



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 Алгоритм конвейерной обработки

Студент Богаченко А.Е.

Группа ИУ7-56Б

Оценка (баллы) _____

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

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Параллельное программирование	4
1.2 Организация взаимодействия параллельных потоков	5
2 Конструкторская часть	6
2.1 Распараллеливание программы	6
3 Технологическая часть	7
3.1 Требования к ПО	7
3.2 Структура ПО	7
3.3 Средства реализации	7
3.4 Листинг кода	8
4 Исследовательская часть	11
4.1 Пример работы	11
4.2 Технические характеристики	11
4.3 Время выполнения алгоритмов	11
4.4 Производительность алгоритмов	13
Заключение	14
Литература	15

Введение

Цель работы: изучение возможности конвейерной обработки и использование такого подхода на практике. Необходимо сравнить времени работы алгоритма на нескольких потоках и линейную реализацию.

В ходе лабораторной работы предстоит:

- реализовать конвейер на потоках;
- реализовать линейную обработку;
- провести сравнение времени работы.

1 Аналитическая часть

Конвейер - система поточного производства [1]. В терминах программирования ленты конвейера представлены функциями, выполняющими над неким набором данных операции и предающие их на следующую ленту конвейера. Моделирование конвейерной обработки хорошо сочетается с технологией многопоточного программирования - под каждую ленту конвейера выделяется отдельный поток, все потоки работают в асинхронном режиме.

В качестве предметной области я решил выбрать торты - на первой линии конвейера происходит подготовка, на второй выпекание, на третьей проводят финальные работы по приготовлению.

1.1 Параллельное программирование

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время многоядерные процессоры, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство.

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений

достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки, расширенные некоторым набором операторов для параллельных вычислений.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (API) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер.

1.2 Организация взаимодействия параллельных потоков

Потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия.

Вывод

Была рассмотрена конвейерная обработка данных, технология параллельного программирования и организация взаимодействия параллельных потоков.

2 Конструкторская часть

Требования к вводу:

На ввод подается целое число - желаемое количество изготовленных экземпляров

Требования к программе:

- вывод статистики обработанных экземпляров.

2.1 Распараллеливание программы

Распараллеливание программы должно ускорять время работы. Это достигается за счет перенесения каждой из лент конвейера на отдельный поток.

Вывод

В данном разделе были рассмотрен спосо распараллеливания конвейера.

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- корректная сортировка.

3.2 Структура ПО

В данном разделе будет рассмотрена структура ПО 3.1.

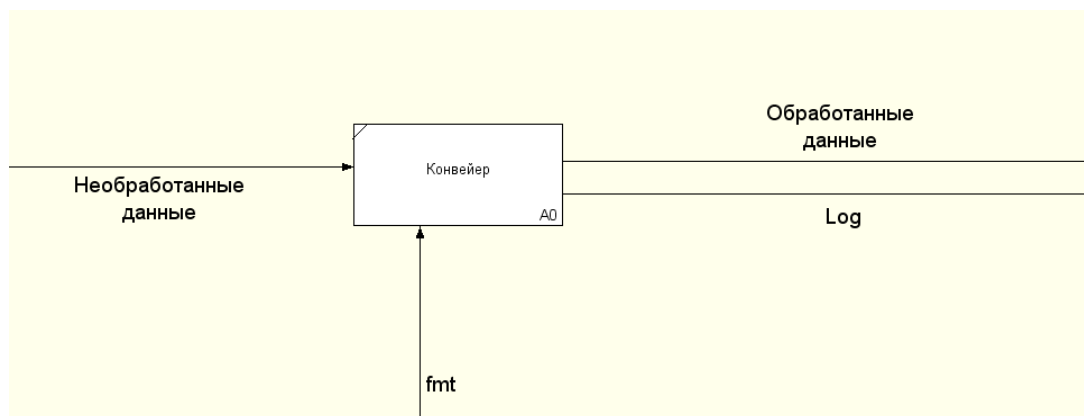


Рисунок 3.1 – Структура ПО

3.3 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка. Так же данный язык предоставляет средства тестирования разработанного ПО.

Время работы алгоритмов было замерено с помощью функции `Now()` из библиотеки `Time` [3].

3.4 Листинг кода

В листингах 3.1 – 3.2 приведены реализации параллельного и линейного конвейеров.

Листинг 3.1 – Реализация параллельного конвейера

```
1 func first(cake *Cake, qBake *Queue) {
2     cake.prepare = true
3
4     cake.sPrepare = time.Now()
5     time.Sleep(time.Duration(200) * time.Millisecond)
6     cake.fPrepare = time.Now()
7
8     qBake.push(cake)
9 }
10
11 func second(cake *Cake, qFinalize *Queue) {
12     cake.bake = true
13
14     cake.sBake = time.Now()
15     time.Sleep(time.Duration(200) * time.Millisecond)
16     cake.fBake = time.Now()
17
18     qFinalize.push(cake)
19 }
20
21 func third(cake *Cake, finished *Queue) {
22     cake.finalize = true
23
24     cake.sFinalize = time.Now()
25     time.Sleep(time.Duration(200) * time.Millisecond)
26     cake.fFinalize = time.Now()
27
28     finished.push(cake)
29 }
30
31 func Parallel(amount int, wait chan int) *Queue {
32     f := make(chan *Cake, 5)
33     s := make(chan *Cake, 5)
34     t := make(chan *Cake, 5)
35     line := createQueue(amount)
36     first := func() {
37         for {
38             select {
39                 case cake := <-f:
40                     cake.prepare = true
```



```

41
42         cake.sPrepare = time.Now()
43         time.Sleep(time.Duration(200) * time.Millisecond)
44         cake.fPrepare = time.Now()
45
46         s <- cake
47     }
48 }
49 }
50
51 second := func() {
52     for {
53         select {
54             case cake := <-s:
55                 cake.bake = true
56
57                 cake.sBake = time.Now()
58                 time.Sleep(time.Duration(200) * time.Millisecond)
59                 cake.fBake = time.Now()
60
61                 t <- cake
62             }
63         }
64     }
65
66 third := func() {
67     for {
68         select {
69             case cake := <-t:
70                 cake.finalize = true
71
72                 cake.sFinalize = time.Now()
73                 time.Sleep(time.Duration(200) * time.Millisecond)
74                 cake.fFinalize = time.Now()
75
76                 line.push(cake)
77                 if cake.num == amount {
78                     wait <- 0
79                 }
80             }
81         }
82     }
83 }
84
85 go first()
86 go second()
87 go third()
88 for i := 0; i <= amount; i++ {

```

```

89     cake := new(Cake)
90     cake.num = i
91     f <- cake
92 }
93
94 return line
95 }

```

Листинг 3.2 – Реализация реализации линейного конвейера

```

1 func Linear(amount int) *Queue {
2     qBake := createQueue(amount)
3     qFinalize := createQueue(amount)
4     finished := createQueue(amount)
5     i := 0
6     for i != -1 {
7         cake := new(Cake)
8         cake.num = i
9         first(cake, qBake)
10        if qBake.l >= 0 {
11            second(qBake.pop(), qFinalize)
12        }
13        if qFinalize.l >= 0 {
14            third(qFinalize.pop(), finished)
15        }
16        if finished.q[len(finished.q)-1] != nil {
17            return finished
18        }
19        i++
20    }
21    return finished
22 }

```

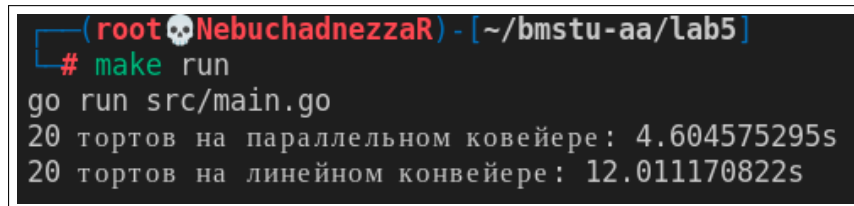
Вывод

В данном разделе была рассмотрена структура ПО и листинги кода программы.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.



```
(root@NebuchadnezzaR) - [~/bmstu-aa/lab5]
# make run
go run src/main.go
20 тортов на параллельном конвейере: 4.604575295s
20 тортов на линейном конвейере: 12.011170822s
```

Рисунок 4.1 – Демонстрация работы конвейеров

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Kali [4] Linux [5] 5.9.0-kali1-amd64.
- Память: 8 GB.
- Процессор: Intel® Core™ i5-8250U [6] CPU @ 1.60GHz
- Количество логических потоков: 8

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [7], предоставляемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N , которая динамически

варьируется в зависимости от того, был ли получен стабильный результат или нет.

В листинге 4.1 пример реализации бенчмарка.

Листинг 4.1 – Реализация бенчмарка

```
1 func BenchmarkParallel10(b *testing.B) {  
2     for i := 0; i < b.N; i++ {  
3         wait := make(chan int)  
4         Parallel(10, wait)  
5         <-wait  
6     }  
7 }
```

На рисунках 4.2 приведён график сравнения производительности конвейеров.

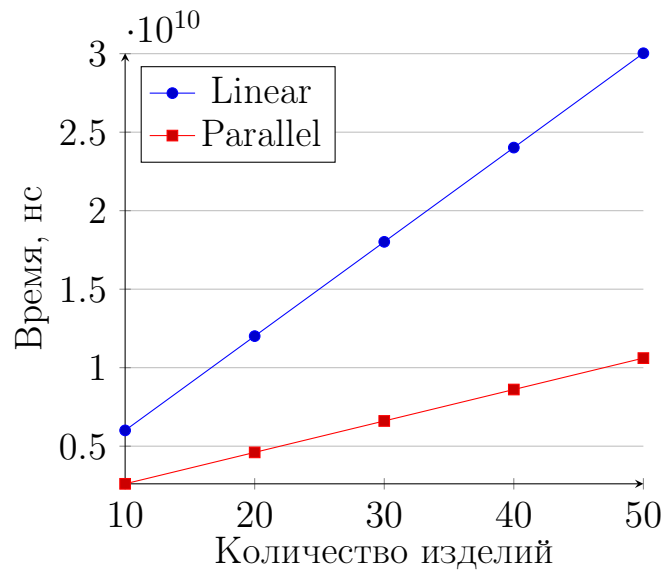
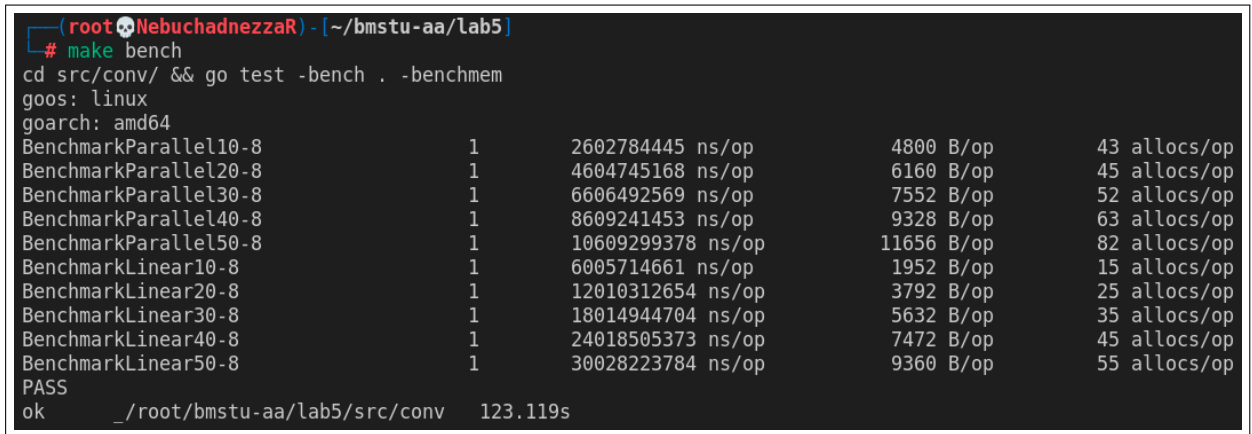


Рисунок 4.2 – Сравнение конвейеров.

4.4 Производительность алгоритмов

Производительность и объем выделенной памяти при работе алгоритмов указаны на рисунке 4.3.



```
(root@NebuchadnezzaR) - [~/bmstu-aa/lab5]
# make bench
cd src/conv/ && go test -bench . -benchmem
goos: linux
goarch: amd64
BenchmarkParallel10-8      1      2602784445 ns/op      4800 B/op      43 allocs/op
BenchmarkParallel20-8      1      4604745168 ns/op      6160 B/op      45 allocs/op
BenchmarkParallel30-8      1      6606492569 ns/op      7552 B/op      52 allocs/op
BenchmarkParallel40-8      1      8609241453 ns/op      9328 B/op      63 allocs/op
BenchmarkParallel50-8      1     10609299378 ns/op     11656 B/op      82 allocs/op
BenchmarkLinear10-8        1      6005714661 ns/op      1952 B/op      15 allocs/op
BenchmarkLinear20-8        1     12010312654 ns/op      3792 B/op      25 allocs/op
BenchmarkLinear30-8        1     18014944704 ns/op      5632 B/op      35 allocs/op
BenchmarkLinear40-8        1     24018505373 ns/op      7472 B/op      45 allocs/op
BenchmarkLinear50-8        1     30028223784 ns/op      9360 B/op      55 allocs/op
PASS
ok      _/root/bmstu-aa/lab5/src/conv  123.119s
```

Рисунок 4.3 – Замеры производительности алгоритмов, выполненные при помощи команды `go test -bench . -benchmem`

Вывод

Эксперимент показывает, что использование нескольких потоков для реализации конвейерной обработки данных ускоряет алгоритм в несколько раз. При этом возникает ситуация при которой ленты не простаивают. Тратится лишь малое время для передачи данных на линию.

Заключение

В ходе лабораторной работы были изучены возможности параллельных вычислений, реализован алгоритм конвейерной обработки данных с помощью параллельных вычислений.

Было проведено сравнение синхронной версии того же алгоритма и асинхронной. Выяснилось, что при использовании потоков, время работы алгоритма не просто сокращается, но и снижается скорость роста времени при увеличении числа изготавливаемых экземпляров.

Литература

- [1] В. П. Меднов Е. П. Бондарев. Транспортные, распределительные и рабочие конвейеры. – М, 1970.
- [2] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 09.09.2020).
- [3] testing – Package time [Электронный ресурс]. Режим доступа: <https://golang.org/pkg/time/> (дата обращения: 18.11.2020).
- [4] Our Most Advanced Penetration Testing Distribution, Ever. [Электронный ресурс]. Режим доступа: <https://kali.org/> (дата обращения: 12.09.2020).
- [5] Linux – Википедия [Электронный ресурс]. Режим доступа: <https://ru.wikipedia.org/wiki/Linux> (дата обращения: 12.09.2020).
- [6] Intel Processors [Электронный ресурс]. Режим доступа: <https://www.intel.com/content/www/us/en/products/processors/core/i5-processors.html> (дата обращения: 12.09.2020).
- [7] testing – The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/pkg/testing/> (дата обращения: 12.09.2020).