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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

Название: Конвейерные вычисления. Алгоритм Бойера – Мура

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Т. А. Казаева
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

Время работы - одна из основных характеристик, влияющих на оценку алгоритма. Существует множество способов улучшения этой характеристики. В данной работе будет рассмотрен один из них – организация вычислительного конвейера. Такой подход предполагает разделение алгоритма на несколько независимых этапов, выходные данные каждого из которых являются входными этапами следующего.

В представленной работе исследуется реализация вычислительного конвейера на параллельных процессах. В качестве алгоритма, который будет декомпозирован на этапы и обернут в вычислительный конвейер, был выбран алгоритм Бойера – Мура – Хорспула для поиска подстрок.

Цель лабораторной работы – исследование параллельных конвейерных вычислений. Для достижения поставленной цели необходимо выполнить следующие задачи:

- исследовать организацию конвейерной обработки данных;
- разработать ПО, реализующее конвейер с количеством лент не менее трех в однопоточной и многопоточной среде;
- исследовать зависимость времени работы конвейера от количества потоков, на которых он работает.

Результаты сравнительного анализа будут приведены в виде таблиц, из которых можно будет сделать вывод об эффективности работы алгоритма Бойера – Мура на многопоточном конвейере, предложенной в данной работе.

1. Аналитический раздел

В данном разделе будут представлены сведения об организации конвейерной обработки данных и способы её оценки, описан реализуемый алгоритм.

1.1 Конвейерные вычисления

Способ организации процесса в качестве вычислительного конвейера позволяет построить процесс, содержащий несколько независимых этапов[1], на нескольких потоках. Выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду. Для контроля стадии используются три основные метрики, описанные ниже.

1. Время процесса - это время, необходимое для выполнения одной стадии.
2. Время выполнения - это время, которое требуется с момента, когда работа была выполнена на предыдущем этапе, до выполнения на текущем.
3. Время простоя - это время, когда никакой работы не происходит и линии простаивают.

Для того, чтобы время простоя было минимальным, стадии обработки должны быть одинаковы по времени в пределах погрешности. При возникновении ситуации, в которой время процесса одной из линий больше, чем время других в N раз, эту линию стоит распараллелить на N потоков.

В зарубежной литературе[2] конвейерная обработка именуется как *pipeline* (англ. - трубопровод) из - за аналогии со строительным трубопроводом, в котором вода льется в одном направлении, от одной части трубопровода к следующей.

1.2 Алгоритм Бойера – Мура – Хорспула

В качестве алгоритма, который будет декомпозирован на этапы для конвейерной обработки, был выбран алгоритм Бойера – Мура – Хорспула. Это упрощённый вариант алгоритма Бойера — Мура[3], предназначенный для поиска подстроки в строке. Выбор алгоритма обусловлен возможностью деления на независимые этапы:

1. Создание словаря для шаблона по следующей формуле:

$$shift(pattern_i) = |string| - \\ -last\_position(c, pattern[1..|pattern| - 1]) \tag{1.1}$$

   где $last\_position$ — последнее вхождение символа в строку, $pattern[a..b]$ – операция взятия подстроки.

2. Сопоставление шаблона строке согласно следующему правилу: Совмещение начала текста и шаблона, сравнение. Если все символы совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. Иначе, шаблон сдвигается на несколько символов вправо.

Смещение происходит согласно значению в словаре шаблона. Для символов, не вошедших в шаблон, величина смещения устанавливается равной длине шаблона.

Представленный алгоритм имеет два независимых этапа, каждый из которых может быть обработан конвейерной линией. В качестве третьего этапа предлагается генерировать случайные строки и извлекать из них подстроки.

1.3 Вывод

В качетсве входных и выходных данных для конвейера с тремя линиями достаточно использовать размер очереди, обрабатываемой конвейером. Конвейер может быть реализован следующим способом:

Линия 1. Генерация случайной строки и подстроки, которая содержится в сгенерированной строке.

Линия 2. Создание словаря для шаблона по формуле 1.1.

Линия 3. Сопоставление шаблона строке.

Для оптимальной работы программы строки, генерируемые линией 1 будут иметь фиксированную длину. Это обеспечит меньший простой конвейерных линий. Для чистоты эксперимента генерируемая линией 1 подстрока обязательно будет содержаться в генерируемой той же линией строке.

Для анализа работы алгоритма, согласно критериям, описанным в подразделе 1.1, для конвейера следует реализовать временные штампы в начале и завершении обработки каждой поступающей в очередь заявки.

2. Конструкторский раздел

Раздел содержит описание и иллюстрации для представленных алгоритмов обработки и организацию системы тестирования программного обеспечения.

2.1 Описание структур данных

Линии вычислительного конвейера реализованы с помощью очереди[4], реализованной на одномерном динамическом массиве. Выбор типа данных обусловлен тем, что заявки обрабатываются в порядке их поступления, выполнив их последовательно.

2.2 Описание способов тестирования

Тестирование алгоритма разделено на следующие этапы:

- тестирование алгоритма генерации подстроки на предмет присутствия подстроки в строке;
- тестирование алгоритма составления словаря шаблона;
- тестирование непосредственно алгоритма Бойера – Мура – Хорспула.

2.3 Схемы алгоритмов

Обращение к элементам массивов на схеме алгоритмов обозначено подстрочными индексами. Операция присваивания обозначена как "← операции "равно"и "не равно"как "="и "#"соответственно.

На рисунках 2.1, 2.2 представлена схема алгоритма работы конвейера.

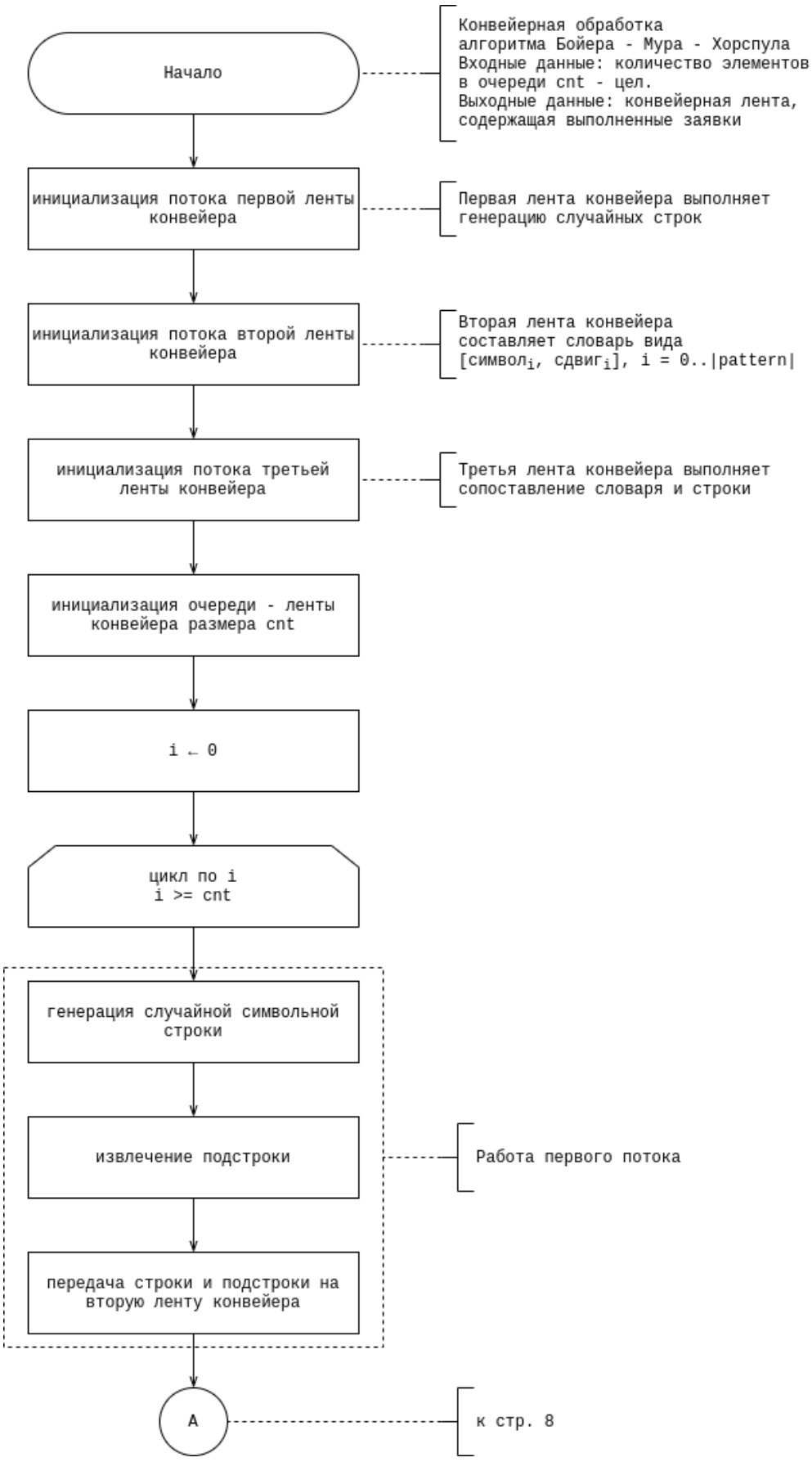

Рисунок 2.1 – Схема работы конвейера (начало)

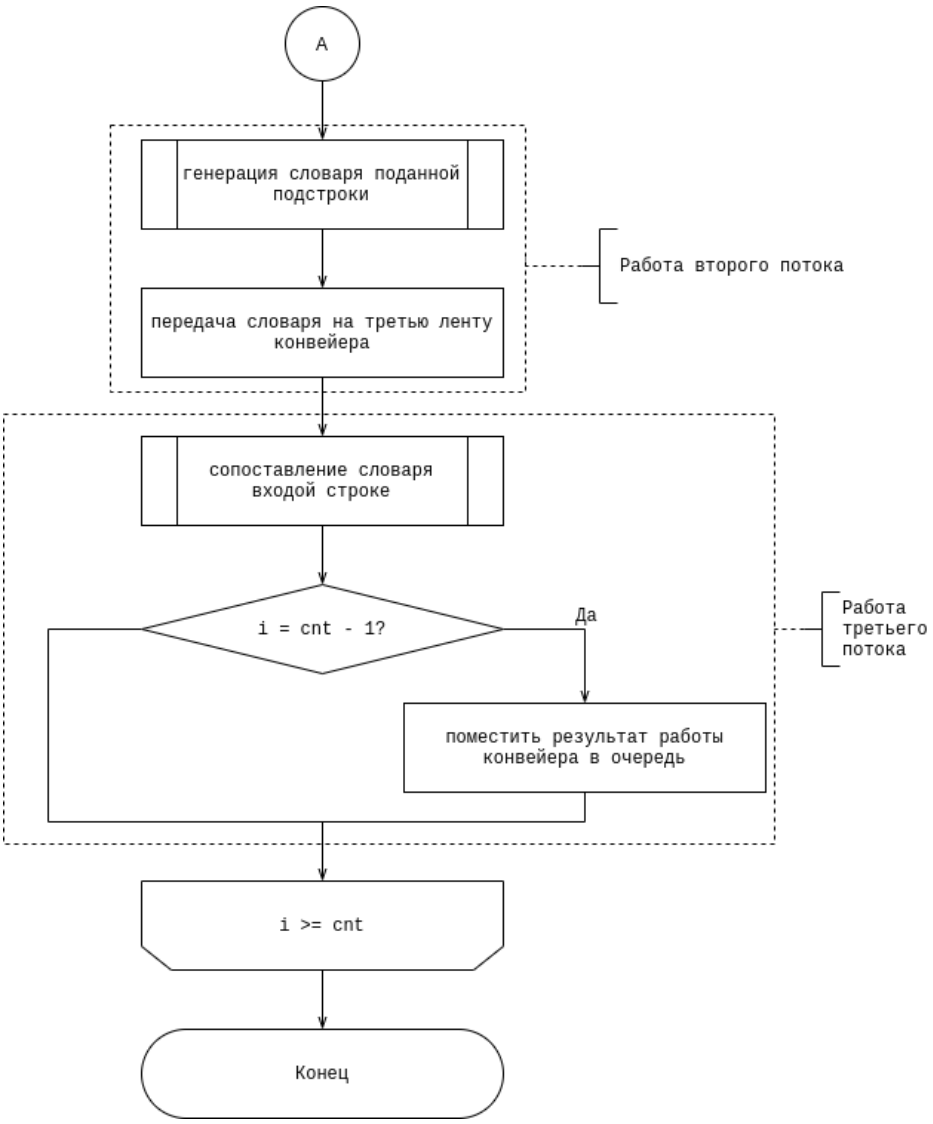

Рисунок 2.2 – Схема работы конвейера (завершение)

На рисунке 2.3 представлена схема получения словаря подстроки для алгоритма поиска.

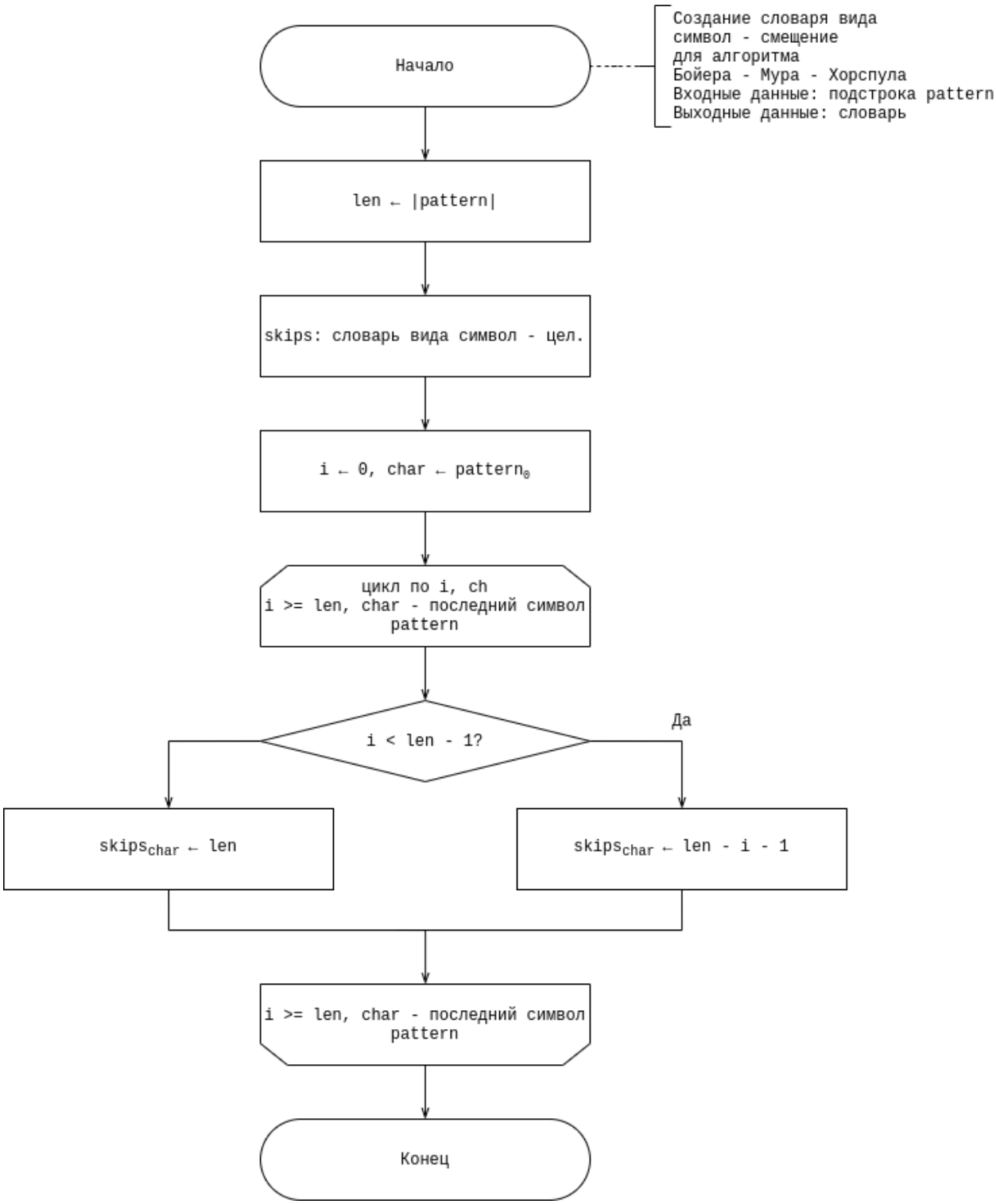

Рисунок 2.3 – Схема алгоритма получения словаря подстроки

На рисунках 2.4, 2.5 представлена схема алгоритма Бойера – Мура – Хорспула.

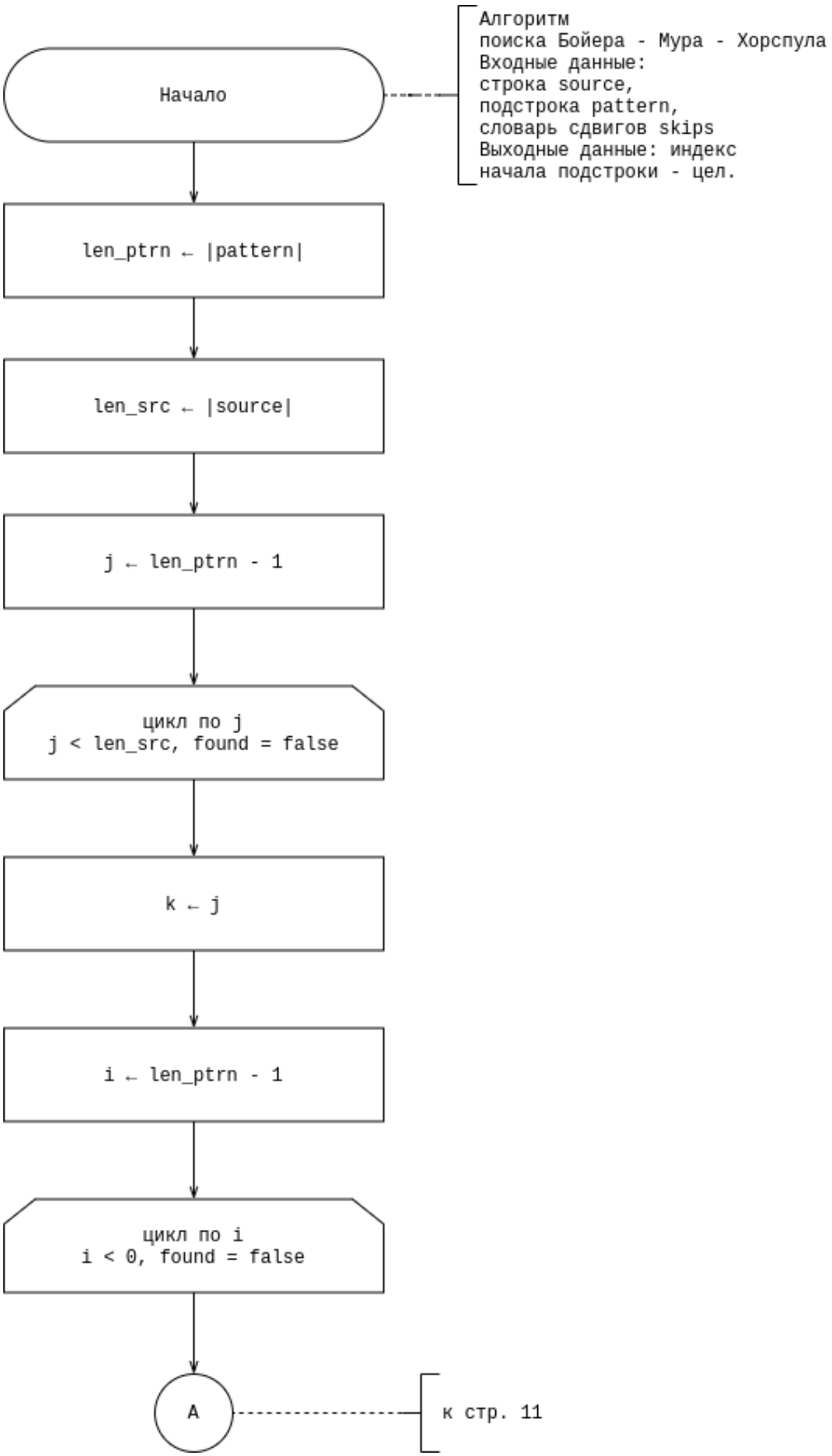

Рисунок 2.4 – Схема алгоритма Бойера – Мура – Хорспула(начало)

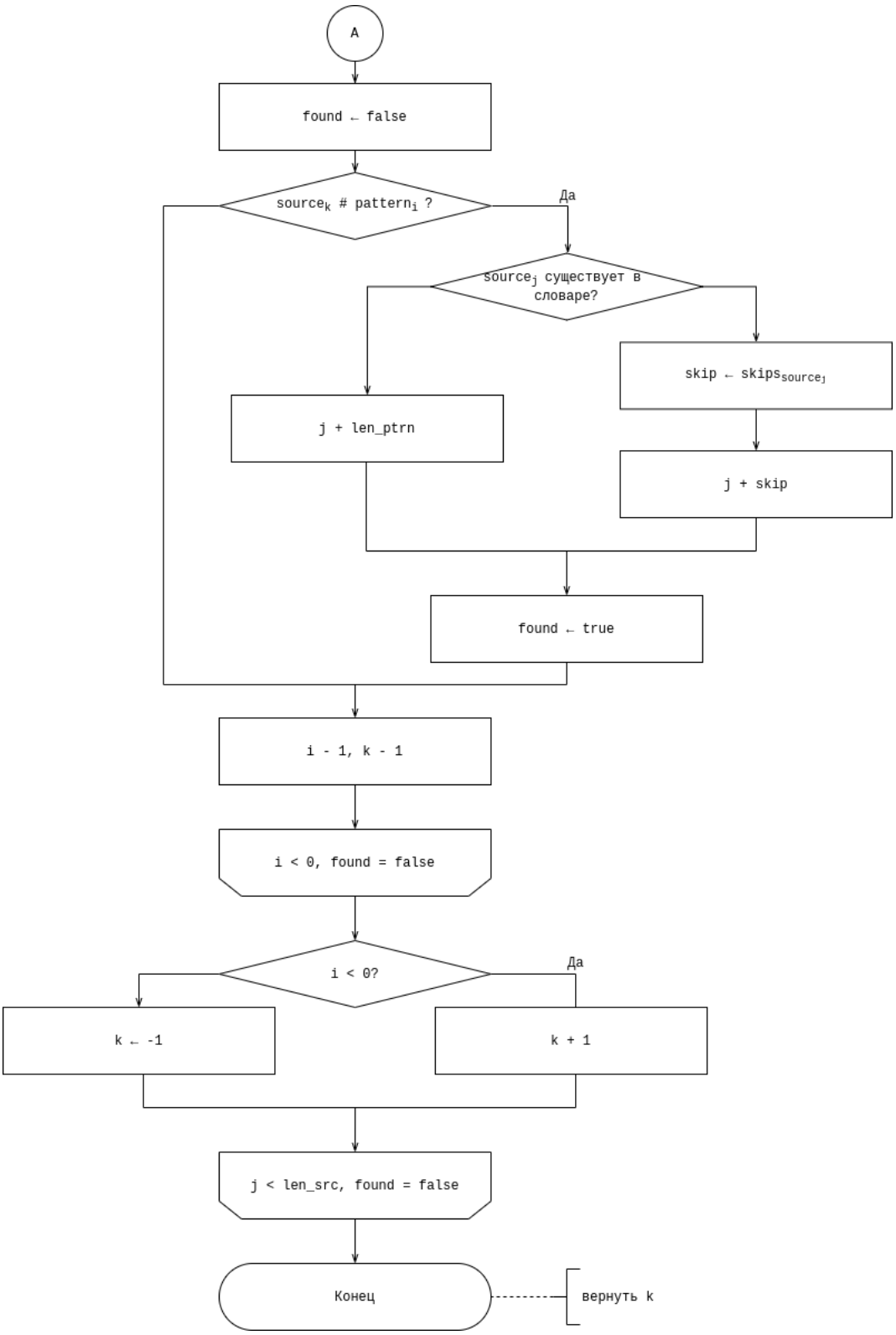

Рисунок 2.5 – Схема алгоритма Бойера – Мура – Хорспула(завершение)

2.4 Описание памяти, используемой алгоритмом

Затраты с точки зрения памяти складываются из:

- размера очереди, реализованной на массиве, состоящей из заявок;
- размера каждой заявки, содержащей 6 временных штампов, целочисленный словарь и два целочисленных массива.

2.5 Структура ПО

Программа поделена на ряд смысловых модулей, описанных ниже.

1. Модуль «bmsearch», в котором содержатся процедуры и функции, связанные с алгоритмом поиска подстрок и их генерации.
2. Модуль «pipeline», включающий в себя функции работы параллельного и синхронного конвейера, функции работы с очередью и журналирование.

Программа имеет консольный интерфейс.

2.6 Вывод

Были разработаны схемы алгоритмов, необходимых для решения задачи. Получено достаточно теоретической информации для написания программного обеспечения.

3. Технологический раздел

Раздел содержит требования к реализованному программному обеспечению, листинги кода и результаты рестирования программного обеспечения.

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- программа получает на вход размер очереди;
- программа выдает журналирование работы конвейера;
- генератор подстрок выдает ненулевые строки, состоящие из строчных и заглавных символов;
- подстрока, которую выдает генератор подстрок, должна содержаться в исходной строке.

3.2 Средства реализации

Для реализации ПО был выбран компилируемый многопоточный язык программирования Golang, поскольку язык отличается легкой и быстрой сборкой программ и автоматическим управлением памяти. В качестве среды разработки была выбрана среда VS Code, написание сценариев осуществлялось утилитой make.

3.3 Листинги кода

Структуры и типы

Заявка в очереди имеет следующую структуру: (3.1)

Листинг 3.1 – Структура заявки в очереди

```
type PipeTask struct {
    num              int
    generated        bool
    skip_table_made  bool
    pattern_mached   bool

    start_generating time.Time
    end_generatig    time.Time
    start_table      time.Time
    end_table        time.Time
    start_match      time.Time
    end_match        time.Time

    source          []rune
    pattern         []rune
    skip_table      map[rune]int
    pattern_index   int
}
```

Структура очереди продемонстрирована на листинге 3.2:

Листинг 3.2 – Структура очереди

```
type Queue struct {
    queue [](*PipeTask)
    size  int
}
```

Очередь

Реализация очереди продемонстрирована на листинге 3.3.

Листинг 3.3 – Очередь

```
func initQueue(size int) *Queue {
    qu := new(Queue)
    qu.queue = make([](*PipeTask), size)
    qu.size = -1

    return qu
}

func (qu *Queue) enqueue (t *PipeTask) {
    if (qu.size != len(qu.queue) - 1) {
        qu.queue[qu.size + 1] = t
```

```
        qu.size++
    }
}

func (qu *Queue) dequeue () *PipeTask {
    t := qu.queue[0]
    qu.queue = qu.queue[1:]
    qu.size--
    return t
}
```

Реализация параллельного конвейера

Реализация параллельного конвейера продемонстрирована на листинге 3.4.

Листинг 3.4 – Реализация параллельного конвейера

```
func Pipeline(count int, ch chan int) *Queue {
    first := make(chan *PipeTask, count)
    second := make(chan *PipeTask, count)
    third := make(chan *PipeTask, count)

    line := initQueue(count)

    gen_string := func() {
        for {
            select {
                case pipe_task := <-first:
                pipe_task.generated = true

                pipe_task.start_generating = time.Now()

                pipe_task.source = bmsearch.GenerateRune(STRING_SIZE)
                pipe_task.pattern = bmsearch.GeneratePattern(
                                    pipe_task.source, PATTERN_SIZE)

                pipe_task.end_generatig = time.Now()

                second <- pipe_task
            }
        }
    }

    get_table := func() {
```

```
        for {
            select {
                case pipe_task := <-second:
                pipe_task.skip_table_made = true

                pipe_task.start_table = time.Now()

                pipe_task.skip_table = bmsearch.ConstructSkipTable(
                                       pipe_task.pattern)
                pipe_task.end_table = time.Now()

                third <- pipe_task
            }
        }
    }

    match := func() {
        for {
            select {
                case pipe_task := <-third:
                pipe_task.pattern_mached = true

                pipe_task.start_match = time.Now()
                pipe_task.pattern_index = bmsearch.FindFirstIndex(
                                          pipe_task.source,
                                          pipe_task.pattern,
                                          pipe_task.skip_table)
                pipe_task.end_match = time.Now()

                line.enqueue(pipe_task)
                if (pipe_task.num == count - 1) {
                    ch <- 0
                }
            }
        }
    }
    go gen_string()
    go get_table()
    go match()


    for i := 0; i < count; i++ {
        pipe_task := new(PipeTask)
        pipe_task.num = i + 1
        first <- pipe_task
    }
    return line
}
```

Реализация синхронного конвейера

Процедуры, необходимые для реализации синхронного конвейера представлены на листингах 3.5 – 3.7.

Листинг 3.5 – Получение подстрок

```
func gen_string_sync(task *PipeTask) *PipeTask {
    task.generated = true

    task.start_generating = time.Now()

    task.source = bmsearch.GenerateRune(STRING_SIZE)
    task.pattern = bmsearch.GeneratePattern(task.source, PATTERN_SIZE)

    task.end_generatig = time.Now()

    return task
}
```

Листинг 3.6 – Получение словаря

```
func get_table_sync(task *PipeTask) *PipeTask {
    task.skip_table_made = true

    task.start_table = time.Now()
    task.skip_table = bmsearch.ConstructSkipTable(task.pattern)
    task.end_table = time.Now()

    return task
}
```

Листинг 3.7 – Поиск подстроки

```
func match_sync(task *PipeTask) *PipeTask {
    task.pattern_mached = true

    task.start_match = time.Now()
    task.pattern_index = bmsearch.FindFirstIndex(task.source,
                                task.pattern, task.skip_table)
    task.end_match = time.Now()

    return task
}
```

На листинге 3.8 представлена реализация синхронного конвейера.

Листинг 3.8 – Синхронный конвейер

```
func Sync(count int) *Queue {
    line_first := initQueue(count)
    line_second := initQueue(count)
    line_third := initQueue(count)

    for i := 0; i < count; i++ {
        pipe_task := new(PipeTask)
        pipe_task = gen_string_sync(pipe_task)
        line_first.enqueue(pipe_task)
        if (len(line_first.queue) != 0) {
            pipe_task = get_table_sync(line_first.dequeue())
            line_second.enqueue(pipe_task)
            if (len(line_second.queue) != 0) {
                pipe_task = match_sync(line_second.dequeue())
                line_third.enqueue(pipe_task)
            }
        }
    }
    return line_third
}
```

Поиск подстроки

Реализация получения словаря вида символ - сдвиг представлена на листинге 3.9.

Листинг 3.9 – Получение словаря вида символ - сдвиг

```
func ConstructSkipTable(pattern []rune) map[rune]int {
    length := len(pattern)
    skips := make(map[rune]int)
    for i, char := range pattern {
        if i < length - 1 {
            skips[char] = length - i - 1
        } else {
            skips[char] = length
        }
    }
    return skips
}
```

Генерация строк и подстрок представлена на листинге 3.10. Реализация алгоритма Бойера – Мура – Хорспула представлена на листинге 3.11.

Листинг 3.10 – Генерация строк и подстрок

```
    func GenerateRune(size int) []rune {
        var letters = []rune("abcdefghijklmnopqrstuvwxyz
                              ABCDEFGHIJKLMNOPQRSTUVWXYZ")
        rand.Seed(time.Now().UnixNano())
        b := make([]rune, size)
        for i := range b {
            b[i] = letters[rand.Intn(len(letters))]
        }
        return b
    }
    func GeneratePattern(src []rune, size int) []rune {
        rand.Seed(time.Now().UnixNano())
        start := rand.Intn(len(src) - size)
        length := len(src)

        return src[start : length - 1]
    }
```

Листинг 3.11 – Алгоритм Бойера – Мура – Хорспула

```
func FindFirstIndex(source, pattern []rune, skips map[rune]int) int {
    len_ptrn := len(pattern)
    len_src := len(source)

    j := len_ptrn - 1
    for j < len_src {
        k := j
        i := len_ptrn - 1
        for i >= 0 {
            if source[k] != pattern[i] {
                if skip, found := skips[source[j]]; found {
                    j += skip
                } else {
                    j += len_ptrn
                }
                break
            }
            k--
            i--
        }
        if i < 0 {
            return k + 1
        }
    }
    return -1
}
```

3.4 Тестирование ПО

Тестирование ПО проводилось написанием журналирования для вывода содержания заявок очереди(3.12).

Листинг 3.12 – Тестирование ПО

```
func print_map(m map[rune]int) {
    fmt.Print("( ")
    for i, ch := range m {
        fmt.Printf("%c:%v ", i, ch)
    }
    fmt.Print(") \\\\ \n")
}

func Log(qu *Queue) {
    fmt.Println("")
    line := qu.queue
    for i := range line {
        if line[i] != nil {
            fmt.Printf("%3v & %12v & %10v & ", i,
            string(line[i].source),
            string(line[i].pattern))
            print_map(line[i].skip_table)
            fmt.Printf(" & %v \\\\ \n", line[i].pattern_index)
        }
    }
}
```

На очереди размером в 5 элементов программа показала следующие результаты(3.1):

Таблица 3.1 – Тестирование ПО

№	Строка	Подстрока	Присутствие	Словарь	Результат
1	MeHJWZigCQ	MeHJWZigC	+	(W:4 g:1 C:9 e:7 H:6 J:5 M:8 Z:3 i:2	0
2	quOFEdkUfL	uOFEdkUf	+	(E:4 d:3 k:2 U:1 f:8 u:7 O:6 F:5	1
3	XKkbxZMRku	bxZMRk	+	(R:1 k:6 b:5 x:4 Z:3 M:2	3
4	SyHEexBMWH	yHEexBMW	+	(x:3 B:2 M:1 W:8 y:7 H:6 E:5 e:4	1
5	bXbEVQAUEZ	bXbEVQAUE	+	(E:9 V:4 Q:3 A:2 U:1 b:6 X:7	0

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4. Экспериментальный раздел

Раздел содержит результат сравнительного анализа работы конвейера, работающего на одном потоке и многопоточного конвейера, приведена таблица результатов эксперимента и дана оценка эффективности предложенной в работе реализации алгоритма.

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- операционная система Ubuntu 20.04.1 LTS;
- память 7 GiB;
- процессор Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz [5].

4.2 Описание системы тестирования

Получение характеристик работы очереди осуществлялось с помощью установки штампов на начало и завершение работы каждой из лент конвейера. Переменные для хранения временных штампов описаны в структуре заявки очереди (листинг 3.1). Работа с временными штампами продемонстрирована на листинге 3.4

4.3 Постановка эксперимента

В данном эксперименте тестируется влияние распараллеливания конвейерной обработки на время работы конвейера. Эксперимент проведен на данных типа rune, который в языке Golang является псевдонимом для целочисленного типа размерностью 32 бита[6]. Было выполнено одно снятие временных штампов. Данные не усреднялись. Во время тестирования

устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования. Оптимизация компилятора была отключена.

4.4 Результаты эксперимента

Результаты эксперимента представлены в таблице 4.1.

Таблица 4.1 – Замеры времени работы на очереди размером 20

№	Начало обработки заявки					
	Параллельно, ns			Синхронно, ns.		
	Линия 1	Линия 2	Линия 3	Линия 1	Линия 2	Линия 3
1	0	53934	111522	0	40521	41301
2	42507	96550	112526	41805	83047	83872
3	87564	240258	248300	84239	124716	126116
4	133080	244788	249092	126667	167664	168461
5	174362	246233	249644	168860	209765	210664
6	214072	318217	341519	211164	256655	274355
7	261739	320344	342130	275205	323431	324521
8	302760	472094	482306	325112	366840	368582
9	348170	477270	483114	369605	413718	414936
10	389616	479263	483587	415422	463733	465791
11	429963	480493	484016	466713	507510	508254
12	470433	593634	600897	508604	548992	550028
13	516046	597457	601563	550442	590621	591488
14	556898	598806	601930	591959	632602	633636
15	596784	716283	743636	634021	674088	677474
16	641117	719814	744380	677841	718884	720048
17	680814	741599	744850	735219	777608	779706
18	722195	850056	876476	780280	852850	853711
19	767730	873968	877044	854184	894956	896102
20	808537	875006	878601	896482	936600	937634

Из таблицы можно сделать вывод, что распараллеленный конвейер выполняет работу на 6% быстрее, чем синхронный. Для более подробного анализа работы конвейера следует рассмотреть ряд характеристик, представленных в разделе 1. Таблица с результатами анализа приведена ниже(4.2):

Таблица 4.2 – Анализ временных замеров

Характеристика		Параллельно, ns,			Синхронно, ns.		
Линия		1	2	3	1	2	3
Простой очереди	gen.	23807	700566	761270	68686	858165	891498
	min	0	0	0	1200	40554	41244
	max	5421	141752	141500	18744	73787	73762
	avg	1190	35028	38063	3434	42908	44574
Время заявки в системе	min	45238			41325		
	max	179982			73655		
	avg	102424			43826		

При распараллеливании конвейера на три потока возникает ситуация, когда простоя в очереди нет вовсе. Так же время простоя первой, второй и третьей линии больше на синхронном конвейере в $2.8, 1, 2$ и $1, 7$ раз, чем на параллельном соответственно.

Однако, среднее время заявки в системе на синхронном конвейере в ≈ 2.3 раза меньше, чем время заявки в синхронном конвейере. Соответственно, выигрыш происходит исключительно за счет обеспечения меньшего простоя очереди и ситуаций, когда простоя нет вовсе.

4.5 Вывод

Конвейер, реализованный на параллельных процессах обеспечивает ситуации с минимальным или вовсе отсутствующим простоем лент. Однако, за счет затрат на обеспечение реентерабельности функций, работы с атомарными операциями и буферизации потоков, в среднем заявка находится в системе дольше, чем заявка в синхронном конвейере.

Заключение

В работе была теоретически проанализирована организация конвейерная обработка данных. Вычислительный конвейер был реализован на трех потоках и на одном потоке. Алгоритм Бойера – Мура – Хорспула был декомпозирован для обработки конвейерными линиями. Эксперимент показал, что распараллеливание алгоритма Бойера – Мура – Хорспула приводит к выигрышу в 6%. Выигрыш происходит исключительно за счет обеспечения меньшего простоя очереди и ситуаций, когда простоя нет вовсе.

Этапы обработки стабильны по времени в пределах погрешности лишь за счет некоторых ограничений, которые были указаны в разделе 2 – такое решение было принято с целью усреднения времени работы каждой из лент конвейера.

Следовательно, реализация данного алгоритма на параллельном конвейере хоть и эффективна, но на ограниченном объеме данных.

Список литературы

[1] *Pipeline Processing in digital logic design.* URL: https://www.fullchipdesign.com/pipeline_space_time_architecture.htm (дата обр. 19.10.2021).

[2] Michael J. (Michael Jay) Quinn. *Parallel programming in C with MPI and openMP.* Dubuque, Iowa : McGraw-Hill, 2004.

[3] Moore J. S. Boyer R. S. "A fast string searching algorithm". в: *Communications of the ACM.* (1977).

[4] Donald Knuth. "The Art of Computer Programming. Volume 1: Fundamental Algorithms, Third Edition." в: Addison-Wesley, 1997. гл. Stacks, Queues, and Dequeues, с. 238—243.

[5] *Процессор Intel® Core™ i3-8145U.* URL: https://ark.intel.com/content/www/ru/ru/ark/products/149090/intel-core-i3-8145u-processor-4m-cache-up-to-3-90-ghz.html (дата обр. 29.09.2021).

[6] *Strings, bytes, runes and characters in Go.* URL: https://go.dev/blog/strings (дата обр. 22.09.2021).