МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГБОУ ВО «НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет автоматики и вычислительной техники

Кафедра АСУ

ОТЧЕТ

по лабораторным работам и расчетно-графической работе

Дисциплина «Параллельное программирование», 5 семестр

Преподаватель: _Ландовский В.В.___

Группа: ____АП-026____

Студенты:

Почуев Н. А.

Кусургашев Р. А.

г. Новосибирск

2022 год

Лабораторная работа 1

Цель работы: создать программу, которая выполняет интегрирование, методом средних прямоугольников, с использованием распараллеливания процессов.

Описание задачи

Реализовать заданный метод численного интегрирования на языке C++ с использованием стандарта POSIX (функций pthread_create, pthread_join).
Входные параметры: пределы интегрирования, количество потоков (от 1 до 8), величина шага или количество шагов.
Выходная информация: значение интеграла, время, затраченное на вычисления.
Подынтегральная функция выбирается самостоятельно.

Блок-схема описание алгоритма:

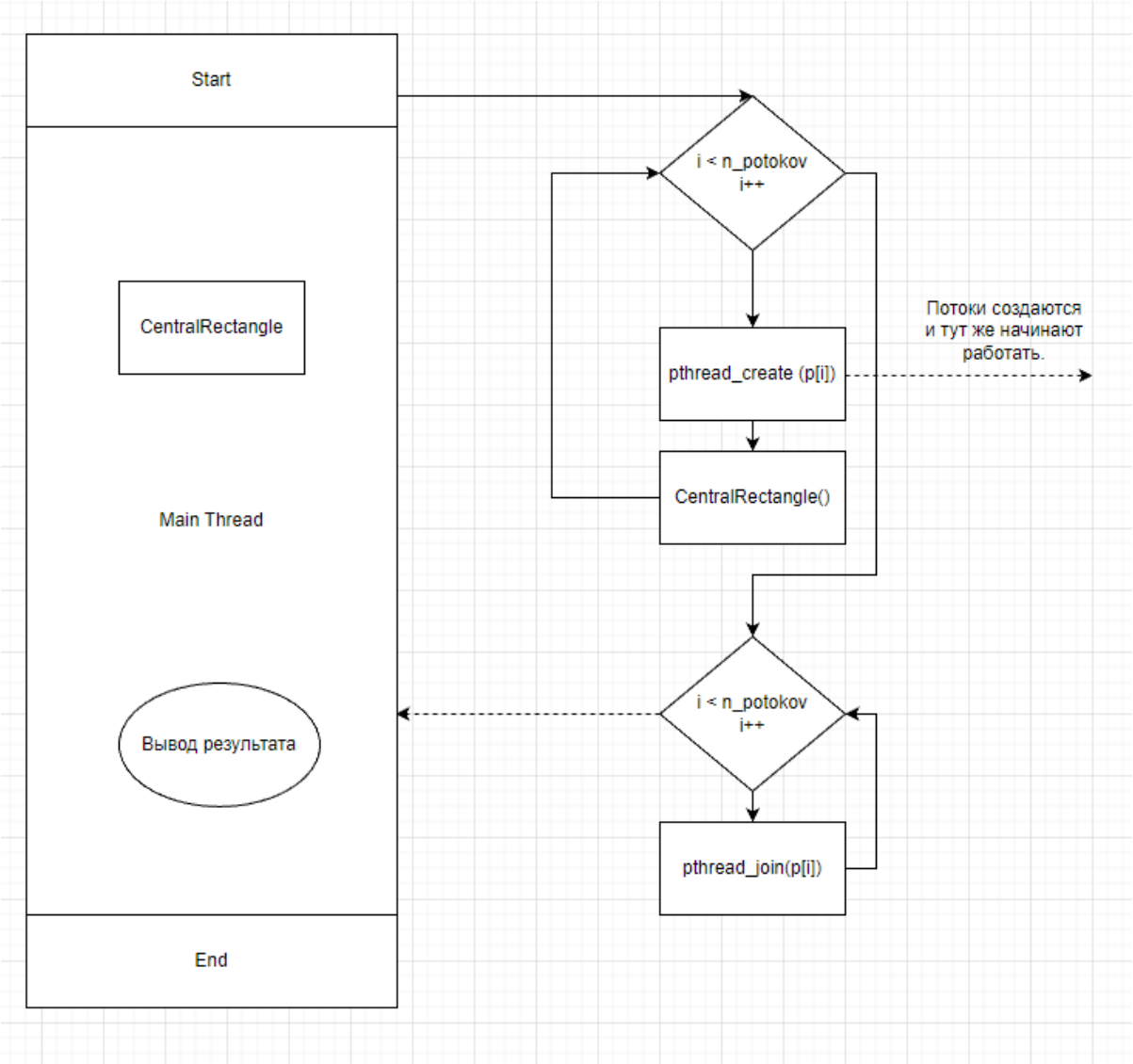

Рисунок 1. Блок-схема основной программы

Пример работы программы:

F(x) = 10 - x

```
Кол-во потоков: 1
a= 100 b= 300 n= 10000000
 Ответ: -38000
 Время работы потоков: 0.506
```

Рисунок 2. Вывод программы при использовании 1 потока

```
Кол-во потоков: 2
a= 100 b=
a= 200 b= 300 n= 5000000200 n= 5000000
 Ответ: -38000
 Время работы потоков: 0.263
```

Рисунок 3. Вывод программы при использовании 2 потоков

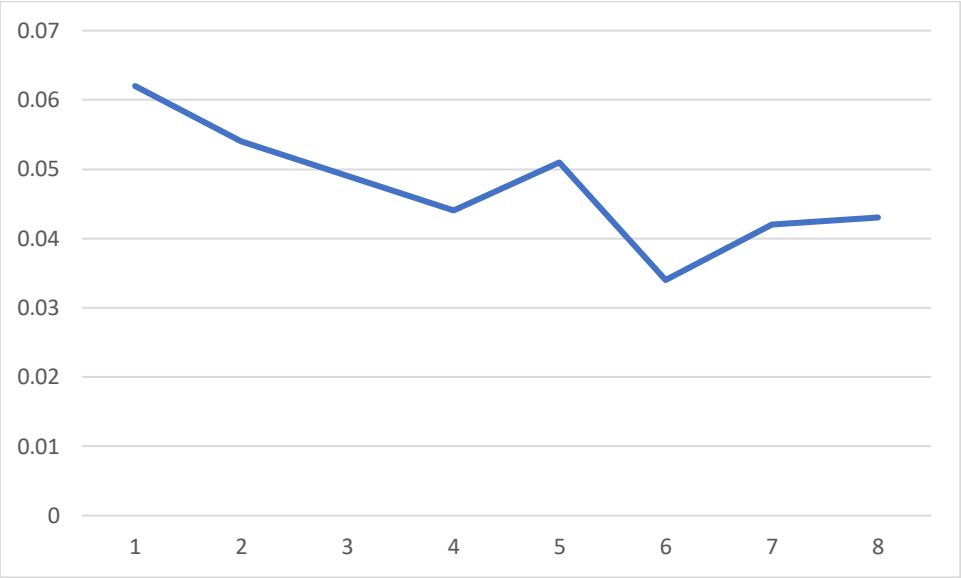

Рисунок 4. Зависимость времени от количества потоков, n = 1000000

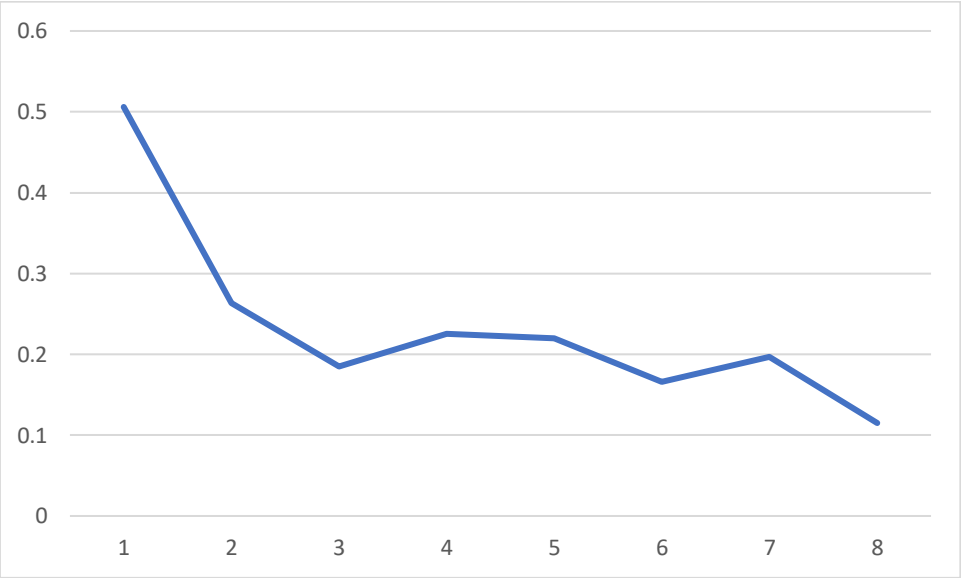

Рисунок 5. Зависимость времени от количества потоков, n = 10000000

Вывод: как мы видим, с увеличением шагов зависимость времени от количества потоков становится более наглядной, а также, что значимые изменения приходятся на смену потоков из 1 в 2 и из 2 в 3, далее же производительность растёт менее заметно.

Листинг программы:

```
#include "pthread.h"
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <omp.h>
#pragma comment(lib,"pthreadVCE2.lib")
double f(double x) {
```

```
	return (10 - x);
}

struct Param
{
	double a;
	double b;
	int n;
	double sum;
};

void* CentralRectangle(void* all_param)
{

	Param* param = (Param*)all_param;

	double a = param->a;
	double b = param->b;
	int n = param->n;

	std::cout << "\na= " << a << " b= " << b << " n= " << n;



	const double h = (b - a) / n;
	double sum = (f(a) + f(b)) / 2;
	for (int i = 1; i < n; i++) {
		double x = a + h * i;
		sum = sum + f(x);
	}
	double result = h * sum;
	param->sum = result;
	return 0;
}

int main()
{
	setlocale(LC_ALL, "Russian");


	double a = 100;
	double b = 300;
	int n = 10000000;
	const int n_potokov = 8;

	std::cout << "Кол-во потоков: " << n_potokov;
	struct Param arr_parall[n_potokov];
	pthread_t p[n_potokov];

	clock_t start = clock();

	double b_h = (b - a) / n_potokov;

	for (int i = 0; i < n_potokov; i++)
	{
		arr_parall[i].a = a + b_h * i;
		arr_parall[i].b = a + b_h * (i + 1);
		arr_parall[i].n = n / n_potokov;
		arr_parall[i].sum = 0;
		pthread_create(&p[i], NULL, CentralRectangle, &arr_parall[i]);
	}


	double sum = 0;
```

```
        for (int i = 0; i < n_potokov; i++)
        {
                pthread_join(p[i], 0);
                sum += arr_parall[i].sum;
        }



        std::cout << "\n Ответ: " << sum;
        clock_t end = clock();
        double seconds = (double)(end - start) / CLOCKS_PER_SEC;
        std::cout << "\n Время работы потоков: " << seconds;

        return 0;


}
```

Лабораторная 2

Цель работы: выполнить задание лабораторной работы №1 используя OpenMP (директивы распараллеливания цикла #pragma omp parallel for с опциями num_threads и reduction).

Блок-схема описание алгоритма:

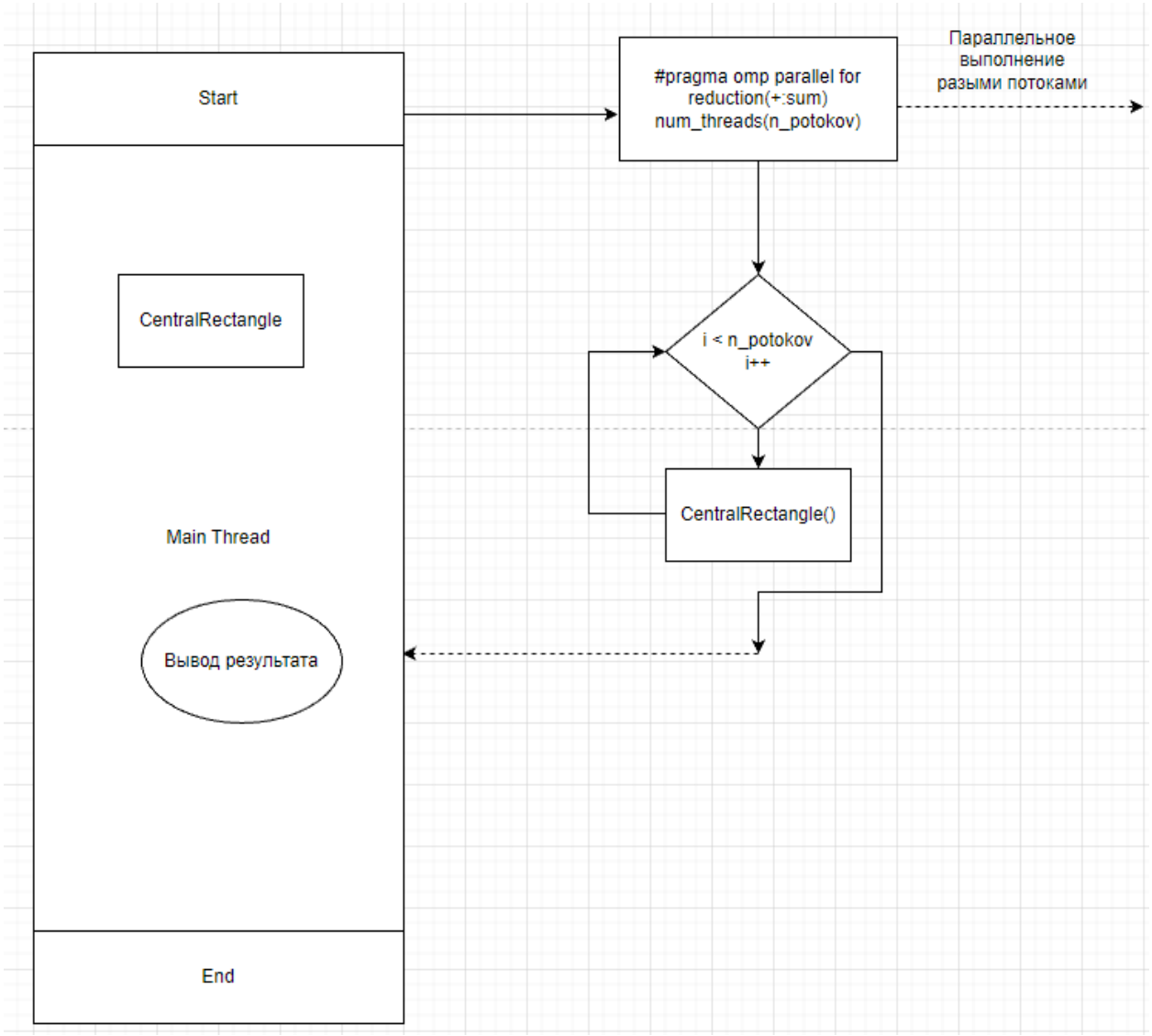

Рисунок 6. Блок-схема основной программы

Пример работы программы:

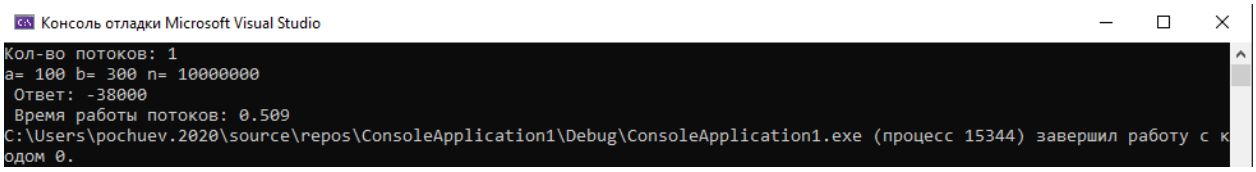

Рисунок 6. Выполнение программы, количество потоков = 1

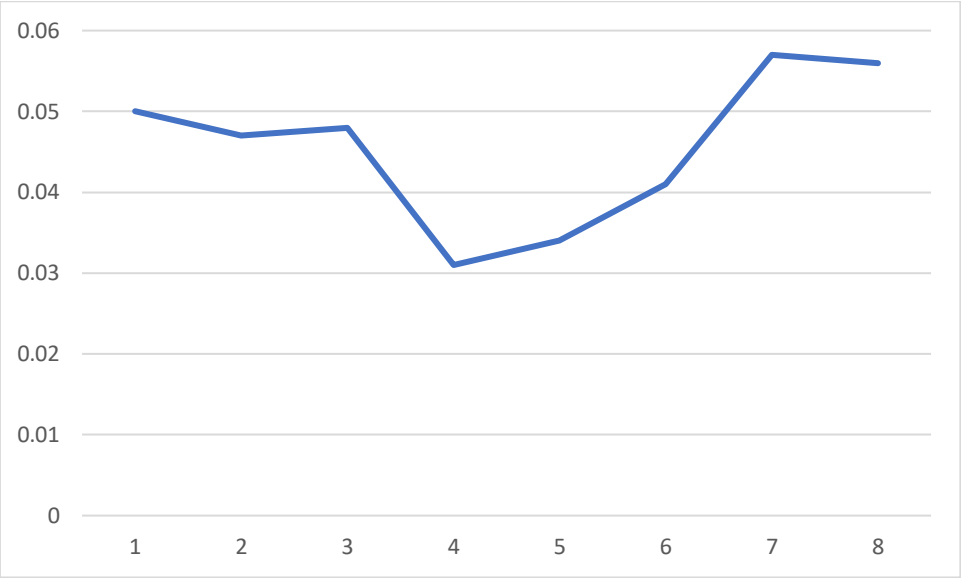

Рисунок 7. График зависимости времени от количества потоков, n = 1000000

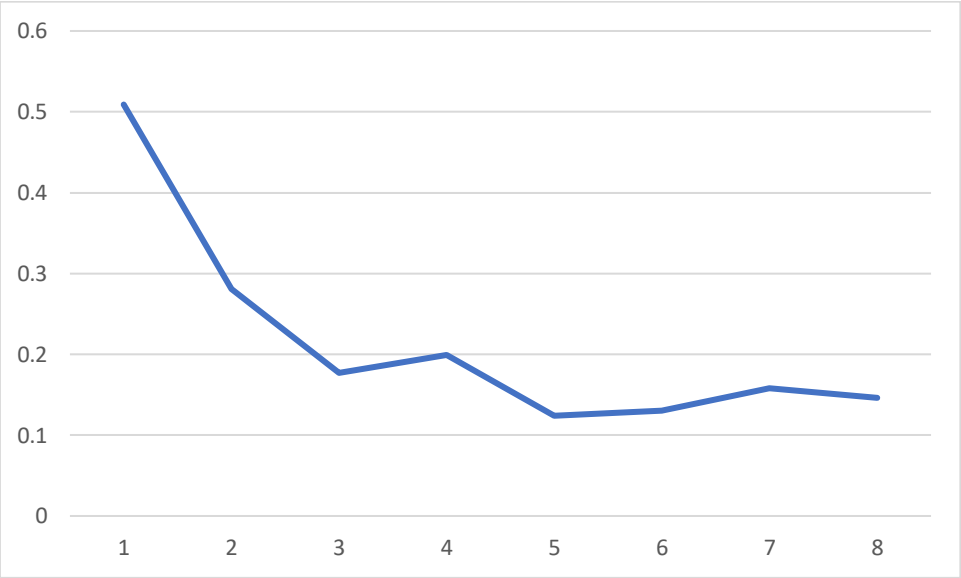

Рисунок 8. График зависимости времени от количества потоков, n = 10000000

Вывод: как мы видим, с увеличением шагов зависимость времени от количества потоков становится более наглядной, а также, что значимые изменения приходятся на смену потоков из 1 в 2 и из 2 в 3, далее же производительность растёт менее заметно.

Листинг программы:

```
#include "pthread.h"
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <omp.h>
#pragma comment(lib,"pthreadVCE2.lib")
double f(double x) {
```

```
	return (10 - x);
}

struct Param
{
	double a;
	double b;
	int n;

};

double CentralRectangle(void* all_param)
{

	Param* param = (Param*)all_param;

	double a = param->a;
	double b = param->b;
	int n = param->n;

	std::cout << "\na= " << a << " b= " << b << " n= " << n;




	const double h = (b - a) / n;
	double sum = (f(a) + f(b)) / 2;
	for (int i = 1; i < n; i++) {
		double x = a + h * i;
		sum = sum + f(x);
	}
	double result = h * sum;

	return result;
}

int main()
{
	setlocale(LC_ALL, "Russian");


	double a = 100;
	double b = 300;
	int n = 1000000;
	const int n_potokov = 8;

	std::cout << "Кол-во потоков: " << n_potokov;
	struct Param arr_parall[n_potokov];
	pthread_t p[n_potokov];

	clock_t start = clock();

	double b_h = (b - a) / n_potokov;
	double sum = 0;

#pragma omp parallel for reduction(+:sum) num_threads(n_potokov)
	for (int i = 0; i < n_potokov; i++)
	{
		arr_parall[i].a = a + b_h * i;
		arr_parall[i].b = a + b_h * (i + 1);
		arr_parall[i].n = n / n_potokov;
		sum += CentralRectangle(&arr_parall[i]);

	}
```

```
        std::cout << "\n Ответ: " << sum;
        clock_t end = clock();
        double seconds = (double)(end - start) / CLOCKS_PER_SEC;
        std::cout << "\n Время работы потоков: " << seconds;

        return 0;


}
```

Лабораторная работа 3

Задача: Программа решает множество независимых однотипных задач. В варианте задается модель распределения работы между потоками и тип решаемых задач. В каждом варианте для простоты предполагается, что условия задач (входные данные) расположены в файле, причем сложность задач может сильно отличаться друг от друга и заранее разделить их на равные по объему вычислений части невозможно. Результаты должны записываться в выходные файлы. Использовать POSIX Threads или C++ threads.

МОДЕЛЬ: конвейер. Первый поток читает входной файл, второй вычисляет результаты, третий записывает выходной файл.

нод - наибольший общий делитель двух чисел, исходные данные - пара чисел, результат - число.

Пример работы программы:

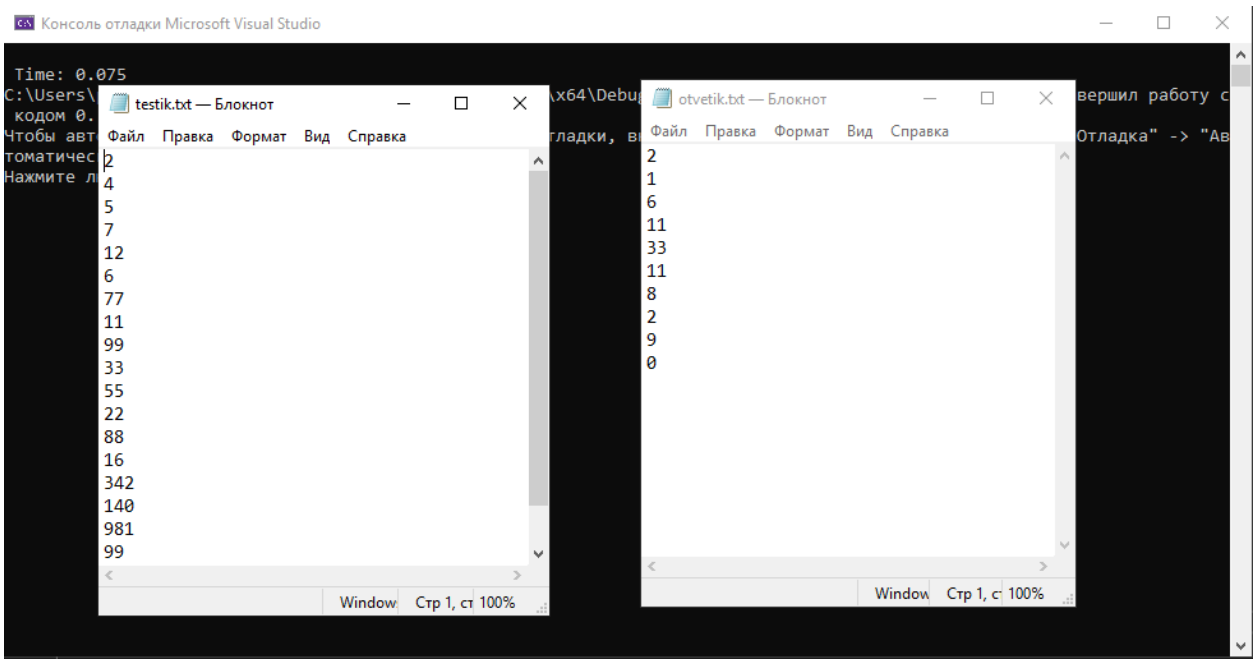

Рисунок 9. Выполнение программы

Для получения полной картины заполним входной файл на 300 строк.

Рисунок 10.Время с использованием mutex

Рисунок 11. Время, если бы потоки работали последовательно один за другим

Вывод: использование модели конвейера сильно увеличивает производительность, чем больший объём данных у нас есть, тем большую разницу мы увидим, реализация модели была произведена с помощью mutex.

Листинг программы:

```
#include <iostream>
#include <filesystem>
#include <stdlib.h>
#include "pthread.h"
#include <string.h>
#include <thread>
#include <mutex>
#include <vector>
#include <fstream>
#include <condition_variable>
#include <string>
#include <queue>
#include <mutex>
```

```
#include <sstream>
#include <clocale>
#include <concurrent_queue.h>
using namespace std;
mutex mtx;
mutex mtx2;


queue<string> queue1;
queue<string> queue2;




int NOD(int a, int b)
{
	while (a > 0 && b > 0)

		if (a > b)
			a %= b;

		else
			b %= a;

	return a + b;
}

template <typename T>
std::string toString(T val)
{
	std::ostringstream oss;
	oss << val;
	return oss.str();
}


void threadaction1(string s) {
	string line;
	ifstream in;
	in.open(s);
	while (getline(in, line))
	{
		mtx.lock();
		queue1.push(line);
		mtx.unlock();
	}
	in.close();
	mtx.lock();
	queue1.push("выход");
	queue1.push("выход");
	mtx.unlock();
}

void threadaction2()
{

	while (true)
	{
		string data1;
		string data2;
		int x1;
		int x2;
```

```
		if (!queue1.empty()) {

			if (queue1.front() == "выход") {
				mtx2.lock();
				queue2.push("выход");
				mtx2.unlock();
				break;
			}
			else {
				mtx.lock();
				data1 = queue1.front();
				queue1.pop();
				data2 = queue1.front();
				queue1.pop();
				mtx.unlock();
			}
		}
		else continue;

			const char* s1 = data1.c_str();
			const char* s2 = data2.c_str();

		x1 = atoi(s1);
		x2 = atoi(s2);

			mtx2.lock();
			queue2.push(toString(NOD(x1, x2)));
			mtx2.unlock();


		}

	}

void threadaction3(string s)
{

	string data;
		while (true)
		{

			if (queue2.empty())continue;
			else
			{

				string line;
				mtx2.lock();
				line = queue2.front();
				queue2.pop();
				mtx2.unlock();
				if (line == "выход") { break; }
				ofstream out;
				out.open(s, ios::app);


				out << line << endl;
				out.close();

			}

		}
}
int main() {

	clock_t start = clock();
```

```
        thread t1(threadaction1, "testik.txt");
        thread t2(threadaction2);
        thread t3(threadaction3, "otvetik.txt");

        t1.join();
        t2.join();
        t3.join();

        clock_t end = clock();
        double seconds = (double)(end - start) / CLOCKS_PER_SEC;
        std::cout << "\n Time: " << seconds;

        return 0;
}
```

Лабораторная работа 4

Цель работы

Реализовать параллельный алгоритм численного интегрирования методом трапеций с помощью MPI.

Вариант 5:

Границы интервалов вычисляются каждым процессом на основе общего числа процессов и своего номера;

Окончательный результат получают все процессы используя allreduce.

Текст программы

```
#include <iostream>
#include <clocale>
#include "mpi.h"

double function(double x)
{
   return sqrt((10* x * x + 9) / (x * x + 13));
}

double trapezium_function(double a, double b)
{
   return (function(a) + function(b)) / 2 * (b - a);
}


double Integral_Method(double a, double b, int d)
{
   const double width = (b - a) / (d * 1.0);
   double integral = 0;
   for (size_t i = 0; i < d; i++) {
      const double x1 = a + i * width;
      const double x2 = a + (i + 1) * width;
      integral += trapezium_function(x1, x2);
   }
   return integral;
}

int main(int argc, char** argv)
{
   setlocale(LC_ALL, "Rus");
   double start, end;
   int numtasks, rank;
   int a = 10;
```

```
    int b = 30;
    double d = 10000;
    double Result;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    const double width = (b - a) / (numtasks * 1.0);
    const double x1 = a + rank * width;
    const double x2 = a + (rank + 1) * width;
    int sum = 0;
    int d_now;
    for (size_t i = 0; i < numtasks; i++)
    {
        double length = (d - sum) / (numtasks - i);
        int new_d = round(length);
        sum += new_d;
        if (i == rank)
            d_now = new_d;
    }
    start = MPI_Wtime();
    double res = Integral_Method(x1, x2, d_now);
    end = MPI_Wtime();
    MPI_Allreduce(&res, &Result, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    double time_to_solve = end - start;

    MPI_Finalize();
    if (rank == 0)
    {


       // std::cout << "Result  = " << Result << "   Wasted time = " << (double)time_to_solve << std::endl;
        printf("Result = %f,  Wasted Time =  %f s\n", Result, time_to_solve);

    }
}
```

Примеры работы программы.

```
C:\Users\rkusu\source\repos\pp_lab4\x64\Debug>mpiexec -n 1 pp_lab4.exe
Result = 62,027528,  Wasted Time =  0,000244 s

C:\Users\rkusu\source\repos\pp_lab4\x64\Debug>mpiexec -n 2 pp_lab4.exe
Result = 62,027528,  Wasted Time =  0,000123 s

C:\Users\rkusu\source\repos\pp_lab4\x64\Debug>mpiexec -n 3 pp_lab4.exe
Result = 62,027528,  Wasted Time =  0,000082 s

C:\Users\rkusu\source\repos\pp_lab4\x64\Debug>mpiexec -n 4 pp_lab4.exe
Result = 62,027528,  Wasted Time =  0,000063 s

C:\Users\rkusu\source\repos\pp_lab4\x64\Debug>_
```

Рисунок 12. Пример выполнения программы

Результаты экспериментов
Входные данные:

Границы от 10 до 30
Количество отрезков – 10000

Подынтегральная функция:

$$f(x) = \sqrt{\frac{10x^2+9}{x^2+13}}$$

	1	2	3	4	5	6	7	8
Затраченное время	0,000244	0,000123	0,000082	0,000068	0,000063	0,000056	0,000051	0,000049
Значение интеграла	62,027528	62,027528	62,027528	62,027528	62,027528	62,027528	62,027528	62,027528

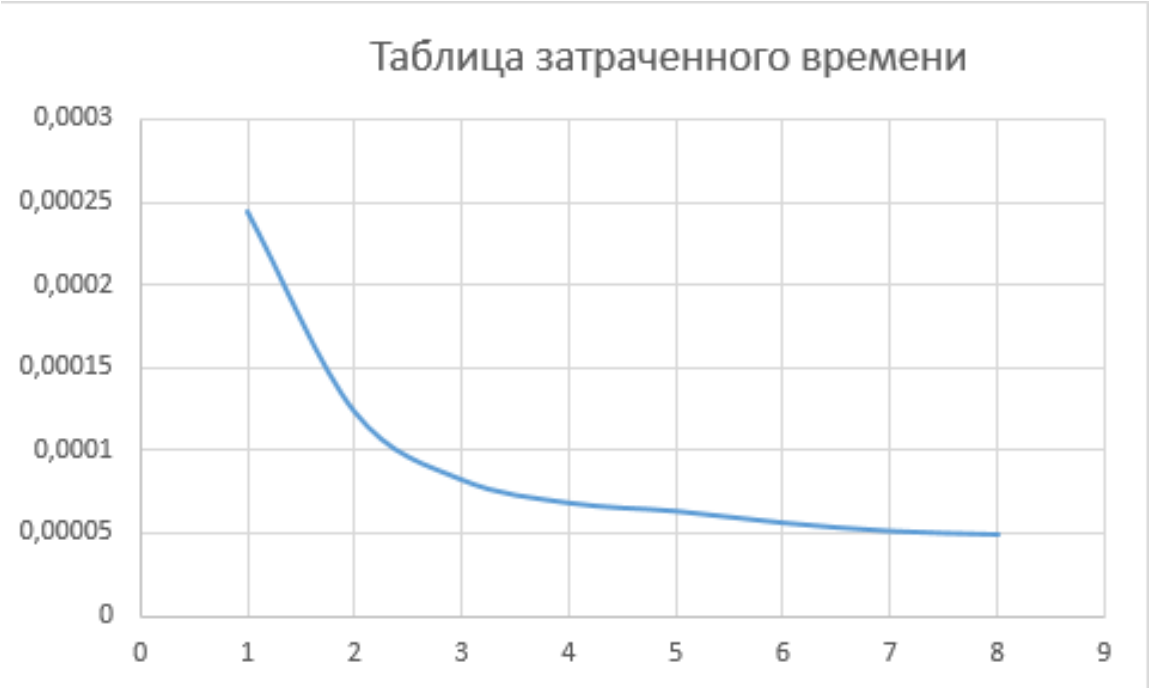

Рисунок 13. График изменения времени выполнения в зависимости от количества процессов

Значение интеграла не изменялось, несмотря на увеличение числа процессов.

Выводы

MPI - очень быстрая и простая библиотека для работы с многопоточностью. В ходе выполнения лабораторной работы, с увеличением числа процессов, время на исполнение уменьшалось. В то время как значение интеграла было правильно найдено при 1 процессе. И в дальнейшем это значение не изменялось.

РГЗ

Задача: выполнить задание лабораторной работы №3 средствами C#.

Блок-схема описание алгоритма:

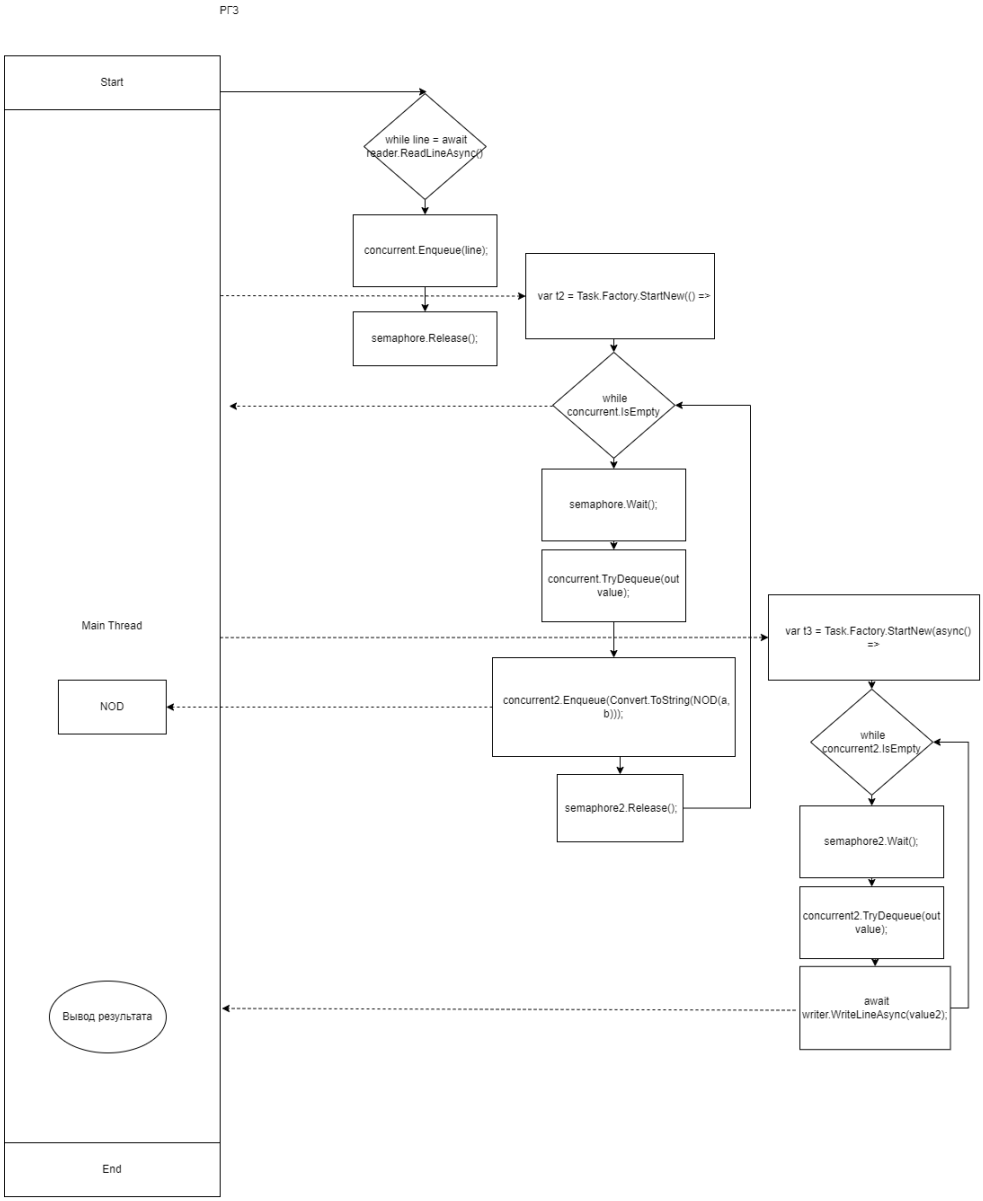

Рисунок 14. Блок-схема основной программы

Пример работы программы:

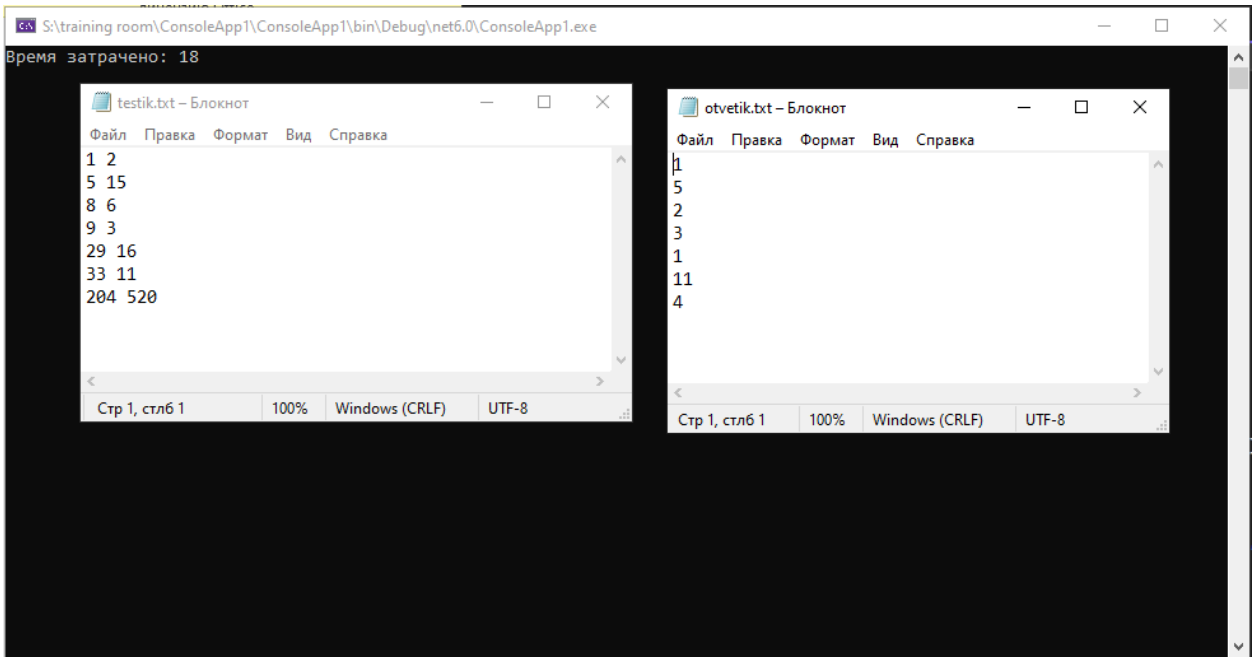

Рисунок 15. Итог программы

Рассмотрим разницу во времени, если у нас потоки работают по модели конвейера, а то есть первый поток читает входной файл, второй вычисляет результаты, третий записывает выходной файл и, если бы каждый поток ждал выполнения предыдущего. Во входной файл запишем 300 значений для наглядности.

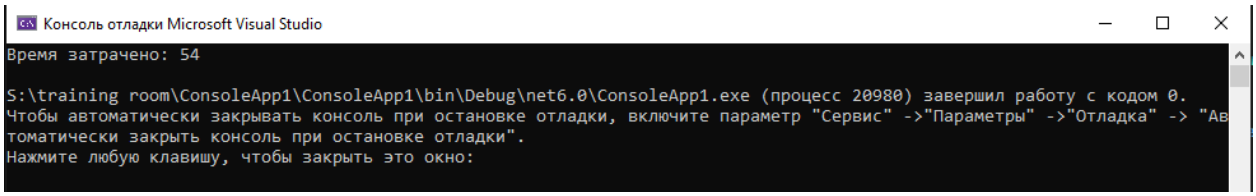

Рисунок 16. Работа приложения

Рисунок 17. Работа приложения при отсутствии конвейера

Как мы видим конвейер работает лучше, проверим с файлом в 1500 значений.

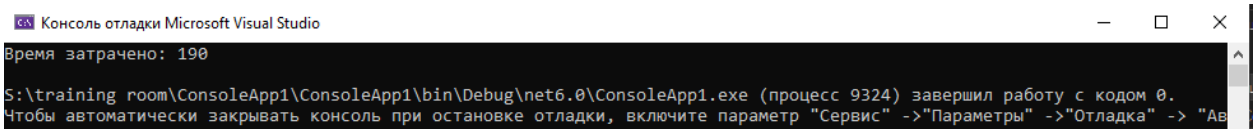

Рисунок 18. Работа приложения

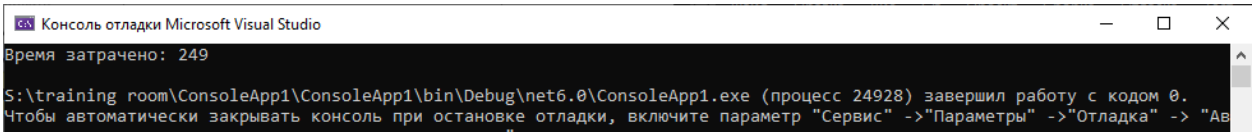

Рисунок 19. Работа приложения при отсутствии конвейера

Выводы: использование модели конвейера сильно увеличивает производительность, чем больший объём данных у нас есть, тем большую разницу мы увидим, реализация модели была произведена с помощью semaphore.

Листинг программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using System.IO;
using System.Collections.Concurrent;

class Program
{


    static int NOD(int x, int y)
    {
        while (x != y)
        {
            if (x > y)
                x = x - y;
            else
                y = y - x;
        }
        return x;
    }

    static void Main(string[] args)
    {
        //$"{Environment.CurrentDirectory}\\list.txt";
        string path = $"{Environment.CurrentDirectory}\\testik.txt";
        string path2 = $"{Environment.CurrentDirectory}\\otvetik.txt";

        var semaphore = new SemaphoreSlim(0);
        var semaphore2 = new SemaphoreSlim(0);

        var concurrent = new ConcurrentQueue<string>();
        var concurrent2 = new ConcurrentQueue<string>();


        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();


        Thread myThread0 = new Thread(threadaction0);
        Thread myThread1 = new Thread(threadaction1);
        Thread myThread2 = new Thread(threadaction2);

        void threadaction0()
        {
```

```
    using (StreamReader reader = new StreamReader(path))

    {

        string line;

        while ((line = reader.ReadLine()) != null)
        {

            concurrent.Enqueue(line);
            semaphore.Release();
        }


    }
}


void threadaction1()
{
    semaphore.Wait();

    while (concurrent.IsEmpty == false)
    {

        string value;
        concurrent.TryDequeue(out value);
        string[] xxx = value.Split(char.Parse(" "));
        int a = Convert.ToInt32(xxx[0]);
        int b = Convert.ToInt32(xxx[1]);

        concurrent2.Enqueue(Convert.ToString(NOD(a, b)));
        semaphore2.Release();
    }
}



void threadaction2()
{
    semaphore2.Wait();
    while (concurrent2.IsEmpty == false)
    {
        using (StreamWriter writer = new StreamWriter(path2, true))
        {

            string value2;
            concurrent2.TryDequeue(out value2);
            writer.WriteLineAsync(value2);
        }
    }




    stopwatch.Stop();
    Console.WriteLine($"Время затрачено: {stopwatch.ElapsedMilliseconds}");
    Console.ReadKey();
}


myThread0.Start();
myThread1.Start();
myThread2.Start();

myThread0.Join();
```

```
        myThread1.Join();
        myThread2.Join();


    }
}
```