Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **«ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 4

Название: Разработка параллельных алгоритмов

Дисциплина: Анализ алгоритмов

Студент	ИУ7-52Б		В.А. Иванов
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Оглавление

Введение

В данной лабораторной реализуется и оценивается параллельный алгоритм поиска множества простых чисел.

Параллелизм — выполнение нескольких вычислений в различных потоках. Параллельное программирование интересно тем, что в процессоре с многоядерной архитектурой несколько процессов могут выполнятся одновременно на разных ядрах. Это приводит к тому, что время выполнения параллельного алгоритма может быть ощутимо меньше, чем у его однопоточного аналога.

Стандартный алгоритм поиска таких чисел подразумевает проход чисел от 2 до N с проверкой каждого из них на простоту. Так как каждая проверка независима, этот алгоритм подходит для реализации покоординатного параллелизма.

1. Аналитическая часть

1.1. Цели и задачи работы

Целью лабораторной работы является разработка и исследование параллельного алгоритма нахождения простых чисел.

Выделены следующие задачи лабораторной работы:

- описание понятия простого числа и параллелизма;
- описание и реализация параллельного алгоритма нахождения множества простых чисел;
- проведение замеров процессорного времени работы алгоритмов при различном количестве потоков;
- анализ полученных результатов.

1.2. Понятие простого числа

Простое число — натуральное число, которое делится на себя и на 1. При этом 1 простым числом не является[1].

Стандартный алгоритм проверки числа N на простоту заключается в переборе всех чисел, которые могут быть делителем данного числа, то есть от 2 до $\sqrt{N+1}$. В случае, если ни одно из этих чисел не делит N без остатка, N является простым числом.

2. Конструкторская часть

Рассмотрим параллельную реализацию алгоритма поиска простых чисел.

2.1. Описание алгоритма

Пусть максимальное число поиска = N, а число потоков = T. Стандандартный алгоритм произведёт перебор чисел от 2 до N, проверяя каждое число на простоту. В случае, если проверка пройдена, число добавляется массив простых чисел. Параллельная версия отличается тем, что для i-го потока выделяются числа от 2+i до N с шагом T.

Схема алгоритма приведена на рисунке 2.1, 2.2

2.2. Требования к программному обеспечению

Для полноценной проверки и оценки алгоритмов необходимо выполнить следующее.

1. Обеспечить возможность консольного ввода предела поиска и количества используемых потоков. Программа должна вывести множество простых чисел.

2. Реализовать функцию замера процессорного времени, затраченного функцией.

2.3. Заготовки тестов

При проверке алгоритма необходимо будет использовать следующие классы тестов:

- один поток, несколько потоков;

- предел = 2, предел = произвольное число;

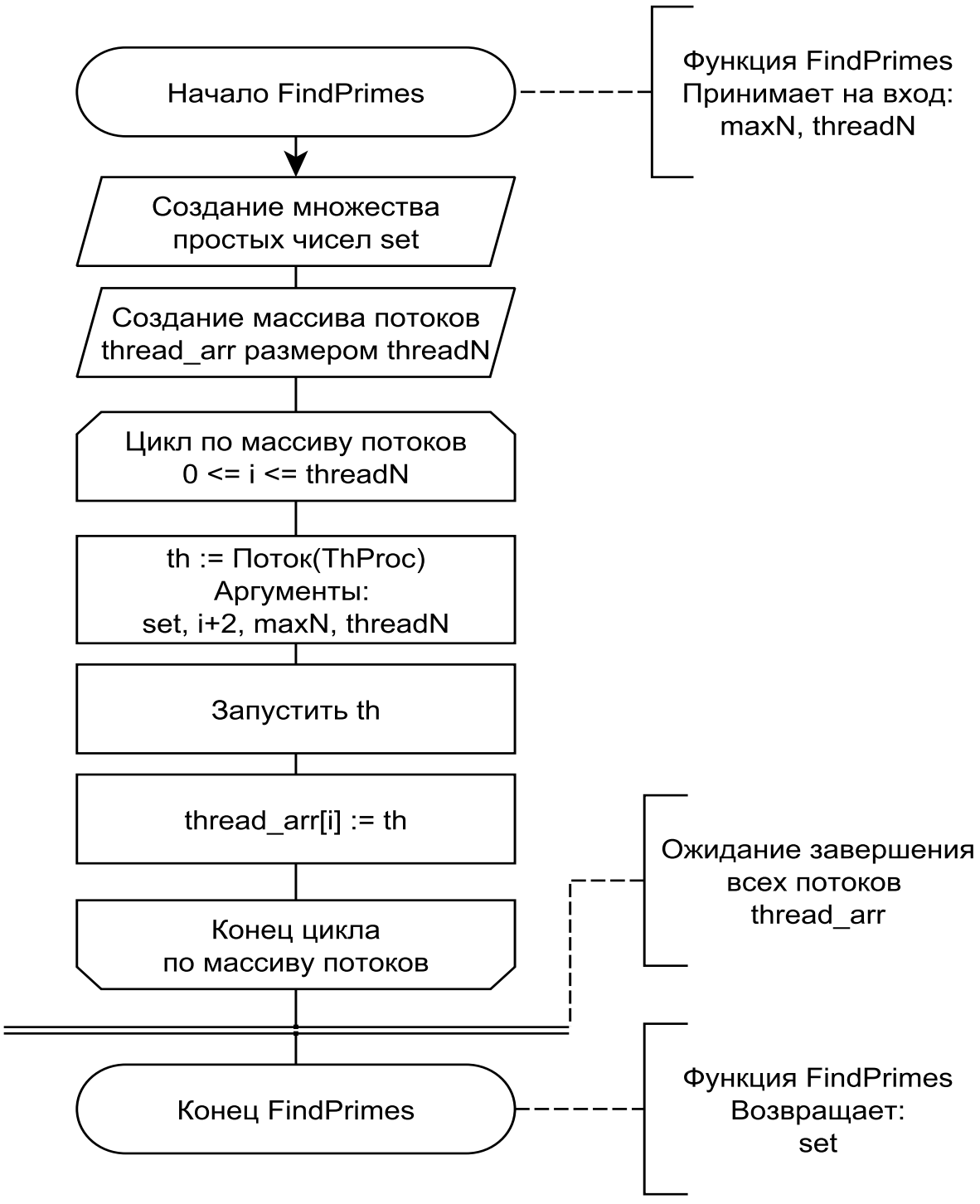

Рис. 2.1 — Главный поток

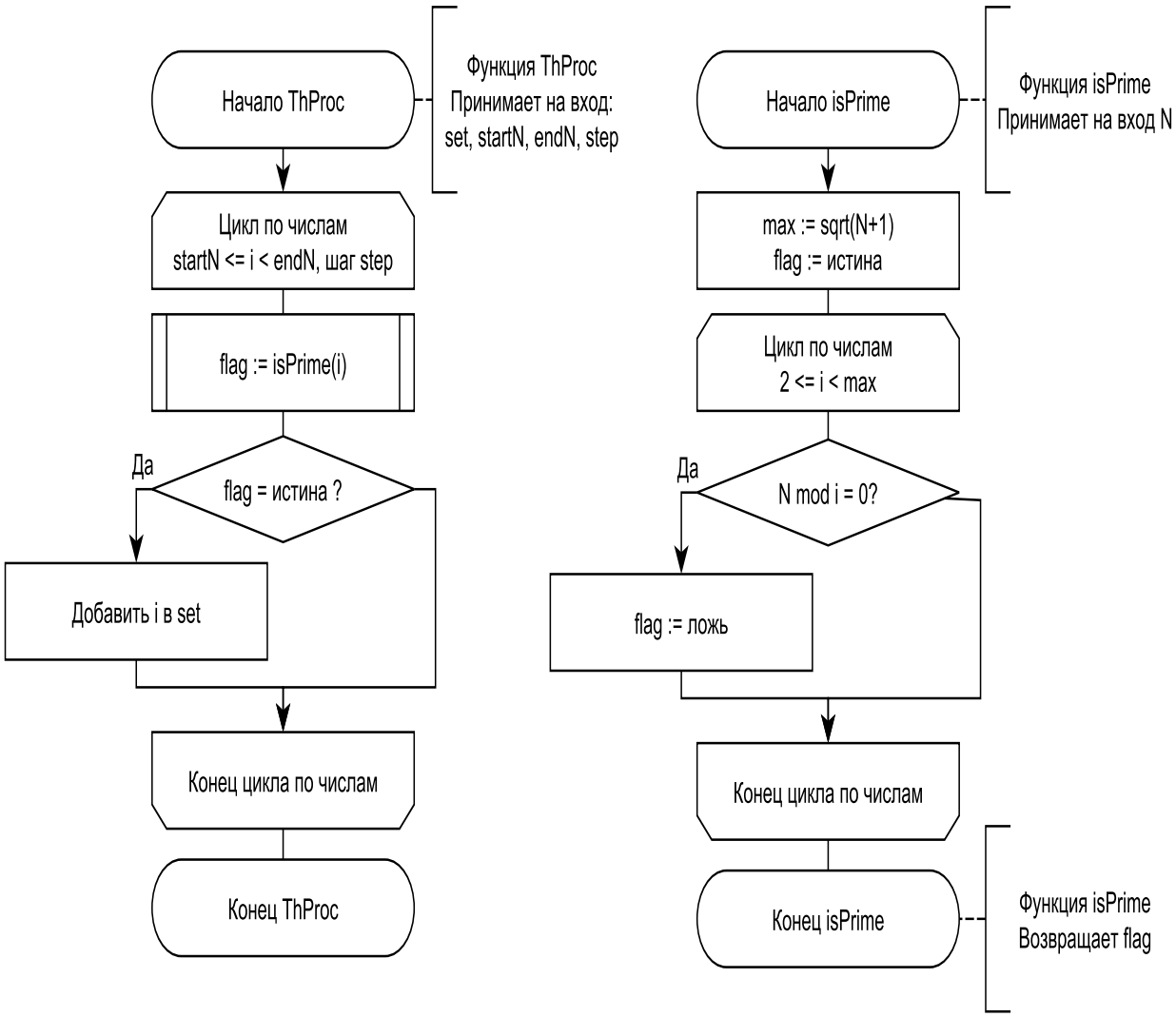

Рис. 2.2 — Рабочий поток

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран C++[2], так как имеется опыт работы с ним, и с библиотеками, позволяющими провести исследование и тестирование программы. Также в языке имеются средства для использования многопоточности. Разработка проводилась в среде Visual Studio 2019[3].

3.2. Листинг кода

Реализация алгоритма поиска простых чисел представлена на листинге 3.1

Листинг 3.1 — Функция параллельного поиска простых чисел.

```
1 #include "primes.h"
2
3 using namespace std;
4 std::mutex set_mtx;
5
6 inline bool is_prime_n(n_t n)
7 {
8   double max_check = sqrt(n + 1);
9   for (n_t i = 2; i < max_check; i++)
10  if (n % i == 0)
11  return false;
12  return true;
13 }
14
15 void thread_proc(set<n_t>& set, n_t start_n, n_t end_n, int
       step)
16 {
17  for (n_t i = start_n; i <= end_n; i += step)
18  if (is_prime_n(i))
19  {
```

```
20      set_mtx.lock();
21      set.insert(i);
22      set_mtx.unlock();
23    }
24  }
25
26  set<n_t> find_primes(n_t max_n, int thread_n)
27  {
28    set<n_t> set;
29    vector<thread> thread_arr;
30
31    n_t start_n;
32    for (int i = 0; i < thread_n; i++)
33    {
34      start_n = static_cast<n_t>(i) + 2;
35      thread_arr.push_back(thread(thread_proc, ref(set),
      start_n, max_n, thread_n));
36    }
37
38    for (int i = 0; i < thread_n; i++)
39    thread_arr[i].join();
40
41    return set;
42  }
```

3.3. Результаты тестирования

Для тестирования написанных функций был создан отдельный файл с ранее описанными классами тестов. Тестирование функций проводилось за счёт сравнения результов функций друг с другом (в случае разного количества потоков) и заданным ожидаемым результатом.

Состав тестов приведён в листинге 3.2.

Листинг 3.2 — Модульные тесты

```
#include "tests.h"

using namespace std;

// Сравнение множеств
bool cmp_sets(vector<set<n_t>>& all_results)
{
  int res_size = all_results.size();
  for (int i = 0; i < res_size -1; i++)
  if (all_results[i].size() != all_results[i + 1].size())
  return false;

  for (auto n : all_results[0])
  for (int i = 1; i < res_size; i++)
  all_results[i].erase(n);
  all_results[0].clear();

  for (int i = 0; i < res_size-1; i++)
  if (all_results[i].size() != all_results[i + 1].size())
  return false;
  return true;
}

// Сравнение разультата работы при разном количестве потоков
void _test_threads(void)
{
  std::string msg;
  msg = __FUNCTION__;    msg += " - OK";

  n_t max_n = 1000;
  vector<set<n_t>> all_results;
```

```
32
33    for (int i = 0; i < 20; i++)
34    all_results.push_back(find_primes(max_n, i+1));
35
36    if (!cmp_sets(all_results))
37    {
38      msg = __FUNCTION__;    msg += " - FAILED";
39    }
40
41    std::cout << msg << std::endl;
42  }
43
44  // Тест при максимальном числе = 2
45  void _test_2(void)
46  {
47    std::string msg;
48    msg = __FUNCTION__;    msg += " - OK";
49
50    vector<set<n_t>> all_results;
51
52    all_results.push_back(set<n_t>{2});
53    all_results.push_back(find_primes(2, 1));
54
55    if (!cmp_sets(all_results))
56    {
57      msg = __FUNCTION__;    msg += " - FAILED";
58    }
59
60    std::cout << msg << std::endl;
61  }
62
63  // Тест при максимальном числе = 200
64  void _test_200(void)
65  {
```

```
66    std::string msg;
67    msg = __FUNCTION__;    msg += " - OK";
68
69    vector<set<n_t>> all_results;
70
71    all_results.push_back(set<n_t>{2, 3, 5, 7, 11, 13, 17, 19,
       23,
72      29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
       89,
73      97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
       151,
74      157, 163, 167, 173, 179, 181, 191, 193, 197, 199});
75    all_results.push_back(find_primes(200, 1));
76
77    if (!cmp_sets(all_results))
78    {
79      msg = __FUNCTION__;    msg += " - FAILED";
80    }
81
82    std::cout << msg << std::endl;
83  }
84
85  void run_tests()
86  {
87    _test_threads();
88    _test_2();
89    _test_200();
90  }
```

Все тесты пройдены успешно.

3.4. Оценка времени

Для замера процессорного времени исполнения функции используется функция QueryPerformanceCounter библиотеки windows.h[4]. Проведение измерений производится в функциях, приведённых в листинге 3.3.

Листинг 3.3 — Функции замера процессорного времени работы функции

```
1  void test_time(n_t max_n, int thread_n)
2  {
3    int count = 0;
4    start_counter();
5    while (get_counter() < 3.0 * 1000)
6    {
7      find_primes(max_n, thread_n);
8      count++;
9    }
10   double t = get_counter() / 1000;
11   cout << "Выполнено " << count << " операций за " << t << "
       секунд" << endl;
12   cout << "Время: " << t / count << endl;
13 }
14
15 void experiments_series(n_t max_n, vector<int>& thread_arr)
16 {
17   cout << "Размер: " << max_n << endl;
18   for (int i : thread_arr)
19   {
20     cout << "\Числоп потоков = " << i << endl;
21     test_time(max_n, i);
22   }
23 }
```

4. Исследовательская часть

4.1. План экспериментов

Измерения процессорного времени проводятся для поиска простых чисел до 10^5. Изучается серия экспериментов с количеством потоков: $1, 2, 4, 8, 16, 32$

Для повышения точности, каждый замер производится пять раз, за результат берётся среднее арифметическое.

4.2. Результат экспериментов

По результатам измерений процессорного времени можно составить таблицу 4.1

Таблица 4.1 — Результат измерений процессорного времени (в секундах)

Потоки	1	2	4	8	16	32
Время	0.033	0.036	0.013	0.009	0.012	0.013

Эксперименты проводились на компьютере с характеристиками:

- ОС - Windows 10, 64 бит;
- Процессор - Intel Core i7 8550U (1800 МГц, 4 ядра, 8 логических процессоров);
- Объем ОЗУ: 8 ГБ.

4.3. Вывод

По результатам экспериментов можно заключить следующее.

- Наиболее быстродейственно алгоритм действует на 8 потоках, что равно количеству логических процессоров на испытуемом компьютере.

- Время при работе одного и двух потоков практически не отличаются. При этом, использование четырёх потоков даёт значительный выигрыш по времени.
- Использование 16 и 32 потоков показывает результат по времени хуже, чем при 8 потоках, из чего следует, что увеличение потоков даёт выигрыш по времени лишь до достижения количества логических ядер машины.

Заключение

В ходе лабораторной работы достигнута поставленная цель: разработка и исследование параллельного алгоритма нахождения простых чисел. Решены все задачи работы.

Были изучены и описаны понятия простого числа и параллелизма. Также были описан и реализован параллельный алгоритм нахождения множества простых чисел. Проведены замеры процессорного времени работы алгоритма при различном количестве потоков. На основании экспериментов проведён сравнительный анализ.

Из проведённых экспериментов было выявлено, что наиболее быстродейственным является использование количества потоков, которое равно количеству логических ядер компьютера. Увеличение или уменьшение количества потоков ведёт к большему времени выполнения вычислений. Также было установлено, что версия алгоритма с двумя потоками не даёт выигрыша по времени по сравнению с однопоточной версией.

Список литературы

1. Простое число // Математическая энциклопедия (в 5 томах). — М.: Советская Энциклопедия, 1977. — Т. 4.

2. Документация языка C++ 98 [Электронный ресурс]. Режим доступа: http://www.open-std.org/JTC1/SC22/WG21/, свободный (дата обращения: 14.10.2020)

3. Документация среды разработки Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 14.10.2020)

4. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 28.09.2020).