**Процессор:**

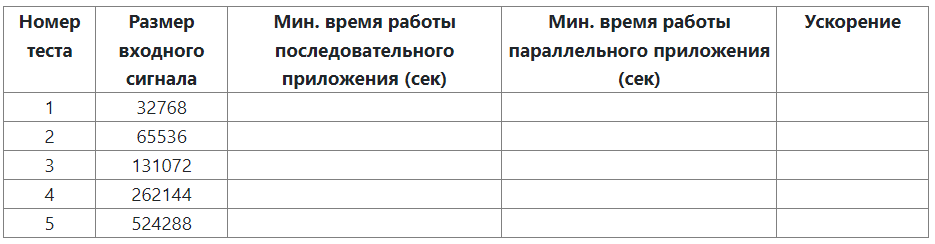
AMD Ryzen 5 5600H with Radeon Graphics, 3301 МГц,   
ядер: 6, логических процессоров: 12

**Память:** 16,0 ГБ (доступно: 15,4 ГБ)

**Задание 7**

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

Таблица 1. Результаты вычислительных экспериментов и ускорение вычислений



Оцените эффективность динамического планирования в OpenMP-версии по сравнению со статическим.

**Код последовательного метода**

#include <iomanip>

#include <iostream>

#include <cmath>

#include <complex>

#include <time.h>

using namespace std;

#define PI (3.14159265358979323846)

//Function for simple initialization of input signal elements

void DummyDataInitialization(complex<double>\* mas, int size) {

for (int i = 0; i < size; i++)

mas[i] = 0;

mas[size - size / 4] = 1;

}

// Function for random initialization of objects' elements

void RandomDataInitialization(complex<double>\* mas, int size)

{

srand(unsigned(1));

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

//DummyDataInitialization(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 < 10; 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;

}

**Код параллельного метода**

#include <iomanip>

#include <iostream>

#include <cmath>

#include <complex>

#include <time.h>

#include <omp.h>

using namespace std;

#define PI (3.14159265358979323846)

//Function for simple initialization of input signal elements

void DummyDataInitialization(complex<double>\* mas, int size) {

for (int i = 0; i < size; i++)

mas[i] = 0;

mas[size - size / 4] = 1;

}

// 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

DummyDataInitialization(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 ParallelFFTCalculation(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 //schedule(dynamic, 16)

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 ParallelFFT(complex<double>\* inputSignal,

complex<double>\* outputSignal, int size) {

BitReversing(inputSignal, outputSignal, size);

ParallelFFTCalculation(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)

{

// Buffer for storing the result of serial FFT

complex<double>\* testSerialSignal;

double Accuracy = 1.e-6; // Comparison accuracy

// Flag, that shows whether the vectors are identical

int equal = 0;

int i; // Loop variable

testSerialSignal = new complex<double>[size];

SerialFFT(inputSignal, testSerialSignal, size);

for (i = 0; i < size; i++)

{

if (abs(outputSignal[i] - testSerialSignal[i]) >= Accuracy)

equal = 1;

}

if (equal == 1)

printf("The results of serial and parallel algorithms are NOT identical.Check your code.");

else

printf("The results of serial and parallel algorithms are identical.");

delete[] testSerialSignal;

}

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

ParallelFFT(inputSignal, outputSignal, size);

duration = (clock() - startTime) / CLOCKS\_PER\_SEC;

if (duration < minDuration)

minDuration = duration;

}

cout << setprecision(10) << fixed;

cout << "Execution time is " << minDuration << " s. " << endl;

// Result signal output

//PrintSignal(outputSignal, size);

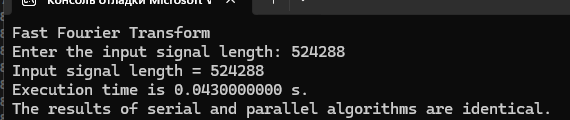
TestResult(inputSignal, outputSignal, size);

// Computational process termination

ProcessTermination(inputSignal, outputSignal);

return 0;

}

****

**Таблица 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер теста | Размер входного сигнала | Мин. время работы последовательного приложения (сек) | Мин. время работы параллельного приложения (сек) | Ускорение |
| 1 | 32768 | 0,005 | 0,002 | 2,5 |
| 2 | 65536 | 0,012 | 0,005 | 2,4 |
| 3 | 131072 | 0,026 | 0,011 | 2,363636364 |
| 4 | 262144 | 0,055 | 0,019 | 2,894736842 |
| 5 | 524288 | 0,119 | 0,041 | 2,902439024 |

**Вывод:** Параллельный метод увеличивает ускорение по сравнению с последовательным при увеличении размера входного сигнала. В среднем ускорение, при 56% загруженности процессора из-за отсутствующего подключения к зарядке, составляет 2,75 раза