МГТУ им. Баумана

Лабораторная работа №5

По курсу: "Анализ алгоритмов"

Конвейерная обработка данных

Работу выполнил: Рязанов М. С., ИУ7-52Б

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

Москва, 2019

Оглавление

Введение

При выполнении одинаковых операций над большим количеством одинаковых объектов для сокращения времени работы программы, может использоваться конвейерный способ организации вычислений. Конвейры находят свое применение в неименованных программных каналах(pipe) в ОС Unix/Linux[3], в современных процессорах для обработки команд[4]При выполнении одинаковых операций над большим количеством одинаковых объектов для сокращения времени работы программы, может использоваться конвейерный способ организации вычислений. Конвейры находят свое применение в неименованных программных каналах(pipe) в ОС Unix/Linux[3], в современных процессорах для обработки команд[4].

.

Целью данной лабораторной работы является изучение конвейрной организации вычислений. Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать основные принципы построения конвейера;
- разработать алгоритмы решения задачи;
- выбрать технологии для последующей реализации и иследования алгоритмов;
- реализовать разработанные алгоритмы;
- произвести тестирование корректности работы реализаций;
- сравнить быстродействие конвейерной и неконвейрной организации вычислений;

1 | Аналитическая часть

Идея заключается в параллельной обработке нескольких объектов, подаваемых на вход конвейеру. Сложные операции представляются в виде последовательности более простых стадий. Вместо обработки объектов последовательно (ожидания завершения конца обработки одного и перехода к следующему), следующий объект может начать обрабатываться через несколько стадий выполнения первого объекта.

Ожидаемое время выполнения неконвейерной функции

$$T = N * T_o \tag{1.1}$$

где T - общее время обработки, N - число объектов, T_o - время обработки одного объекта

Время выполнения конвейерной функции ограничено временем выполнения самой долгой стадии, но не меньше чем сумарное время всех стадий для одного объекта.

$$T \sim N * max(T_1, ..., T_n) \tag{1.2}$$

где T_i - время работы i-й стадии

Для конвейера из 2 стадий с временами работы: T_1, T_2 ожидаемое время работы всего конвейера:

$$T = T_1 + (N - 1) * max(T_1, T_2) + T_2 \tag{1.3}$$

Выводы:

- Определены главные идеи конвейерной организации вычислений
- Расчетные соотношения времени выполнения конвейерных и неконвейерных вычислений приведены в формулах 1.2 и 1.1.

2 | Конструкторская часть

2.1 Разработка алгоритмов

Из идей изложенных в аналитическом разделе следует, основной алгоритм запуска конвейера должна осуществлять следующие действия:

- Создание каналов передачи данных между стадиями
- Запуск стадий конвейера в отдельных потоках
- Контроль завершения выполнения всех стадий

Выводы:

- Разработан алгоритм запуска конвейерных вычислений 2.1

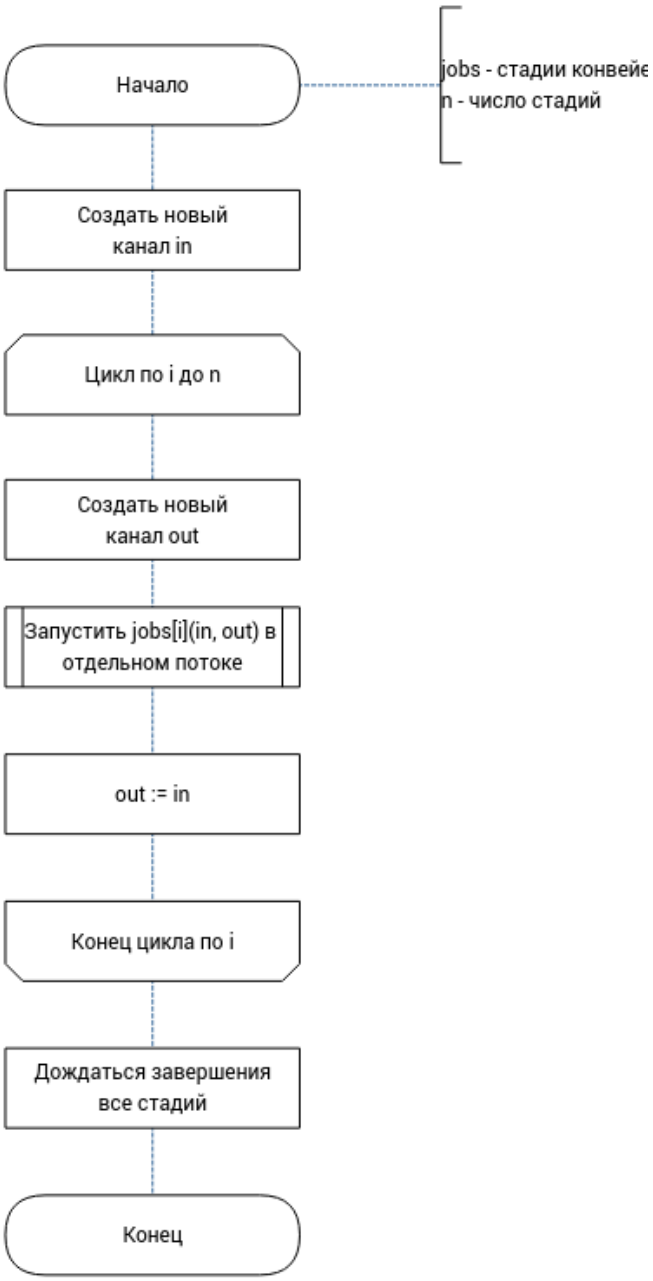

Рис. 2.1: Схема алгоритма запуска конвейера

3 | Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран go[2] т.к он предоставляет богатый инструментарий для написания многопоточных приложений, в том числе каналы[1] для передачи информации между потоками, что обеспечит наибольшую скорость разработки.

3.2 Сведения о модулях программы

Программа состоит из:

- ExecutePipeline - функция запускающая и осуществляющая контроль над исполнителями, запущенными в отдельных потоках;
- CreateWritingJob - функция, создающая исполнитель, читающий объекты из массива и записывающий их в выходной канал;
- CreateSleeperJob - функция, создающая исполнитель, передающий объекты из входного канала в выходной с заданной задержкой;
- CreateReadingJob - функция, создающая испольнитель, читающий объекты из входного потока и записывающий их в массив.

Листинг 3.1: Функция запуска конвейера

```
1 func ExecutePipeline(chanSize int, args ...job) {
2   wg := &sync.WaitGroup{}
3   // Создаем входной канал размера chanSize
```

```
4    in := make(chan transferObject, chanSize)
5    for _, val := range args {
6      // Создаем входной канал размера chanSize
7      out := make(chan transferObject, chanSize)
8      // Инкрементируем число запущенных исполнителей
9      wg.Add(1)
10     go func(in, out chan transferObject, f job) {
11       defer wg.Done()  // Декрементируем число запущенных
     исполнителей
12       defer close(out) // Закрываем выходной канал
13       f(in, out)       // Запуск исполнителя
14     }(in, out, val)
15     // Заменяем входной канал следующего выходным текущего
16     in = out
17   }
18   // Ждем завершения всех исполнителей
19   wg.Wait()
20 }
```

Листинг 3.2: Функция создания исполнителя - писателя

```
1 func CreateWritingJob(objs []transferObject) job {
2   return func(in, out chan transferObject) {
3     for _, val := range objs {
4       log.Printf("Writer: sending object with id = %d\n",
     val.id)
5       out <- val
6     }
7   }
8 }
```

Листинг 3.3: Функция создания исполнителя - "стадии конвейера"

```
1 func CreateSleeperJob(seconds, delta, id int) job {
2   return func(in, out chan transferObject) {
3     for val := range in {
4       log.Printf("Worker %d: got object with id = %d\n",
5         id, val.id)
6       time.Sleep(time.Duration(seconds)*time.Second +
```

```
7           time.Duration(-delta/2+rand.Intn(delta))*
8             time.Millisecond)
9         log.Printf("Worker %d: sending object with id = %d\n"
      ,
10          id, val.id)
11        out <- val
12      }
13    }
14  }
15  }
```

записывающего результы

Листинг 3.4: Функция создания исполнителя

```
1  func CreateReadingJob(objs []transferObject) job {
2    return func(in, out chan transferObject) {
3      count := 0
4      for val := range in {
5        log.Printf("Reader: got object with id = %d\n",
6          val.id)
7        objs[count] = val
8        count++
9      }
10   }
11 }
```

3.3 Тестовые случаи

Тестирование было организовано с помощью пакета **testing**.

Для проверки корректности работы функции ExecutePipeline был разработан следующий тест. Суть которого заключается в проверке свободного прохождения объектов по конвейеру

```
1  func TestPipeline(t *testing.T) {
2
3    var ok = true
4    var recieved uint32
5    freeFlowJobs := []job{
```

```
6       job(func(in, out chan interface{}) {
7         out <- 1
8         time.Sleep(10 * time.Millisecond)
9         currRecieved := atomic.LoadUint32(&recieved)
10        if currRecieved == 0 {
11          ok = false
12        }
13      }),
14      job(func(in, out chan interface{}) {
15        for _ = range in {
16          atomic.AddUint32(&recieved, 1)
17        }
18      }),
19    }
20    ExecutePipeline(freeFlowJobs...)
21    if !ok || recieved == 0 {
22      t.Errorf("no value free flow - dont collect them")
23    }
24  }
```

Выводы:

- В качестве языка программирования выбран Golang, так как предоставляет удобный инструментарий, необходимый в данной задаче
- Алгоритмы реализованы в виде подпрограмм, код которых приведен в листингах : 3.1, 3.2, 3.3, 3.4
- Разработы тестовые случаи и проведено тестирование

4 | Исследовательская часть

Был проведен замер времени работы при конвейерной и неконвейерной организации вычислений.

Проведем замеры времени для конвейера из 3 стадий с временем выполнения 1 секунда.

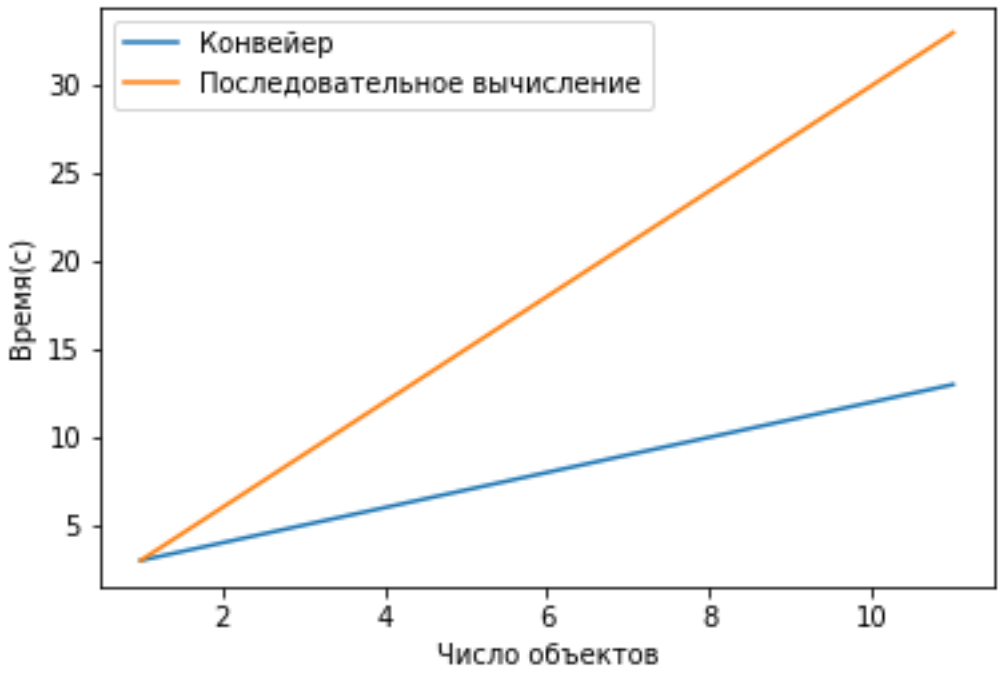

Рис. 4.1: Время выполнения на одинаковых стадиях

Как видно из графика 4.1 конвейерная и неконвейерная реализации

имеют линейную зависимость от числа обрабатываемых объектов, но коэфициент конвейрной реализации меньше.

Убедимся, что угол наклона прямой времени исполнения конвейерной реализации зависит от времени выполнения максимальной из стадий, заменим время выполнения одной из стадий на 3 секунды и повторим замеры.

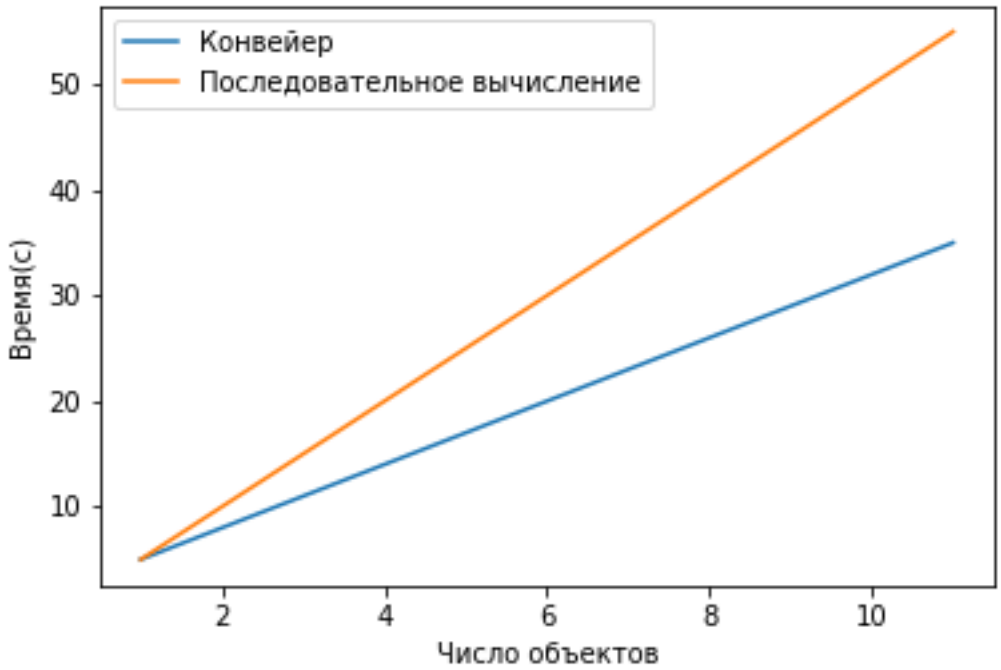

Рис. 4.2: Время выполнения на разных стадиях

Из графика видно 4.2, что угол наклона прямой времени выполнения конвейера относительно прямо времени выполнения неконвейерно реализации уменьшился, это подтверждает, то что время выполнения конвейера ограничивается временем выполнения его наиболее медленной стадии.

Выводы:

- Конвейерная организация дает выигрыш в скорости работы
- Выигрыш от конвейерной организации вычислений ограничивается самой медленной из стадий

Заключение

Был изучен принцип конвейерной организации вычислений, разработан алгоритм запуска конвейера состоящего из произвольного числа стадий. Алгоритм был реализован на языке Golang.

Экспериментально было подтверждено различие во временной эффективности конвейерной и неконвейерной организации вычислений, на основе анализа времени работы 2 реализаций на различном числе обрабатываемых объектов, с стадиями конвейера разной эффективности.

В результате исследований пришел к выводу, что конвейерная реализация дает выигрыш по времени, однако выигрыш ограничивается самой медленной из стадий конвейера.

Список литературы

[1] *Gobyexample*. URL: `https://gobyexample.com/channels`. (accessed: 04.12.2019).

[2] *Golang documentation*. URL: `https://godoc.org/`. (accessed: 04.12.2019).

[3] Ю. Вахалия. *Unix изнутри*. Классика Computer Scince. Питер, 2003.

[4] А. Ю. Попов. *Организация суперскалярных процессоров*. Организация ЭВМ. МГТУ им Н. Э. Баумана, 2011.