**Процессор:**

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

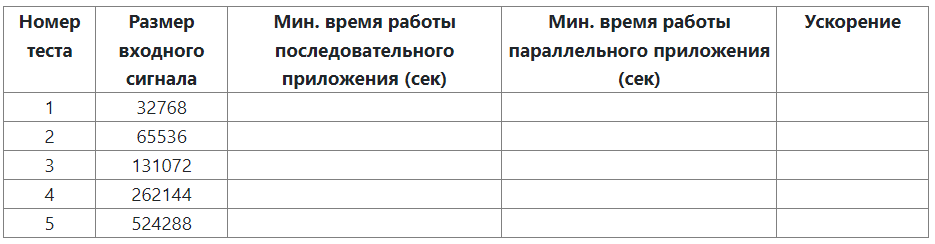
**Память:** 16,0 ГБ (доступно: 15,4 ГБ)

**Задание 16**

**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



**Код программы:**

#include <iomanip>

#include <iostream>

#include <cmath>

#include <complex>

#include <time.h>

#include <math.h>

#include <omp.h>

#include <mpi.h>

using namespace std;

#define T 1

#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, int NProc, int ProcId) {

// Setting the size of signals

if (ProcId == 0) {

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];

if (ProcId == 0) {

//Initialization of input signal elements - tests

// DummyDataInitialization(inputSignal, size);

//Computational experiments

RandomDataInitialization(inputSignal, size);

}

MPI\_Bcast(&(inputSignal[0]), size, MPI\_DOUBLE\_COMPLEX, 0, MPI\_COMM\_WORLD);

}

//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++;

}

}

void ParallelBitReversing(complex<double>\* inputSignal,

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

int bitsCount = 0;

//bitsCount = log2(size)

for (int tmp\_size = size; tmp\_size > 1; tmp\_size /= 2, bitsCount++);

//ind - index in input array

//revInd - correspondent to ind index in output array

int n1 = size / NProc;

int n2 = (ProcId + 1) \* n1;

if (NProc == ProcId + 1) {

n2 = size;

}

int st = ProcId \* n1;

#pragma omp parallel for

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

{

//if (i < )

int mask = 1 << (bitsCount - 1);

int revInd = 0;

for (int i = 0; i < bitsCount; i++) //bit-reversing

{

bool val = ind & mask;

revInd |= val << i;

mask = mask >> 1;

}

outputSignal[revInd] = inputSignal[ind];

}

}

\_\_inline void Butterfly(complex<double>\* signal1, complex<double>\* signal,

complex<double> u, int offset, int butterflySize, int NProc, int ProcId) {

complex<double> tem = signal[offset + butterflySize] \* u;

signal1[offset + butterflySize] = signal[offset] - tem;

signal1[offset] += tem;

}

/\*

void SerialFFTCalculation(complex<double>\* signal, int size, int NProc, int ProcId) {

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, NProc, ProcId);

}

}

\*/

void ParallelFFTCalculation(complex<double>\* signal, int size, int NProc, int ProcId) {

int m = 0;

int n1;

int n2;

int st;

int h;

complex<double>\* signal1 = new complex<double>[size];

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

signal1[i] = 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;

n1 = (size / butterflyOffset) / NProc;

n2 = (ProcId + 1) \* n1;

if (NProc == ProcId + 1) {

n2 = size / butterflyOffset;

}

st = ProcId \* n1;

for (int i = st; i < n2; i++) {

for (int j = 0; j < butterflySize; j++)

Butterfly(signal1, signal, complex<double>(cos(-j \* coeff),

sin(-j \* coeff)), j + i \* butterflyOffset, butterflySize, NProc, ProcId);

}

MPI\_Reduce(&(signal1[0]), &(signal[0]), size, MPI\_DOUBLE\_COMPLEX, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&(signal[0]), size, MPI\_DOUBLE\_COMPLEX, 0, MPI\_COMM\_WORLD);

}

}

/\*

// FFT computation

void SerialFFT(complex<double>\* inputSignal,

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

BitReversing(inputSignal, outputSignal, size);

SerialFFTCalculation(outputSignal, size, NProc, ProcId);

}

\*/

void ParallelFFT(complex<double>\* inputSignal,

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

ParallelBitReversing(inputSignal, outputSignal, size, NProc, ProcId);

ParallelFFTCalculation(outputSignal, size, 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 Pretreatment(int NProc, int ProcId) {

int count = 128;

double eps = 1e-6;

complex<double>\* inputSignal = new complex<double>[count];

complex<double>\* outputSignal = new complex<double>[count];

double t;

#pragma parallel omp for

for (int i = 0; i < count; i++) {

t = 1.0 \* i / count;

inputSignal[i] = complex<double>(sin(2.0 \* PI \* 10 \* t), 0);

}

//SerialFFT(inputSignal, outputSignal, count);

ParallelFFT(inputSignal, outputSignal, count, NProc, ProcId);

cout << "Pretreatment result:" << endl;

for (int i = 0; i < count; i++) {

if (abs(outputSignal[i].real()) > eps || abs(outputSignal[i].imag()) > eps) {

cout << i << " " << outputSignal[i] << endl;

}

}

double norm = sqrt(outputSignal[10].imag() \* outputSignal[10].imag() +

outputSignal[10].real() \* outputSignal[10].real());

cout << "AMPLITUDE:" << norm / count \* 2;

}

double f(complex<double>\* signal, int size, double t) {

double res = signal[0].real() / 2.0;

for (int i = 1; i < 512; i++) {

res += signal[i].real() \* cos(i \* 2.0 \* PI \* t / T)

- signal[i].imag() \* sin(i \* 2.0 \* PI \* t / T);

}

return res;

}

double StandardSum(double t) {

//сумма ряда

double eps = 1e-9;

double res = 0;

double s = 0;

int k = 1;

do {

s = sin(k \* 2.0 \* PI \* t / T) / k;

res += s;

k++;

} while (fabs(s) > eps);

return res;

}

void var1(int NProc, int ProcId) {

int count = 1024;

complex<double>\* inputSignal = new complex<double>[count];

complex<double>\* outputSignal = new complex<double>[count];

for (int i = 1; i < count; i++) {

double t = i \* 1.0 / count;

inputSignal[i] = complex<double>((PI / 2 - PI \* t / T), 0);

}

//SerialFFT(inputSignal, outputSignal, count);

ParallelFFT(inputSignal, outputSignal, count, NProc, ProcId);

for (int i = 0; i < count; i++) {

outputSignal[i] = outputSignal[i] / ((double)count / 2.);

}

cout << left << setw(10) << "Function" << " "

<< setw(10) << "Fourier" << " " << setw(10) << "Exact value" << endl;

for (int i = 1; i < count; i++) {

double t = i \* 1.0 / count;

cout << setw(10) << f(outputSignal, count, t) << " "

<< setw(10) << StandardSum(t) << " " << setw(10) << (PI / 2 - PI \* t / T) << endl;

}

}

void var2(int NProc, int ProcId) {

int count = 1024;

complex<double>\* inputSignal = new complex<double>[count];

complex<double>\* outputSignal = new complex<double>[count];

for (int i = 1; i < count; i++) {

double t = i \* 1.0 / count;

inputSignal[i] = complex<double>((PI / 2 - PI \* t / T), 0);

}

//SerialFFT(inputSignal, outputSignal, count);

ParallelFFT(inputSignal, outputSignal, count, NProc, ProcId);

for (int i = 0; i < count; i++) {

outputSignal[i] = outputSignal[i] / ((double)count / 2.);

}

cout << left << setw(10) << "Function" << " "

<< setw(10) << "Fourier" << " " << setw(10) << "Exact value" << endl;

for (int i = 1; i < count; i++) {

double t = i \* 1.0 / count;

cout << setw(10) << f(outputSignal, count, t) << " "

<< setw(10) << StandardSum(t) << " " << setw(10) << -log(2 \* sin(PI \* t / T)) << endl;

}

}

int main() {

MPI\_Init(NULL, NULL);

//Pretreatment();

//var1();

complex<double>\* inputSignal = NULL;

complex<double>\* outputSignal = NULL;

int size = 1524288;

//int size = 8;

const int repeatCount = 16;

double startTime;

double duration;

double minDuration = DBL\_MAX;

int NProc, ProcId;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &NProc);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcId);

if (ProcId == 0)

cout << "Fast Fourier Transform" << endl;

// Memory allocation and data initialization

ProcessInitialization(inputSignal, outputSignal, size, NProc, ProcId);

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

{

startTime = clock();

// FFT computation

ParallelFFT(inputSignal, outputSignal, size, NProc, ProcId); duration = (clock() - startTime) / CLOCKS\_PER\_SEC;

//SerialFFT(inputSignal, outputSignal, size); duration = (clock() - startTime) / CLOCKS\_PER\_SEC;

if (duration < minDuration)

minDuration = duration;

}

if (ProcId == 0) {

cout << setprecision(6);

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

// Result signal output

//PrintSignal(outputSignal, size);

// Computational process termination

}

ProcessTermination(inputSignal, outputSignal);

MPI\_Finalize();

return 0;

}

**Таблица 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| номер теста | Размер входного сигнала | Мин. время работы последовательного приложения (сек) | Мин. время работы параллельного приложения (сек) | Ускорение |
| 1 | 32768 | 0,0128 | 0,008 | 1,6 |
| 2 | 65536 | 0,0285 | 0,016 | 1,782608696 |
| 3 | 131072 | 0,075 | 0,041 | 1,849056604 |
| 4 | 262144 | 0,2546 | 0,132 | 1,929078014 |
| 5 | 524288 | 0,567 | 0,326 | 1,739263804 |

**Вывод:** Параллельная версия выигрывает по времени у последовательной почти в 2 раза.