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 №4
по курсу «Анализ алгоритмов»

«Параллельное программирование»

Студент Маслова Марина Дмитриевна

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватель Волкова Лилия Леонидовна

2021 г.

Содержание

Введение

Сегодня программирование используется во многих научных и социальных областях. Компьютерам требуется производить все более трудоемкие вычисления на больших объемах данных. При этом предъявляются требования к скорости вычислений.

Одним из возможных решений увеличения производительности компьютера, то есть скорости решения задач является параллельное программирование. На одном устройстве паралельные вычисления можно организовать с помощью **многопоточности** – способности цетрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. При последовательной реализации какого-либо алгоритма, его программу выполняет только одно ядро процессора. Если же реализовать алгоритм так, что независимые вычислительные задачи смогут выполнять несколько ядер параллельно, то это приведет к ускорению решения всей задачи в целом[**?**].

Для реализации паралельных вычислений требуется выделить те участки алгоритма, которые могут выполняться паралльно без изменения итогового резульатат, также необходми правильно организовать работу с данными, чтобы не потерять вычисленные значения.

Одной из распространненых задач, решение которой используется в различных областях, является численное итегрирование, поэтому в данной лабораторной работе будет предпринята попытка ускорить вычисления определенных интегралов.

Целью данной работы является получение навыков параллельного программирования на основе алгоритма численного интегрирования методом средних прямоугольников.

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

- изучить метод средних прямоугольников для численного интегрирования;
- описать возможности распараллеливания данного алгоритма;
- разработать последовательный и параллельный алгоритмы;

– реализовать каждый алгоритм;
– провести тестирование реализованных алгоритмов;
– провести сравнительный анализ алгоритмов по времени работы реализаций;
– сделать выводы по полученным результатам.

1 Аналитическая часть

В данном разделе представлено теоретическое описание алгоритмов численного интегрирования методом средних прямоугольников.

1.1 Конвейерная обработка данных

1.2 Генерация зашифрованных сообщений

Для проведения исследования в данной лабораторной работе на конвейере будут генерироваться зашифрованные сообщения. На первой ленте будет генерироваться случайное сообщение на английском языке. На второй ленте каждое слово в сообщении будет записываться в обратном порядке. На третьей ленте к сообщению будет применен шифр Веженера.

1.2.1 Шифр Веженера

Шифр Веженера является частным случаем многоалфавитной замены. Формально даннный алгоритм шифрования можно описать следующим образом. Выбирается ключ шифравания

http://window.edu.ru/resource/256/20256/files/rsu572.pdf

1.3 Вывод

В данном разделе был рассмотрен алгоритм численного интегрирования методом средних прямоугольнико, так же был описан механизм распараллеливания данного алгоритма. Из представленных описаний можно предъявить ряд требований к разрабатываемому программному обеспечению:

- на вход должны подаваться пределы интегрирования, заданная точность, а также число потоков для параллельного алгоритма;
- на выходе должны выдаваться вычисленные значения определенного интеграла каждым из алгоритмов, причем результаты должны совпадать;
- интегрируемые функции могут быть предложены пользователю на выбор.

2 Конструкторская часть

В данном разделе разрабатываются последовательный и параллельный алгоритмы, численного интегрирования методом средних прямоугольников, структура программы и способы её тестироваия.

2.1 Разработка алгоритмов

На рисунках **??-??** представлена схема алгоритма последовательного алгоритма численного интегрирования методом средних прямоугольников с заданной точностью, на рисунках **??-??** — схема параллельного алгоритма.

2.2 Структура разрабатываемого ПО

Для реализации разрабатываемого программного обеспечения будет использоваться метод структурного программирования. Каждый из алгоритмов будет представлен отдельной функцией, при необходимости будут выделены подпрограммы для каждой из них. Также будут реализованы функции для ввода-вывода и функция, вызывающая все подпрограммы для связности и полноценности программы.

2.3 Классы эквивалентности при тестировании

Для тестирования программного обеспечения во множестве тестов будут выделены следующие классы эквивалентности:

- совпадение верхнего и нижнего пределов интегрирования;
- положительные пределы интегрирования;
- верхний предел интегрирования меньше нижнего предела;
- произвольные пределы интегрирования.

2.4 Вывод

В данном разделе были разработаны последовательный и параллельный алгоритмы, была описана структура разрабатываемого ПО. Для дальнейшей

проверки правильности работы программы были выделены классы эквивалентности тестов.

3 Технологическая часть

В данном разделе описаны требования к программному обеспечению, средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Требования к программному обеспечению

Программа должна предоставлять следующие возможности:

- ввода количества обрабатываемых заявок;
- вывода лога работы программы для введенного количества заявок, включающих для каждой заявки время начала и конца обработки на каждой из лент;
- получения времени обработки всех заявок.

3.2 Средства реализации

Для реализации данной лабораторной работы выбран компилируемый многопоточный язык програмирования `Go`[1], так как он предоставляет необходимый функционал для работы с потоками и простой реализациии их взаимодействия. Интерпретируемый язык программирования высокого уровня Python[2] был выбран для визуализации данных эксперимента, так как он предоставляет большое число настроек параметров графика с использованием простого синтаксиса.

В качестве среды разработки выбран текстовый редактор Vim[3] с установленными плагинами автодополнения и поиска ошибок в процессе написания, так как он реализует быстрое перемещение по тексту программы и простое взаимодействие с командной строкой.

Замеры времени проводились при помощи функции `Now()` из библиотеки `time`[4].

3.3 Листинги кода

В данном подразделе представлены листинги кода алгоритмов:

– реализация генерации сообщений (листинг 3.1);

– реализация перестановки букв в каждом слове в обратном порядке (листинг 3.2);

– реализация шифра Веженера (листинг 3.3);

– последовательная реализация конвейера (листинг 3.4);

– параллельная реализация конвейера (листинг 3.5).

Листинг 3.1 – Функция генерации сообщений

```
func GenerateMsg(msg *Cipher) {
    msg.Msg = gofakeit.Sentence(msg.wordsNum)
    msg.Msg = strings.TrimRight(strings.ToLower(msg.Msg), ".")
}
```

Листинг 3.2 – Функции перестановки букв в каждом слове в обратном порядке

```
func reverseString(str string) (res string) {
    for _, ch := range str {
        res = string(ch) + res
    }

    return res
}

func ReverseWords(msg *Cipher) {
    words := strings.Fields(msg.Msg)

    for i, word := range words {
        words[i] = reverseString(word)
    }

    msg.Msg = strings.Join(words, " ")
}
```

Листинг 3.3 – Функция шифра Веженера

```
func CodeByVegenere(msg *Cipher) {
    key := []rune(gofakeit.Word())
    keyInd := 0
    res := ""
```

Листинг 3.3 (продолжение)

```
5
6       for _, ch := range msg.Msg {
7           if ch >= 'a' && ch <= 'z' {
8               ch -= 'a'
9               keyCh := key[keyInd] - 'a'
10
11              ch = (ch+keyCh)%26 + 'a'
12
13              keyInd++
14              keyInd %= len(key)
15          }
16
17          res += string(ch)
18      }
19
20      msg.Msg = res
21  }
```

Листинг 3.4 – Последовательная обработка заявок

```
1  func LinearConveyor(inQueue *Queue) *Queue {
2      res := CreateQueue(inQueue.capacity)
3
4      c := inQueue.Pop()
5      for c != nil {
6          c.startMsg = time.Now()
7          GenerateMsg(c)
8          c.endMsg = time.Now()
9
10         c.startReplace = time.Now()
11         ReverseWords(c)
12         c.endReplace = time.Now()
13
14         c.startVegenere = time.Now()
15         CodeByVegenere(c)
16         c.endVegenere = time.Now()
17
18         res.Push(c)
19         c = inQueue.Pop()
20     }
21
22     return res
23 }
```

Листинг 3.5 – Конвейерная обработка заявок

```
func ParallelConveyor(inQueue *Queue) *Queue {
    mainToF := make(chan *Cipher, inQueue.capacity)
    fToS := make(chan *Cipher, inQueue.capacity)
    sToT := make(chan *Cipher, inQueue.capacity)
    tToMain := make(chan int)
    res := CreateQueue(inQueue.capacity)

    first := func() {
        for {
            select {
            case c := <-mainToF:
                c.startMsg = time.Now()
                GenerateMsg(c)
                c.endMsg = time.Now()

                fToS <- c
            }
        }
    }

    second := func() {
        for {
            select {
            case c := <-fToS:
                c.startReplace = time.Now()
                ReverseWords(c)
                c.endReplace = time.Now()

                sToT <- c

            }
        }
    }

    third := func() {
        for {
            select {
            case c := <-sToT:
                c.startVegenere = time.Now()
                CodeByVegenere(c)
                c.endVegenere = time.Now()

                res.Push(c)
                if res.tail == res.capacity-1 {
                    tToMain <- 0
```

Листинг 3.5 (продолжение)

```
                }
            }
        }
    }

    go first()
    go second()
    go third()

    cip := inQueue.Pop()
    for cip != nil {
        mainToF <- cip
        cip = inQueue.Pop()
    }

    <-tToMain

    return res
}
```

3.4 Описание тестирования

В таблице 3.1 приведены функциональные тесты программы.

Таблица 3.1 – Функциональные тесты

Количество заявок	**Ожидаемый результат**
-1	Сообщение об ошибке
0	Сообщение об ошибке
j	Сообщение об ошибке
10	Лог работы каждого из алгоритмов
100	Лог работы каждого из алгоритмов
1000	Лог работы каждого из алгоритмов

3.5 Вывод

В данном разделе были реализованы последовательный и конвейрный алгоритмы поэтапной генерации зашифрованных сообщений. Также были написаны тесты для каждого класса эквивалентности, описанного в конструкторском разделе.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

– Операционная система: Manjaro [5] Linux x86_64.

– Память: 8 GiB.

– Процессор: Intel® Core™ i5-8265U, 4 физических ядра, 8 логических ядра[6].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Примеры работы программы

На рисунке 4.1 представлены результаты работы последовательной обработки, на рисунке 4.2 – конвейерной.

4.3 Результаты тестирования

Программа была протестирована на входных данных, приведенных в таблице 3.1. Полученные результаты работы программы совпали с ожидаемыми результатами.

4.4 Постановка эксперимента по замеру времени

Для оценки времени работы последовательной и конвейерной реализации алгоритмов шифрования был проведен эксперимент, в котором определялось влияние количества заявок и количества слов в сообщении на время работы каждого из алгоритмов. Тестирование проводилось на количестве заявок от 5 до 10 с шагом 5, от 25 до 100 с шагом 25 и от 100 до 1000 с

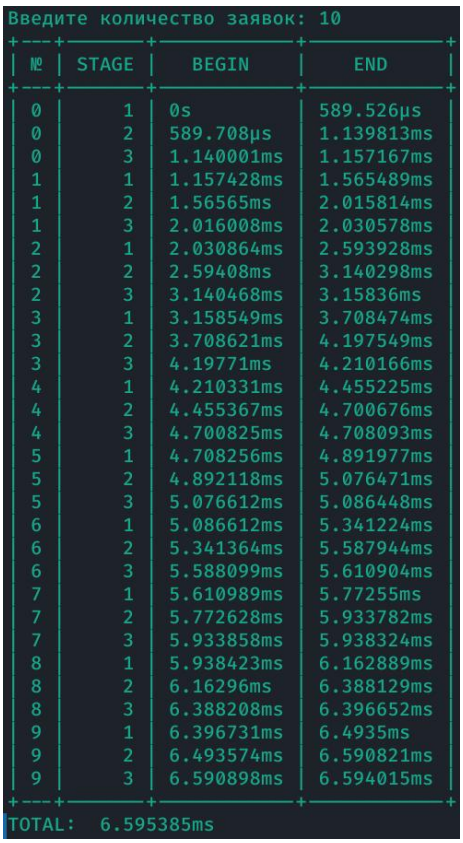

Рисунок 4.1 – Лог последовательной обработки

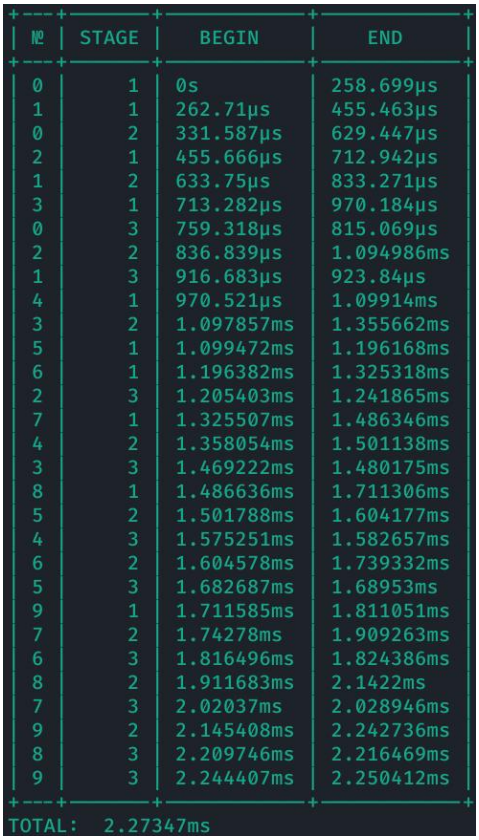

Рисунок 4.2 – Лог конвейерной обработки

шагом 250, а количество слов принимало значения $5, 10, 25, 50, 75, 100$. Время работы на каждом из значений было получено с помощью *бенчмарков*[7], являющимися встроенными средствами языка Go. В них количество повторов измерений времени изменяется динамически до тех пор, пока не будет получен стабильный результат.

Результаты эксперимента были представлены в виде таблиц и графиков, приведенных в следующем подразделе.

4.5 Результаты эксперимента

В таблице 4.1 представлены результаты измерения времени работы последовательной и конвейерной реализаций в зависимости от числа заявок. На рисунке 4.3 представлен соответствующий график.

В таблице 4.2 представлены результаты измерения времени работы последовательной и конвейерной реализаций в зависимости от числа слов в сообщениях при фиксированном числе заявок, равном 50. На рисунке 4.4 представлен соответствующий график.

Таблица 4.1 – Время работы от числа заявок

Число заявок	**Последовательная, нс**	**Конвейерная, нс**
5	1398118	1744339
10	2842780	2770063
25	6987262	5289069
50	13423468	8961571
75	20151882	13339268
100	27811972	17249171
250	71697350	40135084
500	133232370	84669318
750	209148024	127482157
100	276952383	160373144

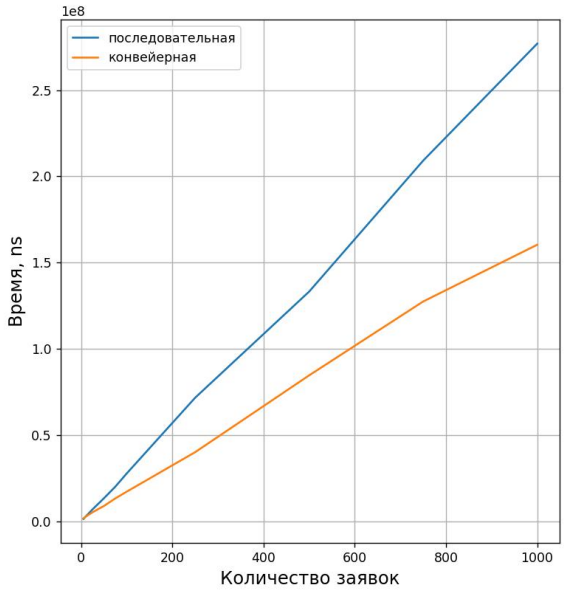

Рисунок 4.3 – График зависимости времени работы от числа заявок

Таблица 4.2 – Время работы от количества слов

Точность	**Последовательная, нс**	**Параллельная, нс**
5	12787683	8451608
10	25006741	14918758
25	62884400	37303679
50	132833197	73853581
75	201132505	108303512
100	259405110	143977382

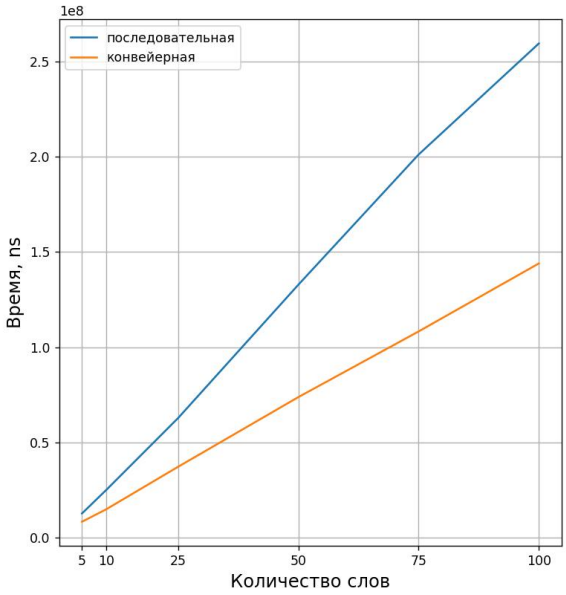

Рисунок 4.4 – График зависимости времени работы реализаций от количества слов в сообщениях

4.6 Вывод

По результатам эксперимента можно сделать следующие выводы:

- при количестве заявок до 10 последовательная и конвейерная реализация генерации зашифрованных сообщений отрабатывают за одинаковое время, а при количестве заявок до 5 последовательная реализация работает 1.3 раза быстрее, что объясняется затратами на передачу данных между лентами с помощью очередей/каналов в конвейерной реализации;
- при большем количестве заявок от 25 конвейерная реализация работает в 1.7 раза быстрее последовательной;
- при фиксированном количестве заявок при различных количествах слов в сообщениях конвейерная реализация работает в 1.7 раза быстрее последовательной.

Таким образом, для обработки количества заявок большего 10 для достижения оптимальной скорости вычислений необходимо использовать конвейерную реализацию. Если работа просходит с количеством заявок до 10 достаточно последовательной реализации, то есть нет необходимости реализовывать более сложный конвейерный алгоритм.

Заключение

В ходе исследования был проведен сравнительный аналализ последовательной и конвейерной реализации алгорима поэтапного шифрования сообщений. В результате исследования было выяснено, что при количестве заявок более 10 конвейерная реализация дает выигрыш в 1.7 раза по сравнению с последовательной, что говорит о преимуществе параллельной реализации этапов конвейера при решении поставленной задачи.

В ходе выполения лабораторной работы:

ПОМЕНЯТЬ

- были описаны и разработаны последовательный и параллельный алгоритм численного интегрирования методом средних прямоугольников;
- был реализован каждый из описанных алгоритмов;
- по экспериментальным данным были сделаны выводы об эффективности по времени каждого из реализованных алгоритмов;
- были получены зависимости времени работы паралелльного алгоритма от числа потоков и времени работы каждого из алгритмов от точности вычислений.

Таким образом, все поставленные задачи были выполнены, а цель достигнута.

Список литературы

[1] The Go Programming Language [Электронный ресурс]. Режим доступа: https://golang.org/ (дата обращения: 07.12.2021).

[2] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 12.10.2021).

[3] welcome home : vim online [Электронный ресурс]. Режим доступа: https://www.vim.org/ (дата обращения: 12.10.2021).

[4] Package time [Электронный ресурс]. Режим доступа: https://golang.org/pkg/time/ (дата обращения: 07.12.2021).

[5] Manjaro — enjoy the simplicity [Электронный ресурс]. Режим доступа: https://manjaro.org/ (дата обращения: 17.10.2021).

[6] Процессор Intel® Core™ i5-8265U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/149088/intel-core-i5-8265u-processor-6m-cache-up-to-3-90-ghz.html (дата обращения: 17.10.2021).

[7] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: https://golang.org/pkg/testing/ (дата обращения: 07.12.2021).