независимые циклы с помощью директивы for.

В данном варианте реализации были использованы следующие директивы OpenMP:

- parallel - определяет параллельный участок кода, который будет выполняться несколькими потоками параллельно;
- for - приводит к тому, что работа в цикле for внутри параллельного участка будет разделена между потоками;
- schedule(static) - указывает, что все итерации цикла делятся на равные блоки;
- num_threads - установки количества потоков;
- nowait - используется для отключения неявной барьерной синхронизации в конце параллельных конструкций;
- sections и section - используются для распределения независимых блоков кода между потоками.

4 Листинг программ

```
#include <iostream>
#include <vector>
#include <complex>
#include <chrono>
#include <random>
#include <cmath>
#include <thread>
#include <iomanip>
#include <omp.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

using namespace std;

const int NUM_THREADS = 2;
// прямое преобразование Фурье
vector<complex<double>> FFT(const vector<complex<double>>& vect) {
    long n = vect.size();
    // Выход из рекурсии - преобразование Фурье для одного элемента просто
    if (n == 1) {
        return vector<complex<double>>(1, vect[0]);
    }
```

```
    // вектор для комплексных экспонент
    vector<complex<double>> w(n);
    // вычисление комплексных экспонент
    for (long i = 0; i < n; i++) {
        // =2x/n - вычисление полярного угла
        double alpha = 2 * M_PI * i / n;
        // i=cos+isin - корень из 1
        w[i] = complex<double>(cos(alpha), sin(alpha));
    }

    // Считаем коэффициенты полинома A и B
    vector<complex<double>> A(n / 2); // четные
    vector<complex<double>> B(n / 2); // нечетные
    // заполнение вектороф коэффицентами полинома

    for (long i = 0; i < n / 2; i++) {
        A[i] = vect[i * 2];
        B[i] = vect[i * 2 + 1];
    }

    // получение коэффицентов Фурье
    vector<complex<double>> Av = FFT(A);
    vector<complex<double>> Bv = FFT(B);

    vector<complex<double>> res(n);

    for (long i = 0; i < n; i++) {
        // P(i)=A(2i)+iB(2i) - значение преобразования Фурье для частоты i
        res[i] = Av[i % (n / 2)] + w[i] * Bv[i % (n / 2)];
    }

    return res;
}

vector<complex<double>> FFT2(const vector<complex<double>>& vect) {
    long n = vect.size();
    // Выход из рекурсии - преобразование Фурье для одного элемента просто
    if (n == 1) {
        return vector<complex<double>>(1, vect[0]);
    }

    // вектор для комплексных экспонент
    vector<complex<double>> w(n);
```

```
    // вычисление комплексных экспонент
    for (long i = 0; i < n; i++) {
        // =2x/n - вычисление полярного угла
        double alpha = 2 * M_PI * i / n;
        // i=cos+isin - корень из 1
        w[i] = complex<double>(cos(alpha), sin(alpha));
    }

    // Считаем коэффициенты полинома A и B
    vector<complex<double>> A(n / 2); // четные
    vector<complex<double>> B(n / 2); // нечетные
    // заполнение вектороф коэффицентами полинома
    for (long i = 0; i < n / 2; i++) {
        A[i] = vect[i * 2];
        B[i] = vect[i * 2 + 1];
    }

    // получение коэффицентов Фурье
    vector<complex<double>> Av = FFT2(A);
    vector<complex<double>> Bv = FFT2(B);

    vector<complex<double>> res(n);
    for (long i = 0; i < n; i++) {
        // P(i)=A(2i)+iB(2i) - значение преобразования Фурье для частоты i
        res[i] = Av[i % (n / 2)] + w[i] * Bv[i % (n / 2)];
    }

    return res;
}

// обратное преобразование Фурье
void IFFT(vector<complex<double>>& x) {
    long N = x.size();
    // реализация комплексного сопряжения -> a+ib = a-ib
    for (long i = 0; i < N; i++) {
        x[i] = conj(x[i]);
    }
    // Применяем FFT к комплексно-сопряженным элементам
    x = FFT(x);
    // Обратное комплексное сопряжение и нормализация
    for (long i = 0; i < N; i++) {
        x[i] = conj(x[i]) / static_cast<double>(N);
    }
```

```
}

// обратное преобразование Фурье
void IFFT2(vector<complex<double>>& x) {
    long N = x.size();
    // реализация комплексного сопряжения -> a+ib = a-ib
    for (long i = 0; i < N; i++) {
        x[i] = conj(x[i]);
    }
    // Применяем FFT к комплексно-сопряженным элементам
    x = FFT2(x);
    // Обратное комплексное сопряжение и нормализация
    for (long i = 0; i < N; i++) {
        x[i] = conj(x[i]) / static_cast<double>(N);
    }
}

// умножение полиномов с применением быстрого преобразования 2 потока
vector<long> FFTMult(const vector<double>& p1, const vector<double>& p2) {
    long n = p1.size() + p2.size() - 1;
    long size = 1;
    // корректировка длины к степени двойки
    while (size < n) size <<= 1; // Даем размеры 2, 4, 8, 16

    vector<complex<double>> f1(size), f2(size);

    #pragma omp parallel num_threads(NUM_THREADS)
    {
        // переход к комплексным числам
        #pragma omp for
        for (long i = 0; i < p1.size(); ++i) {
            f1[i] = complex<double>(p1[i], 0);
        }

        #pragma omp for
        for (long i = 0; i < p2.size(); ++i) {
            f2[i] = complex<double>(p2[i], 0);
        }
    }

    // Векторы для результатов FFT
    vector<complex<double>> resF1(size);
    vector<complex<double>> resF2(size);
```

```
    #pragma omp parallel
    {
        // разделение работы между потоками
        #pragma omp sections
        {
            #pragma omp section
            {
                resF1 = FFT(f1);
            }
            #pragma omp section
            {
                resF2 = FFT(f2);
            }
        }
    }

    // Перемножение (суть ускорения)
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for (long i = 0; i < size; i++) {
            resF1[i] *= resF2[i];
        }
    }

    // Выполнение IFFT
    IFFT(resF1);

    //Получение результата
    vector<long> result(n);
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for (long i = 0; i < n; i++) {
            result[i] = round(resF1[i].real()); // Округляем до ближайшего
        }
    }

    return result;
}

// перемножение полиномов без параллельки
```

```
vector<long> MultPoly(vector<double>& p1, vector<double>& p2) {
    long n = p1.size() + p2.size() - 1;
    long size = 1;
    // корректировка длины к степени двойки
    while (size < n) size <<= 1;  // Даем размеры 2, 4, 8, 16, ...

    vector<complex<double>> f1(size), f2(size);

    // переход к комплексным числам
    for (long i = 0; i < p1.size(); ++i) {
        f1[i] = complex<double>(p1[i], 0);
    }
    for (long i = 0; i < p2.size(); ++i) {
        f2[i] = complex<double>(p2[i], 0);
    }
    // Выполнение FFT
    vector<complex<double>> resF1 = FFT2(f1);
    vector<complex<double>> resF2 = FFT2(f2);
    // Перемножение (суть ускорения)
    for (long i = 0; i < size; ++i) {
        resF1[i] *= resF2[i];
    }
    // Выполнение IFFT
    IFFT2(resF1);
    //Получение результата
    vector<long> result(n);
    for (long i = 0; i < n; ++i) {
        result[i] = round(resF1[i].real()); // Округляем до ближайшего цел
    }
    return result;
}

// вывод итогового полинома
void PrintResult(vector<double>& data) {
    cout << "Mult result: ";
    for (size_t i = 0; i < data.size(); i++) {
        cout << data[i] << (i < data.size() - 1 ? " + " : " ");
    }
    cout << std::endl;
}

// вывод времени в секундах
chrono::duration<double> PrintTime(chrono::duration<double> time) {
```

```
    cout << "Mult time: " << time.count() << "s" << endl;
    return time;
}

// генерация случайных чисел типа double
double generateRandomDouble(double lower, double upper) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<double> dis(lower, upper);
    return dis(gen);
}

// проверка результата
void checkAnswers(const vector<long> vect1, const vector<long> vect2) {
    bool allOk = true;
    long badNum1, badNum2;
    for (long i = 0; i < vect1.size(); i++) {
        if (vect1[i] != vect2[i]) {
            allOk = false;
            badNum1 = vect1[i];
            badNum2 = vect2[i];
            break;
        }
    }
    if (allOk) {
        cout << "Answer is Ok" << endl;
        return;
    }
    cout << "Bad answer!!!" << endl;
    cout << badNum1 << " != " << badNum2 << endl;
}

int main()
{
    // Входные данные
    const long LEN = pow(2, 16);
    vector<double> poly1;
    vector<double> poly2;
    poly1.resize(LEN);
    poly2.resize(LEN);

    // заполнение полиномов
    #pragma omp parallel for schedule(static) num_threads(6)
```

```
    for (long i = 0; i < LEN; i++)
    {
        //poly1[i] = generateRandomDouble(1, 100);
        //poly2[i] = generateRandomDouble(1, 100);
         poly1[i] = 222;
        poly2[i] = 222;
    }

    std::cout << "gen end" << endl;

    // Умножение полиномов с применением быстрого преобразования
    auto start = chrono::high_resolution_clock::now();
    vector<long> res1 = FFTMult(poly1, poly2);
    auto end = chrono::high_resolution_clock::now();
    std::cout << "FFT paral -> ";
    chrono::duration<double> t1 = PrintTime(end - start);


    // Классическое умножение полиномов
    start = chrono::high_resolution_clock::now();
    vector<long> res2 = MultPoly(poly1, poly2);
    end = chrono::high_resolution_clock::now();
    std::cout << "FFT def -> ";
    chrono::duration<double> t2 = PrintTime(end - start);
    std::cout << "k = " << t2 / t1 << endl;

    checkAnswers(res1, res2);
}
```

5 В чем же была проблема

Линейная реализация алгоритма была бастрее из-за того, что в параллельной версии производилось распараллеливание циклов внутри функции FFT, которая является рекурсивной. Глубина рекурсии FFT = длина полинома - 1, а размеры полиномов лежат в диапозоне $[2^{16}; 2^{25}]$ степени. Следовательно появляются накладные расходы на создание потоков: рекурсивные вызовы могут создавать большое количество задач, что может привести к значительным накладным расходам на управление потоками. Плюсом, большая глубина рекурсии может привести к переполнению стека, что тоже влияет на производительность. Также функция IFFT,отвечающая за ОПФ, содержит циклы, использующие функцию conj, которая также является рекурсивной.

Из этого следует, что функции прямого и обратного преобразования Фурье должны быть линейными. Следует распараллеливать только преобразования самих полиномов, а также их перемножение в комплексной плоскости.

6 Результаты тестов

Тестирование проводилось на ноутбуке под управлением 64-разрядной ОС Windows 11, с 16 Гб оперативной памяти и процессором 12th Gen Intel(R) Core(TM) i5-1235U 1.30 GHz(10 физических ядер и 12 логических ядер). Программа запускалась в Visual Studio 2022 с флагом оптимизации O_2 и поддержкой OpenMp.

Размер каждого полинома	Время умножения с помощью быстрого преобразования Фурье(парал), с	Время умножения с помощью быстрого преобразования Фурье(прост), с	Ускорение k
2^16	0.186856	0.233442	1.24932
2^17	0.379817	0.517922	1.36361
2^18	0.77139	1.21792	1.57886
2^19	1.55177	2.03332	1.31032
2^20	3.31225	4.15843	1.25547
2^21	6.92025	12.123	1.75181
2^22	17.3434	26.559	1.53136
2^23	40.671	55.8981	1.3744
2^24	93.5681	123.359	1.31839
2^25	194.267	232.381	1.1962

Рисунок 1 – Результаты тестов для 2 потоков

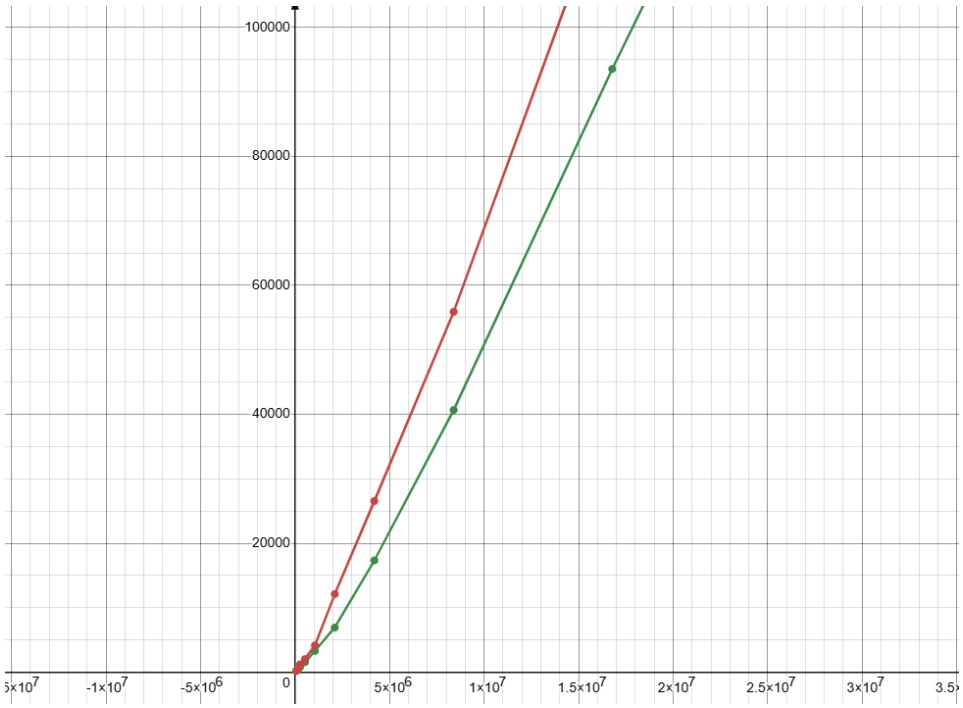

Рисунок 2 – График для 2 потоков

Размер каждого полинома	Время умножения с помощью быстрого преобразования Фурье(парал), с	Время умножения с помощью быстрого преобразования Фурье(прост), с	Ускорение k
2^16	0.242201	0.306144	1.26401
2^17	0.401285	0.518072	1.29103
2^18	0.770502	1.18986	1.54427
2^19	1.54305	2.00419	1.29885
2^20	3.12973	4.08825	1.30626
2^21	6.67581	11.3844	1.70531
2^22	16.3984	25.1562	1.53406
2^23	37.5891	51.9537	1.38215
2^24	84.8664	110.886	1.30659
2^25	199.372	258.204	1.29509

Рисунок 3 – Результаты тестов для 4 потоков

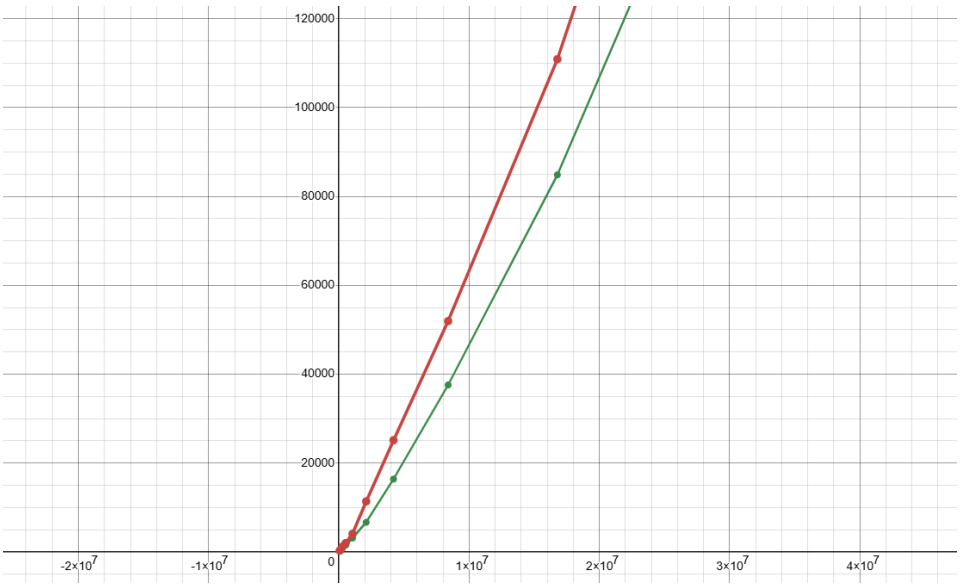

Рисунок 4 – График для 4 потоков

Размер каждого полинома	Время умножения с помощью быстрого преобразования Фурье(парал), с	Время умножения с помощью быстрого преобразования Фурье(прост), с	Ускорение k
2^16	0.211283	0.271045	1.28285
2^17	0.36825	0.484084	1.31455
2^18	0.794964	1.19263	1.50023
2^19	1.57064	2.04278	1.30061
2^20	3.10685	4.1504	1.33589
2^21	6.80349	11.8563	1.74268
2^22	16.2706	24.3597	1.49716
2^23	37.4461	52.8789	1.41213
2^24	83.8333	109.814	1.30991
2^25	138.03	177.987	1.28948

Рисунок 5 – Результаты тестов для 8 потоков

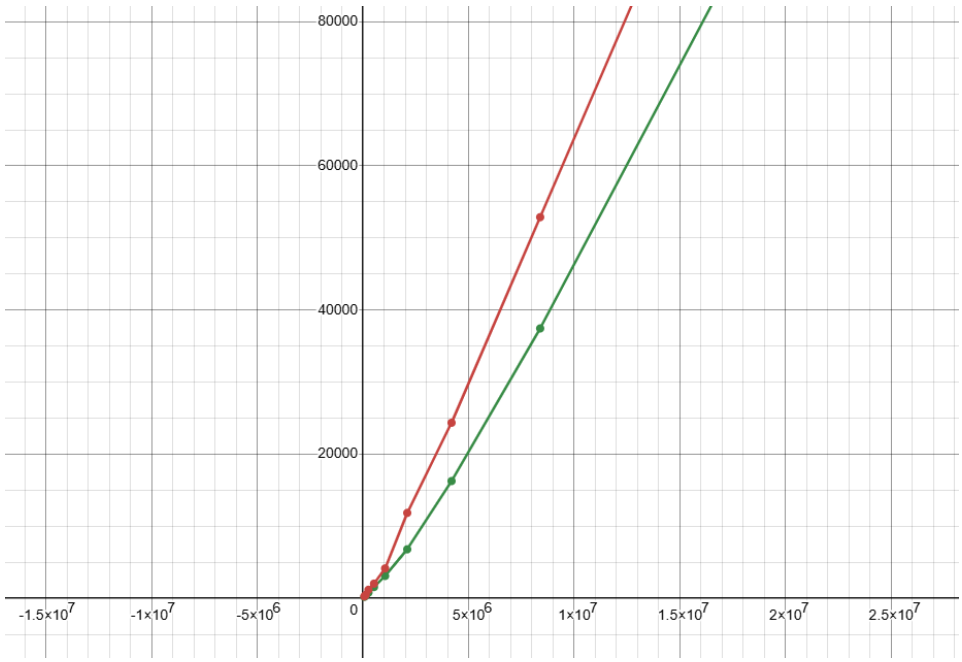

Рисунок 6 – График для 8 потоков

По графикам видно, что существенной разницы во времени выполнения программы с 2 потоками, 4 потоками и 8 потоками почти нет.

По времени выполнения многопоточная программа (8 потоков) примерно в 1.2,1.3 раза превосходит последовательную.

7 Вывод

В ходе лабораторной работы была изучен стандарт написания параллельных приложений OpenMP. Был разработан параллельный алгоритм умножения полиномов при помощи быстрого преобразования Фурье с использованием библиотеки OpenMP.

Параллельный алгоритм, реализованный с помощью OpenMP, оказался быстрее линейного почти в 1.2,1.3 раза.