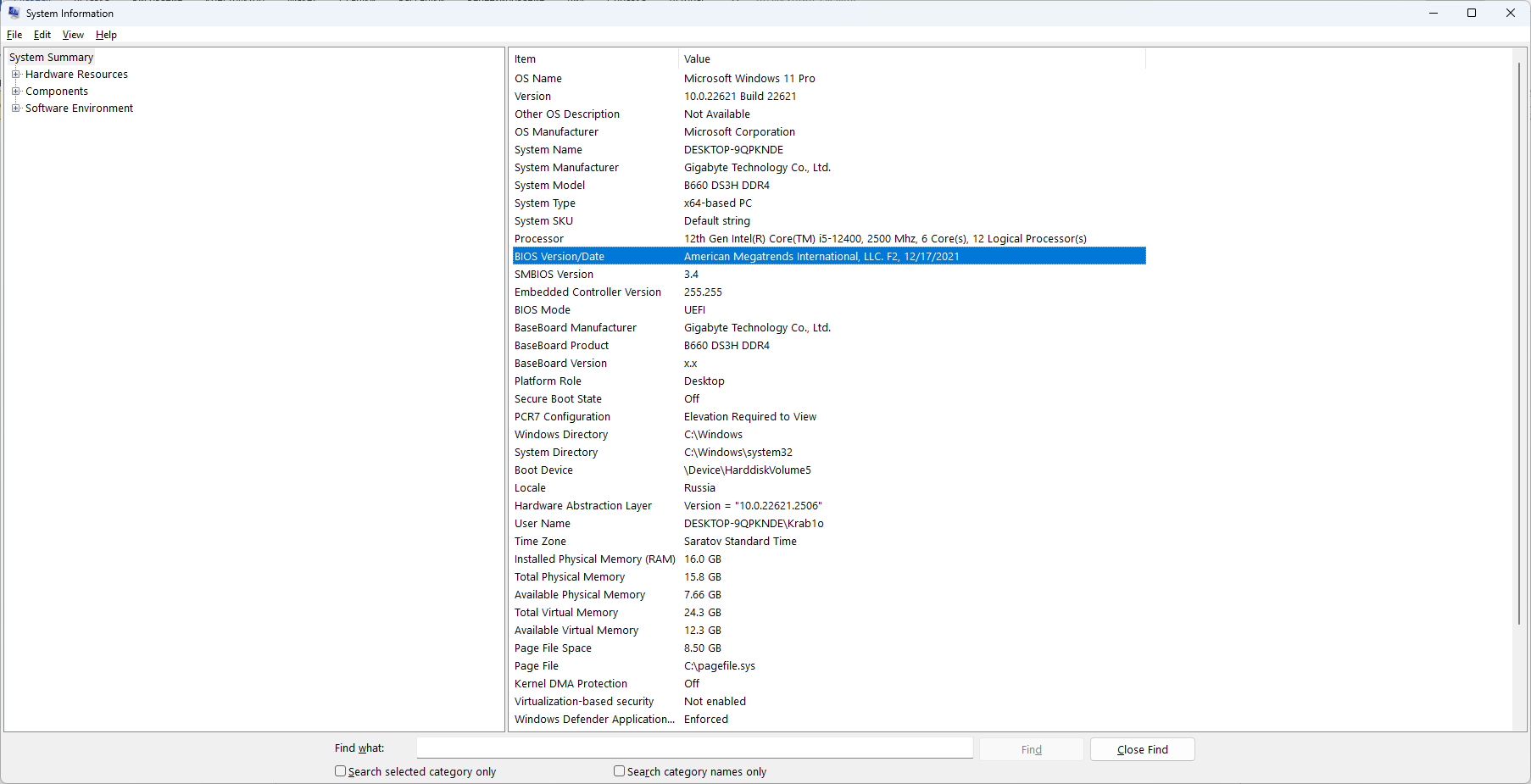
Задание выполнялось на компьютере со следующими параметрами:



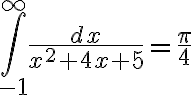
Из важного:

CPU: 12th Gen Intel(R) Core(TM) i5-12400, 2500 Mhz, 6 Core(s), 12 Logical Processor(s)

Motherboard: B660 DS3H DDR4

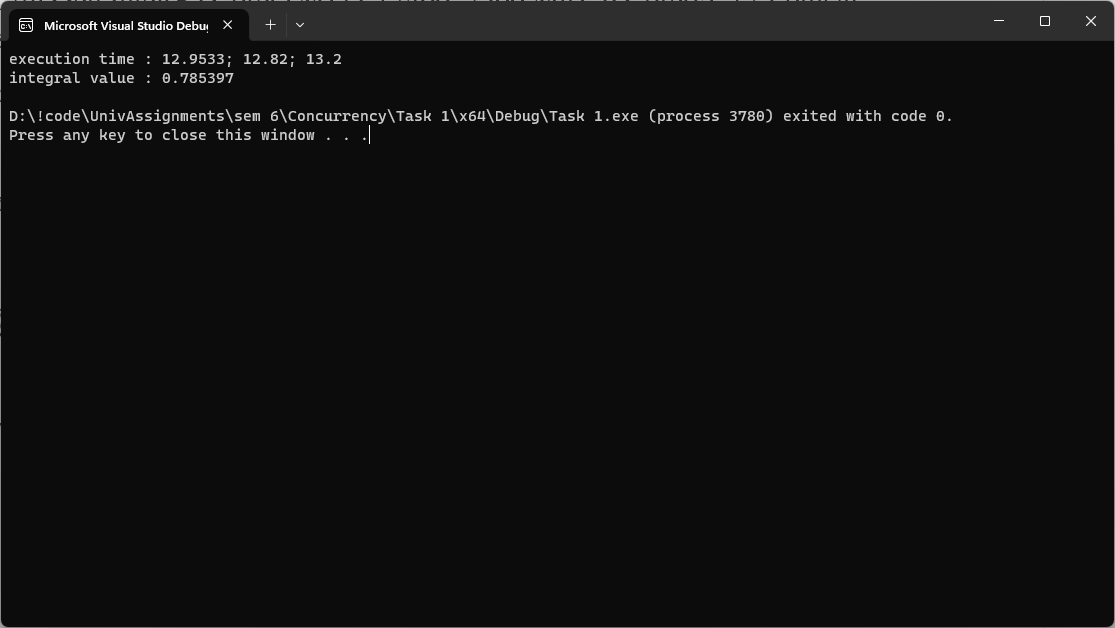
RAM: 16.0 GB со скоростью 2400MHz

Задание 1 (вариант 6). Реализовать подсчёт следующего интеграла:

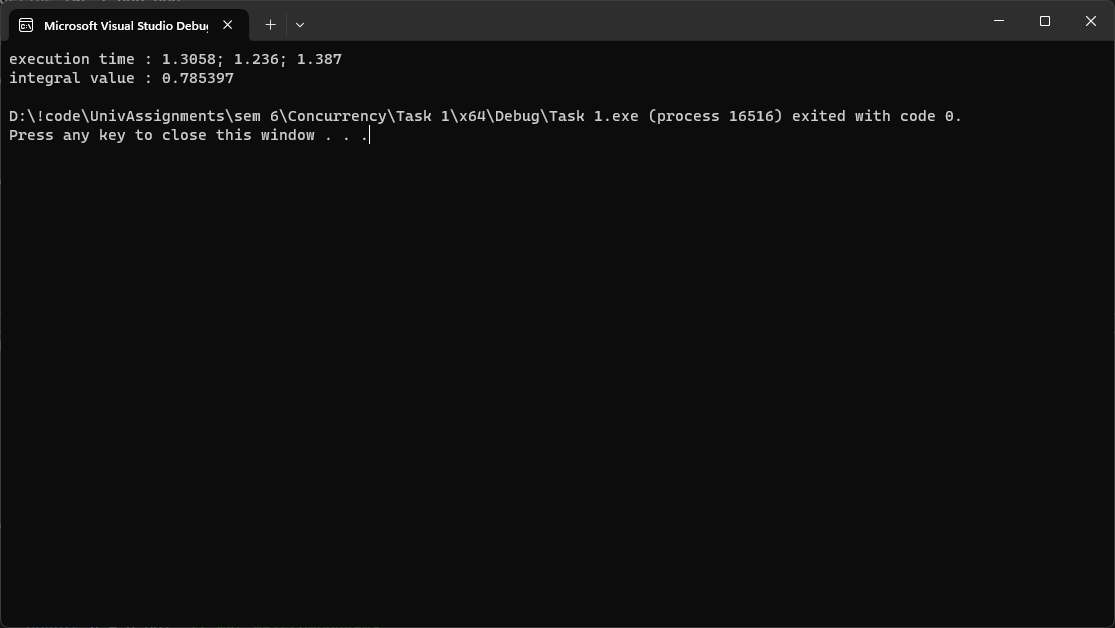


Совершим вычисления 10 раз, посчитаем среднее.

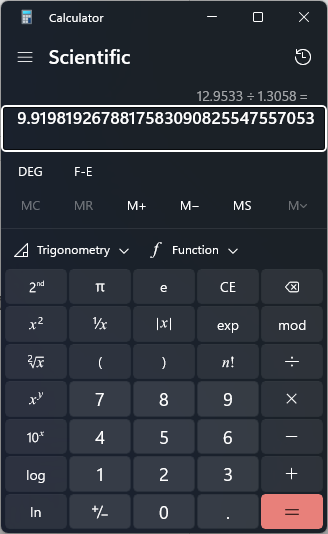
Результат решения (без применения параллелизма, методом прямоугольников):



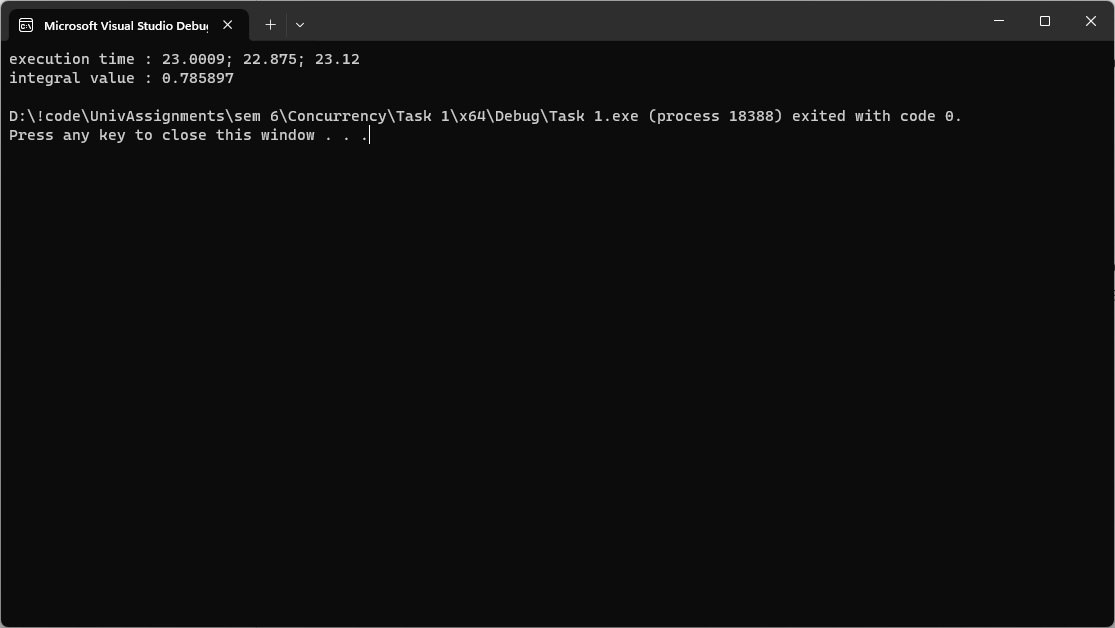
Результат решения (с применением параллелизма, методом прямоугольников):



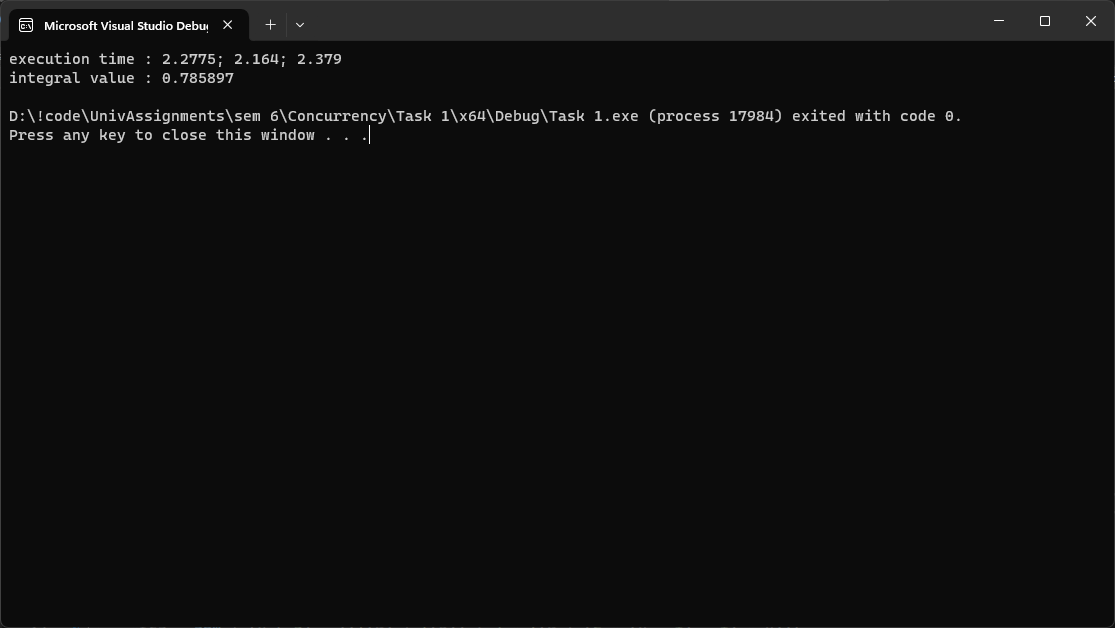
Выигрыш в скорости почти в 10 раз, что соотносится с количеством логических процессоров системы (12): часть ресурсов требуется на диспетчеризацию задач между потоками.



Результат решения (без применения параллелизма, метод Симпсона):



Результат решения (с применением параллелизма, метод Симпсона):



Отношение практические такое же:



Метод прямоугольников:

#include <iostream>

#include <omp.h>

#include <vector>

#define PI 3.1415926535897932384626433832795

#define INF 1'000'000

using namespace std;

double f(double x)

{

return 1 / (x \* x + 4 \* x + 5);

}

void integral(const double a, const double b,

const double h, double\* res)

{

int i, n;

double sum;

double x;

n = (int)((b - a) / h);

sum = 0.0;

#pragma omp parallel for private(x) reduction(+: sum)

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

{

x = a + i \* h + h / 2.0;

sum += f(x) \* h;

}

\*res = sum;

}

double experiment(double\* res)

{

double stime, ftime; // время начала и конца расчета

double a = -1.0; // левая граница интегрирования

double b = INF; // правая граница интегрирования

double h = 0.001; // шаг интегрирования

stime = clock();

integral(a, b, h, res); // вызов функции интегрирования

ftime = clock();

return (ftime - stime) / CLOCKS\_PER\_SEC;

}

int main()

{

int i; // переменная цикла

double time; // время проведенного эксперимента

double res; // значение вычисленного интеграла

double min\_time;

double max\_time;

double avg\_time;

int numbExp = 10;

min\_time = max\_time = avg\_time = experiment(&res);

// оставшиеся запуски

for (i = 0; i < numbExp - 1; i++)

{

time = experiment(&res);

avg\_time += time;

if (max\_time < time) max\_time = time;

if (min\_time > time) min\_time = time;

}

// вывод результатов эксперимента

cout << "execution time : " << avg\_time / numbExp << "; " <<

min\_time << "; " << max\_time << endl;

cout << "integral value : " << res << endl;

return 0;

}

Метод Симпсона:

#include <iostream>

#include <omp.h>

#include <vector>

#define PI 3.1415926535897932384626433832795

#define INF 1'000'000

using namespace std;

double f(double x)

{

return 1 / (x \* x + 4 \* x + 5);

}

void integral(const double a, const double b,

const double h, const long long int n, double\* res)

{

long long int i;

double sum = 0;

#pragma omp parallel for private(x) reduction(+: sum)

for (i = 0; i < n - 1; i++)

{

sum += (h / 3) \* (4 \* f(a + (2 \* i - 1) \* h) + 2 \* f(a + 2 \* i \* h));

}

\*res = sum + (h / 3) \* ((f(a) + f(b)) + 4 \* f(a + (2 \* (n - 1) - 1) \* h));

}

double experiment(double\* res)

{

double stime, ftime; // время начала и конца расчета

double a = -1.0; // левая граница интегрирования

double b = INF; // правая граница интегрирования

long long int n = 1'000'000'000;

double h = (b - a) / (2 \* n); // шаг интегрирования

stime = clock();

integral(a, b, h, n, res); // вызов функции интегрирования

ftime = clock();

return (ftime - stime) / CLOCKS\_PER\_SEC;

}

int main()

{

int i; // переменная цикла

double time; // время проведенного эксперимента

double res; // значение вычисленного интеграла

double min\_time;

double max\_time;

double avg\_time;

int numbExp = 10;

min\_time = max\_time = avg\_time = experiment(&res);

// оставшиеся запуски

for (i = 0; i < numbExp - 1; i++)

{

time = experiment(&res);

avg\_time += time;

if (max\_time < time) max\_time = time;

if (min\_time > time) min\_time = time;

}

// вывод результатов эксперимента

cout << "execution time : " << avg\_time / numbExp << "; " <<

min\_time << "; " << max\_time << endl;

cout << "integral value : " << res << endl;

return 0;

}