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

Тема: Реализация конвейера

Студент: Сироткина П.Ю.

Группа: ИУ7-56Б

Оценка: _____

Преподаватель: Волкова Л.Л.

Москва, 2021 г.

Содержание

Введение

В данной лабораторной работе рассматривается реализация конвейерной обработки данных.

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).

Алгоритм конвейерной обработки может быть последовательным и параллельным. Последовательный алгоритм делает все "в одну ленту". По сути, это обычная программная линейная реализация. Подробно рассматривать этот способ реализации не имеет смысла.

Идея второй реализации заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции 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.

Понятие конвейра исторически связано с декомпозицией задач и автоматизацией производства, такой, что однотипную работу, т.е. например одну стадию технического процесса, выполняет один человек или бригада, и эти исполнители предварительно были обучены конкретному типу работ. Конвейер часто снабжен так называемой лентой (передающим механизмом), которая передает работникам изделия, которые им в свою очередь нужно обработать.

Целью лабораторной работы является изучение и реализация парраллельного алгоритма конвейерной обработки данных.

Для достижения поставленной цели необходимо выполнить следующие задачи:

• Изучить теоретическую часть алгоритма параллельной конвейерной обработки данных;

• Выбрать алгоритм для параллельной конвейерной обработки и дать его описание;

• Привести схемы алгоритма параллельной конвейерной обработки данных, а также схемы выбранного алгоритма для обработки;

• Описать используемые типы данных;

• Описать структуру разрабатываемого ПО;

• Реализовать конвейер с количеством лент не меньше трех в многопоточной среде;

• Протестировать разработанное ПО;

• Исследовать эффективность конвейерной реализации по сравнению с последовательной реализацией;

• На основании проделанной работы сделать выводы.

1 Аналитический раздел

В данном разделе описаны основные идеи алгоритма конвейерных вычислений. Поставлена задача, для которой будет применяться этот алгоритм в качестве примера[1].

1.1 Описание алгоритма параллельной конвейерной обратки данных

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).

На рисунке 1.1 изображена схема, иллюстрирующая работу конвейера с параллельной реализацией.

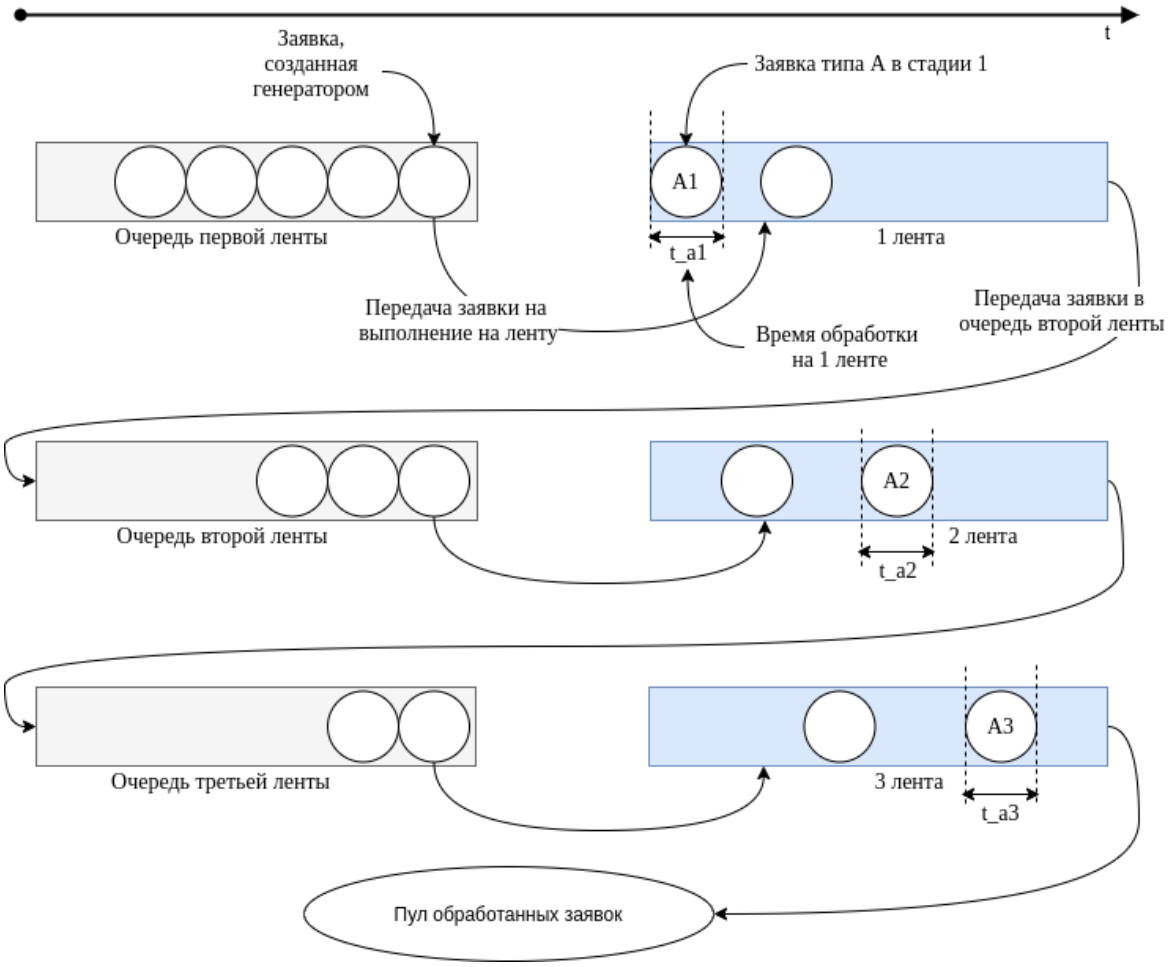

Рисунок 1.1 — Схема, иллюстрирующая работу конвейера с параллельной реализацией

Примечание:

Для примера рассмотрено количество лент, равное 3, однако на практике это число не фиксировано.

Рассмотрим, как заявка перемещается между разными лентами (см. рисунок 1.2).

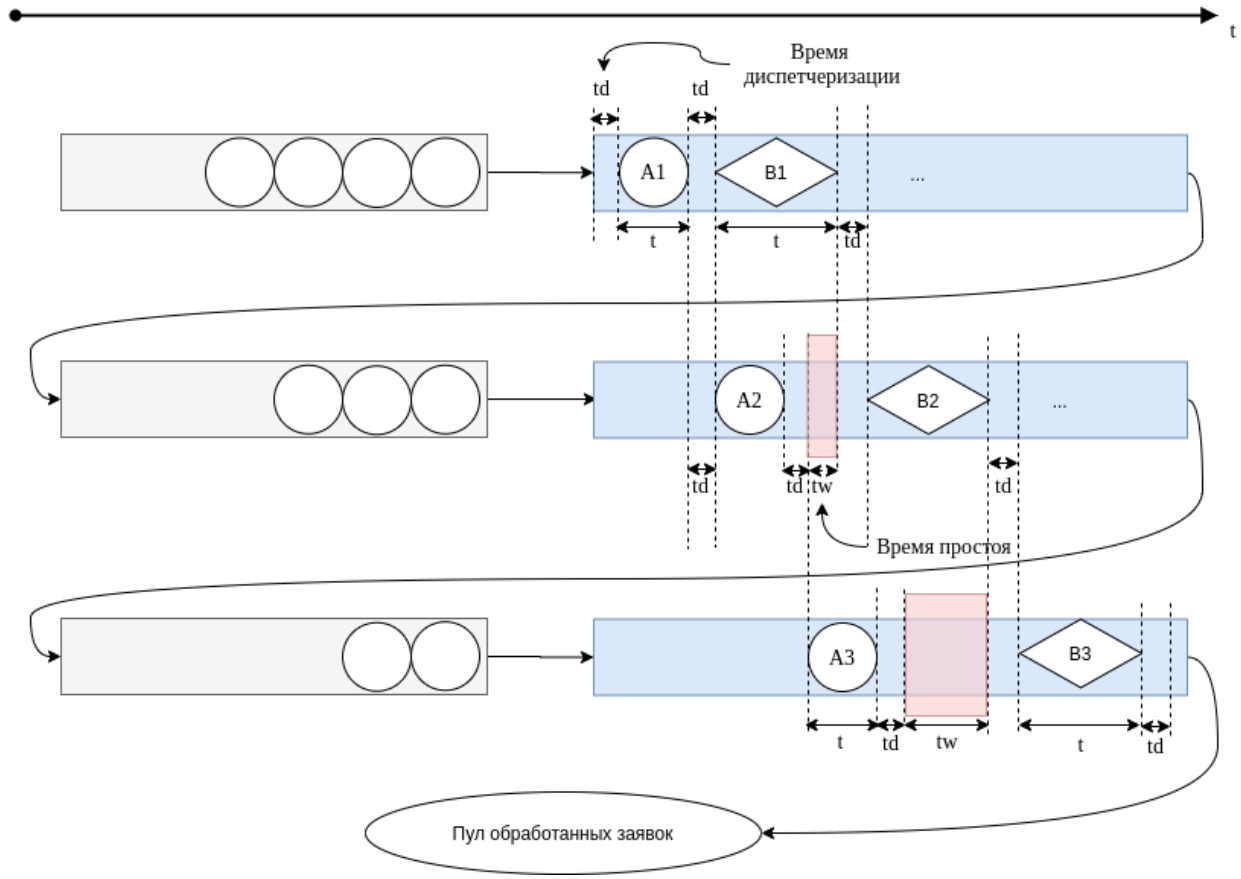

Рисунок 1.2 — Схема, иллюстрирующая пример работы конвейра с параллельной реализацией

Время диспетчеризации (td) - время, в течение которого заявка снимается/добавляется в очередь.

Время простоя (tw) - время, которое тратится на ожидание заявки.

По этому рисунку можно сделать вывод, что в зависимости от сложности заявок может сложиться такая ситуация, что лента будет простаивать.

Например, если две первые заявки более трудоемкие, чем третья, то третья заявка будет долго ждать обработки на других лентах, а кон-

кретно до тех пор, пока не обработаются первые две. Однако, если бы она была первой в очереди на обработку первой лентой, то время простоя бы сократилось.

Может возникнуть ситуация, когда добавление новой заявки в очередь и снятие заявки из этой же очереди происходят одновременно, поэтому доступ к очереди целесообразно осуществлять с использованием мьютекса для целостности данных.

Лента "живет"в терминологии потока, пока либо не выполнит план, либо не получит сигнал о завершении работы.

Нельзя завершить задачу, пока она не обработана до конца, поэтому, даже если был получен сигнал о завершении, необходимо дождаться завершения выполнения заявок, которые на момент получения сигнала находились в состоянии обработки.

Можно провести аналогию с реальной жизнью, например, сотрудники в некоторых банках перестают запускать людей за 10-15 минут до закрытия и в этот период консультируют оставшихся клиентов("заявки").

Генератор заявок может быть статический или динамический. В случае статического очередь формируется заранее ("план на день"), в случае динамического - очередь может формироваться в течение некого периода в зависимости от условий.

1.2 Постановка и описание задачи для конвейерной обработки

В данной лабораторной работе алгоритм конвейерных вычислений изучается на примере алгоритмов поворота, переноса и масштабирования фигуры, представленной в виде массива точек в двумерном растре.

Эта задача является весьма актуальной, т.к. компьютерная графика стала неотъемлемой частью повседневной интернет-жизни человека, и существует потребность в быстром рендеринге изображения.

Если, например, сначала поворачивать отдельно каждую точку, а затем отдельно масштабировать ее, то очевидно, что при большом количестве и сложности фигур изображение будет генерироваться ощутимо

долго, что будет приносить человеку дискомфорт при восприятии изображения.

Если же использовать конвейерный подход и, например, повернув первую точку фигуры, дальше приступить к ее масштабированию и одновременно с этим начать поворачивать вторую точку, то процессорное время будет использоваться более эффективно и изображение будет генерироваться быстрее.

Например, потребность выполнения таких операций может иметь место в анимации компьютерных игр, т.е. при процессировании динамических изображений.

Как известно, в экранной плоскости изображение представляет из себя набор пикселей (точек). Очевидно, что какая-либо фигура - это тоже набор точек экранной плоскоти, и для поворота, переноса или масштабирования фигуры требуется над каждой ее точкой произвести преобразование для получения новой позиции.

Если использовать один поток для рендеринга изображения, то при большом количестве и сложности фигур изображение будет генерироваться ощутимо долго, что будет приносить человеку дискомфорт при восприятии, однако если распараллелить этот процесс, т.е. параллельно генерировать части изображения (т.к. эта операция выполняется независимо для каждой точки), то это может дать коллосальной прирост производительности.

Перейдем к рассмотрению алгоритмов поворота, переноса и масштабирования точек двумерного растра.

Система уравнений (1.1) описывает поворот точки вокруг некоторой точки $(x;\ y)$ относительно точки $(x_c;\ y_c)$ на угол ϕ:

$$\begin{cases} x_{new} = x_c + (x - x_c) \cdot cos(\phi) + (y - y_c) \cdot sin(\phi); \\ y_{new} = y_c - (x - x_c) \cdot sin(\phi) + (y - y_c) \cdot cos(\phi), \end{cases} \tag{1.1}$$

Система уравнений (1.2) описывает масштабирование с коэффициентами $(k_x;\ k_y)$ для точки $(x;\ y)$ относительно точки $(x_c;\ y_c)$:

$$\begin{cases} x_{new} = (x - x_c) \cdot k_x + x_c; \\ y_{new} = (y - y_c) \cdot k_y + y_c \end{cases} \tag{1.2}$$

Система уравнений (1.3) описывает перенос точки $(x;\ y)$, описываемый смещением $(dx;\ dy)$:

$$\begin{cases} x_{new} = x + dx; \\ y_{new} = y + dy, \end{cases} \tag{1.3}$$

Во всех системах уравнений (1.1)-(1.3) значения $(x_{new};\ y_{new})$ - новое положение точки после операции преобразования.

1.3 Вывод

В данном разделе были описаны основные идеи алгоритма параллельной конвейерной обработки данных, а также поставлена задача, для которой будет применяться этот алгоритм в качестве примера.

2 Конструкторский раздел

В данном разделе представлена схема, поясняющая принцип работы алгоритма конвейерной обработки данных, а также схема алгоритма поставленной задачи.

2.1 Схема работы алгоритма конвейерной обработки данных

На рисунке 2.1 представлена схема, поясняющая принцип работы конвейерной обработки данных.

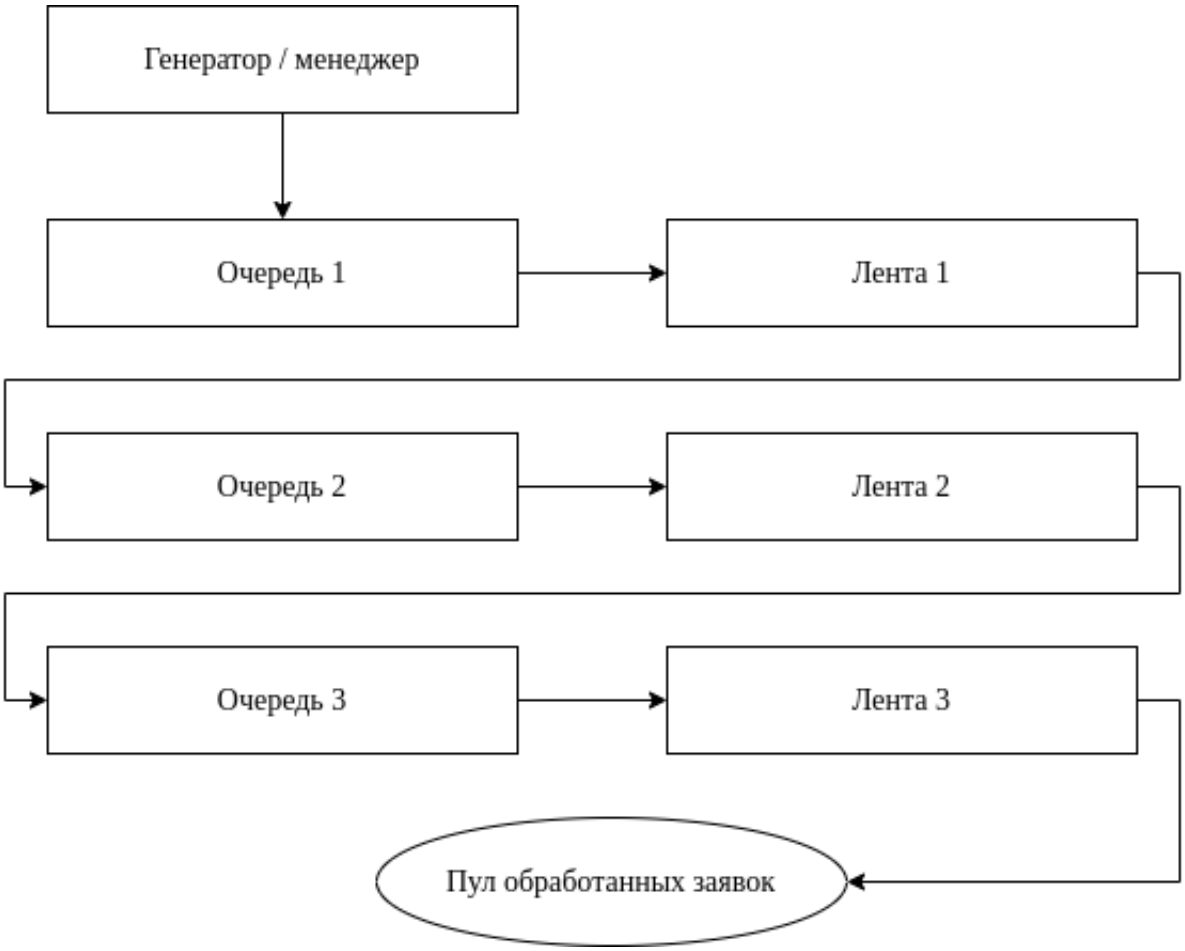

Рисунок 2.1 — Схема, иллюстрирующая работу конвейра с параллельной реализацией

2.2 Схема алгоритма поставленной задачи

На рисунке 2.2 представлена схема алгоритма поставленной задачи (трансформация массива точек в думерном растре).

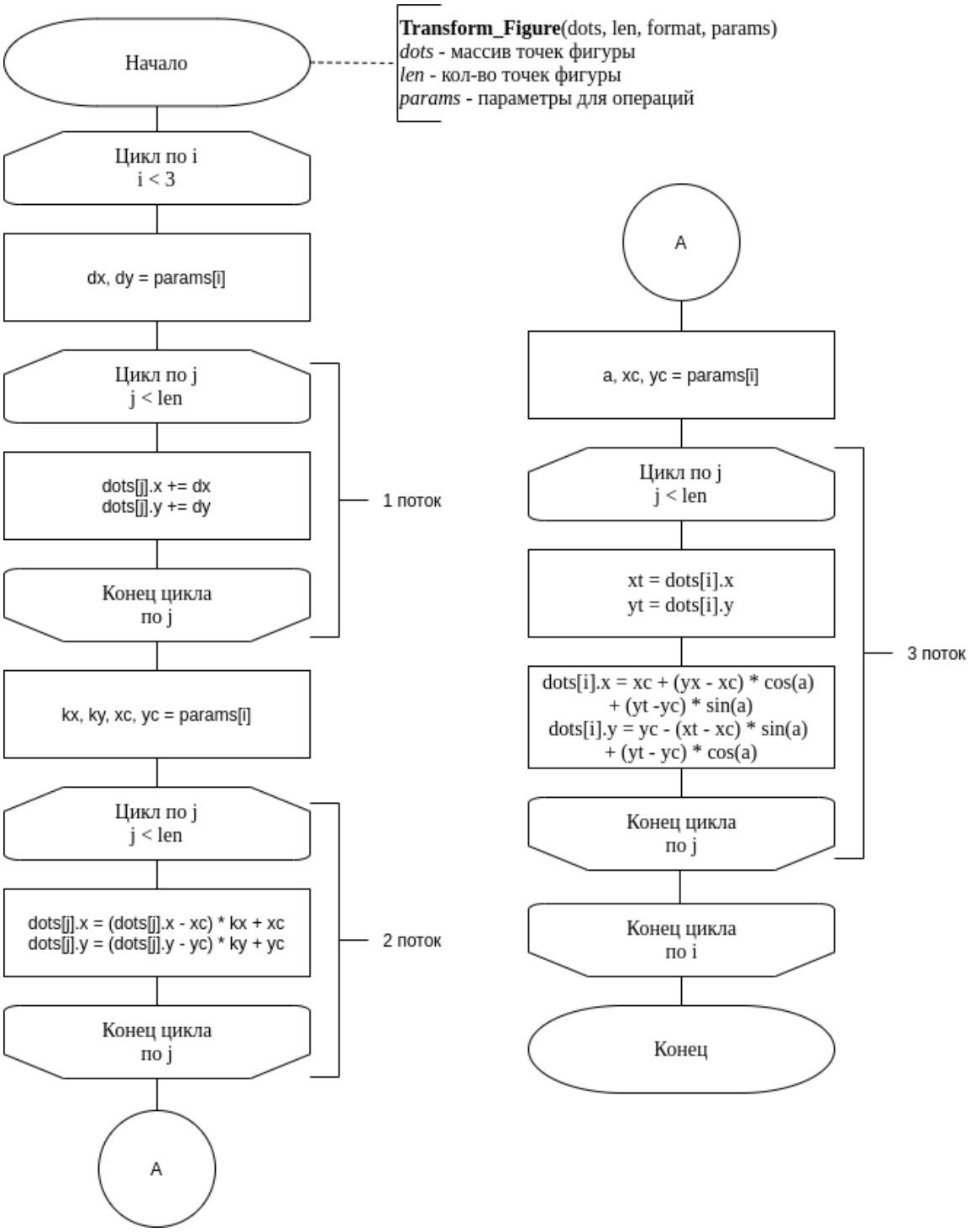

Рисунок 2.2 — Схема алгоритма поставленной задачи

2.3 Описание памяти, используемой программой

Затраты с точки зрения памяти на функционирование ПО складываются из:

а) Количество точек фигуры, т.е. размер очереди.

б) Размер структуры, определяющей заявку - содержит 6 временных меток, описание точки и ее порядковый номер.

2.4 Структура ПО

Программа будет включать в себя один смысловой модуль, называемый **pipeline**, который содержит в себе процедуры и функции, связанные с реализацией параллельного и синхронного конвейера. Независимо от модуля будет существовать файл main.go, реализующий мост между пользователем и модулем, описанным ранее.

2.5 Вывод

На основе теоретических данных, полученных из аналитического раздела, были разработаны схемы требуемых алгоритмов. Рассмотрены затраты по памяти и описана структура ПО.

3 Технологический раздел

В данном разделе приведны требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляются следующие требования:

• На вход программе подается файл с именем localfile.txt, который определен системой заранее. Порядок осуществления операций: перемещение, поворот, масштабирование. В начале файла идут 3 строки, каждая из которых хранит через пробел параметры, необоходимые для каждой операций: для перемещения - (xc, yc, dx, dy), для поворота - (xc, yc, a), для масштабирования - (xc, yc, kx, ky). Затем идет строка, в которой записано количество точек в фигуре. Дальше до конца файла описываются координаты каждой точки фигуры в двумерном растре, каждая пара координат записывается на отдельной строке, координаты разделены пробелом;

• Результатом работы программы является новый файл PipelineResult.txt, содержащий в себе массив точек преобразованной фигуры. В начале файла идет строка, содержащая количество точек, а затем построчно новые координаты. Также создается служебный файл SyncResult.txt для контроля корректности решения;

• Программа должна выполнять логирование: должен быть отдельный пункт меню, задача которого - выводить в упорядоченном по времени порядке события начала и конца обработки каждой заявки в каждой стадии на ленте, а также собрать и вывести требуемую статистику.

• Программа не проверяет корректность служебных файлов.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран язык Golang. Выбор этого языка обусловлен наличием требуемого функционала для организации параллельных вы-

числений. Также выбор обусловлен моим желанием получить больше практики в написании программ на этом языке.

3.3 Листинг кода

В листинге 3.1 представлены пользовательские типы данных.

Листинг 3.1 — Пользовательские типы данных

```
1  package pipeline
2
3  import "time"
4
5  type Dot_t struct {
6      x int
7      y int
8  }
9
10 type Figure_t struct {
11     Size int
12     Dots []Dot_t
13 }
14
15 type File_data_t struct {
16     Figure Figure_t
17     params [3][4]int
18 }
19
20 type Task_t struct {
21     dot Dot_t
22     index int
23
24     start_move   time.Time
25     end_move     time.Time
26
27     start_rotate time.Time
28     end_rotate   time.Time
29
30     start_scale  time.Time
31     end_scale    time.Time
32 }
33
34 type Queue_t struct {
35     Q [](*Task_t)
36     Size int
37 }
```

В листинге 3.2 представлена реализация очереди.

Листинг 3.2 — Реализация очереди

```
38 package pipeline
39
40 func CreateQueue(n int) *Queue_t {
41     q := new(Queue_t)
42     q.Q = make([](*Task_t), n, n)
43     q.Size = -1
44     return q
45 }
46
47 func (q *Queue_t) push(item *Task_t){
48     if q.Size != len(q.Q) - 1 {
49         q.Q[q.Size + 1] = item
50         q.Size++
51     }
52 }
53
54 func (q *Queue_t) pop() *Task_t {
55     item := q.Q[0]
56     q.Q = q.Q[1:]
57     q.Size--
58
59     return item
60 }
```

В листинге 3.3 представлена реализация синхронного конвейера[2].

Листинг 3.3 — Реализация синхронного конвейера

```
61 package pipeline
62
63 import (
64     "math"
65     "time"
66 )
67
68 func Sync(data File_data_t) *Queue_t {
69     pipeline_1 := CreateQueue(data.Figure.Size)
70     pipeline_2 := CreateQueue(data.Figure.Size)
71     pipeline_3 := CreateQueue(data.Figure.Size)
72
73     dx := data.params[0][0]
74     dy := data.params[0][1]
75
```

```
76      kx := data.params[2][2]
77      ky := data.params[2][3]
78
79      a := float64(data.params[1][2])
80
81      r_xc := data.params[1][0]
82      r_yc := data.params[1][1]
83
84      s_xc := data.params[2][0]
85      s_yc := data.params[2][1]
86
87      for i := 0; i < data.Figure.Size; i++ {
88          task := new(Task_t)
89          task.start_move = time.Now()
90          task.dot.x, task.dot.y = smove(data.Figure.Dots[i].x,
                data.Figure.Dots[i].y, dx, dy)
91          task.end_move = time.Now()
92          pipeline_1.push(task)
93      }
94
95      for i := 0; i < data.Figure.Size; i++ {
96          task := pipeline_1.pop()
97          task.start_rotate = time.Now()
98          task.dot.x, task.dot.y = srotate(task.dot.x, task.dot.y, r_xc,
                r_yc, a)
99          task.end_rotate = time.Now()
100         pipeline_2.push(task)
101     }
102
103     for i := 0; i < data.Figure.Size; i++ {
104         task := pipeline_2.pop()
105         task.start_scale = time.Now()
106         task.dot.x, task.dot.y = sscale(task.dot.x, task.dot.y, s_xc,
                s_yc, kx, ky)
107         pipeline_3.push(task)
108         task.end_scale = time.Now()
109     }
110
111     return pipeline_3
112 }
113
114 func smove(x int, y int, dx int, dy int) (int, int) {
115     x += dx
116     y += dy
117
118     return x, y
```

```
119 }
120
121 func sscale(x int, y int, xc int, yc int, kx int, ky int) (int, int) {
122     x = (x - xc) * kx + xc
123     y = (y - yc) * ky + yc
124     return x, y
125 }
126
127 func srotate(x int, y int, xc int, yc int, angle int) (int, int) {
128     xt := x
129     yt := y
130
131     a := float64(float64(angle) * 3.14 / 180)
132
133     x = xc + int(float64(xt - xc) * math.Cos(a) + float64(yt - yc) *
            math.Sin(a))
134     y = yc - int(float64(xt - xc) * math.Sin(a) - float64(yt - yc) *
            math.Cos(a))
135
136     return x, y
137 }
```

В листинге 3.4 представлена реализация параллельного конвейера.

Листинг 3.4 — Реализация параллельного конвейера

```
138 func Pipeline(n int, ch chan int, data *File_data_t) *Queue_t {
139     pipeline_1 := make(chan *Task_t, 50000)
140     pipeline_2 := make(chan *Task_t, 50000)
141     pipeline_3 := make(chan *Task_t, 50000)
142
143     q := CreateQueue(n)
144
145     dx := data.params[0][0]
146     dy := data.params[0][1]
147
148     r_xc := data.params[1][0]
149     r_yc := data.params[1][1]
150     angle := float64(data.params[1][2])
151     a := float64(float64(angle) * 3.14 / 180)
152
153     s_xc := data.params[2][0]
154     s_yc := data.params[2][1]
155     kx := data.params[2][2]
156     ky := data.params[2][3]
157
```

```
158     go func() {
159         for {
160             select {
161             case task := <-pipeline_1:
162 
163                 task.start_move = time.Now()
164                 task.dot.x += dx
165                 task.dot.y += dy
166                 task.end_move = time.Now()
167 
168                 pipeline_2 <- task
169 
170             case task := <- pipeline_2:
171 
172                 task.start_rotate = time.Now()
173                 xt := task.dot.x
174                 yt := task.dot.y
175                 task.dot.x = int(float64(r_xc) + float64((xt - r_xc)) *
                        math.Cos(a) + float64((yt - r_yc)) * math.Sin(a))
176                 task.dot.y = int(float64(r_yc) - float64((xt - r_xc)) *
                        math.Sin(a) + float64((yt - r_yc)) * math.Cos(a))
177                 task.end_rotate = time.Now()
178 
179                 pipeline_3 <- task
180 
181             case task:= <- pipeline_3:
182 
183                 task.start_scale = time.Now()
184                 task.dot.x = (task.dot.x - s_xc) * kx + s_xc
185                 task.dot.y = (task.dot.y - s_yc) * ky + s_yc
186                 task.end_scale = time.Now()
187 
188                 q.push(task)
189                 if task.index == (n-1) {
190                     ch <- 0
191                 }
192             }
193         }
194     }()
195 
196     for i := 0; i < n; i++ {
197         task := new(Task_t)
198         task.dot = data.Figure.Dots[i]
199         task.index = i
200         pipeline_1 <- task
201     }
```

```
202
203     return q
204 }
```

3.4 Тестирование ПО

В таблицах 3.1-3.4 приведены тестовые данные, на которых было протестировано разработанное ПО. Все тесты были успешно пройдены.

Таблица 3.1 — Тестирование операции переноса

(x; y)	(dx; dy)	Ожидание	Результаты
[55; 55]	[10; 10]	[65;65]	[65;65]
[55; 55]	[0; 0]	[55;55]	[55;55]
[55; 55]	[-10; -10]	[45;45]	[45;45]
[55; 55]	[10; -10]	[65;45]	[65;45]
[55; 55]	[-60; -60]	[-5;-5]	[-5;-5]

Таблица 3.2 — Тестирование операции поворота

(x; y)	angle	(xc; yc)	Ожидание	Результаты
[55;55]	0	[0;0]	[55;55]	[55;55]
[55;55]	60	[0;0]	[75;-20]	[75;-20]
[55;55]	-60	[0;0]	[-20;75]	[-20;75]
[55;55]	37	[0;0]	[77;10]	[77;10]
[55;55]	-37	[0;0]	[10;77]	[10;77]
[55;55]	0	[1;1]	[55;55]	[55;55]
[55;55]	60	[-1;-1]	[75;-21]	[75;-21]
[55;55]	-60	[100;100]	[116;39]	[116;39]
[55;55]	37	[55;55]	[55;55]	[55;55]
[55;55]	-37	[10;20]	[24;75]	[24;75]

Таблица 3.3 — Тестирование операции масштабирования

(x; y)	(kx; ky)	(xc; yc)	Ожидание	Результаты
[55;55]	[2;2]	[55;55]	[55;55]	[55;55]
[55;55]	[2;2]	[0;0]	[110; 110]	[110; 110]
[55;55]	[2;2]	[-55;-55]	[165;165]	[165;165]
[55;55]	[2;2]	[100;-100]	[10;210]	[10;210]
[55;55]	[0;0]	[55;55]	[55;55]	[55;55]
[55;55]	[0;0]	[10;-10]	[10;-10]	[10;-10]
[55;55]	[1;1]	[55;55]	[55;55]	[55;55]
[55;55]	[1;1]	[11;11]	[55;55]	[55;55]
[55;55]	[1;1]	[-11;11]	[55;55]	[55;55]
[55;55]	[-2;-2]	[55;55]	[55;55]	[55;55]
[55;55]	[-2;-2]	[0;0]	[-110;-100]	[-110;-100]
[55;55]	[-2;-2]	[-55;-55]	[-275;-275]	[-275;-275]
[55;55]	[-2;-2]	[100;-100]	[190;-410]	[190;-410]

Здесь (x;y) - начальные координаты точки, (xc;yc) - точка центра преобразования, (dx; dy) - коэффициенты переноса, (kx; ky) - коэффициенты масштабирования, a - угол поворота.

3.5 Вывод

В данном разделе было представленно реализованное ПО для решения поставленной задачи, а также проведено тестирование.

4 Экспериментальный раздел

В данном разделе представлен пользовательский интерфейс, а также проведена оценка эффективности алгоритмов.

4.1 Пример работы

На рисунке 4.1 приведен пример работы программы.

```
polina@polina-IdeaPad-5-14ARE05:~/aa/bmstu_aa/lab_05/src$ go run main.go

Выберите действие:
1) Считать данные из файла и произвести операции преобразования с помощью конвейера
2) Показать таблицу логирования
3) Завершить программу

Ваш ответ: 1
1 лента. Перемещение точки:     (115;116), №  0, time:2021-11-06 21:37:27.806196132 +0300 MSK m=+1.685695077
1 лента. Перемещение точки:     (226;227), №  1, time:2021-11-06 21:37:27.806574131 +0300 MSK m=+1.686073076
1 лента. Перемещение точки:     (337;338), №  2, time:2021-11-06 21:37:27.806589721 +0300 MSK m=+1.686088676
2 лента. Поворот точки:         (142; 42), №  0, time:2021-11-06 21:37:27.806603327 +0300 MSK m=+1.686102282
2 лента. Поворот точки:         (298; 29), №  1, time:2021-11-06 21:37:27.806619869 +0300 MSK m=+1.686118814
3 лента. Масштабирование точки: (234; 34), №  0, time:2021-11-06 21:37:27.806629487 +0300 MSK m=+1.686128442
1 лента. Перемещение точки:     (448;449), №  3, time:2021-11-06 21:37:27.806640559 +0300 MSK m=+1.686139504
3 лента. Масштабирование точки: (546;  8), №  1, time:2021-11-06 21:37:27.806650107 +0300 MSK m=+1.686149062
1 лента. Перемещение точки:     (559;560), №  4, time:2021-11-06 21:37:27.806659526 +0300 MSK m=+1.686158481
2 лента. Поворот точки:         (455; 15), №  2, time:2021-11-06 21:37:27.806669264 +0300 MSK m=+1.686168209
2 лента. Поворот точки:         (611;  1), №  3, time:2021-11-06 21:37:27.806679284 +0300 MSK m=+1.686178229
3 лента. Масштабирование точки: (860;-20), №  2, time:2021-11-06 21:37:27.806689143 +0300 MSK m=+1.686188098
3 лента. Масштабирование точки: (1172;-48), №  3, time:2021-11-06 21:37:27.806704693 +0300 MSK m=+1.686203638
2 лента. Поворот точки:         (767;-11), №  4, time:2021-11-06 21:37:27.806717878 +0300 MSK m=+1.686216823
3 лента. Масштабирование точки: (1484;-72), №  4, time:2021-11-06 21:37:27.806728548 +0300 MSK m=+1.686227504

Время, затраченное на обработку файла local.txt: 864.927µs

Выберите действие:
1) Считать данные из файла и произвести операции преобразования с помощью конвейера
2) Показать таблицу логирования
3) Завершить программу

Ваш ответ:
```

Рисунок 4.1 — Пример работы программы

4.2 Технические характеристики

Технические характеритиски машины, на которой выполнялось тестирование:

— Операционная система: Ubuntu[3] Linux[4] 20.04 64-bit.
— Оперативная память: 16 Gb.
— Процессор: AMD(R) Ryzen(TM)[5] 5 4500U CPU @ 2.3 CHz

4.3 Постановка эксперимента

Проводятся эксперименты следующего типа: анализируя временные метки, составляются следующие сравнительные таблицы:

а) Таблица, содержащая максимальное, минимальное и среднее время нахождения заявки в системе для каждой из реализаций.

б) Таблица, содержащая максимальное, минимальное и среднее время простоя системы для каждой из реализаций.

в) Таблица, содержащая время начала обработки каждой заявки для каждой очереди для каждой из реализаций.

4.4 Результаты эксперимента

Результаты эксперимента представлены в таблицах 4.1-4.3.

Таблица 4.1 — Максимальное, минимальное и среднее время нахождения заявки в системе

Время, наносекунды	Параллельная реализация	Синхронная реализация
Мин.	1833	18855
Макс.	21360	27882
Среднее	12053	31647

Таблица 4.2 — Максимальное, минимальное и среднее время в наносекундах простоя системы

Время, наносекунды	Параллельная реализация	Синхронная реализация
Мин.	1253	40887
Макс.	19517	40887
Среднее	975	2044

Таблица 4.3 — Время в наносекундах начала обработки заявок

№ заявки	Л1(П)	Л2(П)	Л3(П)	Л1(С)	Л2(С)	Л3(С)
1	0	3387	8867	0	10239	18935
2	7254	7935	12784	500	10760	19226
3	11472	14057	14849	871	11191	19517
4	12083	15450	16281	1262	11622	19807
5	13346	16952	17834	4338	12042	20097
6	18435	19798	24627	4729	12473	20388
7	19106	23084	25208	5129	12894	20679
8	20589	23845	26420	5500	13325	20969
9	21240	26981	32101	5891	13746	21260
10	21892	27743	33383	6252	14176	21550
11	22473	28544	33965	6612	14637	21841
12	25789	29306	36018	6983	15058	22132
13	30107	31329	37311	7344	15489	22422
14	30688	32662	39295	7714	15910	22712
15	34516	35137	42070	8075	16340	22993
16	36659	37872	43863	8435	16761	23284
17	38673	40497	45176	8806	17192	23574
18	39856	41308	46608	9167	17613	23865
19	42611	44414	47179	9528	18044	24155
20	43232	45787	47781	9898	18464	24446

Здесь Л1, Л2, Л3 означает Линия 1, Линия 2, Линия 3. П, С - параллельная или синхронная реализация.

4.5 Вывод

Как видно из таблицы 4.1, среднее время нахождения заявки в системе, реализующей параллельную версию конвейера, почти в 3 раза меньше, чем среднее время для синхронной реализации. Это объясняется тем, что при параллельной реализации заявка, пройдя первую стадию,

переходит на вторую при первой же возможности, в то время как при синхронной реализации первая заявка с первой ленты будет "ждать когда обработаются остальные заявки на первой ленте, следственно, вторая лента простаивает без дела.

Как видно из таблицы 4.2, среднее время простоя заявки в системе, реализующей параллельную версию конвейера, более чем в 2 раза меньше, чем среднее время для синхронной реализации. Этим объясняется различие в среднем времени нахождения заявки в системе для каждой из реализаций.

Однако, как видно из таблицы 4.3, при поставленной задаче преимущества параллельной реализации не настолько ощутимы. Очевидно, что переключение между лентами занимает больше единиц процессорного времени, чем последовательная передача управления при синхронной реализации (примерно 3000 единиц процессорного времени при параллельной реализации против 500 единиц при синхронной реализации). Выигрыш осуществляется за счет уменьшенного времени простоя лент, однако при данной реализации поставленная задача имеет низкую трудоемкость.

Несмотря на то, что сложилась такая ситуация, тем не менее из нее можно сделать важный вывод. Решая, какую реализацию использовать (параллельную или синхронную), стоит оценить трудоемкость задачи. Если затраты на переключение между лентами (диспетчеризация) нивелируется трудоемкостью задачи, то лучше отдать предпочтение параллельной реализации. Если задача имеет низкую трудоемкость, то следует использовать синхронную реализацию.

Заключение

Работая с алгоритмами конвейерных вычислений, следует учитывать следующие факторы:

— Минимальное, максимальное и среднее время простоя заявки в системе;

— Минимальное, максимальное и среднее время решения каждого из этапов задачи;

— Время переключения между лентами;

— Время последовательной передачи управления.

Если затраты на переключение между лентами (диспетчеризация) нивелируется трудоемкостью задачи, то лучше отдать предпочтение параллельной реализации. Если задача имеет низкую трудоемкость, то следует использовать синхронную реализацию.

При решении поставленной задачи сложилась ситуация, что затраты на создание и переключение между лентами конвейера с параллельной реализацией не окупаются затратами на решение задачи на каждой з лент. Несмотря на это, был сделан важный вывод: не каждую задачу следует программно оптимизировать. Оптимизация часто заключается в использовании некого механизма, который требует больше ресурсов, чем обычная реализация, однако его идея должна заключаться в более удачном использовании больших ресурсов, но следует понимать, что так может происходить не всегда. Решать, следует ли оптимизировать поставленную задачу или нет, следует отталкиваясь от конкретного способа оптимизации вычислений.

Список использованных источников

1. Реализация параллельных вычислений: MPI, OpenMP, кластеры, грид, многоядерные процессоры, графические процессоры, квантовые компьютеры. — Г.И. Шпаковский, 01.09.2011.

2. Анатомия каналов в Go. — [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/490336/., 06.11.2021.

3. Ubuntu. — [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Ubuntu., 16.09.2021.

4. Linux. — [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Linux., 16.09.2021.

5. Процессор AMD Ryzen(TM) 5. — [Электронный ресурс]. Режим доступа: https://shop.lenovo.ru/product/81YM007FRU/., 21.09.2021.