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

Кафедра математической кибернетики и компьютерных наук

ОТЧЕТ ПО ПРАКТИКЕ WORK16

ОТЧЕТ

Студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Забоева Максима Владиславовича

Проверил
Старший преподаватель _____ М. С. Портенко

Саратов 2023

СОДЕРЖАНИЕ

1 Условие задачи 3
2 Практическая часть 4
3 Результаты работы 7
3.1 Характеристики компьютера 7
3.2 Таблицы результатов 7
3.3 Фото результатов 7

1 Условие задачи

Аналогично работе с OMP выполните следующее задание через MPI.

Проведите эксперименты для последовательного и параллельного вычислений БПФ, результаты занесите в таблицу 1.

Таблица 1. Результаты вычислительных экспериментов и ускорение вычислений

Номер теста	Размер входного сигнала	Мин. время работы последовательного приложения (сек)	Мин. время работы параллельного приложения (сек)	Ускорение
1	32768			
2	65536			
3	131072			
4	262144			
5	524288			

2 Практическая часть

Код программы:

```
#include <iomanip>
#include <iostream>
#include <cmath>
#include <complex>
#include <time.h>
#include <mpi.h>
#include <algorithm>
using namespace std;
#define PI 3.14159265358979323846
int NProc, ProcId;
void PrintSignal(complex<double>* signal, int size) {
        cout << "Result signal" << endl;
        for (int i = 0; i < size; i++)
                cout << signal[i] << endl;
}
void DummyDataInitialization(complex<double>* mas, int size) {
        for (int i = 0; i < size; i++)
                mas[i] = 0;
        mas[size - size / 4] = 1;
}
void ProcessInitialization(complex<double>*& inputSignal, complex<double>*& outputSignal,int& size) {
        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);
        inputSignal = new complex<double>[size];
        outputSignal = new complex<double>[size];
        DummyDataInitialization(inputSignal, size);
}
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 bitsCount = 0;
        for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, bitsCount++);
        for (int ind = 0; ind < size; ind++) {
                int mask = 1 << (bitsCount - 1);
```

```
                int revInd = 0;
                for (int i = 0; i < bitsCount; i++) {
                        bool val = ind & mask;
                        revInd |= val << i;
                        mask = mask >> 1;
                }
                outputSignal[revInd] = inputSignal[ind];
        }
}
__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 ParallelFFTCalculation(complex<double>* signal, int size) {
        int m = 0;
        for (int tmp_size = size; tmp_size > 1; tmp_size >>= 1, m++);
        for (int p = 1; p <= m; p++) {
                int butterflyOffset = 1 << p;
                int butterflySize = butterflyOffset >> 1;
                double coeff = PI / butterflySize;
                int i_amount = size / butterflyOffset;
                if (i_amount >= butterflySize and i_amount >= NProc) {
                        for (int i = ProcId; i < i_amount; i += NProc) {
                                for (int j = 0; j < butterflySize; j++) {
                                        double j_coeff = -j * coeff;
                                        Butterfly(signal, complex<double>(cos(j_coeff),sin(j_coeff)), j + i * butterflyOffse
                                }
                                int zero = i - ProcId;
                                int sup = min(zero + NProc, i_amount);
                                for (int k = zero; k < sup; k++) {
                                                int buff = k * butterflyOffset;
                                        int CurProc = k % NProc;
                                        MPI_Bcast(&signal[buff], butterflySize,MPI_DOUBLE_COMPLEX, CurProc, MPI_COMM_WORLD);
                                        MPI_Bcast(&signal[buff + butterflySize],butterflySize, MPI_DOUBLE_COMPLEX, CurProc,
                                }
                        }
                }
                else {
                        for (int i = 0; i < i_amount; i++) {
                                for (int j = 0; j < butterflySize; j++) {
                                        double j_coeff = -j * coeff;
                                        Butterfly(signal, complex<double>(cos(j_coeff),sin(j_coeff)), j + i * butterflyOffse
                                }
                        }
                }
        }
}
void ParallelFFT(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        BitReversing(inputSignal, outputSignal, size);
        ParallelFFTCalculation(outputSignal, size);
}
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 = 1; p <= m; p++) {
                int butterflyOffset = 1 << p;
                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);
        }
}
void SerialFFT(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        BitReversing(inputSignal, outputSignal, size);
        SerialFFTCalculation(outputSignal, size);
}
void TestResult(complex<double>* inputSignal, complex<double>* outputSignal, int size) {
        complex<double>* testSerialSignal;
        double Accuracy = 1.e-6;
        bool equal = true;
        int i;
        testSerialSignal = new complex<double>[size];
        SerialFFT(inputSignal, testSerialSignal, size);
        for (i = 0; i < size; i++) {
                if (abs(outputSignal[i] - testSerialSignal[i]) >= Accuracy)
                        equal = false;
        }
        delete[] testSerialSignal;
}
int main() {
        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &NProc);
        MPI_Comm_rank(MPI_COMM_WORLD, &ProcId);
        complex<double>* inputSignal = NULL;
        complex<double>* outputSignal = NULL;
        int size = 0;
        const int repeatCount = 16;
        double startTime, finishTime;
        double duration;
        double minDuration = DBL_MAX;
        ProcessInitialization(inputSignal, outputSignal, size);
        for (int i = 0; i < repeatCount; i++) {
                if (ProcId == 0) {
                        startTime = MPI_Wtime();
                }
                ParallelFFT(inputSignal, outputSignal, size);
                if (ProcId == 0) {
                        finishTime = MPI_Wtime();
                        duration = (finishTime - startTime) / CLOCKS_PER_SEC;
                        if (duration < minDuration)
                                minDuration = duration;
                }
        }
        if (ProcId == 0) {
                cout << setprecision(6);
                cout << "Execution time is " << minDuration << " s." << endl;
                TestResult(inputSignal, outputSignal, size);
        }
        ProcessTermination(inputSignal, outputSignal);
        MPI_Finalize();
        return 0;
}
```

3 Результаты работы

3.1 Характеристики компьютера

Процессор — 12th Gen Intel Core i5-12600KF, Базовая скорость 3,70ГГц, Кол-во ядер 10, Кол-во процессоров 16 (включая 4 энергоэффективных ядра). 16гб Оперативной памяти, скорость 3200МГц

3.2 Таблицы результатов

Номер теста	Размер входного сигнала	Мин. время работы последовательного приложения (сек)	Мин. время работы параллельного приложения (сек)	Ускорение
1	32768	0,018	0,0019	9,473684211
2	65536	0,037	0,0043	8,604651163
3	131072	0,085	0,0093	9,139784946
4	262144	0,169	0,0193567	8,730827052
5	524288	0,367	0,0413012	8,88594036

3.3 Фото результатов