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

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ. WORK16

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 16 3
1.1 Условие задачи 3
1.2 Решение. Последовательная версия 3
1.3 Решение. Параллельная версия 6
2 Результат работы 11
3 Характеристики компьютера 13

1 Work 16

1.1 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			

1.2 Решение. Последовательная версия

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <time.h>
#include <complex>
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;
        while (i < size)
        {
                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;
                i++;
        }
}
__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
        PrintSignal(outputSignal, size);
        // Computational process termination
        ProcessTermination(inputSignal, outputSignal);
        return 0;
}
```

1.3 Решение. Параллельная версия

```
#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) {
        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 * butterflyOffset, butterflySize);
                                }
                                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, MPI_COMM_WORLD);
                                }
                        }
                }
                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 * butterflyOffset, butterflySize);
                                }
                        }
                }
        }
}

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;
        }
        if (!equal) printf("The results of serial and parallel algorithms are NOT
↪  identical.\n");
        else printf("The results of serial and parallel algorithms are identical.\n");
        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 = 32768;
        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 = clock();
                }
                ParallelFFT(inputSignal, outputSignal, size);
                if (ProcId == 0) {
                        finishTime = clock();
                        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);
                //PrintSignal(outputSignal, size);
        }
        ProcessTermination(inputSignal, outputSignal);
        MPI_Finalize();
        return 0;
```

```
}
```

2 Результат работы

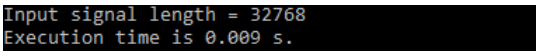

Рисунок 1 – Последовательная 32768

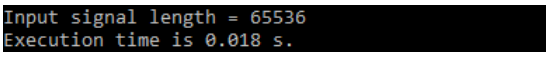

Рисунок 2 – Последовательная 65536

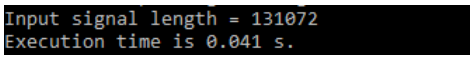

Рисунок 3 – Последовательная 131072

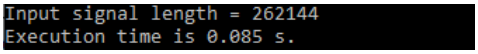

Рисунок 4 – Последовательная 262144

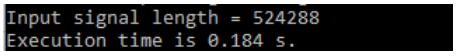

Рисунок 5 – Последовательная 524288

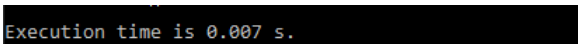

Рисунок 6 – Параллельная 32768

Рисунок 7 – Параллельная 65536

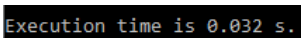

Рисунок 8 – Параллельная 131072

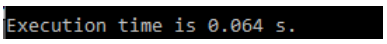

Рисунок 9 – Параллельная 262144

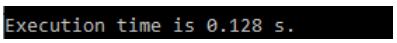

Рисунок 10 – Параллельная 524288

Номер теста	Размер входного сигнала	Мин. время работы последовательного приложения (сек)	Мин. время работы параллельного приложения (сек)	Ускорение
1	32768	0.009	0.007	1,28571
2	65536	0.018	0.014	1,28571
3	131072	0.041	0.032	1,28125
4	262144	0.085	0.064	1,32812
5	524288	0.184	0.128	1,4375

Рисунок 11 – Результирующая таблица

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