МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ. WORK09

ОТЧЕТ О ПРАКТИКЕ

студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Вильцева Данила Денисовича

Проверил
Старший преподаватель _____ М. С. Портенко

Саратов 2023

СОДЕРЖАНИЕ

1 Work 9

1.1 Условие задачи

Периодический сигнал с периодом T, равным 1 секунде, задается функцией слева от знака равенства. Выполните дискретизацию сигнала таким образом, чтобы разрешение по частоте составляло 1 Hz при числе отсчетов 1024. Согласно своему варианту:

- рассчитайте коэффициенты ряда Фурье, используя параллельную программу БПФ;
- вычислите значения функции $f=\frac{a_0}{2}+\sum_{k=1}^{511}(a_k cos(k\frac{2\pi}{T}t)+b_k sin(k\frac{2\pi}{T}t)), 0<t<T$, подставив в выражение рассчитанные коэффициенты ряда Фурье;
- сравните подсчитанные значения функции с полученными аналитическим разложением в ряд Фурье и с точными значениями функции при $0<t<T$.

ВАРИАНТ 2

$$-ln\left(2\sin\left(\frac{\pi}{T}t\right)\right)=\sum_{k=1}^{\infty}\frac{\cos\left(k\frac{2\pi}{T}t\right)}{k}, 0<t<T$$

1.2 Решение

За основу возьмем распараллеленную версию БПФ из задачи 8.

Необходимо исключить процедуру **RandomDataInitialization**

Далее заменим процедуру **TaskDataInitialization**, где рассчитаем амплитудный спектр тестового сигнала в соответствии с вариантом. Размерность полученного массива (size) будет составлять должна составлять 1024 (передаем в процедуру константу)

Распараллеленную версию **BitReversing** оставим без изменений.

В **TestResult** вычисляем функцию:

$$f=\frac{a_0}{2}+\sum_{k=1}^{511}(a_k cos(k\frac{2\pi}{T}t)+b_k sin(k\frac{2\pi}{T}t)), 0<t<T$$

Подставив в неё рассчитанные коэффициенты ряда Фурье.

1.2.1 Исходный код

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <time.h>
#include <complex>
#include <omp.h>
using namespace std;
#define PI (3.14159265358979323846)
// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
        srand(unsigned(clock()));
        for (int i = 0; i < size; i++)
                mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0);
}
//Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal, complex<double>* &outputSignal,
↪  int &size) {
        // Setting the size of signals
        do
        {
                cout << "Enter the input signal length: ";
                cin >> size;
                if (size < 4)
                        cout << "Input signal length should be >= 4" << endl;
                else
                {
                        int tmpSize = size;
                        while (tmpSize != 1)
                        {
                                if (tmpSize % 2 != 0)
                                {
                                        cout << "Input signal length should be powers of
↪  two" << endl;
                                        size = -1;
                                        break;
                                }
                                tmpSize /= 2;
                        }
                }
        } while (size < 4);
        cout << "Input signal length = " << size << endl;
        inputSignal = new complex<double>[size];
        outputSignal = new complex<double>[size];
        //Initialization of input signal elements - tests
        RandomDataInitialization(inputSignal, size);
        //Computational experiments
```

```
        //RandomDataInitialization(inputSignal, size);
}
//Function for computational process temination
void ProcessTermination(complex<double>* &inputSignal, complex<double>* &outputSignal) {
        delete[] inputSignal;
        inputSignal = NULL;
        delete[] outputSignal;
        outputSignal = NULL;
}
void BitReversing(complex<double> *inputSignal, complex<double> *outputSignal, int size) {
        int j = 0, i = 0;
        for (i = 0; i < size; ++i) {
                if (j > i) {
                        outputSignal[i] = inputSignal[j];
                        outputSignal[j] = inputSignal[i];
                }
                else {
                        if (j == i) {
                                outputSignal[i] = inputSignal[i];
                        }
                }
                int m = size >> 1;
                while ((m >= 1) && (j >= m))
                {
                        j -= m;
                        m = m >> 1;
                }
                j += m;
        }
}
__inline void Butterfly(complex<double> *signal,
        complex<double> u, int offset, int butterflySize) {
        complex<double> tem = signal[offset + butterflySize] * u;
        signal[offset + butterflySize] = signal[offset] - tem;
        signal[offset] += tem;
}
void SerialFFTCalculation(complex<double> *signal, int size) {
        int m = 0;

        for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++);
        for (int p = 0; p < m; p++)
        {
                int butterflyOffset = 1 << (p + 1);
                int butterflySize = butterflyOffset >> 1;
                double coeff = PI / butterflySize;
                for (int i = 0; i < size / butterflyOffset; i++)
                        for (int j = 0; j < butterflySize; j++)
```

```
                                Butterfly(signal, complex<double>(cos(-j * coeff), sin(-j *
↪  coeff)), j + i * butterflyOffset, butterflySize);
        }
}
// FFT computation
void SerialFFT(complex<double> *inputSignal, complex<double> *outputSignal, int size) {
        BitReversing(inputSignal, outputSignal, size);
        SerialFFTCalculation(outputSignal, size);
}
void PrintSignal(complex<double> *signal, int size) {
        cout << "Result signal" << endl;
        for (int i = 0; i < size; i++)
                cout << signal[i] << endl;
}
int main()
{
        complex<double> *inputSignal = NULL;
        complex<double> *outputSignal = NULL;
        int size = 0;
        const int repeatCount = 16;
        double startTime;
        double duration;
        double minDuration = DBL_MAX;
        cout << "Fast Fourier Transform" << endl;
        // Memory allocation and data initialization
        ProcessInitialization(inputSignal, outputSignal, size);
        for (int i = 0; i < repeatCount; i++)
        {
                startTime = clock();
                // FFT computation
                SerialFFT(inputSignal, outputSignal, size);
                duration = (clock() - startTime) / CLOCKS_PER_SEC;
                if (duration < minDuration)
                        minDuration = duration;
        }
        cout << setprecision(6);
        cout << "Execution time is " << minDuration << " s. " << endl;
        // Result signal output
        int x = 0;
        PrintSignal(outputSignal, size);
        // Computational process termination
        ProcessTermination(inputSignal, outputSignal);
        return 0;
}
```

1.2.2 Код

```
#include <iomanip>
#include <iostream>
#include <cmath>
#include <complex>
#include <time.h>
#include <omp.h>
#include <vector>
using namespace std;
const double E = 0.000001;

#define PI (3.14159265358979323846)


void TaskDataInitialization(complex<double>* mas, int size)
{
        double pr = 1.0 / (double)size;
        for (int i = 0; i < size; i++) {
                mas[i] = -log(2 * sin(PI * pr * (i + 1)));

        }
}

//Function for memory allocation and data initialization
void ProcessInitialization(complex<double>*& inputSignal, complex<double>*& outputSignal,
↪  int& size) {
        // Setting the size of signals
        cout << "Выставлена длина входящего сигнала = " << size << endl;
        inputSignal = new complex<double>[size];
        outputSignal = new complex<double>[size];
        TaskDataInitialization(inputSignal, size);
}

//Function for computational process temination
void ProcessTermination(complex<double>*& inputSignal, complex<double>*& outputSignal) {
        delete[] inputSignal;
        inputSignal = NULL;
        delete[] outputSignal;
        outputSignal = NULL;
}

void BitReversing(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        int j = 0, i = 0;
#pragma omp parallel for
        for (i = 0; i < size; ++i) {
                if (j > i) {
                        outputSignal[i] = inputSignal[j];
                        outputSignal[j] = inputSignal[i];
```

```
                }
                else {
                        if (j == i) {
                                outputSignal[i] = inputSignal[i];
                        }
                }
                int m = size >> 1;
                while ((m >= 1) && (j >= m))
                {
                        j -= m;
                        m = m >> 1;
                }
                j += m;
        }
}

__inline void Butterfly(complex<double>* signal, complex<double> u, int offset, int
↪  butterflySize) {
        complex<double> tem = signal[offset + butterflySize] * u;
        signal[offset + butterflySize] = signal[offset] - tem;
        signal[offset] += tem;
}

void SerialFFTCalculation(complex<double>* signal, int size) {
        int m = 0;
        for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++);
        for (int p = 0; p < m; p++)
        {
                int butterflyOffset = 1 << (p + 1);
                int butterflySize = butterflyOffset >> 1;
                double coeff = PI / butterflySize;
                for (int i = 0; i < size / butterflyOffset; i++)
                        for (int j = 0; j < butterflySize; j++)
                                Butterfly(signal, complex<double>(cos(-j * coeff),
                                        sin(-j * coeff)), j + i * butterflyOffset,
↪  butterflySize);
        }
}

// FFT computation
void SerialFFT(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        BitReversing(inputSignal, outputSignal, size);
        SerialFFTCalculation(outputSignal, size);
}

void PrintSignal(complex<double>* signal, int size) {
        cout << "Result signal" << endl;
        for (int i = 0; i < size; i++)
```

```
                cout << signal[i] << endl;
}

void TestResult(complex<double>* inputSignal, complex<double>* outputSignal, int size)
{
        complex<double>* testSerialSignal;
        testSerialSignal = new complex<double>[size];

        double pr = 1.0 / (double)size;
        for (int i = 0; i < size; i++) {
                testSerialSignal[i] = outputSignal[0].real() / 4;
                for (int k = 1; k < size; k++) {
                        testSerialSignal[i] += (outputSignal[k].real() / 2 * cos(k * 2 * PI
↪  * pr * (i + 1)) - outputSignal[k].imag() / 2 * cos(k * 2 * PI * pr * (i + 1)));
                }
                testSerialSignal[i] /= size / 2;
        }

        vector <double> eter(size);
        for (int i = 0; i < size; i++) {
                eter[i] = 0;
                for (int k = 1; k < size * 16; k++)
                        eter[i] += cos(k * 2 * PI * pr * (i + 1)) / k;
        }

        cout << "\nИтоговая таблица. Выведено каждое четвёртое значение.\n\n";
        cout << "    Время          Вычисленные          Аналитическое          Точные" <<
↪  endl;
        cout << "                      значения           разложение           значения" <<
↪  endl;

        for (int i = 0; i < size; i += 4) {
                cout << setw(12) << pr * (i + 1) << "  |  " << setw(15) <<
↪  testSerialSignal[i].real() << "  |  " << setw(15) << eter[i] << "  |  " << setw(15) <<
↪  inputSignal[i].real() << endl;
        }

        delete[] testSerialSignal;

}

void ParellelFFTCalculation(complex<double>* signal, int size) {
        int m = 0;
        for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++);
        for (int p = 0; p < m; p++)
        {
                int butterflyOffset = 1 << (p + 1);
                int butterflySize = butterflyOffset >> 1;
```

```
                double coeff = PI / butterflySize;

#pragma omp parallel for
                for (int i = 0; i < size / butterflyOffset; i++)
                        for (int j = 0; j < butterflySize; j++) {
                                Butterfly(signal, complex<double>(cos(-j * coeff), sin(-j *
↪  coeff)), j + i * butterflyOffset, butterflySize);
                        }
        }

}

// FFT computation
void ParellelFFT(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        BitReversing(inputSignal, outputSignal, size);
        ParellelFFTCalculation(outputSignal, size);
}


int main()
{
        setlocale(LC_ALL, "Rus");
        complex<double>* inputSignal = NULL;
        complex<double>* outputSignal = NULL;
        int size = 1024;
        cout << "Быстрое преобразование Фурье (БПФ)" << endl;
        // Memory allocation and data initialization

        cout << "\nПредподготовка.\n";
        ProcessInitialization(inputSignal, outputSignal, size);

        ParellelFFT(inputSignal, outputSignal, size);
        cout << "\nНенулевые нормализованные значения амплитуды:\n";

        int half_size = size / 2;

        for (int i = 0; i < size; i++) {
                if (abs(outputSignal[i].real()) > E || abs(outputSignal[i].imag()) > E) {
                        outputSignal[i] = sqrt(pow(outputSignal[i].real(), 2) +
↪  pow(outputSignal[i].imag(), 2)) / half_size;
                        cout << outputSignal[i].real() << endl;
                }
                else
                        outputSignal[i] = 0;
        }

        ProcessTermination(inputSignal, outputSignal);
```

```
        cout << "\nЗадание\n";
        size = 1024;
        ProcessInitialization(inputSignal, outputSignal, size);
        ParellelFFT(inputSignal, outputSignal, size);
        TestResult(inputSignal, outputSignal, size);

        // Computational process termination
        ProcessTermination(inputSignal, outputSignal);
        system("pause");
        return 0;
}
```

2 Результат работы

Итоговая таблица. Выведено каждое четвёртое значение.

Время	Вычисленные значения	Аналитическое разложение	Точные значения
0.000976562	20.1588	5.09347	5.0936
0.00488281	3.49044	3.48416	3.4842
0.00878906	2.88854	2.89647	2.8965
0.0126953	2.51771	2.52888	2.52891
0.0166016	2.2484	2.2608	2.26083
0.0205078	2.03673	2.04973	2.04976
0.0244141	1.86233	1.87567	1.8757
0.0283203	1.71405	1.72759	1.72762
0.0322266	1.58509	1.59877	1.5988
0.0361328	1.47103	1.48479	1.48483
0.0400391	1.36879	1.38263	1.38266
0.0439453	1.27619	1.29008	1.29011
0.0478516	1.19159	1.20551	1.20554
0.0517578	1.11373	1.12768	1.12771
0.0556641	1.04163	1.05561	1.05565
0.0595703	0.974534	0.988534	0.988565
0.0634766	0.911797	0.925814	0.925844
0.0673828	0.852908	0.866938	0.866968
0.0712891	0.797437	0.811479	0.811509
0.0751953	0.745027	0.759077	0.759108
0.0791016	0.69537	0.709429	0.709459
0.0830078	0.648207	0.662273	0.662303
0.0869141	0.603313	0.617385	0.617415
0.0908203	0.560493	0.57457	0.5746
0.0947266	0.519575	0.533657	0.533688

3 Характеристики компьютера

Характеристики устройства

Имя устройства	DESKTOP-MSS8D39
Процессор	Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz 3.20 GHz
Оперативная память	8,00 ГБ
Код устройства	E3BB953D-13B0-42A7-944B-1ED9FD0E C328
Код продукта	00330-80000-00000-AA153
Тип системы	64-разрядная операционная система, процессор x64