



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 №5 по курсу "Анализ алгоритмов"

Тема Конвейер

Студент Пересторонин П.Г.

Группа ИУ7-53Б

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

Оглавление

Введение	2
1 Аналитическая часть	3
1.1 Описание конвейерной обработки данных	3
1.2 Алгоритм Рабина-Карпа	4
2 Конструкторская часть	5
2.1 Разработка конвейерной обработки данных	5
3 Технологическая часть	9
3.1 Требования к ПО	9
3.2 Средства реализации	9
3.3 Листинг кода	9
3.4 Тестирование функций.	14
4 Исследовательская часть	16
4.1 Технические характеристики	16
4.2 Сравнение параллельного и последовательного конвейера . .	16
4.3 Пример работы и анализ результата	17
Заключение	20
Литература	21

Введение

Конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Сам термин «конвейер» пришёл из промышленности, где используется подобный принцип работы — материал автоматически подтягивается по ленте конвейера к рабочему, который осуществляет с ним необходимые действия, следующий за ним рабочий выполняет свои функции над получившейся заготовкой, следующий делает ещё что-то. Таким образом, к концу конвейера цепочка рабочих полностью выполняет все поставленные задачи, сохраняя высокий темп производства. Например, если на самую медленную операцию затрачивается одна минута, то каждая деталь будет сходиться с конвейера через одну минуту. В процессорах роль рабочих исполняют функциональные модули, входящие в состав процессора.

Цель данной работы: получить навык организации асинхронного взаимодействия потоков на примере конвейерной обработки данных.

В рамках выполнения работы необходимо решить следующие задачи:

- рассмотреть и изучить конвейерную обработку данных;
- реализовать конвейер с количеством лент не меньше трех в многопоточной среде;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

1.1 Описание конвейерной обработки данных

Конвейер[1] — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Идея заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции процессора представляются в виде последовательности более простых стадий. Вместо выполнения инструкций последовательно (ожидания завершения конца одной инструкции и перехода к следующей), следующая инструкция может выполняться через несколько стадий выполнения первой инструкции. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца.

Многие современные процессоры управляются тактовым генератором. Процессор внутри состоит из логических элементов и ячеек памяти — триггеров. Когда приходит сигнал от тактового генератора, триггеры приобретают своё новое значение, и «логике» требуется некоторое время для декодирования новых значений. Затем приходит следующий сигнал от тактового генератора, триггеры принимают новые значения, и так далее. Разбивая последовательности логических элементов на более короткие и помещая триггеры между этими короткими последовательностями, уменьшают время, необходимое логике для обработки сигналов. В этом случае длительность одного такта процессора может быть соответственно уменьшена.

1.2 Алгоритм Рабина-Карпа

Алгоритм, который был выбран для разложения на части конвейера – алгоритм Рабина-Карпа[2] для поиска подстрок сдвигом. Данный алгоритм использует хэширование и используется в качестве замены наивному алгоритму поиска подстрок в случаях, когда, например, осуществляется поиск строки из $m = 10.000$ символов «а», за которыми следует «b», в строке из $n = 10.000.000$ символов «а». В этом случае наивный алгоритм показывает своё худшее время исполнения – $O(mn)$, в то время как алгоритм Рабина-Карпа справляется за линейное время – $O(n)$.

Вывод

В данной работе стоит задача реализации конвейера для алгоритма Рабина-Карпа.

2 Конструкторская часть

2.1 Разработка конвейерной обработки данных

Принцип работы конвейера с 3 лентами представлен на рисунке 2.1. Алгоритм Рабина-Карпа и стадии его обработки представлены на рисунках 2.2 и 2.3.

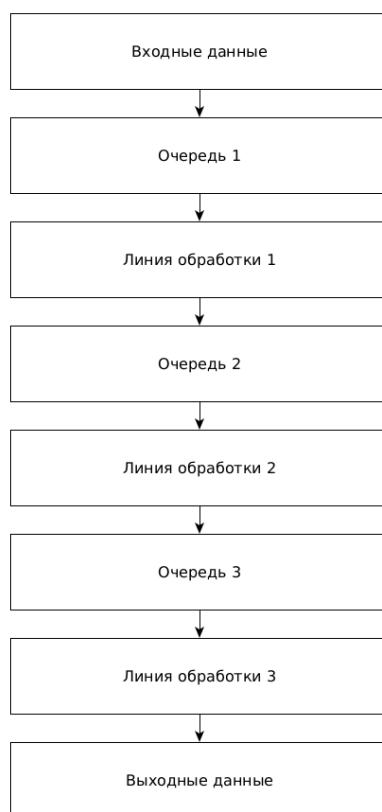


Рис. 2.1: Принцип работы конвейера с 3 лентами.

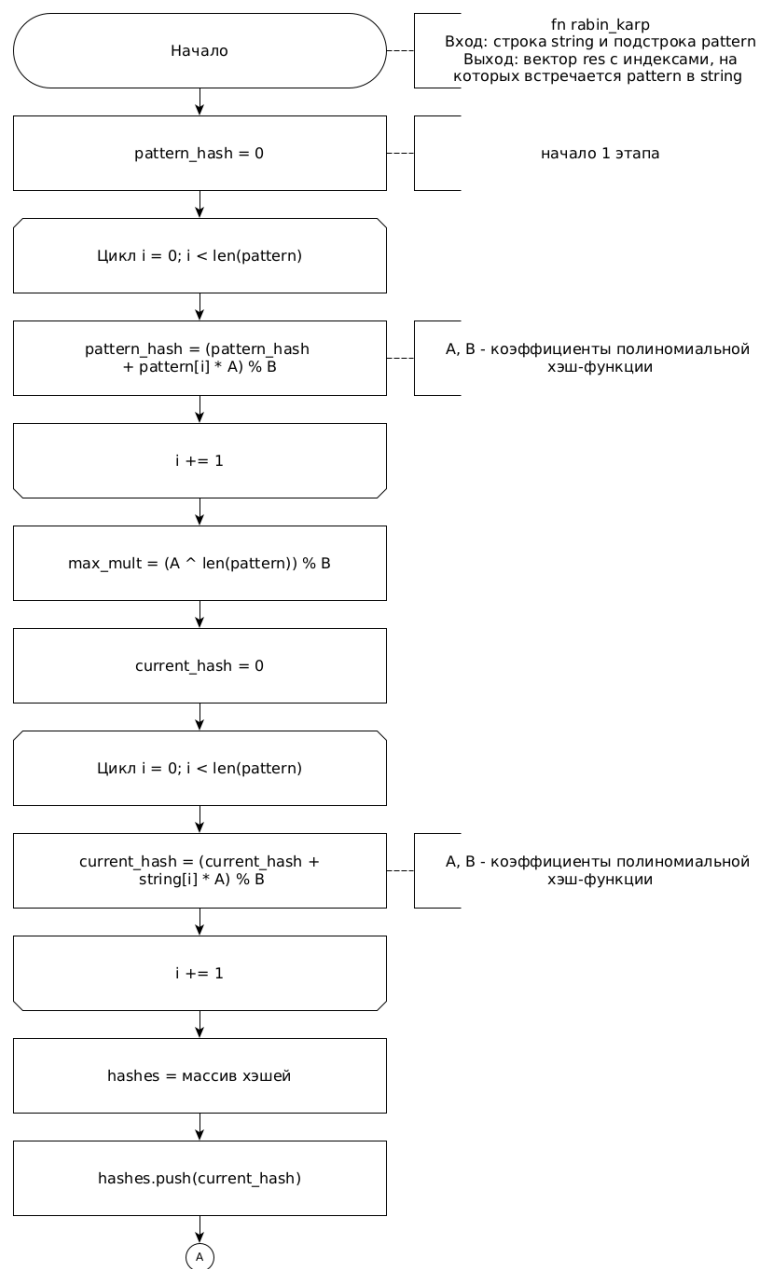


Рис. 2.2: Алгоритм Рабина-Карпа.

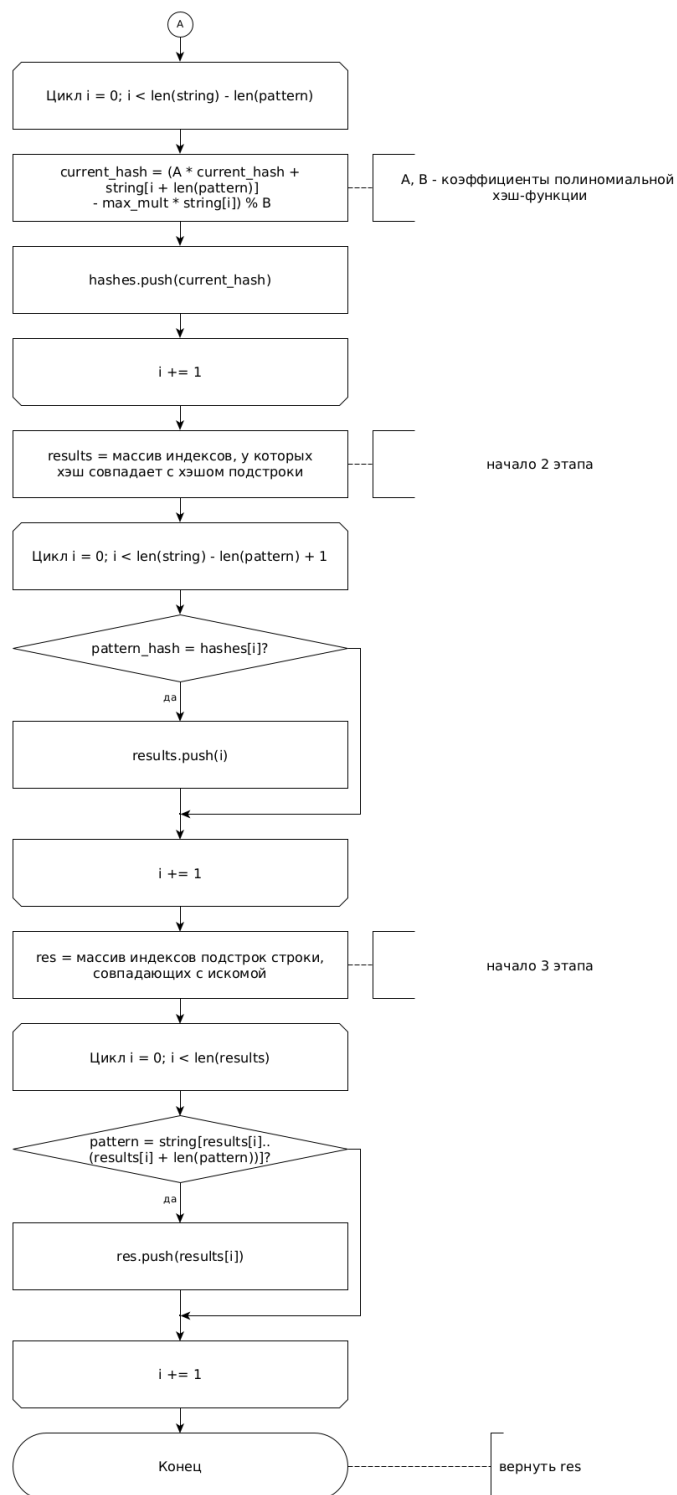


Рис. 2.3: Алгоритм Рабина-Карпа. Продолжение.

Вывод

Был показан принцип работы конвейерной обработки данных, а также алгоритм Рабина-Карпа с разделениями его на этапы для возможности выполнения на конвейере.

3 Технологическая часть

В данном разделе приведены средства программной реализации и листинг кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход конвейера подаётся массив задач, которые на нём нужно обработать;
- на выходе - лог-запись, в которой записаны в упорядоченном по времени порядке события начала и конца обработки определённого задания на ленте.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Rust [3]. Данный выбор обусловлен популярностью языка и скоростью его выполнения, а также тем, что данный язык предоставляет широкие возможности для написания тестов [4].

3.3 Листинг кода

В листинге 3.1 приведена реализация конвейера. В листингах 3.2 и 3.3 приведены реализации задачи и дополнительных используемых структур.

```
1 use std::sync::mpsc::{ Sender, Receiver, channel };
2 use std::thread;
3
4 use super::task::{ RabinKarpTask, Task3 };
5 use super::additional_structs::{ RabinKarpTaskResult };
6
7 pub struct Conveyor3 {
```

```

8     output: Receiver<RabinKarpTask>,
9 }
10
11 impl Conveyor3 {
12     pub fn new(queue: Vec<RabinKarpTask>) -> Self {
13         let (task_input, part1_receiver): (Sender<RabinKarpTask>,
14             Receiver<RabinKarpTask>) = channel();
15         let (part1_sender, part2_receiver): (Sender<RabinKarpTask>,
16             Receiver<RabinKarpTask>) = channel();
17         let (part2_sender, part3_receiver): (Sender<RabinKarpTask>,
18             Receiver<RabinKarpTask>) = channel();
19         let (part3_sender, task_output): (Sender<RabinKarpTask>, Receiver<RabinKarpTask>)
20             = channel();
21
22         thread::spawn(move || {
23             for mut task in part1_receiver.iter() {
24                 task.part1();
25                 part1_sender.send(task).expect("Send task to part2");
26             }
27         });
28
29         thread::spawn(move || {
30             for mut task in part2_receiver.iter() {
31                 task.part2();
32                 part2_sender.send(task).expect("Send task to part3");
33             }
34         });
35
36         thread::spawn(move || {
37             for mut task in part3_receiver.iter() {
38                 task.part3();
39                 part3_sender.send(task).expect("Send task to output");
40             }
41         });
42
43         for task in queue {
44             task_input.send(task).expect("Send task to conveyor");
45         }
46
47         Self {
48             output: task_output,
49         }
50     }
51
52     pub fn recv(&self) -> Option<RabinKarpTaskResult> {
53         self.output.recv().map(|res| res.result()).ok()
54     }
55 }

```

Листинг 3.1: Реализация конвейера

```
1 use super::additional_structs::{RabinKarpTaskResult, StrPat};
2 use chrono::{DateTime, Utc};
3
4 pub const NUMBER_OF_MEASUREMENTS: usize = 6;
5 pub const T1_START: usize = 0;
6 pub const T1_END: usize = 1;
7 pub const T2_START: usize = 2;
8 pub const T2_END: usize = 3;
9 pub const T3_START: usize = 4;
10 pub const T3_END: usize = 5;
11
12 pub trait Task3<T> {
13     fn part1(&mut self);
14     fn part2(&mut self);
15     fn part3(&mut self);
16
17     fn run1(&mut self);
18     fn run2(&mut self);
19     fn run3(&mut self);
20
21     fn result(&self) -> T;
22 }
23
24 #[derive(Debug, Clone)]
25 pub struct RabinKarpTask {
26     data: StrPat,
27     hashes: Option<Vec<u128>>,
28     result: Option<Vec<usize>>,
29     times: [DateTime<Utc>; NUMBER_OF_MEASUREMENTS],
30 }
31
32 impl RabinKarpTask {
33     const BIG_PRIME: u128 = 1_000_000_000_061;
34     const A_COEFF: u128 = 10_000_004_857;
35
36     pub fn new(string: Vec<char>, pattern: Vec<char>) -> Self {
37         let current_time = Utc::now();
38         Self {
39             data: StrPat::new(string, pattern),
40             hashes: None,
41             result: None,
42             times: [current_time; NUMBER_OF_MEASUREMENTS],
43         }
44     }
45 }
```

```

46 fn hash(string: &[char]) -> u128 {
47     let mut result = 0;
48     for &c in string.iter() {
49         result = (result * Self::A_COEFF + c as u128) % Self::BIG_PRIME;
50     }
51     result
52 }
53
54 fn get_mult(len: usize) -> u128 {
55     let mut res = 1;
56     for _ in 0..len {
57         res = (res * Self::A_COEFF) % Self::BIG_PRIME;
58     }
59     res
60 }
61
62 pub fn precompute_hashes(&mut self) {
63     let pattern_len = self.data.pattern.len();
64     let mut result = Vec::with_capacity(self.data.string.len() - pattern_len + 1);
65     let mut res = Self::hash(&self.data.string[0..pattern_len]);
66     let max_mult = Self::get_mult(pattern_len);
67     result.push(res);
68
69     for (&c_1, &c_0) in self.data.string[pattern_len..]
70         .iter()
71         .zip(self.data.string.iter())
72     {
73         res = ((res * Self::A_COEFF) % Self::BIG_PRIME + c_1 as u128 + Self::BIG_PRIME
74             - (max_mult * c_0 as u128) % Self::BIG_PRIME)
75             % Self::BIG_PRIME;
76         result.push(res);
77     }
78
79     self.hashes = Some(result);
80 }
81
82 fn compare_hashes(&mut self) {
83     let pattern_hash = Self::hash(&self.data.pattern);
84     if let Some(hashes) = self.hashes.as_mut() {
85         let mut result = Vec::new();
86         for (i, &hash) in hashes.iter().enumerate() {
87             if hash == pattern_hash {
88                 result.push(i);
89             }
90         }
91         self.result = Some(result);
92     }
93 }

```

```

94
95 fn compare_patterns(&mut self) {
96     let tmp = self.result.take();
97     if let Some(mut result) = tmp {
98         let pattern_len = self.data.pattern.len();
99         result.retain(|&elem| {
100             self.data.string[elem..(elem + pattern_len)] == self.data.pattern[..]
101         });
102         self.result = Some(result);
103     }
104 }
105
106 #[allow(dead_code)]
107 pub fn run_all(&mut self) {
108     self.precompute_hashes();
109     self.compare_hashes();
110     self.compare_patterns();
111 }
112 }
113
114 impl Task3<RabinKarpTaskResult> for RabinKarpTask {
115     fn part1(&mut self) {
116         self.times[T1_START] = Utc::now();
117         self.run1();
118         self.times[T1_END] = Utc::now();
119     }
120
121     fn part2(&mut self) {
122         self.times[T2_START] = Utc::now();
123         self.run2();
124         self.times[T2_END] = Utc::now();
125     }
126
127     fn part3(&mut self) {
128         self.times[T3_START] = Utc::now();
129         self.run3();
130         self.times[T3_END] = Utc::now();
131     }
132
133     fn run1(&mut self) {
134         self.precompute_hashes();
135     }
136
137     fn run2(&mut self) {
138         self.compare_hashes();
139     }
140
141     fn run3(&mut self) {

```

```

142     self.compare_patterns();
143 }
144
145 fn result(&self) -> RabinKarpTaskResult {
146     let result = match self.result.as_ref() {
147         None => Vec::new(),
148         Some(res) => res.clone(),
149     };
150
151     RabinKarpTaskResult {
152         data: self.data.clone(),

```

Листинг 3.2: Структура задачи

```

1 use super::task::{ NUMBER_OF_MEASUREMENTS };
2 use chrono::{ DateTime, Utc };
3
4 #[derive(Debug)]
5 pub struct RabinKarpTaskResult {
6     pub data: StrPat,
7     pub result: Vec<usize>,
8     pub times: [DateTime<Utc>; NUMBER_OF_MEASUREMENTS],
9 }
10
11 #[derive(Default, Clone, Debug)]
12 pub struct StrPat {
13     pub string: Vec<char>,
14     pub pattern: Vec<char>,
15 }
16
17 impl StrPat {
18     pub fn new(string: Vec<char>, pattern: Vec<char>) -> Self {
19         Self {
20             string,
21             pattern,
22         }
23     }
24 }

```

Листинг 3.3: Дополнительные структуры

3.4 Тестирование функций.

В таблице 3.1 представлены данные для тестирования. Все тесты пройдены успешно.

Строка	Подстрока	Ожидаемый результат	Результат
“aaaaaaa”	“aaaa”	[0, 1, 2, 3]	[0, 1, 2, 3]
“aaaabaaaa”	“aaa”	[0, 1, 5, 6]	[0, 1, 5, 6]
“aaaaaaaaa”	“b”	[]	[]

Таблица 3.1: Тестирование функций.

Вывод

Была разработана реализация конвейерных вычислений.

4 Исследовательская часть

В данном разделе приведены примеры работы программы и анализ характеристик разработанного программного обеспечения.

4.1 Технические характеристики

- Операционная система: Manjaro [5] Linux [6] x86_64.
- Память: 8 ГБ.
- Процессор: Intel® Core™ i7-8550U[7].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Сравнение параллельного и последовательного конвейера

В таблице 4.1 приведено сравнение времени выполнения параллельного и последовательного конвейеров в зависимости от длины очереди (при длине строки равной 200000 и длине искомой подстроки равной 200).

Таблица 4.1: Время выполнения параллельного и последовательного конвейеров в зависимости от длины очереди

Длина очереди	Последовательный	Параллельный
2	131913778	101065791
4	264496449	169329387
6	398330949	236552433
8	528620040	299132268
10	684301664	418758929
12	766470705	434427701
14	924169292	510653489
16	1058106910	564733886
18	1162946299	629284580
20	1321981140	697166232

4.3 Пример работы и анализ результата

Пример работы программы приведен на рисунке 4.1.

```

Finished dev [unoptimized + debuginfo] target(s) in 0.04s
Running `target/debug/lab_05`
Queue length: 10, String length: 200000, Pattern length: 200
Task №1 Part 1 Start 2020-12-02 16:06:21.177913996 UTC
Task №1 Part 1 End 2020-12-02 16:06:21.211684534 UTC
Task №2 Part 1 Start 2020-12-02 16:06:21.211696044 UTC
Task №1 Part 2 Start 2020-12-02 16:06:21.211736018 UTC
Task №1 Part 2 End 2020-12-02 16:06:21.222091509 UTC
Task №1 Part 3 Start 2020-12-02 16:06:21.222153654 UTC
Task №1 Part 3 End 2020-12-02 16:06:21.245091171 UTC
Task №2 Part 1 End 2020-12-02 16:06:21.245806488 UTC
Task №3 Part 1 Start 2020-12-02 16:06:21.245821499 UTC
Task №2 Part 2 Start 2020-12-02 16:06:21.245857695 UTC
Task №2 Part 2 End 2020-12-02 16:06:21.256868039 UTC
Task №2 Part 3 Start 2020-12-02 16:06:21.256917227 UTC
Task №2 Part 3 End 2020-12-02 16:06:21.279759968 UTC
Task №3 Part 1 End 2020-12-02 16:06:21.280311129 UTC
Task №4 Part 1 Start 2020-12-02 16:06:21.280319358 UTC
Task №3 Part 2 Start 2020-12-02 16:06:21.280354915 UTC
Task №3 Part 2 End 2020-12-02 16:06:21.290817410 UTC
Task №3 Part 3 Start 2020-12-02 16:06:21.290884525 UTC
Task №3 Part 3 End 2020-12-02 16:06:21.315624896 UTC
Task №4 Part 1 End 2020-12-02 16:06:21.316086030 UTC
Task №5 Part 1 Start 2020-12-02 16:06:21.316100475 UTC
Task №4 Part 2 Start 2020-12-02 16:06:21.316109928 UTC
Task №4 Part 2 End 2020-12-02 16:06:21.334060703 UTC
Task №4 Part 3 Start 2020-12-02 16:06:21.334106343 UTC
Task №5 Part 1 End 2020-12-02 16:06:21.353107315 UTC
Task №6 Part 1 Start 2020-12-02 16:06:21.353121475 UTC
Task №5 Part 2 Start 2020-12-02 16:06:21.353157452 UTC
Task №4 Part 3 End 2020-12-02 16:06:21.358468789 UTC
Task №5 Part 2 End 2020-12-02 16:06:21.364309094 UTC
Task №5 Part 3 Start 2020-12-02 16:06:21.364358866 UTC
Task №5 Part 3 End 2020-12-02 16:06:21.388009747 UTC
Task №6 Part 1 End 2020-12-02 16:06:21.389644213 UTC
Task №7 Part 1 Start 2020-12-02 16:06:21.389652051 UTC
Task №6 Part 2 Start 2020-12-02 16:06:21.389658440 UTC
Task №6 Part 2 End 2020-12-02 16:06:21.399987890 UTC
Task №6 Part 3 Start 2020-12-02 16:06:21.400031627 UTC
Task №7 Part 1 End 2020-12-02 16:06:21.423557480 UTC
Task №8 Part 1 Start 2020-12-02 16:06:21.423564427 UTC
Task №7 Part 2 Start 2020-12-02 16:06:21.423605780 UTC
Task №6 Part 3 End 2020-12-02 16:06:21.423654841 UTC
Task №7 Part 2 End 2020-12-02 16:06:21.434675380 UTC
Task №7 Part 3 Start 2020-12-02 16:06:21.434721510 UTC
Task №8 Part 1 End 2020-12-02 16:06:21.458107939 UTC
Task №9 Part 1 Start 2020-12-02 16:06:21.458120758 UTC
Task №8 Part 2 Start 2020-12-02 16:06:21.458168563 UTC
Task №7 Part 3 End 2020-12-02 16:06:21.459229164 UTC
Task №8 Part 2 End 2020-12-02 16:06:21.469635826 UTC
Task №8 Part 3 Start 2020-12-02 16:06:21.469707887 UTC
Task №9 Part 1 End 2020-12-02 16:06:21.491994862 UTC
Task №10 Part 1 Start 2020-12-02 16:06:21.492002970 UTC
Task №9 Part 2 Start 2020-12-02 16:06:21.492040260 UTC
Task №8 Part 3 End 2020-12-02 16:06:21.492890492 UTC
Task №9 Part 2 End 2020-12-02 16:06:21.502421098 UTC
Task №9 Part 3 Start 2020-12-02 16:06:21.502492862 UTC
Task №10 Part 1 End 2020-12-02 16:06:21.525539971 UTC
Task №10 Part 2 Start 2020-12-02 16:06:21.525590172 UTC
Task №9 Part 3 End 2020-12-02 16:06:21.525895356 UTC
Task №10 Part 2 End 2020-12-02 16:06:21.535825667 UTC
Task №10 Part 3 Start 2020-12-02 16:06:21.535839531 UTC
Task №10 Part 3 End 2020-12-02 16:06:21.558666585 UTC

```

Рис. 4.1: Пример работы программы

Из примера работы видно, что первый этап является самым трудоёмким. Время его работы достаточно стабильно, так как в нём рассчитываются хэш-значения для всех подстрок данной строки, и примерно равняется 0.4 секунды в среднем при длине строк 200000 и длине подстроки 200.

Второй этап выполняется быстрее всех и тоже достаточно стабилен, так как в нём идёт расчет хэш-значения для подстроки, которую надо найти в данной строке, а также сравнение полученного хэш-значения для подстроки и всех хэш-значений последовательных подстрок в данной строке. Среднее время работы примерно равно 0.01 секунды.

Третий этап нестабилен, потому что количество сравниваемых строк в нём равно количеству хэш-значений, которые совпали на втором этапе. Среднее время работы данного этапа примерно равняется 0.2 секунды.

Вывод

Второй этап конвейера получился наименее трудоёмким в среднем (0.01 секунды), а первый - наиболее долгим в среднем (0.4 секунды). Третий этап выполняется в среднем за 0.2 секунды, однако при большем количестве совпадающих подстрок данной строки с данной подстрокой это время может увеличиваться и быть больше, нежели время первого этапа, то есть оно является нестабильным и зависит от данных. Все данные приведены для входных данных, в которых длина строки равна 200000, а длина искомой подстроки равна 200.

Заключение

В рамках лабораторной работы была рассмотрена и изучена конвейерная обработка данных, реализован конвейер с 3 лентами в разных потоках. Благодаря конвейерной обработке данных возможна крайне удобная реализация задач, требующих поэтапной обработки некоторого набора данных. При этом схема обработки данных предоставляет простую схему параллельной обработки задач без конкуренции за данные, так как в определенный момент времени объект принадлежит только одной ленте конвейера.

Литература

- [1] Параллельная обработка данных. Режим доступа: <https://parallel.ru/vvv/lec1.html> (дата обращения: 12.11.2020).
- [2] Rabin M. O. Karp R. M. Efficient randomized pattern-matching algorithms // IBM Journal of Research and Development. 1987. с. 249–260.
- [3] Rust Programming Language [Электронный ресурс]. URL: <https://doc.rust-lang.org/std/index.html>.
- [4] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html> (дата обращения: 10.10.2020).
- [5] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 10.10.2020).
- [6] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: <https://www.linux.org.ru/> (дата обращения: 10.10.2020).
- [7] Процессор Intel® Core™ i7-8550U [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> (дата обращения: 10.10.2020).