**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»

# Отчет по лабораторной работе №4 по курсу "Анализ алгоритмов"

**Тема** Параллельное программирование

**Студент** Криков А.В.

**Группа** ИУ7-53Б

**Преподаватели** Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

# Содержание

# Введение

Многопоточность — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций.

Сутью многопоточности является квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса. Кроме этого, все потоки процесса имеют не только общее адресное пространство, но и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

К достоинствам многопоточной реализации той или иной системы можно отнести следующее:

- облегчение программы посредством использования общего адресного пространства;
- меньшие затраты на создание потока в сравнении с процессами;
- повышение производительности процесса за счёт распараллеливания процессорных вычислений;
- если поток часто теряет кэш, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов [1];
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения [2];
- проблема планирования потоков;

- специфика использования. Вручную настроенные программы на ассемблере, использующие расширения MMX или AltiVec и выполняющие предварительные выборки данных, не страдают от потерь кэша или неиспользуемых вычислительных ресурсов. Таким образом, такие программы не выигрывают от аппаратной многопоточности и действительно могут видеть ухудшенную производительность из-за конкуренции за общие ресурсы.

Однако несмотря на количество недостатков, перечисленных выше, многопоточная парадигма имеет большой потенциал на сегодняшний день и при должном написании кода позволяет значительно ускорить однопоточные алгоритмы.

Целью данной работы является изучение и программная реализация многопоточности на основе алгоритма поиска простых чисел.

В рамках выполнения работы необходимо решить следующие задачи:

- изучить основы параллельных вычислений;
- исследовать и реализовать последовательный и параллельный алгоритмы поиска простых чисел;
- привести схемы последовательной и параллельной реализации алгоритма поиска простых чисел;
- описать структуру разрабатываемого ПО;
- определить средства программной реализации;
- протестировать разработанное ПО;
- провести сравнительный анализ последовательной и параллельной реализации на основе экспериментальных данных;
- подготовить отчет по лабораторной работе.

# 1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы поиска простых чисел.

## 1.1 Описание алгоритмов

### 1.1.1 "Наивный"алгоритм поиска простых чисел

Наиболее наивный подход к поиску простых чисел заключается в следующем. Будем брать по очереди натуральные числа $n$, начиная с двойки, и проверять их на простоту. Проверка на простоту заключается в следующем: перебирая числа $k$ из диапазона от 2 до $n - 1$, будем делить $n$ на $k$ с остатком. Если при каком-то $k$ обнаружится нулевой остаток, значит, $n$ делится на $k$ нацело, и число $n$ составное. Если же при делении обнаруживались только ненулевые остатки, значит, число простое; в этом случае выводим его на экран. Ясно, что, получив нулевой остаток (тем самым обнаружив, что $n$ составное), следует отказаться от дальнейших проб на делимость.

Заметим, что все простые числа, за исключением двойки, нечётные. Если обработать особо случай $n = 2$, то все последующие числа $n$ можно перебирать с шагом 2. Это даст приблизительно двукратное увеличение производительности программы.

### 1.1.2 Оптимизированный алгоритм поиска простых чисел

Ещё одно улучшение возникает благодаря следующему утверждению: наименьший делитель составного числа $n$ не превосходит $\sqrt{n}$. Докажем это утверждение от противного. Пускай число $k$ является наименьшим делителем $n$, причём $k > \sqrt{n}$. Тогда $n = k * l$, где $l \in N$, причём $l <= \sqrt{n}$, то есть $l$ также является делителем числа $n$, кроме того, меньшим, чем $k$, а это противоречит предположению. Всё это означает, что, перебирая потен-

циальные делители, можно оборвать перебор, когда $k$ достигнет $\sqrt{n}$ если до этого момента делителей не найдено, то их нет вообще. Например при проверке на простоту числа 11 то наблюдение позволяет сократить перебор более чем в три раза, а для числа 1111111111111111 — более чем в 1054092553 раза (оба числа — простые).

Можно сделать вывод что оптимизированный алгоритм поиска простых чисел работает в несколько раз быстрее "наивного"алгоритма поиска простых чисел, следовательно в данной лабораторной работе будет рассмотрена реализация оптимизировнного алгоритма поиска простых чисел.

### 1.1.3 Параллельная реализация алгоритма поиска простых чисел

Поскольку для нахождения всех простых чисел в диапазоне от 2 до $n$ нужно проверить каждое число отдельно, можно распараллелить обработку обработку $n$ чисел, разбив диапазон на несколько частей.

## Вывод

В данной работе стоит задача реализации последовательного и параллельного алгоритма поиска простых чисел. Входными данными будет являтся число $n$ - количество чисел, которые необходимо проверить на простоту начиная с 2. Выходными данными будет являтся массив всех простых чисел в диапазоне от 2 до $n$. В связи с ограничениями накладываемыми на ПО, входное число $n$ должно быть корректным. Необходимо дать теоретическую оценку последовательной и параллельной реализации алгоритма поиска простых чисел.

# 2 Конструкторская часть

В данном разделе будут рассмотрены схемы последовательной и параллельной реализации алгоритма поиска простых чисел, описание способов тестирования и структур данных.

## 2.1 Разработка алгоритмов

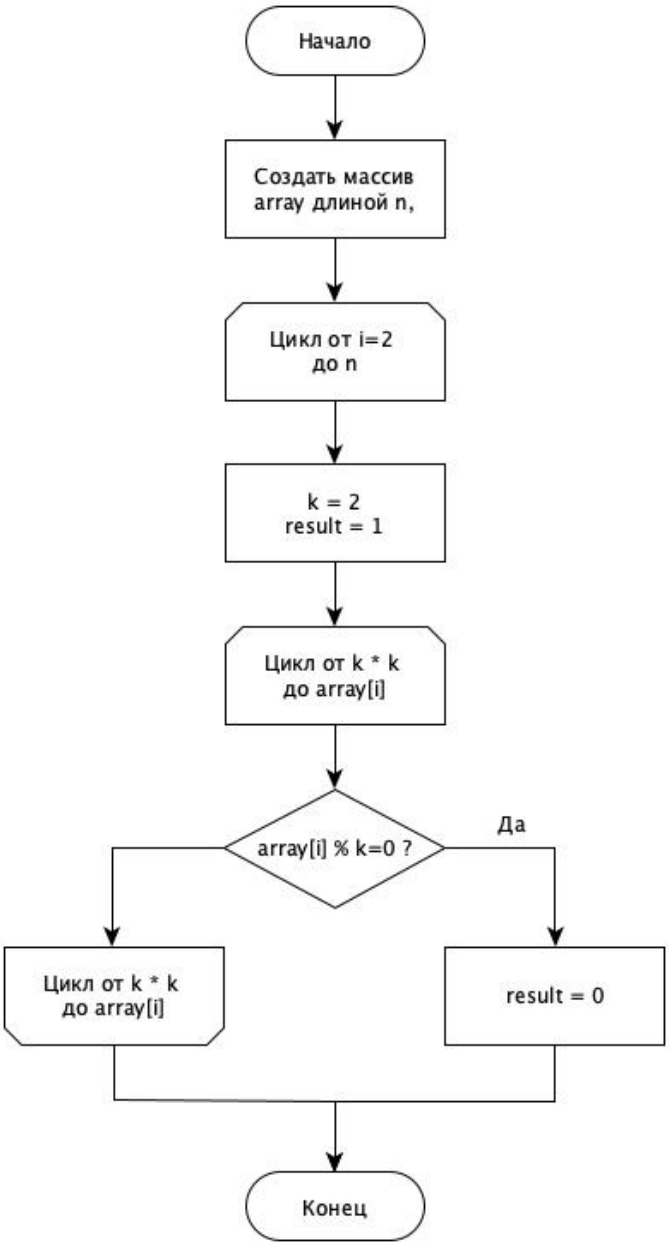


Рис. 2.1: Схема однопоточного алгоритма поиска простых чисел


Рис. 2.2: Схема распараллеливания произвольной функции, способной выполняться в некоторых независимых промежутках

## 2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.2);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. трудоемкость цикла рассчитывается, как (2.3);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.3}$$

4. трудоемкость вызова функции равна 0.

## 2.3 Трудоемкость алгоритмов

### 2.3.1 Алгоритм поиска простых чисел (последовательная реализация)

Во всех последующих алгоритмах не будем учитывать инициализацию матрицу, в которую записывается результат, потому что данное действие есть во всех алгоритмах и при этом не является самым трудоёмким.

Трудоёмкость алгоритма поиска простых чисел (последовательная реализация) Пусть $m = \sqrt{array[i]}$ тогда трудоемкость:

- внешнего цикла по $i \in [1..n]$, трудоёмкость которого: $f = n \cdot f_{body}$;
- цикла по $j \in [2..m]$, трудоёмкость которого: $f = m$;

Учитывая, что трудоёмкость алгоритма равна трудоёмкости внешнего цикла, можно вычислить ее, подставив циклы тела (2.5):

$$f_{standard} \approx n \cdot m; \tag{2.4}$$

### 2.3.2 Алгоритм поиска простых чисел (параллельная реализация)

Трудоёмкость алгоритма поиска простых чисел (параллельная реализация) Пусть $m = \sqrt{array[i]}$, $left, right$ - левая и правая граница диапазона обработатываемого одним потоком, причем $n = (right - left) * count\_threads$, тогда трудоемкость (для одного потока):

- цикла по $i \in [left..right]$, трудоёмкость которого: $f = (right - left) \cdot f_{body}$;
- цикла по $j \in [2..m]$, трудоёмкость которого: $f = m$;

Учитывая, что алгоритм выполняется параллельно итоговая трудоекость будет равна (2.5):

$$f_{parallel} \approx \frac{n \cdot m}{count\_threads}; \tag{2.5}$$

## 2.4 Описание структур данных

Исходя из условия задачи приходим к выводу о том, что для хранения массива найденных простых чисел наиболее удобно использовать структуру данных - массив целых чисел.

## 2.5 Описание способов тестирования

Данный алгоритм можно протестировать функционально. При этом можно выделить следующие классы эквивалентности:

- Входное число $n = 2$;
- Входное число $n > 2$;

## 2.6 Структура ПО

ПО будет состоять из следующих модулей:

- Основной модуль
- Модуль включающий в себя реализацию алгоритма поиска простых чисел
- Модуль для работы с потоками
- Модуль для работы с входными и выходными данными

# Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схема алгоритма поиска простых чисел, а также схема, описывающая работу с многопоточностью. Оценены их трудоёмкости, приведена структура ПО, описаны структуры данных.

# 3 Технологическая часть

В данном разделе приведены средства реализации и листинг алгоритмов.

## 3.1 Средства реализации

Для разработки ПО был выбран язык Java, поскольку он предоставляет разработчику широкий спектр возможностей и позволяет разрабатывать кроссплатформенные приложени, а также предоставляет большой набор инструментов для работы с многопоточностью. В качестве среды разработки использовалась Visual Studio Code. [3]

## 3.2 Листинг кода

Листинг 3.1: Создание и запуск потоков

```
1   public static void main(String[] args) throws InterruptedException, IOException {
2       array = new long[100000000];
3
4       for (int i = 0; i < n; i++) {
5           array[i] = i;
6       }
7
8       PrimeNumbersThread[] threads = new PrimeNumbersThread[N];
9
10      for (int i = 0; i < N; i++) {
11          threads[i] = new PrimeNumbersThread((int) (i * n / N), (int) ((i + 1) * n /
                N));
12      }
13
14      for (int i = 0; i < N; i++) {
15          threads[i].start();
16      }
17
18      for (int i = 0; i < N; i++) {
19          threads[i].join();
20      }
21
22      System.out.println("Success!");
```

```
        IO.printArray(primeNumsArray);
    }
```

Листинг 3.2: Алгоритм поиска простых чисел

```
public class Prime {
    public static int isPrime(long num) {
        long k = 2;
        int flag = 1;

        while (k * k <= num) {
            if (num % k == 0) {
                flag = 0;
                break;
            }
            k++;
        }

        return flag;
    }
}
```

Листинг 3.3: Метод потока поиска простых чисел

```
import java.util.ArrayList;
import java.util.List;

public class PrimeNumbersThread extends Thread {
    private int start;
    private int finish;

    PrimeNumbersThread(int start, int finish) {
        this.start = start;
        this.finish = finish;
    }

    @Override
    public void run() {
        List<Long> array = new ArrayList<>();
        for (int i = start; i < finish; i++) {
            if (Prime.isPrime(Main.array[i]) == 1) {
                array.add(Main.array[i]);
            }
        }
    }
}
```

## 3.3 Тестирование функций

В таблице 3.1 приведены тесты для функций, реализующих алгоритм поиска простых чисел. Тесты пройдены успешно.

| Входное число n | Количество потоков | Ожидаемый результат |
|---|---|---|
| 10 | 4 | (2 3 5 7) |
| 2 | 1 | 2 |
| 20 | 2 | (2 3 5 7 11 13 17 19) |
| -100 | 4 | Входное число n - некорректно |

Таблица 3.1: Тестирование функций

# Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы, настроить модульное тестирование и выполнить исследовательский раздел лабораторной работы.

# 4 Исследовательская часть

В данном разделе будет произведено сравнение реализаций (последовательная, параллельная) алгоритма поиска простых чисел.

## 4.1 Технические характеристики

- Операционная система: Windows 10. [4]
- Память: 16 GiB.
- Процессор: Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz. [5]

## 4.2 Временные характеристики

Для сравнения возьмем количество потоков равное: [1, 2, 3, ... 10]. Так как проверка на простое число считается короткой задачей, возспользуемся усреднением массового эксперимента. Для этого сложим результат работы алгоритма N раз (N >= 10), после чего поделим на N. Тем самым получим достаточно точные характеристики времени. Сравнение произведено при входном параметре n = 5000000. Результат можно увидеть на рис 4.1.

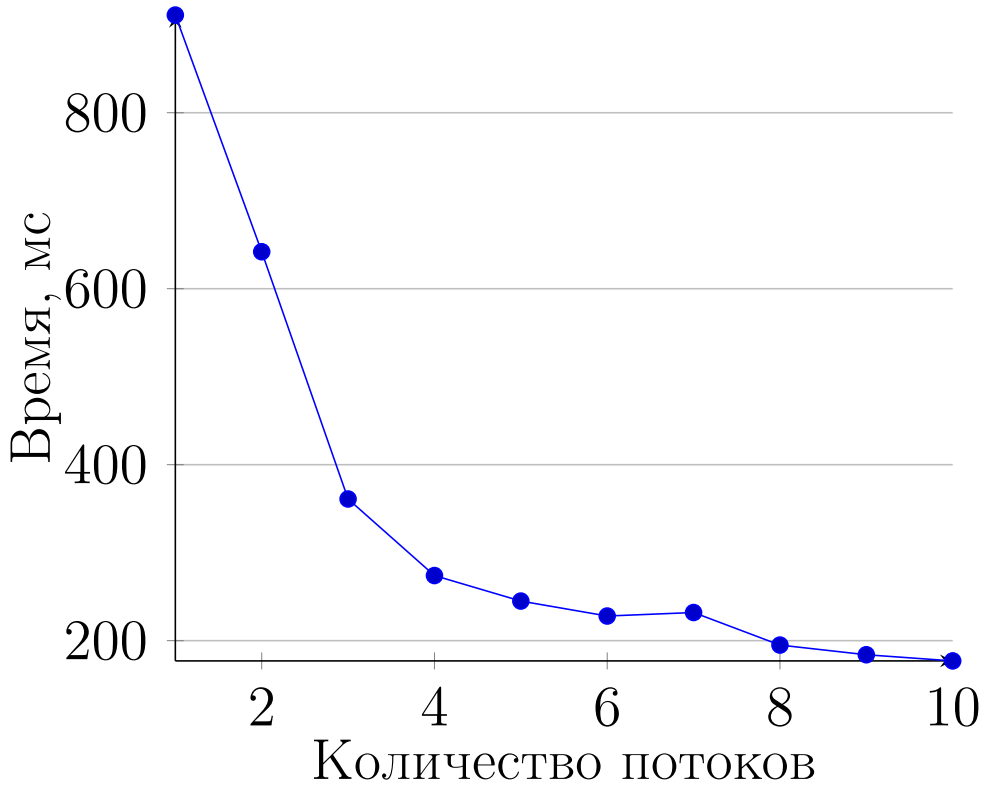


Рис. 4.1: Зависимость времени работы алгоритма от количества потоков

# Вывод

В результате эксперимента было получено, что при количестве потоков = 2 время работы алгоритма поиска простых чисел на 25% быстрее последовательного выполнения алгоритма. При количестве потоков = 6 время выполнения алгоритма на 75% быстрее последовательного выполнения алгоритма. Также стоит отметить, что при дальнейшем увеличении количества потоков время выполнения программы меняется менее обрывисто.

Можно сделать вывод, что для получения большей производительности нужно использовать все ядра процессора.

# Заключение

В рамках данной лабораторной работы:

1. были изучены основы параллельных вычислений;
2. был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
3. был сделан сравнительный анализ паралелельной и однопоточной реализации алгоритма поиска простых чисел;
4. были экспериментально подтверждены различия во временной эффективности реализации однопоточного и многопоточного алгоритма поиска простых чисел;
5. был подготовлен отчет по проделанной работе.

Анализируя рузльтат проведенных экспериментов, приходим к выводу, что, несмотря на более сложный код, параллельные алгоритмы значительно выигрывают по времени аналогичные однопоточные реализации. Наиболее эффективны данные алгоритмы при количестве потоков, совпадающем с количеством логических ядер компьютреа. Так, при входном параметре n удалось улучшить время выполнения алгоритма на 80% (в сравнении с однопоточной реализацией).

# Список литературы

[1] Mario Nemirovsky D. M. T. Multithreading Architecture // Morgan and Claypool Publishers. 2013.

[2] Olukotun K. Chip Multiprocessor Architecture — Techniques to Improve Throughput and Latency // Morgan and Claypool Publishers. 2007. p. 154.

[3] Documentation for Visual Studio Code [Электронный ресурс]. Режим доступа: `https://code.visualstudio.com/docs`.

[4] Клиентская документация по Windows для ИТ-специалистов [Электронный ресурс]. Режим доступа: `https://docs.microsoft.com/ru-ru/windows/resources/`.

[5] Процессор Intel® Core™ i7-4700HQ [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html`.