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»

Отчёт по лабораторной работе №5
по курсу "Анализ алгоритмов"

Тема Конвейерная обработка

Студент Ивахненко Д.А.

Группа ИУ7-56Б

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

В данной лабораторной работе рассматривается реализация конвейерной обработки данных.

Конвейер[1] - способ организации вычислений, используемый в со временных процессорах и контроллерах с целью повышения их произ водительности (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 вы полняться 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. [2]

Целью данной работы является реализация и изучение конвейерной обработки. Для достижения поставленной цели необходимо выполнить следующие задачи:

- исследовать основы конвейерных вычислений;
- исследовать основные методы организации конвейерных вычислений;
- сравнить существующие методы организации конвейерных вычислений;
- привести схемы рассматриваемых алгоритмов, а именно:
  - схему конвейера, содержащего 3 ленты;
  - схему конвейера, содержащего 3 ленты и реализующего fan-in—fan-out подходы;
- описать использующиеся структуры данных;
- описать структуру разрабатываемого програмного обеспечения;
- определить средства программной реализации;

- определить требования к программному обеспечению;
- привести сведения о модулях программы;
- провести тестирование реализованного программного обеспечения;
- провести экспериментальные замеры временных характеристик реализованного конвейера.

1 Аналитическая часть

В данном разделе описаны основные идеи алгоритма конвейерных вычислений. Поставлена задача, для которой будет применяться этот алгоритм в качестве примера[3].

1.1 Описание алгоритма параллельной 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).

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

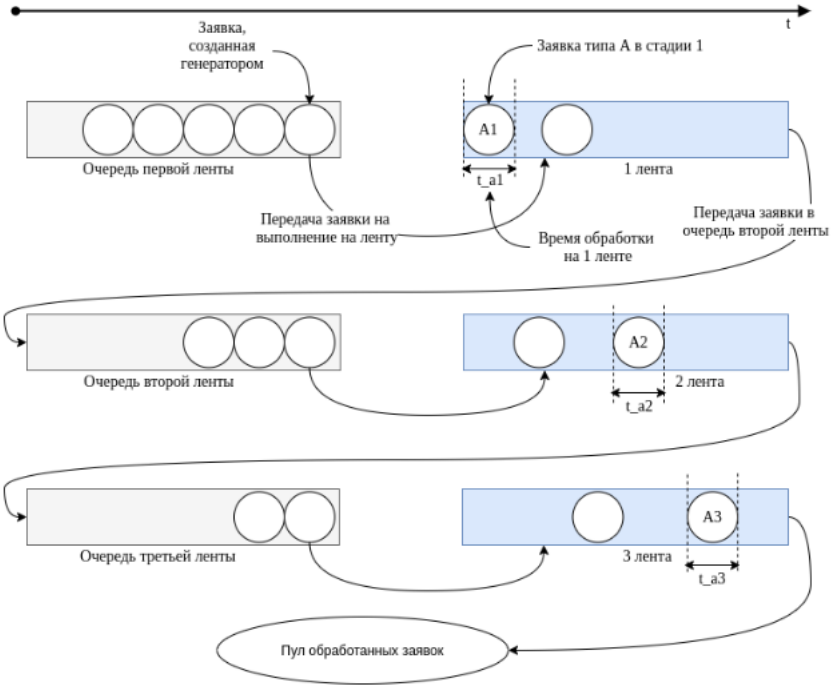

Рисунок 1.1 – Схема, иллюстрирующая работу конвейера с параллельной реализацией

Рассмотрим, как заявка перемещается между разными лентами (1.2)

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 конкретно до тех пор, пока не обра-

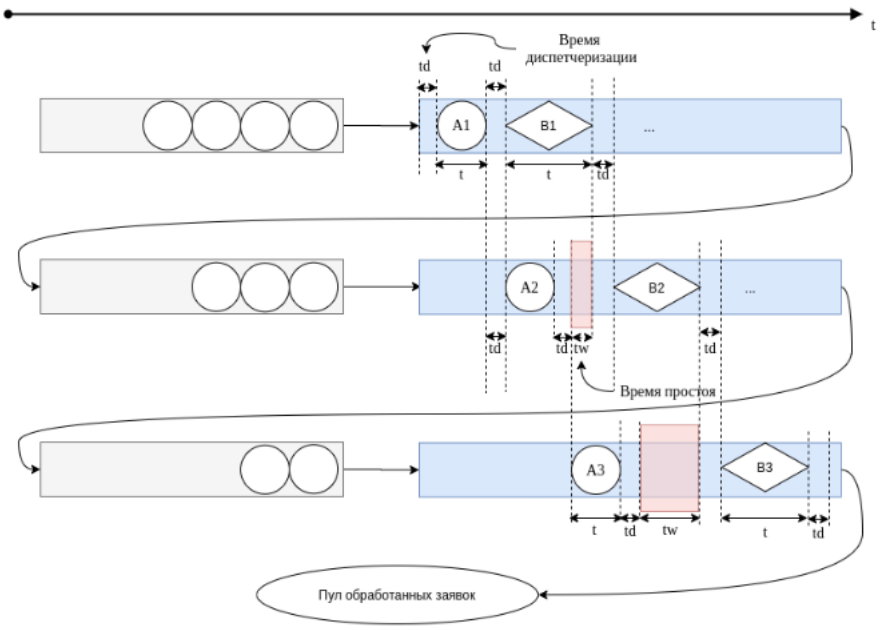

Рисунок 1.2 – Схема, иллюстрирующая пример работы конвейра с параллельной реализацией

ботаются первые две. Однако, если бы она была первой в очереди на обработку первой лентой, то время простоя бы сократилось.

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.

Генератор заявок может быть статический или динамический. В случае статического очередь формируется заранее, в случае динамического – очередь может формироваться в течение некого периода в зависимости от условий.

1.2 Постановка и описание задачи для конвейерной обработки

В качестве алгоритма, реализованного для распределения на конвейере, было выбрана реализация слегка видоизмененная версия команды Unix-подобной операционной системы `md5sum`.

`md5sum` позволяет вычислять значения хеш-сумм файлов по алгоритму MD5. В обычном случае вычисленные хеши выводятся. В других случаях программа сверяет вычисленные значения со значениями, сохранёнными в файле. Наиболее часто программа используется для проверки правильной загрузки файлов по сети.

Наша примерная программа похожа на `md5sum`, однако она принимает в качестве аргу-

мента один каталог и печатает значения хеш-суммы для каждого файла в этом каталоге.

Данный алгоритм состоит из 3-х этапов:

- получение полного пути файла;
- вычисление хеш-суммы при помощи алгоритма md5[?] содержимого файла;
- агрегирование полученных результатов и сохранение в результирующую структуру.

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.

Входными данными для программного обеспечения являются:

- путь к файлу или директории в файловой системе;
- количество потоков для вычисления хеш-суммы.

Выходными данными являются:

- список обработанных файлов и соответствующие им хеш–суммы;
- усредненные временные характеристики конвейера: среднее время обработки заявки каждой из лент, а так же - усредненное время ожидания в каждой из очередей конвейера.

На программное обеспечения накладываются следующие ограничения:

- корректность входных данных;
- обладания соответствующими правами для чтения исходных файлов.

2 Конструкторская часть

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

2.1 Схемы работы алгоритма конвейерной обработки данных

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

Рисунок 2.1 – Схема, иллюстрирующая пример работы конвейра с параллельной реализацией

Заметим, что вычисление хеш-суммы файла не зависит от ранее вычисленных значений. В связи с этим, можно реализовать многопоточное вычисление хеш-сумм файлов, посредством так называемых fan-out и fan-in подходов.

Таким образом, получим схему работы конвейера, отраженную на рисунке 2.2.

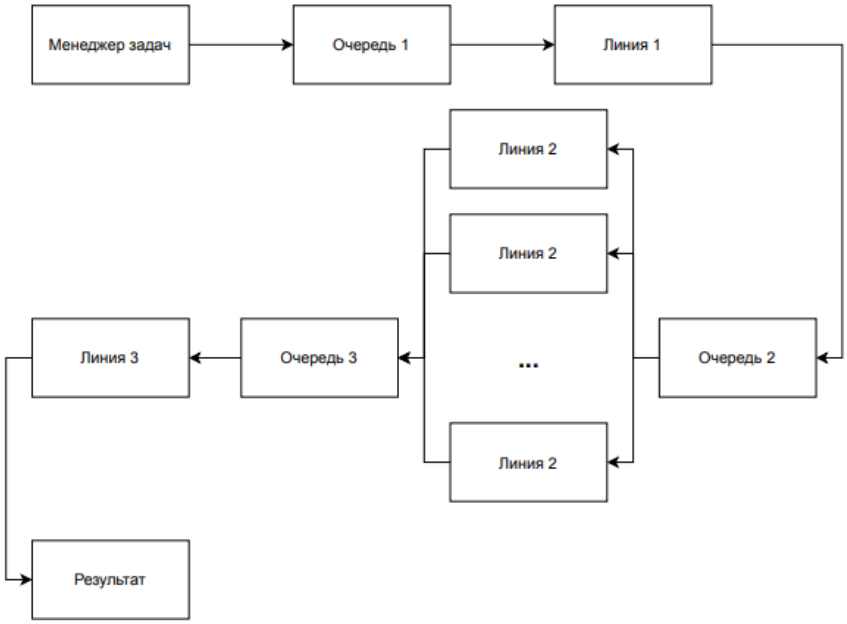

Рисунок 2.2 – Схема организации конвейерных вычислений fan-in/fan-out

Отдельно стоит отметить, что в алгоритме, рассматриваемом в рамках данной лабораторной работы, используется, по сути, статический менеджер задач, поскольку все задачи - это файлы в указанной директории.

2.2 Описание структуры программного обеспечения

Программа будет включать в себя один смысловой модуль, на зываемый `md5pipeline`, который содержит в себе процедуры и функции, связанные с реализацией параллельного и синхронного конвейера. Неза висимо от модуля будет существовать файл `main.go`, одержащий точку входа в программу, а так же `handling.go`, в котором будет реализован подсчет статистики.

Программа разбита на модули:

- `main.go` - файл, содержащий точку входа в программу;
- `handling.go` - файл, содержащий реализацию подсчета статистики;
- `md5pipeline/types.go` - определение пользовательских типов даных;
- `md5pipeline/pipeline.go` - определение структуры конвейера;
- `md5pipeline/filescanner.go` - реализация первой ленты конвейера;
- `md5pipeline/digester.go` - реализация второй ленты конвейера;
- `md5pipeline/md5.go` - реализация третьей ленты конвейера;

На Рисунке 2.3 представлена idef0–диаграмма работы описанного выше конвейера.

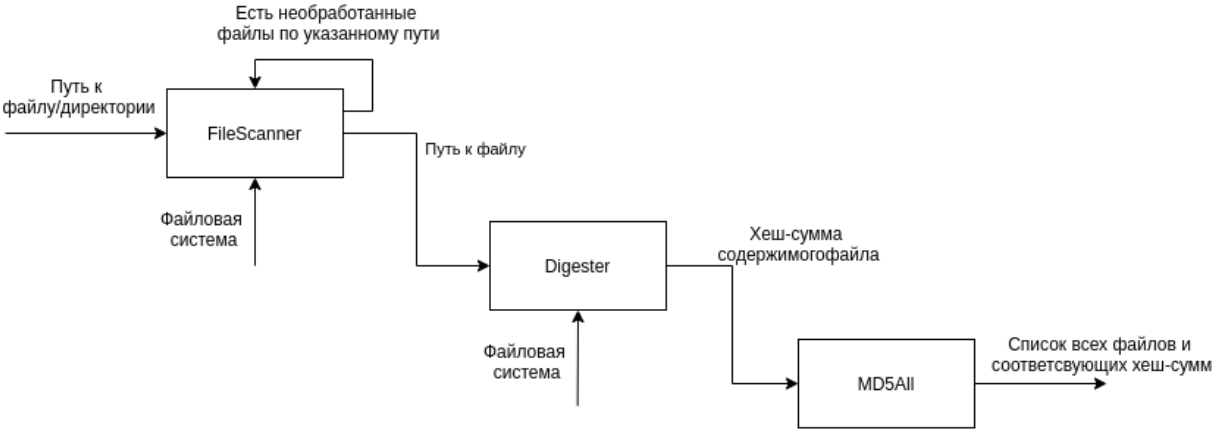

Рисунок 2.3 – Диаграмма работы конвейера

2.3 Описание структур данных

Для реализации конвейерных вычислений, введем некоторые типы данных:

- `FileScannerOutput` – структура, описывающая результат обработки задачи первой лентой конвейера;
- `DigesterOutput` – структура, описывающая результат обработки задачи второй лентой конвейера;
- `ResultingOutput` – структура, описывающая результат обработки задачи конвейером.

Рассмотрим каждый из введенных типов данных

Листинг 2.1 – Определение типов данных. FileScannerOutput

```
type fileScannerOutput struct {
    path string

    startTime          time.Time
    endTime            time.Time
```

На листинге 2.1:

- `path` - путь, к рассматриваемому файлу/директории;
- `startTime` - время начала обработки задачи первой лентой;
- `endTime` - время постановки задачи в очередь.

Листинг 2.2 – Определение типов данных. DigesterOutput

```
type digesterOutput struct {
    path string
    sum  [md5.Size]byte
    err  error

    startFileScannerTime time.Time
    endFileScannerTime   time.Time

    waitingTimeInQueue time.Duration
    startTime          time.Time
    queuingTime        time.Time
}
```

На листинге 2.2:

- `path` - путь, к рассматриваемому файлу/директории;

- `sum` - хеш-сумма содержимого файла;
- `err` - поле, содержащее описание ошибки обработки задачи, при наличии таковой;
- `startFileScannerTime` - время начала обработки задачи первой лентой;
- `endFileScannerTime` - время окончания обработки задачи первой лентой;
- `waitingTimeInQueue` - длительность простоя задачи перед второй лентой;
- `startTime` - время начала обработки задачи второй лентой;
- `queuingTime` - время постановки задачи в очередь.

Листинг 2.3 – Определение типов данных. ResultingOutput

```
type ResultingOutput struct {
    Path string
    Sum  [md5.Size]byte

    Start1                time.Time
    End1                  time.Time

    WaitingForDigester time.Duration
    Start2             time.Time
    End2               time.Time

    WaitingForAggregation time.Duration
    Start3                time.Time
    End3                  time.Time
}
```

На листинге 2.3:

- `Path` - путь, к рассматриваемому файлу/директории;
- `Sum` - хеш-сумма содержимого файла;
- `Start1` - время начала обработки задачи первой лентой;
- `End1` - время окончания обработки задачи первой лентой;
- `WaitWaitingForDigester` - длительность простоя задачи перед второй лентой;
- `Start2` - время начала обработки задачи второй лентой;
- `End2` - время окончания обработки задачи второй лентой;
- `WaitingForAggregation` - длительность простоя задачи перед третьей лентой;
- `Start3` - время начала обработки задачи третьей лентой;
- `End3` - время окончания обработки задачи третьей лентой;

2.4 Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема организации конвейерных вычислений на примере конвейере с тремя лентами (Рисунок 2.1). Так же, было приведено описание пользовательских типов данных, описанных в рамках реализации конвейерных вычислений (Листинги 2.1 – 2.3)

3 Технологическая часть

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

3.1 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран язык Golang[4]. Выбор этого языка обусловлен наличием требуемого функционала для организации параллельных вычислений. Также выбор обусловлен моим желанием получить больше практики в написании программ на этом языке.

3.2 Тестирование

В рамках данной лабораторной работы будет проведено тестирование реализованного программного обеспечения.

При разработке тестов были выделены следующие классы эквивалентности:

- входными данными является путь к файлу;
- входными данными является путь до директории;

В соответствии с данными классами эквивалентности были разработаны тесты, представленные в таблице 3.1.

Все тесты пройдены успешно.

Таблица 3.1 – Тестирование функций

Вход. данные	Ожидаемый результат	Реальный результат
`report.pdf`	`293ae5375e60da69337946e4e88fea5c`	`293ae5375e60da69337946e4e88fea5c`

.	`dfad7f3da4a378ed5d4cebbff9f0814e`
	`ad1b1770cb63d88541cbe86d23ce6e2b`
	`f197857f15b383af48cf83e607fe3173`
	`1e1a8e2e9e494885fdaf038c4505b27d`
	`4193feec424757f73a9a1fb8b1ed0d75`
	`3c60ddab06846a52e8cff94c6bd539e6`
	`f07805bc3115d7df108473490a02476d`
	`9faa535100c9cfa1fee02b5e25507868`
	`7f32a0988dc453353b1785ecf975206d`
		`dfad7f3da4a378ed5d4cebbff9f0814e`
		`ad1b1770cb63d88541cbe86d23ce6e2b`
		`f197857f15b383af48cf83e607fe3173`
		`1e1a8e2e9e494885fdaf038c4505b27d`
		`4193feec424757f73a9a1fb8b1ed0d75`
		`3c60ddab06846a52e8cff94c6bd539e6`
		`f07805bc3115d7df108473490a02476d`
		`9faa535100c9cfa1fee02b5e25507868`
		`7f32a0988dc453353b1785ecf975206d`

3.3 Реализация конвеера

На листингах 3.1 – 3.6 представлен исходный код программы.

Листинг 3.1 – Точка входа в программу

```
func main() {
    parser := argparse.NewParser("md5pipeline", "Hashing files in directory")

    verbose := parser.Flag(
        "v", "verbose", &argparse.Options{Help: "Verbose mode"},
    )

    measures := parser.Flag(
        "m", "measures", &argparse.Options{Help: "Show only measures"},
    )

    serialMode := parser.Flag(
        "s", "serial", &argparse.Options{Help: "Serial Mode"},
    )

    path := parser.String(
        "p", "path", &argparse.Options{
            Required: true,
            Help: "Path to file (dir)",
        },
    )

    numberOfWorkers := parser.Int(
        "n", "numberOfWorkers", &argparse.Options{
            Required: false,
            Help: "Number of digester workers",
        },
    )

    err := parser.Parse(os.Args)
    if err != nil {
        log.Fatalln(parser.Usage(err))
        return
    }


    if *numberOfWorkers == 0 {
        *numberOfWorkers = 16
    }

    pipeline := md5pipeline.NewPipeline(*numberOfWorkers)

    startT := time.Now()
```

```
44
45      var data []md5pipeline.ResultingOutput
46
47      if *serialMode {
48          data, err = pipeline.MD5AllSerial(*path)
49      } else {
50          data, err = pipeline.MD5All(*path)
51      }
52
53      if err != nil {
54          fmt.Println(err)
55          return
56      }
57
58      handle(data, startT, time.Since(startT).Nanoseconds(), *verbose, *measures)
59  }
```

Листинг 3.2 – Определение пользовательских типов данных

```
1  type fileScannerOutput struct {
2      path string
3
4      startTime time.Time
5      endTime   time.Time
6  }
7
8  type digesterOutput struct {
9      path string
10     sum  [md5.Size]byte
11     err  error
12
13     startFileScannerTime time.Time
14     endFileScannerTime   time.Time
15
16     waitingTimeInQueue time.Duration
17     startTime          time.Time
18     endTime            time.Time
19 }
20
21 type ResultingOutput struct {
22     Path string
23     Sum  [md5.Size]byte
24
25     Start1 time.Time
26     End1   time.Time
27
28     WaitingForDigester time.Duration
29     Start2             time.Time
30     End2               time.Time
```

```
31
32      WaitingForAggregation time.Duration
33      Start3                time.Time
34      End3                  time.Time
35  }
```

Листинг 3.3 – Определение структуры конвеера

```
1  type MD5Pipeline struct {
2      numOfWorkers int
3  }
4
5  func NewPipeline(numOfWorkers int) MD5Pipeline {
6      return MD5Pipeline{numOfWorkers: numOfWorkers}
7  }
```

Листинг 3.4 – Лента обхода файловой системы

```
1  func scanFiles(done <-chan struct{}, root string) (<-chan fileScannerOutput, <-
       chan error) {
2      fileScanOutputChan := make(chan fileScannerOutput, queueSize)
3      scanErrors := make(chan error, 1)
4
5      go func() {
6          defer close(fileScanOutputChan)
7          startTime := time.Now()
8          scanErrors <- filepath.Walk(root, func(path string, info os.FileInfo,
               err error) error {
9              if err != nil {
10                 return err
11             }
12
13             if !info.Mode().IsRegular() {
14                 return nil
15             }
16
17             select {
18             case fileScanOutputChan <- fileScannerOutput{
19                 path,
20                 startTime,
21                 time.Now(),
22             }:
23
24                 startTime = time.Now()
25             case <-done:
26                 return errors.New("scanning was canceled")
27             }
28
29             return nil
30         })
```

```
31      }()
32
33      return fileScanOutputChan, scanErrors
34  }
```

Листинг 3.5 – Лента вычисления хеш-суммы

```
1  func digester(done <-chan struct{}, fsOutputChan <-chan fileScannerOutput,
       digesterOutputChan chan<- digesterOutput) {
2      for fileInfo := range fsOutputChan {
3          startT := time.Now()
4          fileData, err := ioutil.ReadFile(fileInfo.path)
5
6          checksum := md5.Sum(fileData)
7
8          select {
9          case digesterOutputChan <- digesterOutput{
10             fileInfo.path,
11             checksum,
12             err,
13             fileInfo.startTime,
14             fileInfo.endTime,
15             startT.Sub(fileInfo.endTime),
16             startT,
17             time.Now(),
18         }:
19         case <-done:
20             return
21         }
22     }
23 }
```

Листинг 3.6 – Запуск конвейера

```
1  func (p *MD5Pipeline) MD5All(root string) ([]ResultingOutput, error) {
2      done := make(chan struct{})
3      defer close(done)
4
5      scanOutputChan, scanErrors := scanFiles(done, root)
6
7      digesterOutputChan := make(chan digesterOutput, queueSize)
8
9      var wg sync.WaitGroup
10     wg.Add(p.numOfWorkers)
11
12     for i := 0; i < p.numOfWorkers; i++ {
13         go func() {
14             digester(done, scanOutputChan, digesterOutputChan)
15             wg.Done()
16         }()
```

```
    }

    go func() {
        wg.Wait()
        close(digesterOutputChan)
    }()

    result := make([]ResultingOutput, 0)

    for digesterRes := range digesterOutputChan {
        startTime := time.Now()

        if digesterRes.err != nil {
            return nil, digesterRes.err
        }

        result = append(result, ResultingOutput{
            digesterRes.path,
            digesterRes.sum,
            digesterRes.startFileScannerTime,
            digesterRes.endFileScannerTime,
            digesterRes.waitingTimeInQueue,
            digesterRes.startTime,
            digesterRes.endTime,
            startTime.Sub(digesterRes.endTime),
            startTime,
            time.Now(),
        })
    }

    if err := <-scanErrors; err != nil {
        return nil, err
    }

    return result, nil
}
```

Листинг 3.7 – Последовательная версия программы

```
func (p *MD5Pipeline) MD5AllSerial(root string) ([]ResultingOutput, error) {
    paths := make([]fileScannerOutput, 0)
    hashes := make([]digesterOutput, 0)
    result := make([]ResultingOutput, 0)
    scanErrors := make(chan error, 1)

    startT := time.Now()
    scanErrors <- filepath.Walk(root, func(path string, info os.FileInfo, err
        error) error {
        if err != nil {
```

```
            return err
        }

        if !info.Mode().IsRegular() {
            return nil
        }

        paths = append(paths, fileScannerOutput{
            path,
            startT,
            time.Now(),
        })

        startT = time.Now()
        return nil
    })

    for _, fileInfo := range paths {
        startT = time.Now()
        fileData, err := ioutil.ReadFile(fileInfo.path)

        checksum := md5.Sum(fileData)

        hashes = append(hashes, digesterOutput{
            fileInfo.path,
            checksum,
            err,
            fileInfo.startTime,
            fileInfo.endTime,
            startT.Sub(fileInfo.endTime),
            startT,
            time.Now(),
        })
    }


    for _, digesterRes := range hashes {
        startT = time.Now()

        if digesterRes.err != nil {
            return nil, digesterRes.err
        }

        result = append(result, ResultingOutput{
            digesterRes.path,
            digesterRes.sum,
            digesterRes.startFileScannerTime,
            digesterRes.endFileScannerTime,
```

```
58                digesterRes.waitingTimeInQueue,
59                digesterRes.startTime,
60                digesterRes.endTime,
61                startT.Sub(digesterRes.endTime),
62                startT,
63                time.Now(),
64            })
65        }
66
67        if err := <-scanErrors; err != nil {
68            return nil, err
69        }
70
71        return result, nil
72    }
```

3.4 Вывод

В данном разделе была представлена реализация ПО на языке golang для ре шения поставленной задачи, а также проведено тестирование в соответствии с выделенными классами эквивалентности.

4 Исследовательская часть

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

4.1 Постановка эксперимента

Технические характеристики устройства, на котором выполнялся эксперимент:

- операционная система: Ubuntu[5] Linux x86_64;
- память: 16 GiB;
- процессор: AMD Ryzen™ 7 4700U[6].

Проводятся эксперименты следующего типа:

1. Сравнение времени обработки заявок на конвейере с разным кол-вом сопрограмм и использование однопоточной последовательной обработки.
2. Подсчет максимального, минимального и среднего времени:
   (a) проведенного задачей в системе;
   (b) проведенного задачей в очередях.
3. Анализ трассировки программы.

Так как в реализуемом конвейере реализуются fan-out и fan-in подходы, особый интерес представляет зависимость временных характеристик системы от количества параллельных сопрограмм. А также сравнение непосредственно однопоточной последовательной реализации и конвеера.

Для сравнения возьмем конвейеры с соответствующим числом сопрограмм из набора [1, 2, 4, ..., 64].

На момент тестирования в файловой системе машины, на которой проводились тесты, насчитывалось 69257 файлов различного типа и размера. В ходе проведения эксперимента данная величина оставалась неизменной.

Так как в общем случае вычисление хеш-суммы файла является достаточно короткой задачей, воспользуемся усреднением массового эксперимента. Для этого вычислим среднее арифметическое значение временных ресурсов, затраченных на выполнение алгоритма, для `n` запусков. Сравнение произведем при `n` = 100.

4.2 Результат эксперимента

В таблице 4.1 приведены результаты сравнения работы однопоточного последовательного алгоритма и конвейера с различным числом сопрограмм.

Видно, что последовательная реализация алгоритма уступает конвейерной даже с 1 сопрограммой почти на 30%, а при конвейере в 16 сопрограммами - уже почти в 4 раза.

Таблица 4.1 – Сравнение последовательного алгоритма и конвейера

Реализация	Время выполнения (с)
Последовательная, 1 поток	8.94
Конвейер (1 сопрограмма)	7.02
Конвейер (2 сопрограммы)	4.17
Конвейер (4 сопрограмма)	2.93
Конвейер (8 сопрограмм)	2.31
Конвейер (16 сопрограмм)	2.28
Конвейер (32 сопрограммы)	2.28
Конвейер (64 сопрограмма)	2.32

В таблицах 4.2 – 4.3 отображены результаты анализа среднего, минимального и максимального времени нахождения задачи в системе и в очереди.

Таблица 4.2 – Время выполнения алгоритмов (в мс) для отсортированного массива

Кол-во сопрограмм	Мин. (с.)	Макс. (с.)	Среднее (с.)
1	0.8e−3	2.72	1.38
2	1.1e−4	1.18	0.69
4	1.5e−5	0.54	0.36
8	1.2e−5	0.34	0.18
16	0.9e−5	0.16	0.06

Таблица 4.3 – Время выполнения алгоритмов (в мс) для отсортированного массива

Кол-во сопрограмм	Мин. (с.)	Макс. (с.)	Среднее (с.)
1	2.5e−3	6.77	3.45
2	3.1e−4	3.52	1.81
4	4.4e−5	1.91	0.94
8	3.4e−5	0.96	0.45
16	2.5e−5	0.34	0.12

Очевидным является тот факт, что при увеличении количества сопрограмм данное время уменьшается.

В зависимости от трудоемкости задач, выполняемых на том или ином этапе конвеера, время ожидания в очереди будет увеличиваться или уменьшаться. На примере решаемой задачи, заметим, что время ожидания занимает примерно треть всей работы системы.

Из данных, приведенных в Таблице 4.4 следует, что обработка задач действительно происходит асинхронно - во время работы одной ленты, другие выполняют свои функции. Кроме того, видна работа принципа fan-in/fan-out - четвертая и пятая задачи одновременно попали на две параллельные ленты номер 2.

Таблица 4.4 – Трассировка конвейера

Лента	Задача	Начало (µs)	Конец (µs)
1	2	164.54	171.25
2	2	193.18	233.20
3	2	273.29	275.73
1	3	174.50	183.26
2	3	278.66	328.46
3	3	358.56	361.35
1	4	186.61	191.78
2	4	238.44	264.625
3	4	293.75	296.54
1	5	194.09	221.88
2	5	238.44	264.62
3	5	282.92	285.79

4.3 Вывод

Исходя из результатов проведенного эксперимента можно сделать следующие выводы. Конвейерная обработка действительно может помочь снизить временные затраты на выполнение задачи, благодаря отстутствию ожидания конца выполнения подзадачи, как в случае последовательного алгоритма. В случае реализованного в рамках лабораторной работы алгоритма, как минимум на 30%, что не так уж много, поэтому прежде чем реализовывать конвейер, необходимо проанализировать необходимость его использования, а также рассмотреть возможность распараллеливания отдельных наиболее трудоемких этапов конвеера.

Использование подходов fan-in / fan-out, то есть использование разветвления – нескольких потоков выполняющих параллельно один из этапов конвейера, существенно снижает временные затраты, особенно в тех случаях, когда на данном этапе конвейера выполняется более трудозатратная работа, чем на других. Данный подход позволил снизить временные затраты алгоритма, рассматриваемого в рамках данной работы почти в 4 раза, по сравнению с последовательным алгоритмом и в 3 раза в сравнении с конвейером без распараллеливания.

Заключение

В ходе выполнения лабораторной работы были выполнены поставленные задачи, а именно:

- исследованы основы конвейерных вычислений;
- исследованы основные методы организации конвейерных вычислений;
- проведено сравнение существующих методов организации конвейерных вычислений;
- приведены схемы рассматриваемых алгоритмов, а именно:
  - схема конвейера, содержащего 3 ленты;
  - схема конвейера, содержащего 3 ленты и реализующего fan–in–fan–out подходы;
- описаны использующиеся структуры данных;
- описана структура разрабатываемого програмного обеспечения;
- определены средства программной реализации;
- определены требования к программному обеспечению;
- приведены сведения о модулях программы;
- проведено тестирование реализованного программного обеспечения;
- проведены экспериментальные замеры временных характеристик реализованного конвейера.

Прежде чем использовать алгоритмы конвейерных вычислений, необходимо произвести анализ поставленной задачи, особенно следует учитывать следующие факторы:

1. Минимальное, максимальное и среднее время простоя заявки в системе.
2. Минимальное, максимальное и среднее время решения каждого из этапов задачи.
3. Время переключения между лентами.
4. Время последовательной передачи управления.

Если затраты на переключение между лентами (диспетчеризация) нивелируются трудоемкостью задачи, то лучше отдать предпочтение конвейерной реализации. Однако, если задача имеет низкую трудоемкость, то, возмонжо, стоит использовать последовательную реализацию, которая будет проще в реализации, что позволит избежать потенциальных ошибок. Так в случае рассматриваемой в рамках данной работы задачи, конвейерная реализация алгоритма снизила временные затраты на 30%, и в некоторых случаях окажется

более выгодно использовать чуть менее эффективный, но более простой и надежный алгоритм.

Также важно найти слабые места в реализации алгоритма - выделить наиболее трудоемкие этапы конвейера, и в том случае, если есть возможность их распараллелить, то следует использовать подход fan-in / fan-out, что позволит снизить время ожидания задач в очереди перед данным этапом. Этот подход позволил более чем в 3 раза снизить временные затраты по сравнению с обычным конвейером.

Список литературы

[1] Дьячков Р.А. Волков А.Н. Гнутов В.К. // Конвейеры. Справочник. Машиностроение, Ленинградское отделение. 1984.

[2] Шпаковский Г.И. // Реализация параллельных вычислений: MPI, OpenMP, кластеры, грид, многоядерные процессоры, графические процессоры, квантовые компьютеры. 2011.

[3] Go Concurrency Patterns: Pipelines and cancellation [Электронный ресурс]. Режим доступа: `https://go.dev/blog/pipelines` (дата обращения: 28.11.2021).

[4] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://go.dev/` (дата обращения: 28.11.2021).

[5] Ubuntu 20.04.3 LTS [Электронный ресурс]. Режим доступа: `https://ubuntu.com/download/desktop` (дата обращения: 28.09.2020).

[6] Проессор AMD Ryzen™ 7 4700U [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-4700u` (дата обращения: 28.09.2021).