НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра параллельных вычислительных технологий

ОТЧЕТ ПРИНЯТ

Оценка _____

_____ / _____ /

(подпись) (расшифровка)

«___» _____ _____ г.

ОТЧЕТ

по лабораторной работе № 3

Параллельные циклы

по дисциплине Параллельное программирование

Студенты гр. ПМИ-02

Дюков Богдан Витальевич,

Сидоров Даниил Игоревич

Новосибирск-2024

I. Цель работы

Знакомство со спецификацией прикладного программного интерфейса OpenMP, применением OpenMP для распараллеливания циклов, использованием компиляторов, реализующих спецификацию OpenMP.

II. Ход работы

1. Скомпилировали и запустили программу из примера 19.

Текст программы:

```
#include <omp.h>
#include <iostream>

using namespace std;

int main()
{
	int a[100];
	int b[100];

	// Инициализация массива b
	for(int i = 0; i < 100; i++)
	{
		b[i] = i;
	}

	// Директива OpenMP для распараллеливания цикла
	#pragma omp parallel for
	for(int i = 0; i < 100; i++)
	{
		a[i] = b[i];
		b[i] = 2 * a[i];
	}

	int result = 0;

	// Далее значения a[i] и b[i] используются, например, так:
	#pragma omp parallel for reduction(+ : result)
	for(int i = 0; i < 100; i++)
	{
		result += (a[i] + b[i]);
	}

	cout << "Result = " << result << endl;

	return 0;
}
```

Результаты работы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp -O3 Task_1.cpp -o Task_1
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_1
Result = 14850
```

Данная программа вычисляет утроенную сумму целых чисел от 0 до 99.

2. Модифицировали программу, заменив a[i] = b[i] в теле цикла на a[i] = f(b[i]), где f – некоторая чистая функция, задающая достаточно сложное вычисление, чтобы время ее работы составляло порядка 1 секунды.

Пусть функция f принимает параметр b[i] и выполняет с его копией бессмысленные операции сложения и вычитания. Отметим, что в результате функция возвращает значение, равное исходному значению b[i] (от чего результат работы программы будет совпадать с предыдущим пунктом). Время работы функции составляет чуть больше 1 секунды.

Текст программы:

```
#include <omp.h>
#include <iostream>

#define N 40000

using namespace std;

// Функция, выполняющая сложные вычисления
int f(int x)
{
	int result = x;

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

		for(int j = 0; j < N; j++)
		{
			result += 3;
		}

		result -= 3 * N;
	}

	return result;
}

int main()
{
	int a[100];
	int b[100];

	// Инициализация массива b
	for(int i = 0; i < 100; i++)
	{
		b[i] = i;
	}

	#pragma omp parallel for
	for(int i = 0; i < 100; i++)
	{
		// Теперь a[i] вычисляет функцию от b[i]
		a[i] = f(b[i]);
		b[i] = 2*a[i];
	}
```

```
	int result = 0;

	#pragma omp parallel for reduction(+ : result)
	for(int i = 0; i < 100; i++)
	{
		result += (a[i] + b[i]);
	}

	cout << "Result = " << result << endl;

	return 0;
}
```

Запустили в терминальном окне программу top, перешли в режим отображения загрузки отдельных ядер вычислительной системы, нажав клавишу «1». Загрузка ядер до запуска программы:

```
top - 21:22:00 up  4:26,  1 user,  load average: 0.22, 0.06, 0.02
Tasks:  45 total,   1 running,  44 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.3 us,  1.0 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :  1.0 us,  0.0 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15930.4 total,  14613.2 free,    553.2 used,    764.0 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  15109.1 avail Mem
```

Видим, что ядра процессора практически не тратят свое время на пользовательские процессы.

В другом окне выполняем компиляцию модифицированной программы с игнорированием директив OpenMp (компиляция будет выполняться без ключа -O3, потому что иначе компилятор заметит глупизну действий в функции f и будет выполнять её моментально) и ее запуск:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ Task_23.cpp -o Task_23
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_23
Result = 14850
```

Загрузка ядер во время работы программы:

```
top - 21:22:27 up  4:27,  1 user,  load average: 0.22, 0.07, 0.02
Tasks:  46 total,   2 running,  44 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15930.4 total,  14612.5 free,    553.9 used,    764.0 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  15108.4 avail Mem
```

Ожидаемо видим, что одно ядро (в данном случае CPU4) полностью используется на 100% в процессах пользовательского пространства, тогда как остальные ядра не загружены.

3. Скомпилировали программу с поддержкой OpenMP и запустили её:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp Task_23.cpp -o Task_23
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_23
Result = 14850
```

Программа стала работать быстрее в разы (заметно без замеров), что является первым хорошим знаком. Во время работы программы была проанализирована загрузка ядер:

```
top - 21:38:22 up  4:43,  1 user,  load average: 2.49, 0.58, 0.23
Tasks:  46 total,   2 running,  44 sleeping,   0 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15930.4 total,  14615.2 free,    551.1 used,    764.1 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  15111.2 avail Mem
```

Теперь интенсивно работают все ядра системы (за счет параллельности).

4. Изучили работу программ из примеров 1-6.

Рассмотрим каждую программу.

Программа из примера 1 демонстрирует нам применение частных и разделяемых переменных. Частные переменные уникальны для каждого потока, тогда как разделяемые переменные доступны всем потокам. В программе создается соответствующая параллельная область, где каждый поток выводит общее число потоков, значение разделяемой переменной массива (hello_string[]) и свой номер (данный номер хранится в частной переменной id). Немного модифицируем программу, чтобы был обеспечен корректный вывод.

Текст программы (только места с модификациями):

```
// Функция для корректного вывода
void print_data(string message)
{
	ostringstream out;
	out << message;
	cout << out.str();
}

// В функции main
// Директива OpenMP: объявление параллельной области
#pragma omp parallel private(id) shared(hello_string)
{
	int size = omp_get_num_threads();
	id = omp_get_thread_num();
	print_data("Number of threads = " + to_string(size) + "\n" + hello_string +
" " + to_string(id) + "\n");
} // Конец параллельной области
```

Результат работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_1
Initial thread: id = 1234, size = 0
Number of threads = 16
Hello World from thread 0
Number of threads = 16
Hello World from thread 2
Number of threads = 16
Hello World from thread 9
Number of threads = 16
Hello World from thread 5
Number of threads = 16
Hello World from thread 14
Number of threads = 16
Hello World from thread 13
Number of threads = 16
Hello World from thread 4
Number of threads = 16
Hello World from thread 12
Number of threads = 16
Hello World from thread 10
Number of threads = 16
Hello World from thread 6
Number of threads = 16
Hello World from thread 15
Number of threads = 16
Hello World from thread 11
Number of threads = 16
Hello World from thread 3
Number of threads = 16
Hello World from thread 1
Number of threads = 16
Hello World from thread 8
Number of threads = 16
Hello World from thread 7
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Число используемых потоков соответствует числу ядер процессора.

Программа из примера 2 демонстрирует применение опции reduction. Эта опция используется в параллельных областях для автоматического применения операции редукции (такой как сложение, умножение и т.д.) к переменным, которые обновляются внутри области. Результаты этих обновлений затем “сводятся” в одну переменную после завершения параллельной области.

В программе создается параллельная область с разделяемой переменной sum, которая используется в качестве переменной редукции с операцией сложения (каждый поток получит свою частную переменную sum). Каждый поток вычисляет сумму значений своего сегмента массива (проинициализированного целыми числами за параллельной областью). После завершения параллельной области, сумма частичных результатов добавляется к значению переменной sum начального потока. Итоговая сумма выводится на экран.

Текст программы (понадобится в дальнейших пунктах):

```
#include <omp.h>
#include <iostream>
#include <sstream>

using namespace std;

// Функция для корректного вывода
void print_data(string message)
{
	ostringstream out;
	out << message;
	cout << out.str();
}

int main()
{
	int sum = 0;
	const int a_size = 100;
	int a[a_size], id, size;

	for(int i = 0; i<100; i++)
	{
		a[i] = i;
	}

	// Применение опции reduction
	#pragma omp parallel private(id, size) reduction(+ : sum)
	{	// Начало параллельной области
		id = omp_get_thread_num();
		size = omp_get_num_threads();

		// Разделяем работу между потоками
		int integer_part = a_size / size;
		int remainder = a_size % size;

		int a_local_size = integer_part + ((id < remainder) ? 1 : 0);

		int start = integer_part * id + ((id < remainder) ? id : remainder);
		int end = start + a_local_size;
```

```
            // Каждый поток суммирует элементы
            // своей части массива
            for(int i = start; i < end; i++)
            {
                  sum += a[i];
            }

            print_data("Thread " + to_string(id) + ", partial sum = " +
to_string(sum) + "\n");
      }

      // Благодаря опции reduction сумма частичных
      // результатов добавлена к значению переменной
      // sum начального потока
      cout << "\nFinal sum = " << sum << endl;
      return 0;
}
```

Результат работы программы (аналогично примеру 1 модифицировали вывод):

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_2
Thread 0, partial sum = 21
Thread 10, partial sum = 399
Thread 4, partial sum = 183
Thread 12, partial sum = 471
Thread 9, partial sum = 363
Thread 13, partial sum = 507
Thread 3, partial sum = 168
Thread 14, partial sum = 543
Thread 5, partial sum = 219
Thread 1, partial sum = 70
Thread 11, partial sum = 435
Thread 6, partial sum = 255
Thread 8, partial sum = 327
Thread 7, partial sum = 291
Thread 2, partial sum = 119
Thread 15, partial sum = 579

Final sum = 4950
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Программа из примера 3 показывает 2 способа управления количеством потоков в параллельной области. Это можно сделать с использованием опции num_threads в конкретной параллельной области или с помощью функции omp_set_num_threads (тогда указанное число потоков будет относиться ко всем последующим параллельным областям в программе, если в них отсутствует соответствующая опция).

Результат работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_3
Number of threads = 2
pi (approximately) = 3.14104
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Программа из примера 4 показывает применение барьерной синхронизации. Это означает, что каждый поток, достигший этой директивы, останавливается и ждет, пока все остальные потоки не достигнут этой точки. После того как все потоки достигли барьера, они продолжают

выполнение. В программе каждый поток выводит информацию о себе. А благодаря барьерам эти выводы происходят в порядке возрастания номеров потоков.

Результаты работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_4
Number of threads = 8
Hello World from thread 0
Number of threads = 8
Hello World from thread 1
Number of threads = 8
Hello World from thread 2
Number of threads = 8
Hello World from thread 3
Number of threads = 8
Hello World from thread 4
Number of threads = 8
Hello World from thread 5
Number of threads = 8
Hello World from thread 6
Number of threads = 8
Hello World from thread 7
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Программа из примера 5 демонстрирует использование директивы #pragma omp for для автоматического распределения итераций цикла между потоками в OpenMP. Она показывает, как может быть сокращен код примера 2.

Результаты работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_5
Thread 5, partial sum = 219
Thread 4, partial sum = 183
Thread 12, partial sum = 471
Thread 7, partial sum = 291
Thread 3, partial sum = 168
Thread 1, partial sum = 70
Thread 9, partial sum = 363
Thread 10, partial sum = 399
Thread 6, partial sum = 255
Thread 8, partial sum = 327
Thread 2, partial sum = 119
Thread 13, partial sum = 507
Thread 15, partial sum = 579
Thread 14, partial sum = 543
Thread 0, partial sum = 21
Thread 11, partial sum = 435
#0 Final sum = 4950
#1 Final sum = 4950
```

Программа из примера 6 показывает недетерминированное поведение в параллельных секциях. Данные конструкции используются в тех случаях, когда вычисления могут быть выполнены параллельно, но естественным образом не могут быть представлены как итерации цикла.

Результаты работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Test_6
section_count 1
section_count 1
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

5. Проведем эксперименты с программой из примера 2. Перед этим немного модифицируем её: уменьшим число потоков в параллельной области до 8, а также добавим в вывод каждого потока информацию об обрабатываемом сегменте (значения start и end).

Текст программы (только части с модификациями):

```
// Модификация №1: Установка числа потоков
omp_set_num_threads(8);

#pragma omp parallel private(id, size) reduction(+ : sum)
{
	id = omp_get_thread_num();
	size = omp_get_num_threads();

	int integer_part = a_size / size;
	int remainder = a_size % size;

	int a_local_size = integer_part + ((id < remainder) ? 1 : 0);

	int start = integer_part * id + ((id < remainder) ? id : remainder);
	int end = start + a_local_size;

	for(int i = start; i < end; i++)
	{
		sum += a[i];
	}

	// Модификация №2: Дополнительная информация в выводе
	print_data("Thread " + to_string(id) + "\tpartial sum = " + to_string(sum) +
"\tstart = " + to_string(start) + "\tend = " + to_string(end) + "\n");
}
```

Результаты работы модифицированной программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_5
Thread 0        partial sum = 78        start = 0       end = 13
Thread 2        partial sum = 416       start = 26      end = 39
Thread 6        partial sum = 978       start = 76      end = 88
Thread 5        partial sum = 834       start = 64      end = 76
Thread 1        partial sum = 247       start = 13      end = 26
Thread 4        partial sum = 690       start = 52      end = 64
Thread 7        partial sum = 1122      start = 88      end = 100
Thread 3        partial sum = 585       start = 39      end = 52

Final sum = 4950
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Теперь мы видим для каждого потока соответствующий сегмент, который он обрабатывает.

Начнем выполнять ошибочные модификации. Вынесем переменную id из класса частных в класс разделяемых переменных:

```
#pragma omp parallel private(size) shared(id) reduction(+ : sum)
```

Оценим результаты:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp -O3 Task_52.cpp -o Task_52
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_52
Thread 7        partial sum = 78        start = 0       end = 13
Thread 7        partial sum = 247       start = 13      end = 26
Thread 7        partial sum = 1122      start = 88      end = 100
Thread 7        partial sum = 978       start = 76      end = 88
Thread 7        partial sum = 834       start = 64      end = 76
Thread 7        partial sum = 416       start = 26      end = 39
Thread 7        partial sum = 585       start = 39      end = 52
Thread 7        partial sum = 690       start = 52      end = 64

Final sum = 4950
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_52
Thread 6        partial sum = 78        start = 0       end = 13
Thread 6        partial sum = 585       start = 39      end = 52
Thread 6        partial sum = 834       start = 64      end = 76
Thread 6        partial sum = 690       start = 52      end = 64
Thread 6        partial sum = 1122      start = 88      end = 100
Thread 6        partial sum = 247       start = 13      end = 26
Thread 6        partial sum = 978       start = 76      end = 88
Thread 6        partial sum = 416       start = 26      end = 39

Final sum = 4950
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Проанализируем возникающие ошибочные ситуации. Вынеся id в класс разделяемых переменных, мы сделали её доступной для чтения и записи всеми потоками. Теперь, когда один поток изменяет значение id, это изменение видно всем потокам. Это объясняет, почему в выводе все потоки имеют одинаковый номер id, который является последним значением id, установленным в параллельной области. Последним выполняющим код из параллельной области потоком не обязательно является поток с максимальным номером, потому что порядок выполнения потоков в OpenMP не гарантирован. Это демонстрируется во втором тесте.

Также можно отметить, что благодаря ключу -O3 обеспечивается корректное определение каждым потоком сегмента массива (на момент определении сегмента каждый поток оперирует корректным номером), и как результат – корректное вычисление итоговой суммы. Скомпилируем код без данного ключа и запустим программу:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp Task_52.cpp -o Task_52
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_52
Thread 7        partial sum = 78        start = 0       end = 13
Thread 7        partial sum = 1122      start = 88      end = 100
Thread 7        partial sum = 978       start = 76      end = 88
Thread 7        partial sum = 978       start = 76      end = 88
Thread 7        partial sum = 834       start = 64      end = 76
Thread 7        partial sum = 247       start = 13      end = 26
Thread 7        partial sum = 690       start = 52      end = 64
Thread 7        partial sum = 416       start = 26      end = 39

Final sum = 5343
```

Теперь наблюдаем ошибочное определение сегментов.

Вернем id в класс частных переменных. Исключим опцию reduction:

```
#pragma omp parallel private(id, size)
```

Также уберем дополнительный вывод start и end для каждого потока, поскольку переменная sum (в любом виде) никак не влияет на вычисление сегментов.

Оценим результаты:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp -O3 Task_51.cpp -o Task_51
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_51
Thread 0        partial sum = 78
Thread 4        partial sum = 768
Thread 3        partial sum = 1641
Thread 5        partial sum = 912
Thread 6        partial sum = 1056
Thread 2        partial sum = 494
Thread 7        partial sum = 2763
Thread 1        partial sum = 325

Final sum = 2763
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_51
Thread 0        partial sum = 78
Thread 7        partial sum = 2306
Thread 4        partial sum = 1184
Thread 6        partial sum = 1890
Thread 5        partial sum = 912
Thread 3        partial sum = 1769
Thread 1        partial sum = 325
Thread 2        partial sum = 494

Final sum = 1890
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_51
Thread 0        partial sum = 78
Thread 3        partial sum = 663
Thread 2        partial sum = 1328
Thread 4        partial sum = 1602
Thread 6        partial sum = 1056
Thread 1        partial sum = 325
Thread 5        partial sum = 912
Thread 7        partial sum = 1447

Final sum = 1602
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

(без опции -O3 выводы не меняются)

В каждом из трех тестов значение итоговой суммы различно, что подтверждает её недетерминированность.

При отсутствии опции reduction в коде, итоговая сумма во всех тестах не соответствует ожидаемому значению (4950). Вместо того, чтобы получить сумму всех элементов массива, мы получаем значение, которое соответствует частичной сумме, вычисленной последним потоком, завершившим свою работу.

Этому есть объяснение. Операция += не является атомарной Это означает, что она может быть прервана в середине своего выполнения. Без опции reduction все восемь потоков пытаются обновить общую переменную sum. Это приводит к состоянию гонки, где только одно обновление будет учтено. Поэтому sum в конце равно сумме для того потока, который завершился последним (не обязательно с максимальным id), потому что это последнее обновление, которое было применено к sum.

Ещё можно отметить, что большая часть частичных сумм вычисляется некорректно. Например, эталонная частичная сумма для первого потока – 247, а в тестах с исключенной

опцией она принимает значения, например, 325. Это в очередной раз подчеркивает важность использования директивы reduction, исключающей проблему гонки данных.

6. Перепишем программу из примера 2 без использования опции reduction, но с применением директивы critical.

Текущая программа примера 2 (слегка модифицированная) использует опцию reduction для суммирования элементов массива в многопоточном режиме. Вместо этого используем директиву critical для обеспечения безопасности потоков при обновлении общей переменной sum. Для этого:

- Удалим опцию reduction(+ : sum) из директивы #pragma omp parallel. Теперь sum больше не будет автоматически обновляться с использованием операции редукции.
- Внутри цикла for, который выполняет суммирование элементов массива, обернем операцию sum += a[i] в директиву #pragma omp critical. Это гарантирует, что в любой момент времени только один поток может обновлять значение sum.

Текст программы:

```
#include <omp.h>
#include <iostream>
#include <sstream>

using namespace std;

// Функция для корректного вывода
void print_data(string message)
{
	ostringstream out;
	out << message;
	cout << out.str();
}

int main()
{
	int sum = 0;
	const int a_size = 100;
	int a[a_size], id, size;

	for(int i = 0; i<100; i++)
	{
		a[i] = i;
	}

	// Установка числа потоков
	omp_set_num_threads(8);

	#pragma omp parallel private(id, size)
	{	// Начало параллельной области
		id = omp_get_thread_num();
		size = omp_get_num_threads();

		// Разделяем работу между потоками
		int integer_part = a_size / size;
		int remainder = a_size % size;
```

```
            int a_local_size = integer_part + ((id < remainder) ? 1 : 0);

            int start = integer_part * id + ((id < remainder) ? id : remainder);
            int end = start + a_local_size;

            // Каждый поток суммирует элементы
            // своей части массива
            for(int i = start; i < end; i++)
            {
                  // Критическая секция обеспечивает безопасность обновления общей
переменной
                  #pragma omp critical
                  {
                        sum += a[i];
                  }

            }

      }

      cout << "\nFinal sum = " << sum << endl;
      return 0;
}
```

Результаты работы программы:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp -O3 Task_6.cpp -o Task_6
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Task_6

Final sum = 4950
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

Важно отметить, что использование директивы critical может привести к снижению производительности из-за ожидания потоками доступа к критической секции.

7. На основании примеров и пояснений в теоретической части разработали программу в соответствии с вариантом задания.

Согласно варианту, нужно выполнить построение изображения множества Мандельброта. За основу была взята готовая последовательная программа на языке C (доступна по ссылке: https://rosettacode.org/wiki/Mandelbrot_set#PPM_non_interactive).

Программа подверглась модификациям. Она была переделана под C++ формат, а также добавился буфер для хранения цветов пикселей, данные из которого в конце программы записываются в файл за раз (вместо одиночной записи в файл изображения цвета каждого пикселя в цикле). Буфер нужен только потому что он необходим в параллельной реализации (чтобы корректно сравнивать её с последовательной реализацией). Был также добавлен код для замеров времени. Программа насытилась подробными комментариями. Текст программы доступен в приложении в конце документа.

Перед тем как распараллеливать циклы кратко опишем алгоритм Мандельброта. Он основан на итерационном процессе, в котором комплексное число Z обновляется согласно формуле: Z=Z^2+C, где C - текущая точка на комплексной плоскости (соответствующая конкретному пикселю изображения), которую мы исследуем, а Z инициализируется как 0 для каждой новой точки.

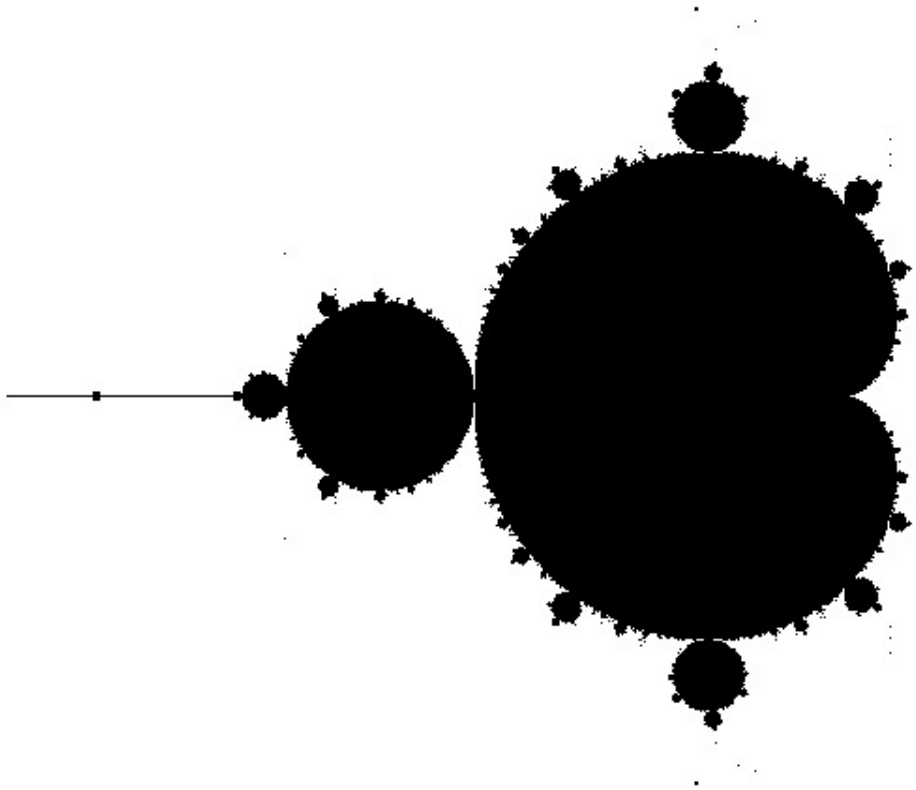
В алгоритме Мандельброта цвет пикселя определяется на основе того, сколько итераций было выполнено до того, как Z ушел на бесконечность, или до того, как было достигнуто максимальное количество итераций.

- Если после максимального числа итераций (IterationMax) Z не ушел на бесконечность, то считается, что точка принадлежит множеству Мандельброта. В этом случае пиксель обычно окрашивается в черный цвет.
- Если Z уходит на бесконечность до того, как будет достигнуто IterationMax, то точка не принадлежит множеству Мандельброта. В этом случае пиксель окрашивается в белый цвет. Часто используются различные цветовые схемы для визуализации количества итераций, что позволяет создать красочные изображения фракталов.

Запустим последовательную программу и оценим результаты:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ g++ -fopenmp -O3 Mandelbrot.cpp -o Mandelbrot
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$ ./Mandelbrot
Время работы: 0.0369961
user@DESKTOP-MDKFVDK:~/labs_PP/lab3$
```

И получившееся изображение множества Мандельброта:



Обоснуем выбранный способ распараллеливания. Было рассмотрено 3 варианта (во всех используется динамическая балансировка и управление числом потоков).

1) Объединение двух вложенных циклов в один с помощью директивы collapse. Тогда параллельная область будет выглядеть следующим образом:

```
#pragma omp parallel for collapse(2) schedule(dynamic) num_threads(num_of_threads)
for(int iY = 0; iY < iYmax; iY++)
{
	for(int iX = 0; iX < iXmax; iX++)
	{
		double Cx = CxMin + iX * PixelWidth;
		double Cy = CyMin + iY * PixelHeight;

		if(fabs(Cy) < PixelHeight / 2)
		{
			Cy = 0.0;
		}

		double Zx = 0.0, Zy = 0.0, Zx2 = 0.0, Zy2 = 0.0;
		int Iteration = 0;

		for(; Iteration < IterationMax && ((Zx2 + Zy2) < ER2); Iteration++)
		{
			double temp = Zx2 - Zy2 + Cx;
			Zy = 2 * Zx * Zy + Cy;
			Zx = temp;

			Zx2 = Zx * Zx;
			Zy2 = Zy * Zy;
		}

		int index = (iY * iXmax + iX) * 3;

		if(Iteration == IterationMax)
		{
			colors[index] = 0;
			colors[index + 1] = 0;
			colors[index + 2] = 0;
		}
		else
		{
			colors[index] = 255;
			colors[index + 1] = 255;
			colors[index + 2] = 255;
		}

	}
}
```

Опция collapse требует отсутствие кода между циклами. От чего часть кода, которая должна выполниться сразу после внешнего цикла, переходит во внутренний цикл и на каждой его итерации впустую перезаписывается.

2) Использование collapse аналогично прошлому варианту, но с вынесением преобразования пиксельных координат в соответствующие координаты на комплексной плоскости в отдельную параллельную область (в ней тоже требуется динамическая балансировка из-за различий во времени обработки итераций):

```
int maxValue = max(iYmax, iXmax);

#pragma omp parallel for schedule(dynamic) num_threads(num_of_threads)
for(int i = 0; i < maxValue; i++)
{

	if(i < iYmax)
	{
		CyValues[i] = CyMin + i * PixelHeight;

		if(fabs(CyValues[i]) < PixelHeight / 2)
		{
			CyValues[i] = 0.0;
		}

	}

	if(i < iXmax)
	{
		CxValues[i] = CxMin + i * PixelWidth;
	}

}

#pragma omp parallel for collapse(2) schedule(dynamic) num_threads(num_of_threads)
for(int iY = 0; iY < iYmax; iY++)
{

	for(int iX = 0; iX<iXmax; iX++)
	{
		double Zx = 0.0;
		double Zy = 0.0;
		double Zx2 = Zx * Zx;
		double Zy2 = Zy * Zy;

		int Iteration = 0;

		for(; Iteration < IterationMax && ((Zx2 + Zy2) < ER2); Iteration++)
		{
			Zy = 2 * Zx * Zy + CyValues[iY];
			Zx = Zx2 - Zy2 + CxValues[iX];

			Zx2 = Zx * Zx;
			Zy2 = Zy * Zy;
		}

		int index = (iY * iXmax + iX) * 3;

		if(Iteration == IterationMax)
		{
			colors[index] = 0;
			colors[index + 1] = 0;
			colors[index + 2] = 0;
		}
		else
		{
```

```
                colors[index] = 255;
                colors[index + 1] = 255;
                colors[index + 2] = 255;
            }

        }

}
```

3) Распараллеливание только внешнего цикла. Это лучший вариант (будет доказано ниже), поэтому приведем полный текст программы:

```
#include <stdio.h>
#include <math.h>
#include <fstream>
#include <string>
#include <vector>
#include <iostream>

#include "../timer.h"

using namespace std;

const int NUM_OF_THREADS_DEFAULT = 16;
const int CHUNK_SIZE_DEFAULT = 1;

// Задание размера изображения в пикселях
const int iXmax = 800;
const int iYmax = 800;

// Минимальные и максимальные значения для
// Реальной (Cx) и мнимой (Cy) частей комплексных чисел на плоскости
const double CxMin = -2.5;
const double CxMax = 1.5;
const double CyMin = -2.0;
const double CyMax = 2.0;

// Цвет кодируется от 0 до 255
const int MaxColorComponentValue = 255;

// Максимальное количество итераций, которые будут
// Выполнены для каждой точки на комплексной плоскости
const int IterationMax=200;

// Радиус пробега
const double EscapeRadius=2;

// Создание файла и запись заголовка
ofstream CreatePPMFile(string filename, string comment)
{
      ofstream fp;
      fp.open(filename, ios::binary);

      // Заголовок содержит тип файла (P6 - PPM), обязательный комментарий (#),
      // Размеры изображения и максимальное значение для каждого цветового
компонента
```

```
	fp << "P6\n" << comment << "\n" << iXmax << "\n" << iYmax << "\n" << MaxColorComponentValue << "\n";

	return fp;
}

int main(int argc, char *argv[])
{
	int num_of_threads = (argc > 1 ? stoi(argv[1]) : NUM_OF_THREADS_DEFAULT);
	int chunk_size = (argc > 2 ? stoi(argv[2]) : CHUNK_SIZE_DEFAULT);

	// Переменные для перебора каждого пикселя на экране (координаты)
	int iX;
	int iY;

	// Координаты на комплексной плоскости, соответствующие пикселям изображения (iX, iY)
	double Cx;
	double Cy;

	// Ширина (высота) каждого пикселя в терминах реальной (мнимой) оси комплексной плоскости
	double PixelWidth = (CxMax - CxMin) / iXmax;
	double PixelHeight = (CyMax - CyMin) / iYmax;

	// Определение комплексного числа Z как пары вещественных чисел (Z=Zx+Zy*i)
	double Zx;
	double Zy;

	// Квадраты реальной и мнимой частей (Zx2=Zx*Zx;  Zy2=Zy*Zy)
	double Zx2;
	double Zy2;

	// Отслеживание текущего количества выполненных итераций
	// Алгоритма Мандельброта для каждой точки на комплексной плоскости
	int Iteration;

	// Квадрат радиуса пробега (для оптимизации)
	double ER2 = EscapeRadius * EscapeRadius;

	ofstream fp = CreatePPMFile("image1.ppm", "# ");

	// Создаем одномерный массив для хранения цветов
	vector<unsigned char> colors(iYmax * iXmax * 3);

	Timer time;

	// Выполним параллельно перебор каждого пикселя по вертикали
	// Каждая итерация цикла будет выполняться в отдельном потоке.
	// schedule(dynamic) говорит OpenMP динамически распределять итерации цикла по потокам.
	// num_threads(num_of_threads) устанавливает количество потоков, которые будут использоваться.
	#pragma omp parallel for schedule(dynamic, chunk_size) num_threads(num_of_threads)
	for(int iY = 0; iY < iYmax; iY++)
```

```
	{
		// Преобразование вертикальной координаты пикселя
		// В соответствующую координату на мнимой оси
		double Cy = CyMin + iY * PixelHeight;

		// Проверяем, находится ли координата достаточно близко к 0
		// Это делается для улучшения визуализации основной “антенны”
		// Множества Мандельброта, которая расположена вдоль реальной оси
(Cx), где Cy равно нулю.
		if(fabs(Cy) < PixelHeight / 2)
		{
			Cy = 0.0;
		}

		// Перебор каждого пикселя по горизонтали
		for(int iX = 0; iX < iXmax; iX++)
		{
			// Преобразование горизонтальной координаты пикселя
			// В соответствующую координату на реальной оси
			double Cx = CxMin + iX * PixelWidth;

			// Инициализация начального значения Z и квадраты его частей
			double Zx = 0.0;
			double Zy = 0.0;
			double Zx2 = Zx * Zx;
			double Zy2 = Zy * Zy;

			int Iteration = 0;

			// Итерационно вычисляем значение комплексного числа до момента,
пока не будет достигнуто максимальное количество итераций
			// Или пока квадрат модуля комплексного числа Z не станет больше
квадрата радиуса побега
			for(; Iteration < IterationMax && ((Zx2 + Zy2) < ER2);
Iteration++)
			{
				double temp = Zx2 - Zy2 + Cx;
				Zy = 2 * Zx * Zy + Cy;
				Zx = temp;

				Zx2 = Zx * Zx;
				Zy2 = Zy * Zy;
			}

			// Используем одномерную индексацию для доступа к элементам
массива
			int index = (iY * iXmax + iX) * 3;

			// Если после максимального числа итераций Z не ушел на
бесконечность
			// => точка принадлежит множеству Мандельброта
			if(Iteration == IterationMax)
			{
				// Поскольку каждый поток записывает результаты в свою
собственную область массива colors
				// Это не приводит к конфликтам между потоками.
```

```
                        // Делаем пиксель черным
                        colors[index] = 0;
                        colors[index + 1] = 0;
                        colors[index + 2] = 0;
                    }
                    else
                    {
                        //Если квадрат модуля Z превышает квадрат радиуса побега,
то Z будет стремиться к бесконечности при дальнейших итерациях и
                        // => точка не принадлежит множеству Мандельброта

                        // Делаем пиксель белым
                        colors[index] = 255;
                        colors[index + 1] = 255;
                        colors[index + 2] = 255;
                    }

                }
            }

            cout << "Время работы: " << time.elapsed() << endl;

            // Записываем данные из буфера в файл
            fp.write(reinterpret_cast<const char*>(colors.data()), colors.size());

            fp.close();
            return 0;
}
```

В данной программе дополнительно добавлена возможность в консольном режиме менять не только число потоков, но и значение параметра <chunk_size>, который определяет число итераций для блока каждого потока. Это понадобится для дальнейших тестов.

Сравним время работы параллельной области для всех вариантов в секундах. Размер изображения в пикселях во всех тестах 800x800, а параметр <chunk_size> берется по умолчанию (1). Среднее время работы последовательного алгоритма: ~0.037. Замеры выполняем с ключом -O3:

Число потоков	Вариант		
	Первый	Второй	Третий
2	0.0368986	0.0362983	0.0192506
4	0.0261421	0.0258767	0.0102114
8	0.0191358	0.0171328	0.0053717
16	0.0118118	0.0117349	0.0037213

Вариант с распараллеливанием только одного цикла наиболее предпочтителен.

8. Разработали систему тестов (все замеры с ключом -O3).

1) Замеры времени выполнения параллельной области с использованием различных типов политик планирования. Время выполнения измерялось для различного числа потоков (2, 4, 8, 16) и различных размеров изображения (200x200, 400x400, 800x800, 1600x1600). <chunk_size> брался по умолчанию.

Тип политики dynamic

Число потоков	Размер изображения			
	200x200	400x400	800x800	1600x1600
2	0.0012454	0.0049055	0.0192506	0.0747571
4	0.0007510	0.0026023	0.0102114	0.0385385
8	0.0005194	0.0014625	0.0053717	0.0209345
16	0.0005099	0.0010578	0.0037213	0.0137022

Тип политики static

Число потоков	Размер изображения			
	200x200	400x400	800x800	1600x1600
2	0.0014567	0.0049512	0.0192109	0.0801278
4	0.0013467	0.0049142	0.0188165	0.0730247
8	0.0012388	0.0045779	0.0156145	0.0597098
16	0.0010101	0.0027512	0.0099939	0.0380071

Тип политики guided

Число потоков	Размер изображения			
	200x200	400x400	800x800	1600x1600
2	0.0012964	0.0048243	0.0195093	0.0746489
4	0.0013948	0.0050945	0.0201781	0.0773805
8	0.0008898	0.0027662	0.0108918	0.0395349
16	0.0007513	0.0017726	0.0058799	0.0225185

Ожидаемо, что при увеличении размерности изображения время работы при любом типе политики увеличивается. Все политики показывают наилучшее время выполнения при использовании 16 потоков для всех размеров изображений. Сравнивая различные типы политик, можем сделать вывод, что динамическое планирование показывает себя наилучшим образом. У неё более резкий спад по времени при увеличении числа потоков и самое меньшее время при максимальном числе потоков.

2) Замеры времени выполнения параллельной области с использованием политики планирования “dynamic” для оценки влияния значения параметра <chunk_size>.

Замеры выполнялись для различного числа потоков (2, 4, 8, 16) и различных значений параметра <chunk_size> (1, 2, 4, 8, 16). Для более корректного вывода выполнили замеры при минимальном и максимальном размере изображения (200x200, 1600x1600).

Размер изображения 200x200

Число потоков	Значение <chunk_size>				
	1	2	4	8	16
2	0.0013098	0.0012774	0.0013073	0.0012675	0.0012966
4	0.0007427	0.0007601	0.0007458	0.0007235	0.0009333
8	0.0004881	0.0005297	0.0005508	0.0006494	0.0010444
16	0.0005189	0.0007118	0.0006654	0.0007808	0.0011688

Размер изображения 1600x1600

Число потоков	Значение <chunk_size>				
	1	2	4	8	16
2	0.0751605	0.0747924	0.0744274	0.0744373	0.0741943
4	0.0384036	0.0385649	0.0381106	0.0381371	0.039035
8	0.0206806	0.0206492	0.0209878	0.0211278	0.0210701
16	0.0115445	0.012365	0.0115763	0.0118068	0.0118189

На основе замеров можно сказать, что в большинстве случаев время выполнения увеличивается при увеличении <chunk_size>. Наиболее отчетливо это видно при малом размере изображения и большем числе потоков.

Выводы. Из-за того, что время расчета для каждой точки изображения в алгоритме Мандельброта индивидуально и неизвестно заранее, необходимо использовать динамический тип планирования, который обеспечивает динамическую балансировку загрузки процессорных ядер. Что касается значения параметра <chunk_size>, при его увеличении мы начинаем проигрывать по времени. При малом <chunk_size> задачи более равномерно распределяются между потоками (меньше потоков простаивают) и увеличенные в таком случае накладные расходы на планирование и синхронизацию становятся менее значительными.

Приложение. Последовательная версия программы построения множества Мандельброта

```cpp
#include <stdio.h>
#include <math.h>
#include <fstream>
#include <string>
#include <vector>
#include <iostream>

#include "../timer.h"

using namespace std;

// Задание размера изображения в пикселях
const int iXmax = 800;
const int iYmax = 800;

// Минимальные и максимальные значения для
// Реальной (Cx) и мнимой (Cy) частей комплексных чисел на плоскости
const double CxMin = -2.5;
const double CxMax = 1.5;
const double CyMin = -2.0;
const double CyMax = 2.0;

// Цвет кодируется от 0 до 255
const int MaxColorComponentValue = 255;

// Максимальное количество итераций, которые будут
// Выполнены для каждой точки на комплексной плоскости
const int IterationMax = 200;

// Радиус пробега
const double EscapeRadius = 2;

// Создание файла и запись заголовка
ofstream CreatePPMFile(string filename, string comment)
{
	ofstream fp;
	fp.open(filename, ios::binary);

	// Заголовок содержит тип файла (P6 - PPM), обязательный комментарий (#),
	// Размеры изображения и максимальное значение для каждого цветового
компонента
	fp << "P6\n" << comment << "\n" << iXmax << "\n" << iYmax << "\n" <<
MaxColorComponentValue << "\n";

	return fp;
}

int main(int argc, char *argv[])
{
	// Переменные для перебора каждого пикселя на экране (координаты)
	int iX;
	int iY;

	// Координаты на комплексной плоскости, соответствующие пикселям изображения
(iX, iY)
	double Cx;
```

```
	double Cy;

	// Ширина (высота) каждого пикселя в терминах реальной (мнимой) оси
комплексной плоскости
	double PixelWidth = (CxMax - CxMin) / iXmax;
	double PixelHeight = (CyMax - CyMin) / iYmax;

	// Определение комплексного числа Z как пары вещественных чисел (Z=Zx+Zy*i)
	double Zx;
	double Zy;

	// Квадраты реальной и мнимой частей (Zx2=Zx*Zx;  Zy2=Zy*Zy)
	double Zx2;
	double Zy2;

	// Отслеживание текущего количества выполненных итераций
	// Алгоритма Мандельброта для каждой точки на комплексной плоскости
	int Iteration;

	// Квадрат радиуса пробега (для оптимизации)
	double ER2 = EscapeRadius * EscapeRadius;

	// Создаем одномерный массив для хранения цветов
	vector<unsigned char> colors(iYmax * iXmax * 3);

	ofstream fp = CreatePPMFile("image.ppm", "# ");

	Timer time;

	// Перебор каждого пикселя по вертикали
	for(iY = 0; iY < iYmax; iY++)
	{
		// Преобразование вертикальной координаты пикселя
		// В соответствующую координату на мнимой оси
		Cy = CyMin + iY * PixelHeight;

		// Проверяем, находится ли координата достаточно близко к 0
		// Это делается для улучшения визуализации основной “антенны”
		// Множества Мандельброта, которая расположена вдоль реальной оси
(Cx), где Cy равно нулю.
		if (fabs(Cy) < PixelHeight / 2)
		{
			Cy = 0.0;
		}

		// Перебор каждого пикселя по горизонтали
		for(iX = 0; iX < iXmax; iX++)
		{
			// Преобразование горизонтальной координаты пикселя
			// В соответствующую координату на реальной оси
			Cx = CxMin + iX * PixelWidth;

			// Инициализация начального значения Z и квадраты его частей
			Zx = 0.0;
			Zy = 0.0;
			Zx2 = Zx * Zx;
```

```
                Zy2 = Zy * Zy;

                // Итерационно вычисляем значение комплексного числа до момента,
пока не будет достигнуто максимальное количество итераций
                // Или пока квадрат модуля комплексного числа Z не станет больше
квадрата радиуса побега
                for (Iteration = 0; Iteration < IterationMax && ((Zx2 + Zy2) <
ER2); Iteration++)
                {
                        // Z = Z^2 + C => Zx + Zy*i = (Zx^2 - Zy^2 + Cx) +
(2*Zx*Zy + Cy)*i
                        Zy = 2 * Zx * Zy + Cy;
                        Zx = Zx2 - Zy2 + Cx;

                        Zx2 = Zx * Zx;
                        Zy2 = Zy * Zy;
                }

                // Используем одномерную индексацию для доступа к элементам
массива
                int index = (iY * iXmax + iX) * 3;

                // Если после максимального числа итераций Z не ушел на
бесконечность
                // => точка принадлежит множеству Мандельброта
                if(Iteration == IterationMax)
                {
                        // Поскольку каждый поток записывает результаты в свою
собственную область массива colors
                        // Это не приводит к конфликтам между потоками.

                        // Делаем пиксель черным
                        colors[index] = 0;
                        colors[index + 1] = 0;
                        colors[index + 2] = 0;
                }
                else
                {
                        //Если квадрат модуля Z превышает квадрат радиуса побега,
то Z будет стремиться к бесконечности при дальнейших итерациях и
                        // => точка не принадлежит множеству Мандельброта

                        // Делаем пиксель белым
                        colors[index] = 255;
                        colors[index + 1] = 255;
                        colors[index + 2] = 255;
                }

            }
        }

        cout << "Время работы: " << time.elapsed() << endl;

        // Записываем данные из буфера в файл
        fp.write(reinterpret_cast<const char*>(colors.data()), colors.size());
```

```
        fp.close();
        return 0;
}
```