Parallel programming in Go and Scala

A performance comparison

Carl Johnell

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Contact Information:
Author:
Carl Johnell
Email: cjohnell@gmail.com

University advisor:
Jürgen Börstler
Department of Software Engineering

Faculty of Computing	Internet	: www.bth.se
Blekinge Institute of Technology	Phone	: +46 455 38 50 00
SE-371 79 Karlskrona, Sweden	Fax	: +46 455 38 50 57

ABSTRACT

This thesis provides a performance comparison of parallel programming in Go and Scala. Go supports concurrency through goroutines and channels. Scala have parallel collections, futures and actors that can be used for concurrent and parallel programming. The experiment used two different types of algorithms to compare the performance between Go and Scala. Parallel versions of matrix multiplication and matrix chain multiplication were implemented with goroutines and channels in Go. Matrix multiplication was implemented with parallel collections and futures in Scala, and chain multiplication was implemented with actors.

The results from the study shows that Scala has better performance than Go, parallel matrix multiplication was about 3x faster in Scala. However, goroutines and channels are more efficient than actors. Go performed better than Scala when the number of goroutines and actors increased in the benchmark for parallel chain multiplication.

Both Go and Scala have features that makes parallel programming easier, but I found Go as a language was easier to learn and understand than Scala. I recommend anyone interested in Go to try it out because of its ease of use.

Table of Contents

0.5 Terminology

Concurrency & **Parallelism:** The term concurrency refers to techniques that makes it easier to design solutions for concurrent problems. The purpose of parallelism is to make a program faster by performing computations in parallel [44]. A web browser is an example of a program whose goal is concurrency and not parallelism. The user is able to switch between tabs and menu options while the browser is loading and rendering documents. The browser is doing lots of things concurrently, loading documents and handling user input. Scientific calculations are examples where parallelism is utilised to speed up the computations by performing them in parallel. The constructs provided by a programming language can often be used for both concurrency and parallelism.

Rob Pike defines concurrency as “dealing with lots of things at once” and parallelism as “doing lots of things at once” [45].

Speedup: A metric for relative performance improvement [14]. For example, if a task takes 20 seconds to execute on 1 core and 10 seconds on 2 cores the speedup is 20 / 10 = 2. The speedup in this example is known as linear speedup.

1 Introduction

Over the last decades processor manufacturers have improved performance through increased clock speeds, execution flow optimization and caches. By increasing clock speed more instructions can be processed within the same time frame, i.e. the same work can be executed faster. Execution flow optimization are techniques that improve the instruction stream so they can be executed more efficiently. Examples of such techniques are pipelining, branch prediction and instruction reordering. Continuous improvements to CPU performance has been made possible by the fact that the number of transistors per square inch appears to double every two years, known as Moore's law [41].

However, around 2003 the rapid increase in clock speeds came to a halt due to problems with heat and power consumption. These physical constraints forced CPU designers to look for performance gains in other areas, mainly multi-core technologies [42]. A multi-core processor is equipped with multiple CPUs, or cores, on the same chip and enables parallel processing of multiple threads.

In 2005 Herb Sutter wrote an article, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software", where he argued that software development had reached a turning point [42]. Developers are now forced to take a concurrent approach to software design in order to maximize performance.

Concurrency and parallelism are still open issues, even though multi-core processors have been mainstream for more than ten years. The traditional method for concurrency has been multithreading and while this model is very simple, it's inherently nondeterministic and hard to reason about [22]. One of the main problems with concurrency is modifying shared data from different threads. The instructions from two threads can interleave, causing a race condition. The result from such a scenario is nondeterministic and depends on the relative timing between the threads [46]. Threads need to synchronize access in order to avoid race conditions, which adds another set of problems, such as deadlocks and starvation.

In order to make parallel and concurrent programming more accessible, programming languages and libraries need to provide abstractions that makes it easier for developers to write concurrent code with good performance.

As multi-cores have become ubiquitous, functional programming has also received more attention. A functional program is composed of functions that have properties similar to ordinary mathematical functions [23], which means that side effects are avoided. A function has a side effect if, in addition to returning a value, it also modifies some state. Functional programming favors immutability over mutability. This means that synchronized access to shared data is unnecessary because it can't be modified. In addition to this, functional programs are easy to run concurrently because the order in which the functions execute doesn't matter. The output value from a function only depends on what the input is. Scala is an object-functional programming language that has full support for functional programming, as well as object-oriented programming. It runs on the JVM and has interoperability with Java. Examples of different methods for concurrency and parallelism in Scala are futures and promises [35], actors [37] and parallel collections [15].

Go is a new programming language from Google that was released in 2012 (version 1). Go was created with focus on concurrency and has built-in support that enables concurrent programming. The concurrency primitives in Go are based on communicating sequential processes (CSP), introduced by Tony Hoare in 1979. The basic concurrency constructs in Go are known as goroutines and channels.

Both languages aim to make concurrency easier, as described on their official sites, and it's one of the main selling points for both Go and Scala.

Based on the literature study, described in section 4.2 and 5.1, no previous study has compared concurrency or parallelism between Go and Scala in the same context and environment. This thesis intends to fill that void by providing insight to their performance, speedup and synchronization / communication capabilities. The results from this study can help developers who are deciding between Go and Scala and it can also help in understanding which concurrency model might best be applied to their situation.

The goal with this thesis is to evaluate parallelism in Go and Scala. More specifically, compare the performance of concurrency primitives in Go with actors, futures and parallel collection in Scala. The experiment measures the execution time of two different types of algorithms. The two algorithms were implemented as a sequential and parallel version in both Go and Scala. Matrix multiplication compares the execution time and speedup of goroutines, futures and parallel collections. Matrix chain multiplication compares goroutines and channels with actors, and compares the impact that synchronization / communication, between multiple processes, have on performance. The design of the experiment was derived from the studies presented in section 5.1, how to compare Go and Scala and what type of algorithms to use in the benchmark. The algorithms are explained in more detail in section 4.3.2 and 4.3.3.

Related work is presented in chapter 2. Chapter 3 further introduces Go and Scala with examples of their syntax and features. Chapter 4 outlines details regarding the research questions, and the design of the literature study and experiment. Chapter 5 presents the results from the literature study and experiment, followed by a more detailed analysis of the experiment in chapter 6. Chapter 7 concludes the thesis and answers the research questions. Finally, suggestions for future work are discussed in chapter 8.

2 Related work

The literature study uncovered several papers that have analysed performance or concurrency in Go or Scala. A large portion of these papers are previous theses that have evaluated Go [7, 8, 9, 10, 11, 12, 13]. I found six performance studies regarding Go or Scala, excluding the previous theses. Of these six studies, three focused on Go [1, 2, 3] and two focused on Scala [4, 17]. One study [5] compared sequential performance between C++, Java, Go and Scala.

Tang [2] analysed an early version of Go in 2010 and measured parallelism performance and speedup. He concluded that Go had good support for concurrency and that it was easy to use. Based on Tangs paper, Togashi and Klyuev [1] evaluated compile time, code size and performance in Go and Java. They found that Go compiles faster and has better parallelism performance, while the sequential Java version was faster. Serfass and Tang [3] compared Go with Threading Building Blocks (TBB) in C++ and concluded that TBB performed better.

Pankratius et al. [4] conducted an empirical study where they had 13 programmers implement three concurrent programs in two versions, one version in Scala and the other in Java. Based on their analysis of these programs they concluded that average Scala runtimes are comparable to Java and code in Scala is more compact. Totoo et al. [17] compared parallelism in Haskell, F# and Scala and found that Haskell had the best speedup, followed by Scala and F# respectively.

Hundt [5] compared sequential performance between C++, Java, Go and Scala. Based on their execution time he ranked the languages in the following descending order (fastest to slowest): C++, Scala, Go, Java.

Jonsson and Starrsjö [8] compared Go and Erlang against Java, and found that Go and Erlang are both better suited for parallel programming than Java is. Both Eriksson and Nilsson [7], Järleberg and Nilsson [12], compared Go, Erlang and F#. They analysed performance, concurrency support and productivity between the languages.

Stjernberg and Annebäck [9] compared language features between Go and C++. They found that concurrency in Go is very easy, but it lacks important constructs such as inheritance and generics. Hjalmarsson and Huss [10] analysed ease of use and performance and found that Go is slightly slower than Java and C++. Derek [11] evaluated Go and Clojure and concluded that "channels" in Go are an efficient means of communication and synchronization.

Åberg and Lindeberg [13] compared F#, Go and Java. They measured performance and conducted a survey on how easy Go and F# is to understand. They found that Go is easier to learn than F# and that F# lags behind in performance, while Go and Java performs similar.

These papers were found using the strategy described in section 4.2. The results from the literature study are described in more depth in section 5.1. Even though there are many papers that have analysed concurrency in Go or Scala, it appears that no previous study has compared them in the same context and environment.

3 Background

This chapter provides a short introduction to some of the features in Go and Scala. An initial description of the features for concurrency and parallelism in Go and Scala is also presented. Concurrency and parallelism in Go and Scala are further explained and summarised in section 5.1.2.

3.1 Go

Go was developed with the goal to combine the best from dynamically and statically typed languages. The expressiveness from dynamic languages coupled with the performance and safety provided by static languages. Go was a reaction to the fact that standard languages, C++ and Java, are hard to use, have slow compilation and they are poorly adapted to the current multi-core environment [43]. Go has garbage collection, fast compilation and support for concurrency. Go also support pointer semantics and user defined types are values by default, not references as in Java. This allows for better memory control and cache efficiency. Go is object-oriented but doesn't have the concept of classes and inheritance. Code reuse is instead encouraged through composition.

Figure 3-1 illustrates how a traditional vector class can be implemented in Go. A function that starts with uppercase in Go is globally accessible, similar to the "public" keyword in Java. Private functions start with lowercases and they cannot be accessed from outside the package. The public functions in the example are *Length* and *Dot*. The only private function is *print*. The `:=` operator on line 24 is used for type inference, i.e. the compiler can infer the type from the right hand side of the assignment so the type of the variable can be omitted. Line 25 show a more traditional variable declaration.

```
1.  package vector
2.  import "math"
3.  import "fmt"
4.
5.  type Vector3 struct {
6.    x float64
```

```go
    y float64
    z float64
}

func(v Vector3) print() {
    fmt.Println(“x:”, v.x, “; y:”, v.y, “; z:”, v.z)
}

func (v Vector3) Dot(w Vector3) float64 {
    return v.x * w.x + v.y * w.y + v.z * w.z
}

func (v Vector3) Length() float64 {
    return math.Sqrt(v.x * v.y * v.z)
}

// Create vector v and w as value types
v := Vector3{x: 1, y: 3, z: 9}
var w Vector3 = Vector3{2, 4, 8}

v.Length()
v.DotProduct(w)
```

Figure 3-1: Vector implementation in Go

Arrays are the main data structure used by the algorithms in this thesis. Arrays are values in Go, assigning one array to another will copy all the elements. The length of an array is also part of its type, i.e. an array of type int with 3 elements is not compatible with an array of type int with 4 elements. The primary purpose of arrays in Go is to support slices. Slices provide a more general and powerful interface to sequences of data. Slices hold a reference to an underlying array, along with the length and capacity. Assigning one slice to another will not copy the elements of the underlying array. Only the slice structure is copied, which is very cheap. Figure 3-2 illustrates several examples with arrays and slices.

```go
// Create an array with 10 elements of type int
var buffer [10]int
buffer[4] = 44

// Create a slice without an underlying array, i.e. a nil slice
var slice1 []int

// Make slice1 point to buffer as its underlying array
// Lower bound can be omitted, i.e. slice1 = buffer[:5] is identical
slice1 = buffer[0:5]
// Will modify buffer[4] to 22 from 44
slice1[4] = 22

// Create a slice from type inference
// Upper bound can also be omitted, slice2 := buffer[5:]
slice2 := buffer[5:10]

// Set entry buffer[5] to 15
// The lower bound of the slice is used as base when indexing to a slice
slice2[0] = 15

// Create an array with 50 elements
// and make a slice that points to this array
slice3 := make([]int, 50)
```

Figure 3-2: Examples of slices and arrays in Go

3.1.1 Examples of concurrency

Concurrency in Go is achieved through goroutines and channels. A goroutine is a function that's executing concurrently with other goroutines. Channels makes it possible for different goroutines to communicate and synchronize [39]. A read from an unbuffered channel will block until data is available. Similarly, a write to an unbuffered channel will block if noone is waiting for data. Line 3 and 9 in figure 3-3 are writing to a channel and line 7 and 18 are reading from a channel. Goroutines are created by prefixing a function call with the keyword go**.** Two goroutines are created in the example, and they need to synchronize in order to print "Hello" and "World" in the correct order. The function *world* will block until *hello* writes to the channel. When the value is available, *world* will continue with its execution and finally notify the main thread (goroutine).

```
1.  func hello(ch chan int) {
2.    fmt.Print("Hello ")
3.    ch <- 1
4.  }
5.
6.  func world(ch chan int, finished chan int) {
7.    <- ch
8.    fmt.Println("world")
9.    finished <- 1
10. }
11.
```

```
ch := make(chan int)
finished := make(chan int)

go world(ch, finished)
go hello(ch)

<- finished
```
Figure 3-3: Goroutines and channels in Go

3.2 Scala

Scala is an object-functional language and has full support for functional and object-oriented programming. Scala runs on the JVM and has seamless interoperability with Java, which gives Scala access to the full ecosystem Java offers. Unlike traditional functional languages, Scala allows for a gradual approach to a more functional style because of its multi paradigm [24]. Scala is functional in the sense that functions are first class citizens, i.e. they are treated as regular objects. Scala supports higher order functions, nested functions, closures and concise syntax for defining anonymous functions, in order to facilitate functional programming. For object-oriented programming, Scala has traits and classes. Traits are similar to interfaces in Java but they allow for default implementations of methods. In addition to this, Scala also supports generics and pattern matching [36].

Figure 3-4 shows a corresponding vector class in Scala. The most noticeable difference is that Scala results in less and more concise code than Go. Scala supports automatic type inference, same as Go, but with a different notation. Line 14 uses type inference to determine the type of *v*, while line 15 declares the type. The type of the return value in methods and functions can also be inferred, as shown by the definition of *length* and *add* (+). All operators in Scala, addition, subtraction, multiplication etc, are implemented as methods, known as operator overloading. Line 7 shows an implementation for vector addition and line 19 shows how to invoke the addition method using infix notation. Infix notation is syntatic sugar for a regular method call, as shown by the comment at line 18. Scala doesn't have an explicit return statement as Java. The last expression in the function body is the return value.

```
1.  class Vector3(val x: Float, val y: Float, val z: Float) {
2.    def length() = {
3.      Math.sqrt(x*x + y*y + z*z)
4.    }
4.
5.    def dot(v: Vector3): Float = v.x * x + v.y * y + v.z * z
```

```

  def +(v: Vector3): = new Vector3(v.x + x, v.y + y, v.z + z)

  private def print() {
    println("x: " + x + "; y: " + y + "; z: " + z)
  }
}

val v = new Vector3(1, 3, 9)
val w: Vector3 = new Vector3(2, 4, 8)

v.length()
// Same as u = v.+(w)
val u = v + w
```
Figure 3-4: Vector implementation in Scala

3.2.1 Examples of concurrency

Three different methods for concurrency and parallelism in Scala were used in this thesis, actors, futures and parallel collections. **Actors** communicate by sending and receiving messages, and each actor has a mailbox that store the messages in the order they are received. These messages are then processed sequentially by the actor [37]. Figure 3-5 shows a possible implementation of the "Hello world"-example with Akka actors. Line 8 and 25 sends a message to an actor using infix notation, i.e. sending a message is just a regular method call. The *receive* method in *Hello* and *World* uses pattern matching to process messages. Line 16 terminates the actor system.

Futures can be used for asynchronous computation. In the example in figure 3-5 a future is created at line 29 which will retrieve users from a database. The *Future* function returns an object that represents a result that is not yet available [35], in this case the users from the database. The body of the *Future* function is executed concurrently. Line 32 defines a callback function for the future that will be invoked once the users are available.

Parallel collections provides parallel implementations for sequences, maps and sets. They're an extension to the standard collections framework in Scala [15]. A sequential collection can be converted to its corresponding parallel implementation by using the *par* method. The method *seq* converts a parallel collection to its sequential version. Line 41 creates a new list with all elements from the original list squared by performing the computations in parallel.

```scala
// Akka actors
case class Print()

class Hello(val world: ActorRef) extends Actor {
  def receive = {
    case Print => {
      print("Hello ")
      world ! Print
    }
  }
}
class World extends Actor {
  def receive = {
    case Print => {
      println("world")
      context.system.shutdown()
    }
  }
}

val system = ActorSystem("system")
val world = system.actorOf(Props[World], "world")
val hello = system.actorOf(Props(new Hello(world)), "hello")

hello ! Print

// Futures

val f: Future[List[User]] = Future {
  db.getUsers()
}
f onSuccess {
  case users => for(user <- users) println(user.name)
}

// Do other stuff

// Parallel collections

val l1 = List(2, 4, 8, 16)
val l2 = l1.par.map(x => x*x).seq
```

Figure 3-5: Akka actors, futures and parallel collections in Scala

4 Method

The thesis was carried out in two steps. The first step being the literature study and the second the execution of the experiment. In this chapter the research questions are introduced, along with a motivation to why they were chosen. The approach taken in the literature study is presented in section 4.2 and the search engines, search terms and the criteria the papers had to fulfill are explained. Section 4.3 introduce the design of the experiment, how it was conducted and why this was the chosen approach. At the end of section 4.2 and 4.3 the limitations and validity threats that the approaches had is presented and the countermeasures I took are also explained. Section 4.3.1 and 4.3.2 present the algorithms used in the experiment in more detail.

4.1 Research questions

The literature study was a prelude to the experimental phase. Previous studies that had analysed performance or concurrency, in Go or Scala, were used as references during the development and execution of the experiment. I had never used Go or Scala before and this approach allowed me to learn about their concurrency support and how other studies had analysed their performance. The focus of this thesis is to analyse parallelism in Go and Scala. The questions related to the experiment reflect this by comparing different aspects of their concurrency constructs.

Literature study

RQ1. What previous performance studies have been conducted regarding Go or Scala?
RQ2. How is concurrency and parallelism facilitated in Go and Scala?

Experiment

RQ3. How does the performance between Go and Scala compare in a multi-core environment?
- Which of the concurrency models achieves the better speedup?

RQ4. How is performance affected by synchronization and communication between multiple processes?

4.2 Design of literature study

The literature study was not a full formal systematic literature review (SLR) according to [18]. However, I used those guidelines to some extent while designing the literature study, as described in this section. This approach was sufficient enough to answer the research questions regarding the literature study which is why a full formal SLR wasn't used. The search engines and the keywords that were used to find references are presented in this section, together with the criteria that the papers had to fulfill. Finally, limitations and validity threats regarding the literature study are explained.

Search engines

These are the search engines that were used to find references from academic journals, articles, conferences and theses.

- Google Scholar [27]
- BTH Summon [28]
- Engineering Village [29]

Keywords

The terms below are related to the research questions and they summarise the main focus of this thesis. Different combinations and variations of the keywords were used in the search engines to find relevant references. Note that I only used search phrases where Go and / or Scala were included. Example of valid search phrase: Go Scala concurrency. Example of invalid search phrase: concurrency performance.

- Scala
- Go / Golang
- Concurrency
- Multi-core / Multicore
- Parallelism
- Parallel programming
- Performance
- Benchmark

Criteria

In order for a paper to be included it had to be related to the research questions regarding the literature study. I.e. performance analysis of Go or Scala, or an analysis of the concurrency features in Go or Scala.

The review process I applied, to find relevant references, was to look at the top 100 - 200 results from the search query. If a paper met the criteria, according to the title and publication, I read the abstract and conclusion. If the paper still seemed relevant I read the remaining of it before finally deciding if it could be used as a reference. If a paper was to be considered as a candidate it had to be in the top 100 - 200 of the search results. The reason for this was to speed up the process and make it more manageable, even if it meant some papers could be overlooked.

4.2.1 Limitations and validity threats

The papers that were found during the literature study have varying degrees of quality. To manage this threat a paper had to reach a certain level of reliability before it could be included as main literature. I only considered a paper as part of the main literature if it had been peer reviewed and published in a conference or journal. The main literature are the studies that were used to derive the design of the experiment. These studies are described in section 5.1. The main limitation during the literature study was that an exhaustive search / study could not be achieved. Some searches yielded thousands of results due to limitations in the search settings, e.g. search phrases including "Go" were hard to limit to the programming language. The consequence of this is that I had to limit the number of search results that could be analysed, which means that some papers could have been overlooked.

4.3 Design of experiment

This experiment was conducted by measuring the execution time of two different algorithms, matrix multiplication and matrix chain multiplication. A sequential and parallel version of both algorithms were implemented in Go and Scala. This is the same method that Tang [2] used when analysing the concurrency performance of an early release of Go. Even though Tang used different algorithms, parallel integration and optimal binary search tree, matrix multiplication and matrix chain multiplication exhibit the same properties from a concurrency point of view. Togashi and Klyuev [1], Serfass and Tang [3] have also conducted similar experiments with Go and the design of both experiments were derived from Tangs original paper [2].

Matrix multiplication requires minimum effort to parallelize, it's easy to divide the work into multiple tasks and no synchronization between the processes is necessary. The purpose of this algorithm was to analyse the speedup. This provide insight to the overhead of creating and running multiple processes in Go and Scala, and their general performance characteristics.

Matrix chain multiplication on the other hand is harder to parallelize because it requires synchronization between the different processes. The purpose of this algorithm was to see the impact synchronization in Go and Scala had on the performance.

See section 4.3.1 and 4.3.2 for a more detailed description of the algorithms.

Parallel matrix multiplication in Scala was implemented in two different versions. One version used parallel collections and the other used blocking futures. Parallel matrix chain multiplication in Scala was implemented with actors, because of its message oriented design. All parallel versions in Go used goroutines and channels. The source code is presented in appendix A and B.

Figure 4-1 show the benchmark procedure. The procedure start by setting the general benchmark parameters (call to *init*), number of cores and which algorithm to execute. The iteration loop start and *setUp* performs the necessary steps for each iteration. The garbage collector is manually invoked to free memory that was allocated in the *setUp* procedure. The specified algorithm is executed and elapsed time is accumulated. Finally, average runtime is calculated and printed to the console. Each benchmark was invoked with 100 iterations. The two versions, sequential and parallel, of both algorithms were measured with this approach.

```
benchmark(iters):
  init()
  sum = 0
  for 1 .. iters
    setUp()
    runGC()
    start = currentTime()
    algorithm()
    sum += (currentTime() - start)
  print sum / iters
  exit
```

Figure 4-1: Pseudocode of the benchmark procdure

The execution time of matrix multiplication was measured with the following sizes: 1024, 1536, 2048, 2560 and 3072. Matrix chain multiplication was measured with 2048 matrices (array with 2049 elements). The variable in this benchmark was the number of processes used to coordinate the computation, ranging from $2^0 - 2^{11}$. Test data for both algorithms, matrices for matrix multiplication and array elements for matrix chain multiplication, were generated randomly in the *setUp* procedure. I.e. new test data was created in each iteration, and the garbage collector was invoked to remove previous test data.

The shell commands listed in figure 4-2 illustrate how the benchmarks were compiled and executed. In this example the matrix multiplication benchmark is compiled and the parallel version is then executed using 4 cores and matrix size 2048×2048. Matrix chain multiplication was executed in a similar fashion.

```
$ go build -o MatrixMult *.go                  #build go benchmark
$ ./MatrixMult -size=2048 -numprocs=4
$ scalac -d classes *.scala                    #build scala benchmark
$ java -cp classes:/path/to/scala-library.jar MatrixMult -Dsize=2048
-Dnumprocs=4
```
Figure 4-2: Shell commands used to compile and execute the benchmarks

The `$ go build` command invoke the 6g compiler and 6l linker to build a Go executable, 6g and 6l is the compiler backend in Go that supports amd64 [30]. There is no need to pass optimization flags to the compiler since optimization is enabled by default in Go [31].

The default settings of the JVM [32] are listed in appendix C. The values of these settings were found with the command `$ java -XX:+PrintFlagsFinal`. See the official Oracle documentation for a more detailed description of these settings.

The hardware and software the benchmarks were executed on:

- Macbook Pro 15” Retina, late 2013; 8 GB 1600 MHz DDR3; 2 GHz Intel i7 quad-core with 32 KB L1-cache, 256 KB L2-cache and 6 MB L3-cache; 250 GB SSD
- OS X Yosemite 10.10.2; Go 1.4.2 darwin/amd64; Scala 2.11.6; JVM 1.7.0_60; Akka 2.3.10

4.3.1 Limitations and validity threats

This experiment was only conducted on one computer. The results from the experiment may vary depending on the hardware and software used. Another threat is that I had never used Go or Scala before. Due to a lack of experience in these languages, the implementations of the two algorithms may not be written in the most idiomatic way and could potentially have performance problems. To accommodate the potential performance problems I avoided things that could have side effects on the performance, e.g. all major memory allocations are performed before measuring the execution times. I also looked at other performance benchmarks in Go [2] and Scala [15] and used some of the code as a reference when implementing the algorithms.

4.3.2 Matrix multiplication

Matrix multiplication takes two matrices A and B of size $p \times q$ and $q \times r$, and produces a third matrix $C = A \cdot B$ of size $p \times r$. Note that matrix multiplication is only defined if the number of columns in A and the number of rows in B are of the same size. Entry i, j in matrix C is denoted as $C[i,j]$ for $i = 1,..,p$ and $j = 1,..,r$. Entry $C[i,j]$ is computed by taking the dot product of row i in A and column j in B: [25]

$$C[i,j] = \sum_{k=1}^{q} A[i,k] \cdot B[k,j]$$

In this experiment p, q and r are all of the same size n to help make it easier to parallelize. The parallel version is just a set of smaller matrices multiplied in parallel. Given m processes we can divide A into m matrices $A_1,..,A_m$, each of size $n/m \times n$. Matrix C is now calculated by multiplying $C_1 = A_1 \cdot B,..,C_m = A_m \cdot B$ in parallel. Since there are no dependencies between the matrix multiplications and no concurrent writes occur, $C_1,..,C_m$ doesn't contain any overlapping rows, we can execute these multiplications simultaneously.

```
matrixMult(A, B):
  if A.cols != B.rows
    return null
  q = A.cols
  C = new Matrix[A.rows,B.cols]
  for i = 1 .. A.rows
    for j = 1 .. B.cols
      C[i,j] = 0
      for k = 1 .. q
        C[i,j] += A[i,k] * B[k,j]
  return C
```

Figure 4-3: Matrix multiplication

4.3.3 Matrix chain multiplication with dynamic programming

Matrix multiplication is associative, $A \cdot (B \cdot C)$ produces the same result as $(A \cdot B) \cdot C$. If we are given a sequence (chain) of n matrices $A_1, .., A_n$, what is the most efficient way of parenthesizing these matrices? Simply putting parentheses from left to right can result in a large amount of unnecessary processing [25].

From line 10 in figure 4-3 we see that there are $p \cdot q \cdot r$ scalar multiplications. The list below shows the number of computations for the matrices A, B and C of size 5×10, 10×20 and 20×100 respectively. The second parenthesization results in more than twice the amount of scalar multiplications.

1. $(A \cdot B) \cdot C$: $(5 \cdot 10 \cdot 20) + (5 \cdot 20 \cdot 100) = 11000$
2. $A \cdot (B \cdot C)$: $(10 \cdot 20 \cdot 100) + (5 \cdot 10 \cdot 100) = 25000$

Let $A_{i..j} = A_i \cdot A_{i+1} \cdot .. A_j$, the product of the chain $A_i, .., A_j$. For any optimal parenthesization, at the last step we are multiplying two matrices $A_{i..j} = A_{i..k} \cdot A_{k+1..j}$. The problem then becomes to decide the value of k and recursively parenthesize the sub-chains $A_i, ..A_k$ and $A_{k+1}, .., A_j$ in an optimal way [26].

In dynamic programming, a sub-problem is only computed once and then stored in a table. The solution to this sub-problem can then later be reused by looking it up in the table. We define M to be our lookup table, where $M[i,j]$ is the minimum cost for $A_{i..j}$. Our input is a list p of matrix sizes, where matrix A_i is of size $p[i-1] \times p[i]$. The following recurrence relation defines the cost of $A_{i..j}$ [25].

$$\begin{array}{lr} M[i,j] = min_{i \le k < j}(M[i,k] + M[k+1,j] + p[i-1] \cdot p[k] \cdot p[j]) & if\ i < j \\ M[i,j] = 0 & if\ i = j \end{array}$$

We try $j - i$ different values of k and choose the value that results in the least amount of computation. Entry $M[1,n]$ contains the minimum number of scalar multiplications in order to compute $A_{1..n}$, which is the answer we want to find. Note that the algorithm in this experiment only gives us the number of scalar multiplications and doesn't say how to actually perform the multiplications. This is easily fixed by adding a second table K which record the value of k in $K[i,j]$. I.e. $K[i,j]$ tells us were to split the chain $A_i, .., A_j$.

The chain $(10,\ 20,\ 30,\ 40,\ 50)$ produces the following lookup table:

	j = 1	j = 2	j = 3	j=4
i = 1	0	6000	18000	38000
i = 2		0	24000	64000
i = 3			0	60000
i=4				0

The algorithm populates the table by executing these steps:

$$
\begin{aligned}
M[4,4] &= 0\\
M[3,3] &= 0\\
M[3,4] &= min(\infty,\ M[3,3] + M[4,4] + p[2]\cdot p[3]\cdot p[4]) = 60000\\
M[2,2] &= 0\\
M[2,3] &= min(\infty,\ M[2,2] + M[3,3] + p[1]\cdot p[2]\cdot p[3]) = 24000\\
M[2,4] &= min(\infty,\ M[2,2] + M[3,4] + p[1]\cdot p[2]\cdot p[4]\\
&\qquad ,\ M[2,3] + M[4,4] + p[1]\cdot p[3]\cdot p[4]) = 64000\\
M[1,1] &= 0\\
M[1,2] &= min(\infty,\ M[1,1] + M[2,2] + p[0]\cdot p[1]\cdot p[2]) = 6000\\
M[1,3] &= min(\infty,\ M[1,1] + M[2,3] + p[0]\cdot p[1]\cdot p[3]\\
&\qquad ,\ M[2,3] + M[3,3] + p[0]\cdot p[2]\cdot p[3]) = 18000\\
M[1,4] &= min(\infty,\ M[1,1] + M[2,4] + p[0]\cdot p[1]\cdot p[4]\\
&\qquad ,\ M[1,2] + M[3,4] + p[0]\cdot p[2]\cdot p[4]\\
&\qquad ,\ M[1,3] + M[4,4] + p[0]\cdot p[3]\cdot p[4]) = 38000
\end{aligned}
$$

The table is created from left to right and bottom to top, because the entry $M[i,j]$ cannot be computed before $M[i+1,j]$ and $M[i,j-1]$. That is, we must compute all entries below i and to the left of j first. These dependencies are the main problem when parallelizing this algorithm.

Figure 4-4 and 4-5 presents the pseudocode of the algorithm. The *computeCost* procedure implements the previously defined recurrence relation. The main procedure *matrixChain* computes each entry in the cost matrix by invoking *computeCost*.

```
computeCost(M, p, i, j):
  if i == j
    M[i,j] = 0
  else
    M[i,j] = INFINITY
    for k = i; k < j; k++
      q = M[i,k] + M[k+1,j] + p[i-1]*p[k]*p[j]
      if q < M[i,j]
        M[i,j] = q
  return
```

Figure 4-4: Implementation of the reccurence relation that computes entry $M[i,j]$ of the cost matrix

```
matrixChain(p):
  n = p.length-1
  M = new Matrix[n,n]
  for i = n; i >= 1; i--
    for j = i; j <= n; j++
      computeCost(M, p, i, j)
  return M[1,n]
```

Figure 4-5: Matrix chain multiplication

To parallelize this algorithm we have to honor the data dependencies previously mentioned, which requires synchronization between the different processes. Given m processes and the chain $A_1,..,A_n$, we divide the cost matrix M into m sub-matrices $M_1,..,M_m$, each of size $n/m \times n$. Process P_i is assigned matrix M_i and then further divides it into $m - i + 1$ sub-matrices $M_{i,i},..,M_{i,m}$, each of size $n/m \times n/m$ [2, 6], called blocks.

Example with 4 processes:

P_1	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{1,4}$
P_2		$M_{2,2}$	$M_{2,3}$	$M_{2,3}$
P_3			$M_{3,3}$	$M_{3,4}$
P_4				$M_{4,4}$

As previously saw, in order to compute $M[i,j]$ we first have to compute $M[i+1,j]$ and $M[i,j-1]$. This is still true, but now we work with blocks instead of single matrix entries. Before we compute block $M_{i,j}$ we first have to compute $M_{i+1,j}$ and $M_{i,j-1}$. The computation of the cost matrix is divided into m steps, or rounds. In the first round the blocks in diagonal $i=j$ are executed simultaneously because there are no dependencies between them. I.e. all processes P_1, P_2, P_3, P_4 are executing. When process P_i finishes its computation of a block, it first has to notify P_{i-1} that the block is ready and then wait for P_{i+1} to finish its computation. After each round one process will exit and when round r have finished there are $m - r$ processes left.

5 Results

This chapter starts by presenting the papers that were found during the literature study. The purpose of the report, how the authors interpreted the results and their conclusion. After this introduction, I give my own interpretation and summary of the papers in the context of this thesis. The research questions related to the literature study are answered and explained at the end of section 5.1.1 and 5.1.2. Section 5.2 presents and explain the results from the experiment, what the charts represent and how they should be interpreted. The research questions related to the experiment are answered in chapter 6.

5.1 Literature study

Below is a summary of the academic papers, theses and articles that were found during the literature study. The papers are grouped into two different categories. The first group are papers where the authors conducted an analysis / comparison of performance in Go or Scala. The second group focuses on different concurrency paradigms.

5.1.1 RQ1: Performance studies in Go or Scala

Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java [4]

An empirical study where different aspects of Scala and Java were compared. Pankratius et al. studied thirteen programmers who worked on three different projects in both Scala and Java. A total of 39 programs in both Scala and Java. The authors analysed different issues such as effort, code, language usage, performance, and programmer satisfaction. They concluded that average Scala execution times are comparable to Java. Lowest execution times are sometimes even better in Scala, but Java scale better on parallel hardware. They also confirmed that code in Scala is more compact than Java, but refuted that Scala lead to less programming and debugging effort.

Multi-core parallel programming in Go [2]

Tang analysed the performance of Go on an octa-core AMD chip (8 processor cores). Parallel integration was used to measure the speedup and parallel dynamic programming to examine the overhead from goroutines and channels. He concluded that writing parallel programs in Go is easy. On a system with 8 cores the speedup was 7.79, almost linear. The overhead from creating and scheduling a goroutine was also very small.

Concurrency in Go and Java: Performance analysis [1]

A short paper where Togashi and Klyuev compared Go and Java by code size, compile times, and performance using matrix multiplication. They concluded that Go compiles faster, has less code and the parallel version achieved better performance. The sequential version in Java performed better..

Comparing parallel performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem [3]
A continuation from Tang's original paper [2], where Tang and Serfass compared Go against Threading Building Blocks (TBB) in C++, using the optimal binary search tree algorithm. They found that the overhead from task scheduling with TBB was smaller than goroutines. The overall performance of TBB was 1.6 to 3.6 times faster than Go.

Loop recognition in c++/java/go/scala [5]
Hundt compared C++, Java, Go and Scala using the loop recognition algorithm. Code size, compile times, memory footprint and performance were analysed. The default data structures in each language, together with their idiomatic loop constructs, were used in the implementation of the algorithm. Hundt avoided concurrency and special optimizations. The optimized version for Go and Scala resulted in the smallest code size. Hundt concluded that C++ achieves the best performance, followed by Scala, Go, and Java respectively. Go had the fastest compile times of these four languages.

Haskell vs. F# vs. Scala: A High-level Language Features and Parallelism Support Comparison [17]
Totoo et al. conducted a programmability and performance comparison between Haskell, F# and Scala. With 8 cores the speedup was 5.62 in Haskell and 4.51 in Scala, and with 4 cores the speedup was 2.62 in F#. The authors also observed that the best speedups are achieved using the highest level of abstraction in these three languages. Regarding Scala actors, they found that the added programming effort doesn't justify their performance gains.

Summary
The table below summarises previous performance studies in Go and Scala in the context of Java. Go and Java performance appears to be quite similar, and Scala appears to be faster than Java.

Comparison	Description
Go and Java	- The fastest Java version performed better than the fastest Go version in Hundt's benchmark [5]. - In the study by Togashi and Klyuev [1], the sequential version in Java was faster while the parallel version in Go performed better.

Scala and Java	- The benchmark from Pankratius et al. [4] shows that median execution time was 83 seconds in Scala and 98 seconds in Java. - The fastest Scala version performed better than the fastest Java version in Hundt's benchmark [5].

An article published on the official Go blog [33] presents some flaws with the benchmark that was used by Hundt [5]. In Hundt's benchmark the best C++ version executed in 23 seconds and the best Go version executed in 126 seconds. When the author of the article ran the same benchmark C++ executed in 17 seconds and Go in 25 seconds. After some modifications to the implementation Go executed in 2.25 seconds and C++ in 1.99 seconds, almost identical performance. This example demonstrates the difficulties with performance measurements.

All studies seem to reach the same conclusion that compile times in Go are very efficient compared to other languages.

All studies that analysed parallelism [1, 2, 3, 17] measured the performance on a different number of cores / threads and compared the absolute execution times and the relative speedup. Some studies also analysed the performance loss caused by synchronization [2, 3].

5.1.2 RQ2: Concurrency and parallelism in Go and Scala

Goroutines and Channels [38, 39]

Concurrency in Go is based on goroutines and channels. A goroutine is a function executing concurrently with other goroutines. Channels allow goroutines to communicate between each other through send and receive operations. A goroutine that's receving from a channel is blocked until another goroutine send a value to the same channel. Goroutines are lightweight threads managed by the Go runtime. They execute in the same address space and a goroutine only maintain a pointer to its own stack, making them very efficient and cheap. Go avoid the problems with access to shared variables by passing data between goroutines through channels. Only one goroutine has access to a shared variable at any given time. Channels also allow for goroutines to synchronize their execution.

Futures and Promises, article from the Scala documentation [35]

This article introduces the concept of futures and promises in Scala. A future is an object that represent a result which may become available at some later point, hence the name "future". Futures are computed concurrently and are by default non-blocking. When the result is available the future can invoke a callback with the data, thus enabling asynchronous computation. It's also possible to block while waiting for the future to complete. Futures can be composed in a similar way as collections using functions such as *map*, *filter*, *flatMap* and *foreach*. With these functions the value from a future is used to produce a new future. However, from a user perspective there is only one future that represents the final result. A future is usually a read only object created by the *Future* function. Promises on the other hand can create a writable future. The result in the future is available when a value from the promise is written to it, thus fulfilling its "promise".

Actor systems in Akka [37]
An actor is the fundamental object for concurrency in this model. Actors communicate by messages and based on the message an actor can either forward it to other actors, create new actors or perform some computation. Each actor has a mailbox that store messages in the order they were received, and the messages are processed sequentially by the actor. Actors usually contain some variables that represent its state. This state can be modified, based on the received message, but since actor code is not executed concurrently there is no need to synchronize access to the data. The ability of an actor to create other actors that it can manage allow for easy task decomposition. A message can be sent to an actor to perform some task, and this message can be further delegated to other actors until it becomes small enough to manage. Actors are made possible by the actor system. The system manages a thread pool and takes care of scheduling, sending and delivering of messages, logging etc. The system is highly configurable, e.g. which type of mailbox an actor should use and how different type of actors should be scheduled.

Inside Scala Actors [20]
In this paper from 2010 Corbat introduced the standard library implementation of actors for Scala, which as of version 2.11 is deprecated in favor of the actors implementation from Akka [34]. The paper describes the functionality of the library, potential pitfalls and performance. Corbat concluded that features from Scala make actors intuitive and he wishes for this concurrency model to gain more popularity.

Software transactional memory for Scala [21]
Goodman et al. provides an overview of different options for implementing software transactional memories (STM) in Scala. They graded the implementation techniques from least invasive to most invasive, in terms of required runtime and compiler modifications. STM is an alternative to the traditional locking mechanism. Instead of using locks to restrict access to a critical region, code can execute concurrently with the exception that information is recorded so that a rollback can be performed if necessary. When two transactions write to the same memory, only one transaction will be committed and the changes from the other transaction will be reverted. The authors evaluated usability and invasiveness of pure library based implementations, bytecode rewriting and compiler modifications. Usability refers to how much involvement from the programmer is required in order to utilize STM. They concluded that the

best implementation technique is not on either end of the spectrum, but somewhere in the middle.

A Generic Parallel Collection Framework [15]
A paper by Odersky et al. where they describe the implementation of a generic parallel collection framework for Scala. Their framework can parallelize operations on several different data structures such as arrays, lists, maps and sets. A parallel collection is created by invoking the *par* function of a standard sequential collection. All functions performed on this new collection such as *map*, *filter, foreach* etc, will be executed in parallel. It's also possible to convert a parallel collection to a sequential by invoking the *seq* function. Tasks are automatically created and scheduled on different threads when a method is invoked on a parallel collection. When the threads have finished, the framework will combine their work in order to produce the final result.

Concurrency Programming Paradigns, A Comparison in Scala [19]
Lesani et al. analysed ease of use and performance of different concurrency paradigms in Scala. Among the paradigms they compared are locking and conditions, non-blocking algorithms, actors and software transactional memory (STM). The authors investigated two common features of the concurrency models, isolation and signaling. The isolation mechanism prevents concurrent operations from accessing shared data in an inconsistent state. Signaling is the mechanism that allows a process to inform another process about an event. Their benchmark shows that fine grained locking gives the best performance, followed by non-blocking algorithms, STM and actors. They concluded that actors or STM are the best choices, depending on the task. STM is best suited when data consistency is the problem while actors make coordination problems easier.

Summary
Concurrency and parallelism in Scala can be achieved through a myriad of different techniques. Software transactional memory wasn't used in this thesis because they don't allow for parallel execution. STM is only used to protect concurrent access to shared data. Concurrency in Go is mainly achieved through the built-in support for goroutines and channels. The table below provides a summary of the concurrency and parallelism features that were just described.

Language	Concurrency / parallelism construct	Features
Go	Goroutines and channels	- Channels can be used for both signaling and isolation - Concurrent execution with goroutines
Scala	Futures and promises	- Mainly used for asynchronous computation

	Actors	- No concurrent execution inside an actor, which means good isolation - Good support for signaling through messaging
	Software transactional memories	- No support for signaling - Good support for isolation
	Parallel collections	- Parallelism without any user effort.

5.2 Experiment

The performance results from the experiment are presented in this section. The matrix multiplication benchmark include two types of charts. The first chart shows the execution time with different matrix sizes and the second chart shows the speedup of the parallel version, relative to the sequential version. The charts from matrix chain multiplication shows the execution time with different levels of granularity. The notation np that's used in the charts is an acronym for **n**umber of **p**rocessors. seq is an abbreviation of sequential. I.e. $np = x$ is the parallel version of the algorithm executed on x processors and seq is the sequential version of the algorithm.

The implementation of matrix multiplication differ slightly from the pseudocode described in section 4.3.2. The innermost loop is swapped with the second innermost, i.e. line 7 and 9 in figure 4-3 are swapped and the initialisation of matrix C is done prior to the multiplication. These changes in the algorithm results in 4x - 10x performance improvement due to better cache efficiency. This optimization technique is called loop interchange [16].

5.2.1 Matrix multiplication

Results from the matrix multiplication benchmark. The benchmarks were executed with the matrix sizes $512, 1024, 1536, 2048, 2560$ and 3070. Figure 5-1 shows that Go was slowest in both the sequential and the parallel case. From figure 5-2 and 5-3 we can see that futures yielded slightly better performance than parallel collections in Scala. With matrix size 3070 and 4 processors, Go took 18.39 seconds to execute, parallel collections in Scala took 6.5 seconds and futures took 5.62 seconds. The sequential version in Go with matrix size 3070 took 65.02 seconds to execute and 19.88 seconds in Scala. This shows that Scala is roughly 3x faster than Go.

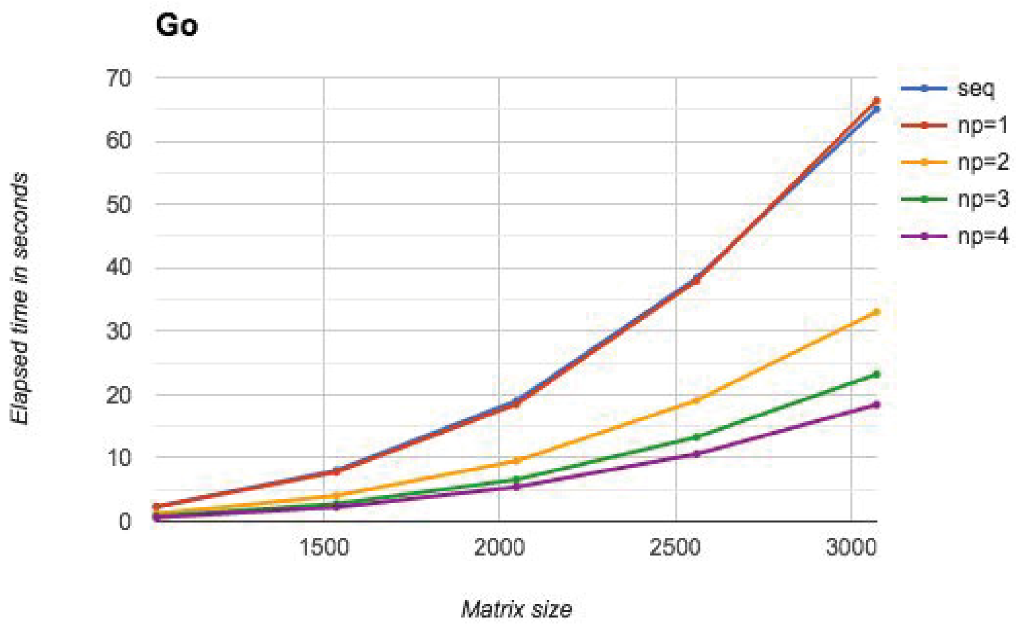

Figure 5-1: Matrix multiplication in Go

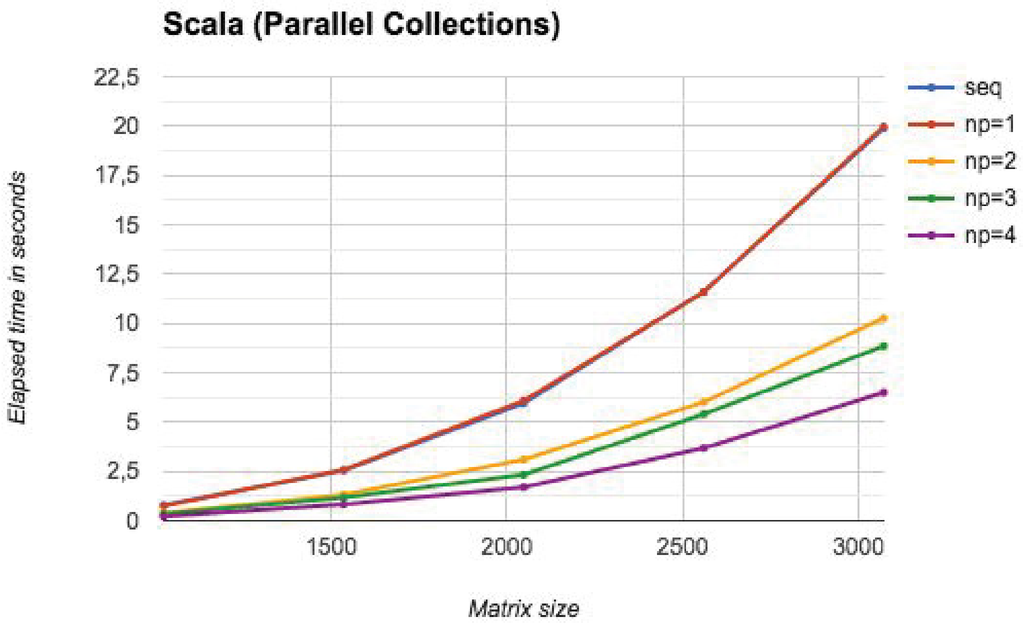

Figure 5-2: Matrix multiplication in Scala with parallel collections

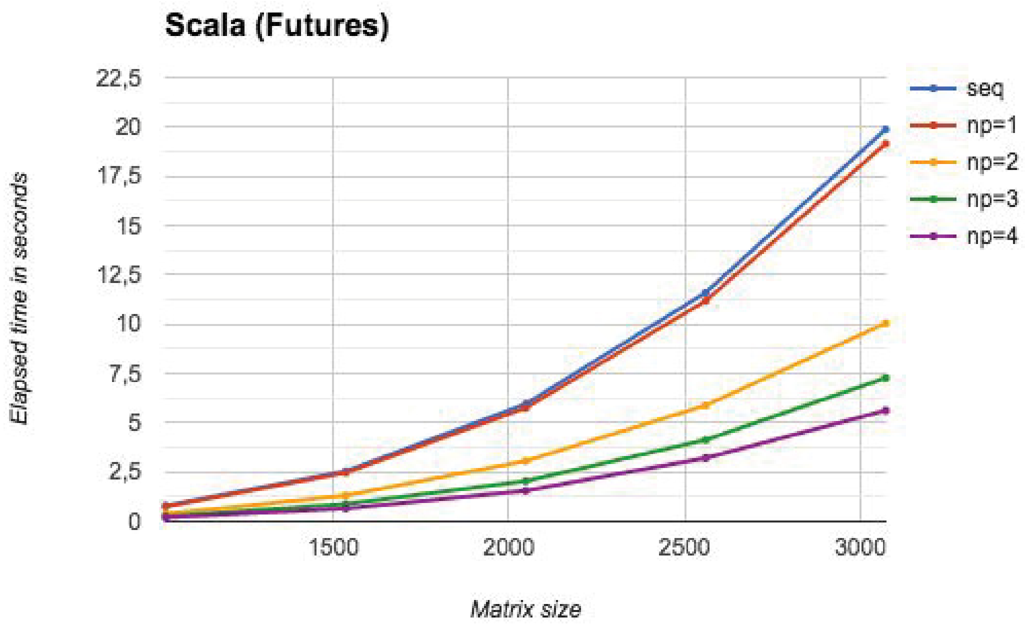

Figure 5-3: Matrix multiplication in Scala with futures

5.2.2 Speedup with parallel matrix multiplication

Relative speedup of the three versions. The purpose is to show how the processing speed of different matrix sizes improved when more processors were used, i.e. how much faster they could be computed. The linear line illustrates the ideal speedup. Figure 5-5 shows that parallel collections had the worst speedup. Figure 5-4 and 5-6 shows that goroutines and futures have quite similar speedup.

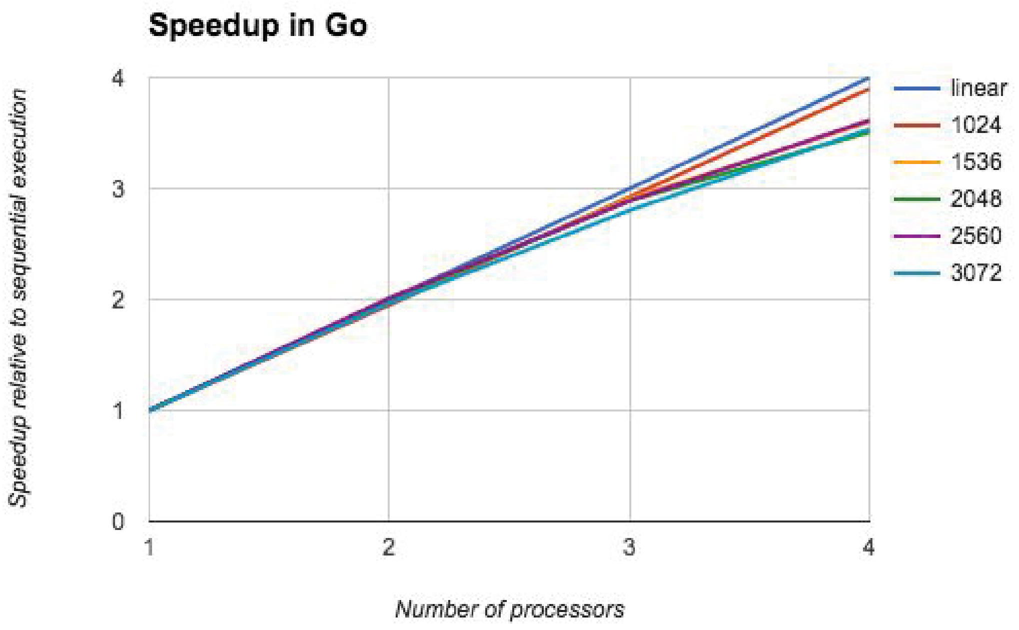

Figure 5-4: Parallel execution, relative to sequential execution, in Go

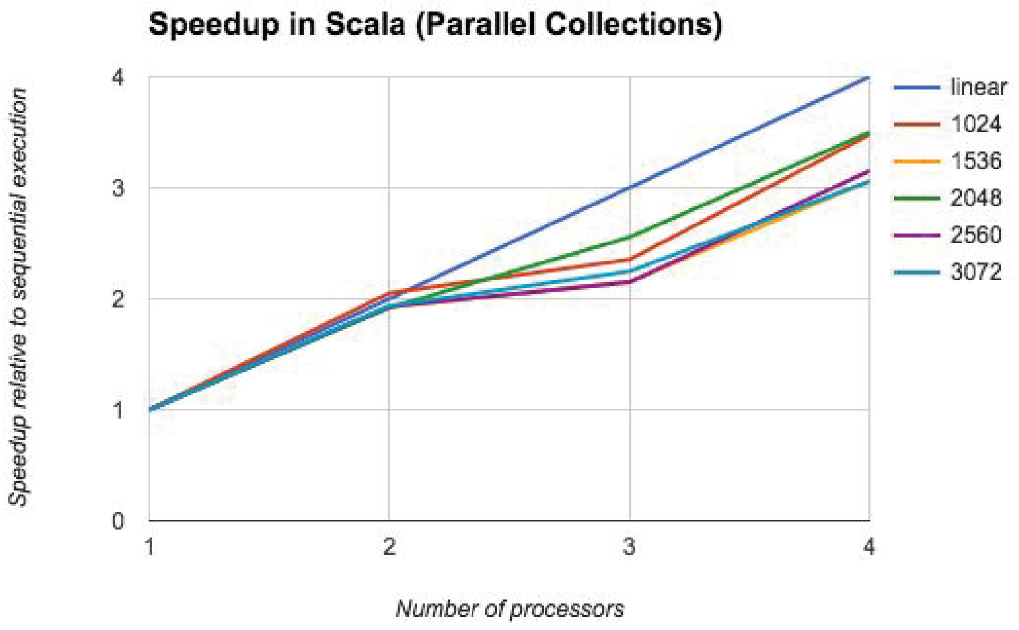

Figure 5-5: Parallel execution, relative to sequential execution, with parallel collections in Scala

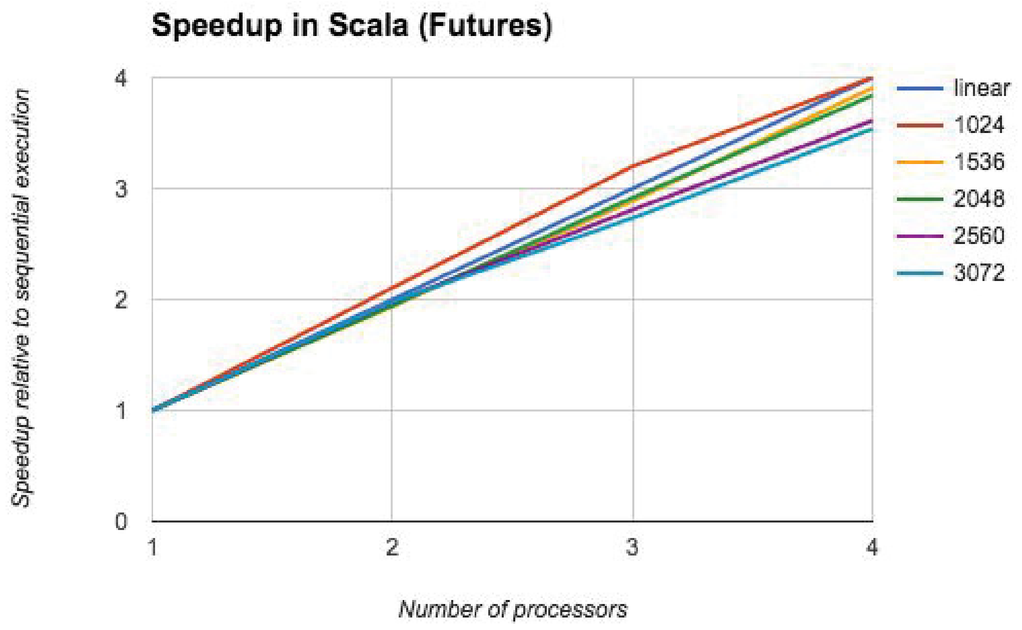

Figure 5-6: Parallel execution, relative to sequential execution, with futures in Scala

5.2.3 Matrix chain multiplication

Results from matrix chain multiplication. The horizontal axes in figure 5-7 and 5-8 are in log scale and it illustrates the number of goroutines and actors that were used. The number of processes on the horizontal axis are 2^0, 2^1, .., 2^{11}. The benchmarks were executed with 2048 matrices as test data (array with 2049 elements). With 2048 actors / goroutines, each actor or goroutine was assigned one row from the cost matrix to compute. The sequential version in Scala took 12.84 seconds to execute and 19.44 seconds in Go. The parallel version in Scala also performs better when the number of actors / goroutines are between 1 - 8, but Go performs better when the number of processes are between 16 - 2048. With 4 processors the best execution time in Go is 2.91 seconds using 64 goroutines, and 3.39 seconds in Scala using 128 actors.

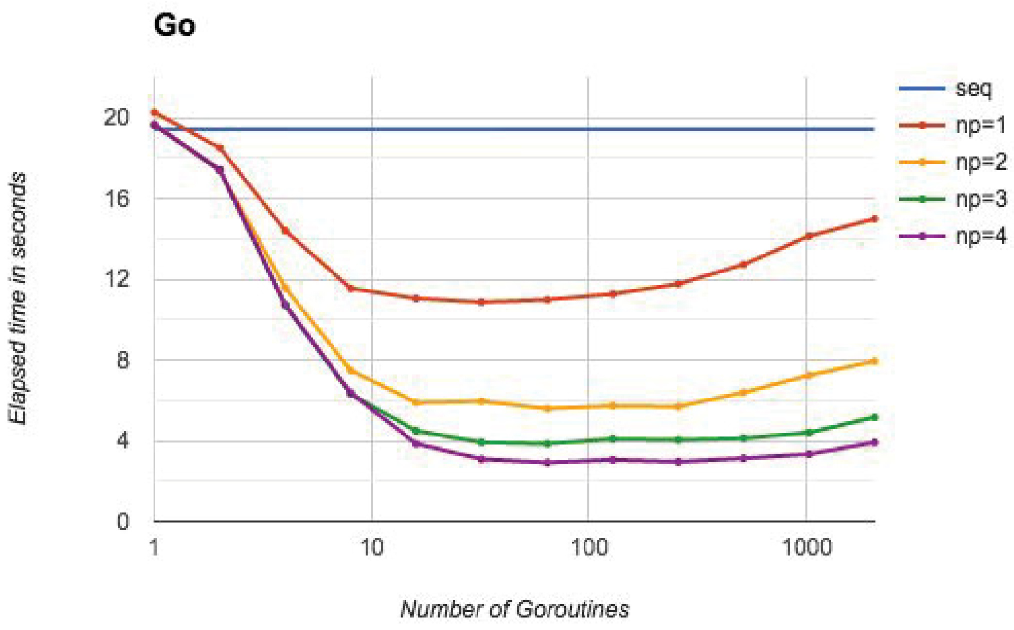

Figure 5-7: Matrix chain multiplication in Go

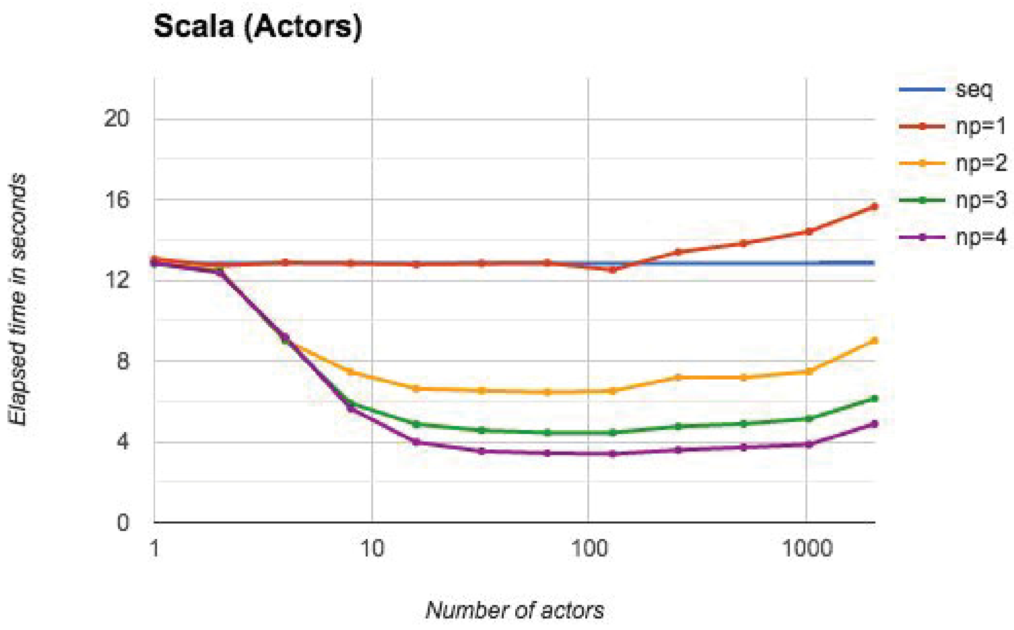

Figure 5-8: Matrix chain multiplication with actors in Scala

6 Analysis

Issues encountered during implementation of the benchmarks are explained in this chapter. The performance results are analysed in the context of the research questions. Absolute execution times of the sequential and parallel version of both algorithms are analysed, along with the relative speedup. Scalability when multiple processes requires synchronization and communication is also discussed. Finally, the results are compared against the performance studies that were presented in section 5.1.1, to see how well they conform.

6.1 Implementation issues

The main data structure for both algorithms is a two-dimensional (2D) array. 2D arrays can be represented as slice-of-slices in Go. A slice-of-slices can either be created by allocating each slice separately, or create a single array and then point the slices into this array [40]. The latter approach is usually faster because of improved cache efficiency. It's also possible to represent a 2D array as a regular slice of size $n \cdot n$. The entry for row *i*, column *j* is then found with the expression $i \cdot n + j$. Figure 6-1 outlines the three alternatives.

```go
//version 1: create each slice separately
matrix := make([][]int, n)
for i := range matrix {
  matrix[i] = make([]int, n)
}

//version 2: create slices from a single array
matrix := make([][]int, n)
buffer := make([]int, n*n)
for i := range matrix {
  matrix[i], buffer = buffer[:n], buffer[n:]
}

//version 3: regular slice
matrix := make([]int, n*n)
// set row i, column j to 0
matrix[i*n+j] = 0
```

Figure 6-1: Different ways of representing a matrix in Go

I tested both algorithms with all three versions. The execution time of version 1 and version 2 were similar. However, the performance of matrix multiplication improved substantially with version 3. Matrix chain multiplication had some improvements with version 3, but it wasn't as noticeable. Figure 6-2 shows the execution time of sequential matrix multiplication with version 2 and version 3. Version 2 is denoted as 2D and version 3 as 1D. Regular slices are 70 - 90 % faster than slice-of-slices, almost twice as fast.

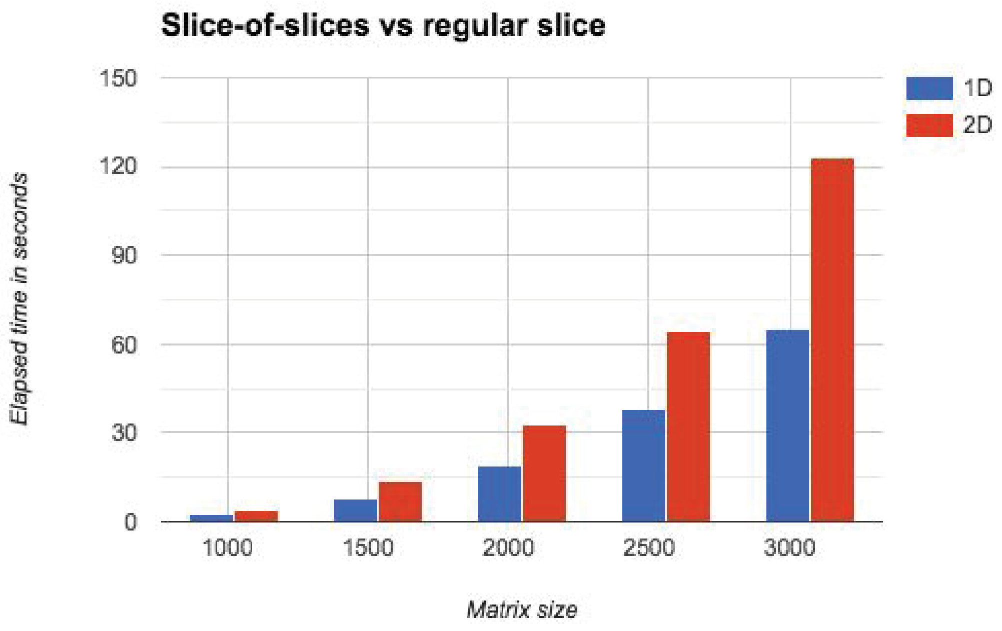

Figure 6-2: Sequential matrix multiplication with slice-of-slices (2D) and regular slice (1D)

The performance loss from slice-of-slices are caused by (1) twice as much bounds checking and (2) extra instructions required to handle the added level of indirection.

(1) Go applies bounds checking before writing or reading from a slice. The runtime in Go will detect if we try to write / read beyond the length of a slice. The program will crash if a bounds violation occurs. It’s possible to disable this feature, although it’s not recommended, by passing the -gcflags -B to the `$ go build` command. With this option enabled, version 3 is 50 - 60% faster than version 2, which is slightly less than before.

(2) By looking at the assembly output from the command `$ go tool 6g -S -B example.go`, bounds checking disabled, we see that slice-of-slices executes 36 instructions while the regular slice version executes 22 instructions. Version 2 executes 39% more instructions than version 1 does, which almost corresponds to the performance difference when bounds checking is disabled (1). All the added instructions in version 2 are of type `MOV`. Note

that these numbers are based on the instructions that computes the scalar multiplication, because this is the only thing that differs from the two versions.

Performance impact from slice-of-slices isn't very well documented, and it's definitely something that Go programmers should be aware of until better support for 2D arrays is added.

This wasn't an issue with the Scala implementation. The corresponding Scala versions of figure 6-1 had the same performance.

6.2 RQ3: General performance

Both sequential versions and the parallel version of matrix multiplication are faster in Scala. Sequential matrix multiplication was 2.9 - 3.3x faster and sequential chain multiplication was 1.5x faster. The results from matrix multiplication also shows that parallelism is faster in Scala. Parallel collections were 2.3 - 3.2x faster than goroutines and futures were 3 - 3.5x faster. The relative performance between goroutines and futures conform to the numbers from sequential execution. This suggests that the overhead from goroutines and futures is roughly the same.

The reason that parallel collections are slower than futures is because of overhead from the more advanced load-balancing technique used by the framework, as described in the paper A Generic Parallel Collection Framework [15]. Figure 6-3 illustrates the load balancing technique that futures and goroutines used, *n* is the number of matrix rows, *np* is the number of processors / goroutines / futures and *id* is the assigned id to the processor. This approach isn't as flexible as the technique used by the framework, but in this case it works pretty good.

```
splitWork(n, np, id):
  size = n / np
  start = id * size
  end = if (id+1) < np then start + size else n
  return start, end

for id = 0 .. np-1
  start, end = splitWork(n, np, id)
  doWork(start, end)
```

Figure 6-3: Pseudocode that manually creates equal sized tasks

However, creating a parallel version of matrix multiplication was easiest with parallel collections. Goroutines and futures required more effort because of manually creating equal sized tasks and assigning them to the goroutines / futures.

Futures achieved linear speedup with matrix size 1024 and goroutines were also close to reach it. The reason that parallel collections have worse speedup is mostly likely due to the previously stated reasons.

6.3 RQ4: Synchronization

The results from parallel chain multiplication aren't as clear cut as matrix multiplication. Scala was 0 - 35% faster than Go up until the 8 processes mark. However, Go was 0 - 20% faster than Scala from 16 - 2048 processes. Go was 19% faster than Scala with 2048 processes on 4 cores. Based on these numbers, it appears that goroutines have better scalability and the overhead from Akka actors is quite significant compared to goroutines and channels. This isn't too suprisring though, considering that goroutines and channels are part of the Go programming language and Akka is a third party framework. I also found it was easier to develop a parallel implementation in Go.

6.4 Previous studies

In section 5.1.1 I concluded that Go and Java had similar performance, while Scala was slightly faster than Java. Based on this observation, and the fact that Scala was faster than Go in Hundt's study [5], the results from the experiment appears to be inline with previous studies.

6.5 Limitations and validity threats

The biggest threat is the fact that I only had time to conduct the experiment on one system. It's possible that executing the benchmarks on Linux or Windows, or another set of hardware, will produce results that differ from what is presented in this thesis. It's also important to remember that other algorithms / programs could have better performance in Go. This study only shows that matrix multiplication and chain multiplication were faster in Scala.

7 Conclusions

In this thesis I compared the performance between Go and Scala with parallel implementations of matrix multiplication and matrix chain multiplication. Both implementations in Go were based on goroutines and channels. Matrix multiplication in Scala was implemented in two different versions. One version with parallel collections and the other with blocking futures. Matrix chain multiplication was implemented with Akka actors.

RQ1: What previous performance studies have been conducted regarding Go or Scala?

From the literature study I found 5 scientific papers and 8 theses that analyse some aspect of performance in Go or Scala. Hundt's paper [5] is the only study that compared Go and Scala in the same context. However, the paper is focused on sequential performance and the study was conducted before version 1 of Go had been released. None of the 13 papers have compared parallelism or concurrency between Go and Scala.

RQ2: How is concurrency and parallelism facilitated in Go and Scala?

Scala has several constructs that simplify concurrency and parallelism. The parallel collections framework provides an easy way to achieve parallelism. The framework should be used for data parallelism and when no synchronization / communication is needed. Futures support monadic methods, similar to collections, which makes it easy to create complex parallel computations. Futures can be applied to most situations, but actors are often the better choice if the tasks requires complicated synchronization / communication. The actor model is the most powerful mechanism for concurrency in Scala. The support for messaging in actors makes it easy to build complex interactions between processes. However, Akka actors are quite extensive and there are a wide range of different configurable options.

Go supports concurrency through the built-in goroutines and channels. Go takes a more general approach towards concurrency, unlike Scala which provides new mechanisms for different use cases.

RQ3: How does the performance between Go and Scala compare in a multi-core environment?

General performance in Scala is better than Go and parallel matrix multiplication is faster in Scala. Overhead from futures and goroutines is similar and both implementations were close to linear speedup. Parallel collections are slightly slower than futures and have worst speedup of the three versions. Parallel collections suffer some performance loss because of the load-balancing technique used by the framework. With 4 cores and matrix size 3070, the speedup is 3.54 with futures, 3.54 with goroutines and 3.06 with parallel collections.

RQ4: How is performance affected by synchronization and communication between multiple processes?
Goroutines have better scalability than actors. With 4 cores and 4 actors / goroutines, the execution time of parallel matrix chain multiplication in Scala is 9.18 seconds and 10.52 seconds in Go. With 2048 actors / goroutines the execution time in Go is 3.92 seconds and 4.88 seconds in Scala.

Contribution
Scala has better performance than Go, but Akka actors aren't as efficient as goroutines and channels. Sequential matrix multiplication is 2.9 - 3.3x faster in Scala and sequential chain multiplication is 1.5x faster in Scala. Parallel matrix multiplication is also faster in Scala, parallel collections performed 2.3 - 3.2x better and futures performed 3 - 3.5x better, compared to goroutines. Parallel chain multiplication is 0 - 35% faster with actors in Scala between 1 - 8 processes. However, goroutines are 0 - 20% faster than actors between 16 - 2048 processes.

Another, accidental, discovery was the performance problems with slice-of-slices in Go. Sequential matrix multiplication is almost 2x faster with regular slices compared to slice-of-slices. The performance differences are similar in the parallel version of matrix multiplication. The official Go documentation doesn't have any information about this in the section where they describe slices and two-dimensional slices [40].

Coming from a background in C-like languages, C++ and Java, and little previous experience in functional languages, I found that Go was easier to learn and understand than Scala. Scala is more complex than Go, more features and an intricate type system, which results in a higher learning curve. I would recommend anyone interested in Go to try it out because of its ease of use.

8 Future Work

This thesis focused on parallelism and performance, but that is only one aspect to consider. An empirical study that analyses the complexity of large concurrent systems in Go and Scala would be interesting. By complexity I mean how easy the system is to understand, learn and maintain. Evaluating performance of such a system would also be interesting since short programs, as in this thesis, do not resemble systems in production.

9 References

[1] TOGASHI, Naohiro; KLYUEV, Vitaly. Concurrency in Go and Java: Performance analysis. In: *4th IEEE International Conference on Information Science and Technology (ICIST)*. IEEE, 2014. p. 213-216.

[2] TANG, Peiyi. Multi-core parallel programming in go. In: *Proceedings of the First International Conference on Advanced Computing and Communications*, 2010. p. 64-69.

[3] SERFASS, Doug; TANG, Peiyi. Comparing parallel performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem. In: *Proceedings of the 50th Annual Southeast Regional Conference*. ACM, 2012. p. 268-273.

[4] PANKRATIUS, Victor; SCHMIDT, Felix; GARRETÓN, Gilda. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012. p. 123-133.

[5] HUNDT, Robert. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days*, 2011.

[6] CÁCERES, Edson Norberto; MONGELI, Henrique; LOUREIRO, Leonardo; NISHIBE, Christiane; WUN SONG, Siang. A parallel chain matrix product algorithm on the integrade grid. In: *International Conference on High Performance Computing, Grid and e-Science in Asia Pacific Region*, 2009. p. 304-311.

[7] ERIKSSON, Linus; NILSSON, Niclas. En utvärdering av Go, Erlang och F# ur ett concurrencyperspektiv, 2012. http://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/rapport/eriksson_linus_OCH_nilsson_niclas_K12021.pdf

[8] JONSSON, Yuuki; STARRSJÖ, Andreas. Parallellprogrammering i Go och Erlang, 2011. http://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/rapport/jonsson_yuuki_OCH_starrsjo_andreas_K11084.pdf

[9] STJERNBERG, Johan; ANNEBÄCK, Joakim. A comparison between Go and C++.
http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group9Alexander/Joakim_Anneback_Johan_Stjernberg.finalreport.2.pdf

[10] HJALMARSSON, Alexander; HUSS, Jakob. Empirisk undersökning av programspråket Go med avseende på snabb utveckling, 2011.
http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group5Lars/Alexander%20Hjalmarsson%20och%20Jakob%20Huss-go_dkand11.pdf

[11] STIMPFILING, Robert Derek. An evaluation of Go and Clojure, 2010.
http://scholar.colorado.edu/cgi/viewcontent.cgi?article=1034&context=csci_ugrad

[12] JÄRLEBERG, Anders; NILSSON, Kim. *Go, F# and Erlang*, 2012.
http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Anders_Jarleberg_Kim_Nilsson.report.pdf

[13] ÅBERG, Marcus; LINDEBERG, Johan. F# and Go compared to Java, 2012.
http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/Marcus_Aberg_Johan_Lindeberg.rapport.pdf

[14] Speedup. Date of reference: 2015-06-06.
http://en.wikipedia.org/wiki/Speedup

[15] PROKOPEC, Aleksandar; ROMPF, Tiark; BAGWELL, Phil; ODERSKY, Martin. A generic parallel collection framework. In: *Euro-Par 2011 Parallel Processing*. Springer Berlin Heidelberg, 2011. p. 136-147.

[16] CHEN, Nicholas; JOHNSON, Ralph. Patterns for cache optimizations on multi-processor machines. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010. p. 2.

[17] TOTOO, Prabhat; DELIGIANNIS, Pantazis; LOIDL, Hans-Wolfgang. Haskell vs. F# vs. Scala: a high-level language features and parallelism support comparison. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*. ACM, 2012. p. 49-60.

[18] KEELE, Staffs. *Guidelines for performing systematic literature reviews in software engineering*. EBSE Technical Report, 2007.

[19] LESANI, Mohsen; ODERSKY, Martin; GUERRAOUI, Rachid. *Concurrent Programming Paradigms, A Comparison in Scala*, 2009.

[20] CORBAT, Thomas. Inside Scala Actors, 2010.
http://sifsv-80011.ifs.hsr.ch/SemProgAnTr/files/Inside_Scala_Actors.pdf

[21] GOODMAN, Daniel; KHAN, Behram; KHAN, Selman; LUJÁN, Mikel; WATSON, Ian. Software transactional memories for Scala. *Journal of Parallel and Distributed Computing*, 2013, 73.2: 150-163.

[22] LEE, Edward A. The problem with threads. *Computer*, 2006, 39.5: 33-42.

[23] HUGHES, John. Why functional programming matters. *The computer journal*, 1989, 32.2: 98-107.

[24] ODERSKEY, Martin. What is Scala? Offical Scala website. Date of reference: 2015-05-26.
http://www.scala-lang.org/what-is-scala.html

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, 3rd edition. MIT Press, 2009.

[26] Chain Matrix Multiplication. Lecture slides. Date of reference: 2015-05-26.
https://www.cse.ust.hk/~dekai/271/notes/L12/L12.pdf

[27] Google Scholar. Date of reference: 2015-05-26.
https://scholar.google.se

[28] BTH Summon. Date of reference: 2015-05-26.
http://bth.summon.serialssolutions.com.miman.bib.bth.se

[29] Engineering Village. Date of reference: 2015-05-26.
http://www.engineeringvillage.com

[30] Official Go documentation. Date of reference: 2015-05-26.
https://golang.org/doc/install/source

[31] Offical Go documentation. Date of reference: 2015-05-26.
https://golang.org/cmd/gc/

[32] Java HotSpot VM Options. Official Oracle documentation. Date of reference: 2015-05-26.
http://www.oracle.com/technetwork/java/javase/community/vmoptions-jsp-140102.html

[33] COX, Russ; MA, Shenghou. Profiling Go Programs. Official Go blog, 2011 & 2013. Date of reference: 2015-05-26.
http://blog.golang.org/profiling-go-programs

[34] JOVANOVIC, Vojin; HALLER, Philipp. The Scala Actors Migration Guide. Official Scala documentation. Date of reference: 2015-05-26.
http://docs.scala-lang.org/overviews/core/actors-migration-guide.html

[35] HALLER, Philipp; PROKOPEC, Aleksandar; MILLER, Heather; KLANG, Viktor; KUHN, Roland; JOVANOVIC, Vojin. Futures and Promises. Official Scala documentation. Date of reference: 2015-05-26.
http://docs.scala-lang.org/overviews/core/futures.html

[36] SCHINZ, Michel; HALLER, Philipp. A Scala Tutorial for Java Programmers. Official Scala documentation, 2014. Date of reference: 2015-06-15.
http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

[37] Official Akka documentation. Date of reference: 2015-05-26.
http://doc.akka.io/docs/akka/2.3.11/scala.html

[38] AIMONETTI, Matt. Go Bootcamp, 2014. Date of reference: 2015-05-26.
http://www.golangbootcamp.com/book/concurrency

[39] Official Go documentation: Date of reference: 2015-05-26.
https://golang.org/doc/effective_go.html#concurrency

[40] Official Go documentation: Date of reference: 2015-05-26.
https://golang.org/doc/effective_go.html#slices

[41] Moore's law. Date of reference: 2015-05-26.
http://www.mooreslaw.org/

[42] SUTTER, Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. In: Dr. Dobb's Journal, 2005. Date of reference: 2015-05-26.
https://www.cs.utexas.edu/~lin/cs380p/Free_Lunch.pdf

[43] PIKE, Rob. Another Go at Language Design, 2010. Date of reference: 2015-05-26.

http://web.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf

[44] Paralleism vs Concurrency. Haskell wiki. Date of reference: 2015-05-27.
https://wiki.haskell.org/Parallelism_vs._Concurrency

[45] GERRAND, Andrew. Concurrency is not paralleism. Official Go blog, 2013. Date of reference: 2015-05-27.
http://blog.golang.org/concurrency-is-not-parallelism

[46] AKHTER, Shareem; ROBERTS, Jason. Avoiding Classic Threading Problems. In: Dr. Dobb's Journal, 2011. Date of reference: 2015-06-17.
http://www.drdobbs.com/tools/avoiding-classic-threading-problems/231000499

Appendix A Go source code

Matrix multiplication

```go
func seqMatrixMult (m1, m2, res []int, n int) []int {
  m1Rows := len(m1) / n
  for i := 0; i < m1Rows; i++ {
    for k := 0; k < n; k++ {
      for j := 0; j < n; j++ {
          res[n*i+j] += m1[i*n+k] * m2[k*n+j]
        }
      }
  }
  return res
}

func parMatrixMult(vp, n int, m1, m2, res []int) []int {
  chunkSize := n / vp
  start := 0
  end := 0
  ch := make(chan int, vp)

  for i := 0; i < vp; i++ {
    start = end
    end += chunkSize
    if (i + 1) == vp {
      end = n
    }

    startPos := start * n
    endPos := end * n
    m1Chunk := m1[startPos:endPos]
    resChunk := res[startPos:endPos]

    go func(m1, m2, res []int, n int, ch chan int) {
      seqMatrixMult(m1, m2, res, n)
      ch <- 1
    }(m1Chunk, m2, resChunk, n, ch)
  }

  for i := 0; i < vp; i++ {
    <-ch
  }
  return res
}
```

Matrix chain multiplication

```go
func computeCost(cost, chain []int, n, i, j int) {
  cost[i*n+j] = math.MaxInt32

  if i == j {
    cost[i*n+j] = 0
  } else {
    for k := i; k <= (j - 1); k++ {
      q := cost[i*n+k]+cost[(k+1)*n+j]+chain[i-1]*chain[k]*chain[j]
      idx := i*n + j
      if q < cost[idx] {
        cost[idx] = q
      }
    }
  }
}

func seqChainMult(cost, chain []int, n int) {
  for i := n; i >= 1; i-- {
    for j := i; j <= n; j++ {
      computeCost(cost, chain, n+1, i, j)
    }
  }
}

func getBounds(n, vp, id int) (int, int) {
  size := n / vp
  low := id * size
  high := low + size

  if (id + 1) == vp {
    high = n
  }
  return low + 1, high
}

func computeBlock(cost, chain []int, n, i, j, il, ih, jl, jh int) {
  if i == j {
    for ii := ih; ii >= il; ii-- {
      for jj := ii; jj <= jh; jj++ {
        computeCost(cost, chain, n, ii, jj)
      }
    }
  } else {
    for ii := ih; ii >= il; ii-- {
      for jj := jl; jj <= jh; jj++ {
        computeCost(cost, chain, n, ii, jj)
      }
    }
  }
}

func partition(rounds []chan int, finish chan int, cost, chain []int, i, n, vp int) {
  j := i
  il, ih := getBounds(n, vp, i)

  for ; j < vp; j++ {
    jl, jh := getBounds(n, vp, j)
    computeBlock(cost, chain, n+1, i, j, il, ih, jl, jh)

    if i > 0 {
      rounds[i-1] <- 1
    }
    if j+1 < vp {
      <-rounds[i]
    }
  }

  if il == 1 {
    finish <- 1
  }
}

func parChainMult(rounds []chan int, finish chan int, cost, chain []int, vp, n int) {
  for i := 0; i < vp; i++ {
    go partition(rounds, finish, cost, chain, i, n, vp)
  }
  <-finish
}
```

Appendix B Scala source code

Matrix multiplication

```
// Sequential matrix multiplication
def seqMatrixMult(start: Int, end: Int, m1: Array[Array[Int]], m2: Array[Array[Int]],
res: Array[Array[Int]]) = {
  val n = m1.length
  var i = start
  var j, k = 0

  while (i < end) {
    while (k < n) {
      while (j < n) {
        res(i)(j) += m1(i)(k) * m2(k)(j)
        j += 1
      }
      j = 0
      k += 1
    }
    k = 0
    i += 1
  }

  m3
}

// Parallel collections
def parMatrixMult(m1: Array[Array[Int]], m2: Array[Array[Int]], res:
Array[Array[Int]]) = {
  val n = m1.length

  (0 until n*n) par foreach(x => {
    val i = x / n
    val k = x % n
    var j = 0
    while(j < n) {
      res(i)(j) += m1(i)(k) * m2(k)(j)
      j += 1
    }
  })

  res
}

// Futures
def futureMatrixMult(vp: Int, m1: Array[Array[Int]], m2: Array[Array[Int]], res:
Array[Array[Int]]) = {
  val chunkSize = m1.length / vp

  val chunks = 0 until vp map {
    x => {
      val start = x*chunkSize
      val end = if ((x+1) == np) m1.length else (x+1) * chunkSize
      (start, end)
    }
  }

  val mult = Future.sequence(
    chunks map {x => Future {seqMatrixMult(x._1, x._2,  m1, m2, res)} }
  )

  Await.result(mult, Inf)
  res
}
```

Matrix chain multiplication

```
object ChainMult {
  def computeCost(cost: Array[Array[Int]], chain: Array[Int], i: Int, j: Int) {
    cost(i)(j) = Int.MaxValue

    if (i == j) {
      cost(i)(j) = 0
    } else {
      var k = i
      while(k < j) {
        val q = cost(i)(k) + cost(k+1)(j) + chain(i-1) * chain(k) * chain(j)
        if (q < cost(i)(j)) {
          cost(i)(j) = q
        }
        k += 1
      }
    }
  }

  def seqChainMult(cost: Array[Array[Int]], chain: Array[Int], n: Int) = {
    var i = n
    var j = i

    while (i >= 1) {
      while(j <= n) {
        computeCost(cost, chain, i, j)
        j += 1
      }
      i -= 1
      j = i
    }
    cost(1)(n)
  }
}

case class CalculateCost()
case class ComputationFinished(res: Int)
case class InitWorker(id: Int)
case class NotifyWorker(id: Int)
case class Work()

class Worker(chain: Array[Int], cost: Array[Array[Int]], n: Int, vp: Int) extends
Actor {
  var i: Int = _
  var j: Int = _
  var iBounds: Tuple2[Int, Int] = _

  def getBounds(id: Int): (Int, Int) = {
    val size = n / vp
    var low = id * size
    var high = low + size

    if (id+1 == vp) {
      high = n
    }

    (low+1, high)
  }

  def computeBlock() {
    val jBounds = getBounds(j)
    var ii = iBounds._2
    var jj = 0

    if (i == j) {
      while(ii >= iBounds._1) {
        jj = ii
        while (jj <= jBounds._2) {
          cost(ii)(jj) = i
          ChainSeq.computeCost(cost, chain, ii, jj)
          jj += 1
        }
        ii -= 1
      }
    } else {
      while (ii >= iBounds._1) {
        jj = jBounds._1
        while(jj <= jBounds._2) {
          ChainSeq.computeCost(cost, chain, ii, jj)
          jj += 1
        }
        ii -= 1
      }
    }

    j += 1
  }

  def sendResponse() {
    if (i > 0) {
      sender ! NotifyWorker(i-1)
    } else if (i == 0 && j == vp) {
      sender ! ComputationFinished(cost(1)(n))
    }

    if (j == vp) {
      context.stop(self)
    }
  }

  def receive = {
    case InitWorker(id) => {
      i = id
      j = i
      iBounds = getBounds(i)
      computeBlock()
      sendResponse()
    }
    case Work() => {
      computeBlock()
      sendResponse()
    }
  }
}

class Master(chain: Array[Int], cost: Array[Array[Int]], size: Int, vp: Int) extends
Actor {
  val workers = Vector.fill(vp) {
    context.actorOf(Props(classOf[Worker], chain, cost, size, vp))
  }

  def receive = {
    case CalculateCost => {
      for (i <- 0 until vp) {
        workers(i) ! InitWorker(i)
      }
    }
    case NotifyWorker(id) => {
      workers(id) ! Work()
    }
    case ComputationFinished(res) => {
      sender ! res
      context.stop(self)
    }
  }
}
```

Appendix C Java Virtual Machine settings

JVM	HotSpot 64-Bit Server
AllowUserSignalHandlers	false
DisableExplicitGC	false
FailOverToOldVerifier	true
MaxFDLimit	true
RelaxAccessControlCheck	false
ScavengeBeforeFullGC	true
UseAltSigs	false
UseConcMarkSweepGC	false
UseGCOverheadLimit	true
UseLWPSynchronization	true
UseParallelGC	true
UseParallelOldGC	true
UseSerialGC	false
UseTLAB	true
UseSplitVerifier	true
UseThreadPriorities	true
UseVMInterruptibleIO	false
UseG1GC	false
MaxGCPauseMillis	18446744073709551615
InitiatingHeapOccupancyPercent	45
NewRatio	2
SurvivorRatio	8
MaxTenuringThreshold	15
ParallelGCThreads	8
ConcGCThreads	0
G1ReservePercent	10
G1HeapRegionSize	0

AggressiveOpts	false
CompileThreshold	10000
LargePageSizeInBytes	0
MaxHeapFreeRatio	100
-XX:MaxNewSize=size	18446744073709486080 bytes
MaxPermSize	85983232 bytes
MinHeapFreeRatio	0
NewSize	1310720 bytes
ReservedCodeCacheSize	50331648 bytes
TargetSurvivorRatio	50
ThreadStackSize	1024
UseBiasedLocking	true
UseFastAccessorMethods	false
UseLargePages	false
UseStringCache	false
AllocatePrefetchLines	4
AllocatePrefetchStyle=1	1
OptimizeStringConcat	true