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

Название: Многопоточная реализация конвейера

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б (Группа) _____ (Подпись, дата) М.А. Козлов (И.О. Фамилия)

Преподователь _____ (Подпись, дата) Л.Л. Волкова (И.О. Фамилия)

Москва, 2020

Содержание

Введение

Система конвейерной обработки – это система, основаная на разделении подлежащей выполнению задачи на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры или потока. При этом конвейеризацию можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Но не каждую задачу можно разделить на несколько ступеней, организовав передачу данных от одного этапа к следующему.

Целью данной лабораторной работы является реализация системы конвейерной обработки.

Задачи данной лабораторной работы:

1) описать алгоритмы конвейерной обработки;
2) реализовать алгоритмы конвейерной обработки;
3) провести замеры процессорного времени работы.

1 Аналитический раздел

В данном разделе будут рассмотрены основные теоритические понятия конвейерной обработки и параллельных вычислений.

1.1 Основные понятия

Параллельное программирование служит для создания программ, эффективно использующих вычислительные ресурсы за счет одновременного исполнения кода на нескольких вычислительных узлах. Для создания параллельных приложений используются параллельные языки программирования и специализированные системы поддержки параллельного программирования, такие как MPI и OpenMP. Параллельное программирование является более сложным по сравнению с последовательным как в написании кода, так и в его отладки. Для облегчения процесса параллельного программирования существуют специализированные инструменты, например, отладчик TotalView [1].

В модели конвейерной обработки (pipelines) поток обрабатываемых данных проходит через несколько этапов. Прохождение этапов осуществляется строго последовательно. Параллелизм достигается за счет одновременной обработки разных элементов на разных этапах. Конвейерная обработка возникает при работе с последовательными потоками данных (потоки событий, потоки видеосигналов, потоки изображений), а также при многоэтапной обработке элементов последовательности в строго заданном порядке [2].

Конвейерную обработку можно реализовать с помощью задач и конкурентных очередей. Каждая задача реализует этап конвейера, очереди выступают ограниченными буферами, накапливающими элементы.

1.2 Проблема ограниченного буфера

Рассмотрим проблему ограниченного буфера, известную также как проблема производителя и потребителя. Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда. Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Пусть максимальное число элементов в буфере равно N, count – текущее количество элементов в буфере. Если значение count равно N, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение count. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной count, чтобы проверить, не равно ли оно нулю, но квант времени процесса-потреби-

теля закончился. Планировщик задач передал управление процессу-производителю, который поместил элемент в буфер и увеличил значение count. Так как значение было равно 0 и потребитель находиться в состоянии ожидания, производитель активизирует его. Но потребитель не был в состоянии ожидания, поэтому сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он обнаружит, что ранее считанное значение count равно нулю и уйдет в состояние ожидания.

Производитель будет наполнять буфер, пока он не заполнится, после чего перейдёт в состояние ожидания. Оба процесса оказались в состоянии ожидания сигнала активации друг от друга (deadlock). Причина возникновения deadlock-а состоит в том, что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает.

Существует несколько решений проблемы производителя и потребителя. Самым простым является ограничение доступа к переменной count с помощью семафоров [3].

1.3 Вывод

В данном разделе были рассмотрены основы конвейерной обработки и технологии параллельного программирования.

2 Конструкторский раздел

В данном разделе будут рассмотрены описание системы, требования к функциональности ПО и определены способы тестирования.

2.1 Описание системы

Система состоит из трех этапов (рисунок 2.1), который последовательно соединены между собой. Каждый этап выполняет возведение в квадрат полученного на вход элемента. Для уменьшения влияния времени диспетчеризации будет вызываться системный вызов sleep с разным временем ожидания для каждого этапа. Время попадания в каждую очередь логируется.

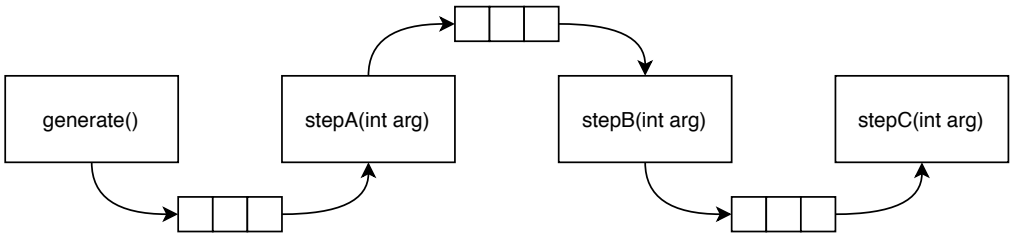

Рисунок 2.1 — Схема конвейерной обработки

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить следующую минимальную функциональность консольного приложения:

1) предоставить возможность ввода количества генерируемых элементов в системе;
2) обеспечить вывод времени получения и отправки элемента конвейера.

2.3 Тесты

Тестирование ПО будет проводиться методом чёрного ящика. Эффект от конвейеризации становиться заметен, если время выполнения этапа много больше времени диспетчеризации. Каждый этап будет возводить число в квадрат и вызывать системный вызов sleep с разным временем ожидания для каждого этапа.

2.4 Вывод

В данном разделе были рассмотрены схема алгоритмов обработки элементов линии конвейера и описаны требования к функциональности ПО.

3 Технологический раздел

В данном разделе будут выбраны средства реализации ПО и представлен листинг кода.

3.1 Средства реализации

В данной работе используется язык программирования golang [4], так как он позволяет написать программу в относительно малый срок за счёт встроенных средств распараллеливания и взаимодействия потоков, таких как gorutine и channels [5]. В качестве среды разработки использовалась Visual Studio Code [6].

Для получения текущего времени была использована функция Now() модуля time. Для упрощения работы с ней используется обёртка getTime, код которой представлен в листинге 3.1.

Листинг 3.1 — Функция получения текущего времени

```
1 func getTime() string {
2     return time.Now().Format("15:04:05.99999999")
3 }
```

3.2 Листинг программы

Ниже представлены листинги кода системы:

1) генерации значений (листинг 3.2);
2) декоратора этапа (листинг 3.3);
3) реализация ступеней конвейера (листинг 3.4);
4) реализация точки входа в программу (листинг 3.5).

Листинг 3.2 — Реализация генерации значений

```
1  func generate(n int) <-chan int {
2      queue := make(chan int)
3
4      go func() {
5          defer close(queue)
6
7          for i := 0; i < n; i++ {
8              value := rand.Intn(100)
9              fmt.Print("time: ", getTime(), " send: ", value, "\n")
10             queue <- value
11         }
12     }()
13
14     return queue
15 }
```

Листинг 3.3 — Реализация декоратора этапа конвейера

```
type actionFunc func(int) int

func action(source <-chan int, name string, f actionFunc) <-chan int {
    result := make(chan int)

    go func() {
        defer close(result)

        for arg := range source {
            fmt.Print("time: ", getTime(), " in ", name, ": ", arg, "\n")
            value := f(arg)
            fmt.Print("time: ", getTime(), " out ", name, ": ", value, "\n")
            result <- value
        }
    }()

    return result
}
```

Листинг 3.4 — Реализация ступеней конвейера

```
func stepA(arg int) int {
    time.Sleep(100)
    return arg * arg
}

func stepB(arg int) int {
    time.Sleep(200)
    return arg * arg
}

func stepC(arg int) int {
    time.Sleep(150)
    return arg * arg
}
```

Листинг 3.5 — Реализация точки входа в программу

```
func main() {
    var count int
    fmt.Print("Enter generate count: ")
    _, err := fmt.Scanf("%d", &count)
    if err != nil {
        fmt.Println(err)
        os.Exit(-1)
    }

```

```
10      var queue = generate(count)
11      queue = action(queue, "A", stepA)
12      queue = action(queue, "B", stepB)
13      queue = action(queue, "C", stepC)
14
15      for res := range queue {
16          fmt.Print("time: ", getTime(), " res: ", res, "\n")
17      }
18  }
```

3.3 Тестирование

На рисунке 3.1 представлен результат работы программы. Из времени поступления и отправки значения из разных этапов можно заключить, что одновременно обрабываются разные элементы на разных этапах, следовательно программа работает корректно.

```
Enter generate count: 4
time: 22:35:25.6318478 send: 81
time: 22:35:25.7128518 send: 87
time: 22:35:25.7128518 in  A: 81
time: 22:35:25.7148525 out A: 6561
time: 22:35:25.7158511 in  A: 87
time: 22:35:25.7158511 in  B: 6561
time: 22:35:25.7158511 send: 47
time: 22:35:25.7178518 out A: 7569
time: 22:35:25.7178518 out B: 43046721
time: 22:35:25.7178518 in  B: 7569
time: 22:35:25.7188532 in  C: 43046721
time: 22:35:25.7188532 in  A: 47
time: 22:35:25.7188532 send: 59
time: 22:35:25.7198649 out B: 57289761
time: 22:35:25.7198649 out C: 1853020188851841
time: 22:35:25.7208506 res: 1853020188851841
time: 22:35:25.7208506 in  C: 57289761
time: 22:35:25.7218841 out A: 2209
time: 22:35:25.7218841 in  A: 59
time: 22:35:25.7218841 in  B: 2209
time: 22:35:25.7238495 out A: 3481
time: 22:35:25.7238495 out C: 3282116715437121
time: 22:35:25.7238495 res: 3282116715437121
time: 22:35:25.7248508 out B: 4879681
time: 22:35:25.7248508 in  B: 3481
time: 22:35:25.7248508 in  C: 4879681
time: 22:35:25.7258529 out B: 12117361
time: 22:35:25.7278524 out C: 23811286661761
time: 22:35:25.7398535 in  C: 12117361
time: 22:35:25.7398535 res: 23811286661761
time: 22:35:25.7418659 out C: 146830437604321
time: 22:35:25.7428559 res: 146830437604321
```

Рисунок 3.1 — Результаты тестирования алгоритмов.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа на основе замеров времени работы алгоритмов.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данной работы были проведёны следующие эксперименты по вычислению:

1) времени работы системы в линейной и конвейерной реализациях (график 4.1);
2) времени работы конвейерной системы от времени работы этапа (график 4.2).

Тестирование проводилось на ноутбуке с процессором Intel(R) Core(TM) i5-7200U CPU 2.50 GHz [7] под управлением Windows 10 с 8 Гб оперативной памяти.

В ходе экспериментов по замеру времени работы в линейной и конвейерной реализациях было установлено, что конвейерная модель обрабатывает элементы в ≈ 2.6 раза быстрее, чем линейная. Это объясняется тем, что одновременно обработываются разные элементы на разных этапах.

По результатам исследования зависимости времени работы конвейерной системы от времени работы этапа можно сделать вывод, если одна из стадий намного более трудоемкая, чем остальные, то конвейерная обработка становится не эффективной, так как производительность всей программы будет упираться в производительность этой стадии, и разницы между обычной обработкой и конвейерной будет малозаметна. В таком случае можно либо разбить трудоемкую стадию на набор менее трудоемких, либо выбрать другой алгоритм, либо отказаться от конвейерной обработки.

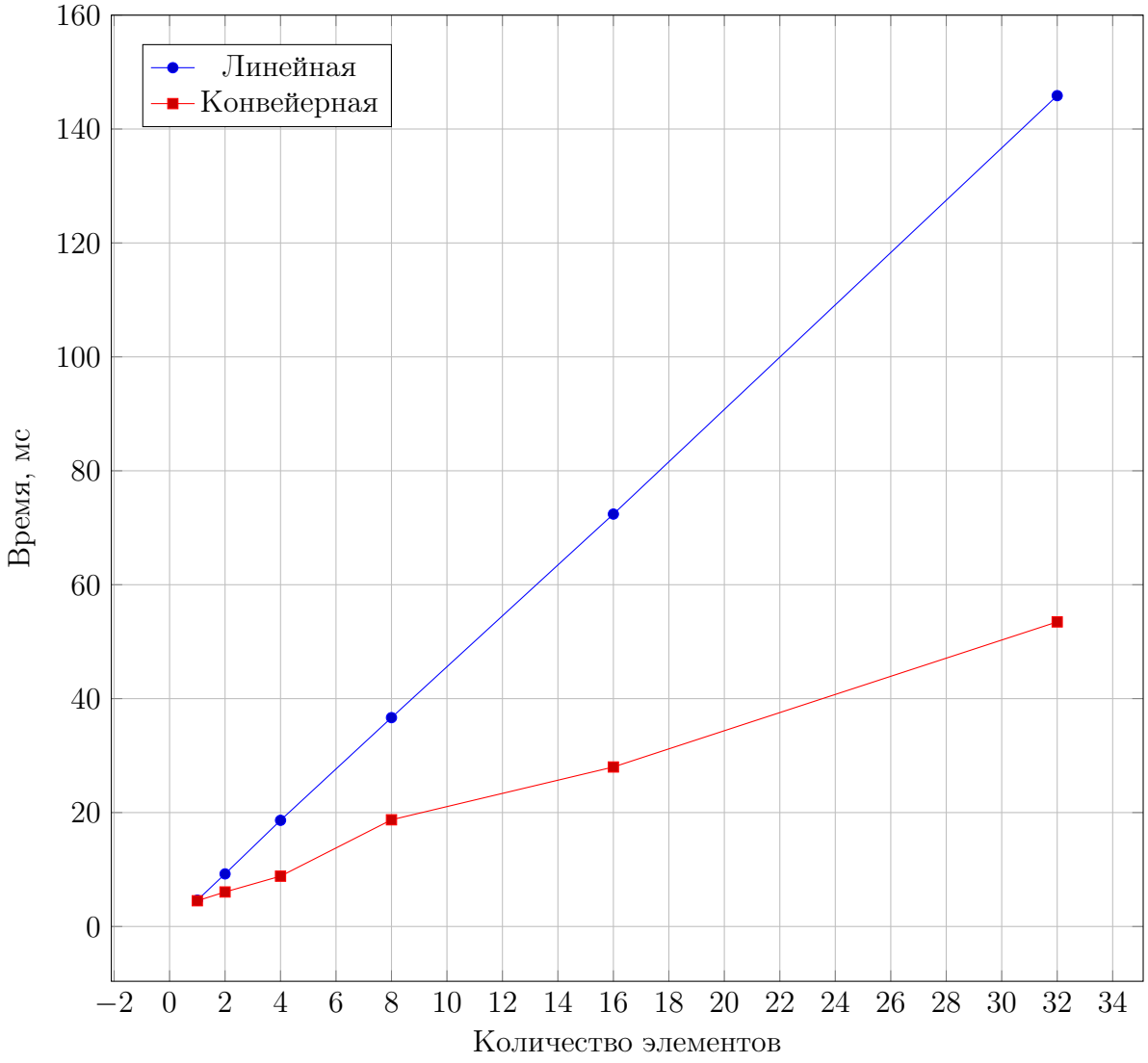

Рисунок 4.1 — График зависимости времени работы от модели системы

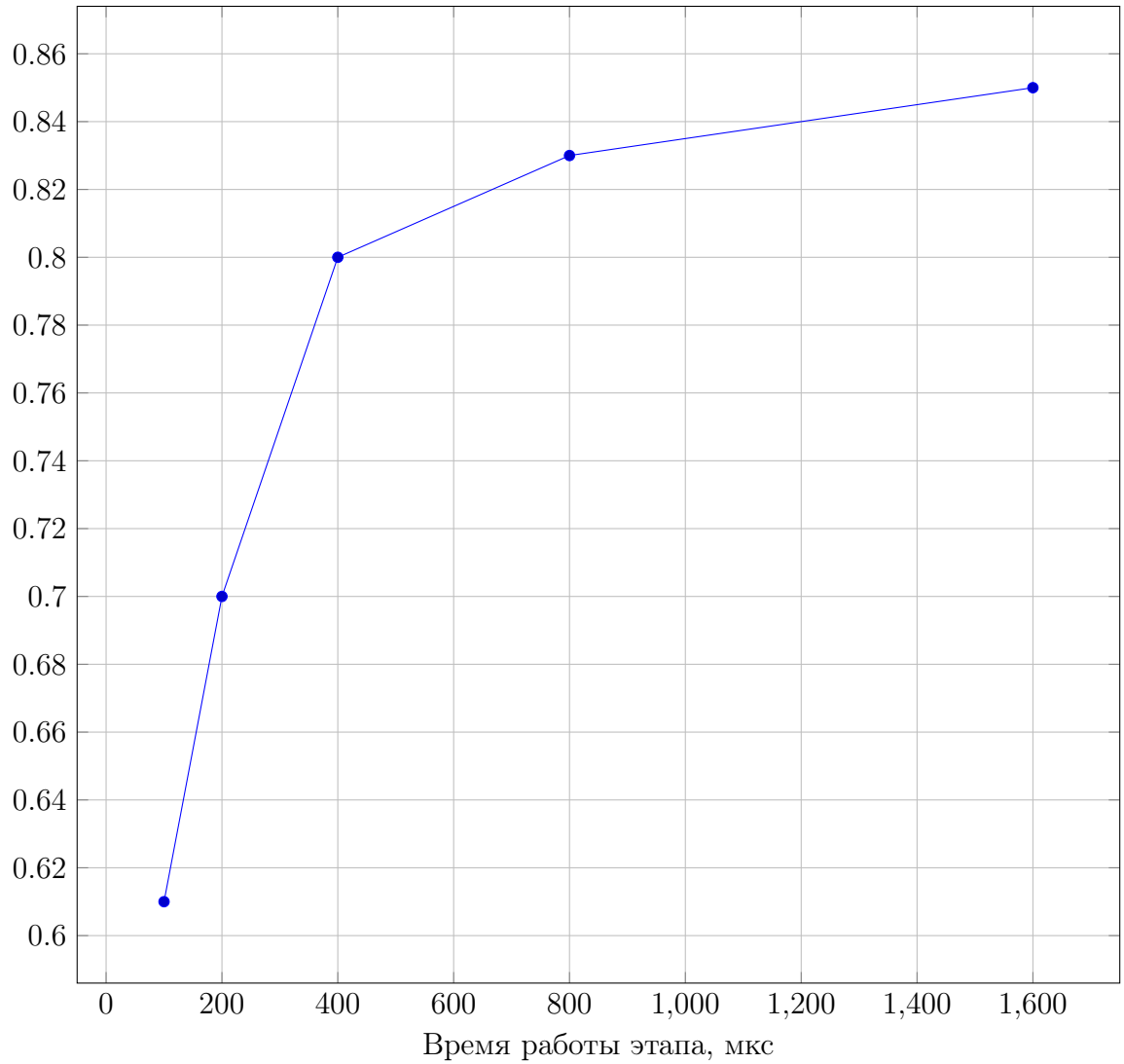

Рисунок 4.2 — График зависимости отношения времени работы первого этапа к времени работы конвейера

Заключение

В ходе выполнения лабораторной работы были описаны и реализованы алгоритмы конвейерной обработки. Разработанный конвейер, с параллельно работающими линиями, позволил ускорить работу программы в 2,6 раз по сравнению с линейной реализацией. Однако если одна из стадий намного более трудоемкая, чем остальные, то конвейерная обработка становится не эффективной, так как производительность всей программы будет упираться в производительность этой стадии, и разницы между обычной обработкой и конвейерной будет малозаметна. В таком случае можно либо разбить трудоемкую стадию на набор менее трудоемких, либо выбрать другой алгоритм, либо отказаться от конвейерной обработки.

Список использованных источников

1. TotalView. // [Электронный ресурс]. Режим доступа: https://totalview.io/products/totalview, (дата обращения: 28.11.2020).

2. Типовые модели параллельных приложений. // [Электронный ресурс]. Режим доступа: https://intuit.ru/studies/courses/5938/1074/lecture/16465?page=3, (дата обращения: 28.11.2020).

3. Синхронизация потоков. Проблема производителя и потребителя. // [Электронный ресурс]. Режим доступа: https://moodle.kstu.ru/mod/page/view.php?id=9274, (дата обращения: 28.11.2020).

4. Go. // [Электронный ресурс]. Режим доступа: https://golang.org/, (дата обращения: 01.10.2020).

5. A Tour of Go. Concurrency. // [Электронный ресурс]. Режим доступа: https://tour.golang.org/concurrency/1, (дата обращения: 01.10.2020).

6. Visual Studio Code - Code Editing. // [Электронный ресурс]. Режим доступа: https://code.visualstudio.com, (дата обращения: 01.10.2020).

7. Intel® Core™ i5-7200U Processor. // [Электронный ресурс]. Режим доступа: https://www.intel.com/content/www/us/en/products/processors/core/i5-processors/i5-7200u.html, (дата обращения: 26.09.2020).