Performance and Usability Evaluation of Concurrency in Modern Functional Programming Languages

dpt908e14, Autumn 2014, Aalborg University

13-01-2014

The Faculty of Engineering and Science
Computer Science 9th term
Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst
Phone no.: 99 40 99 40
Fax no.: 99 40 97 98
Homepage: http://www.cs.aau.dk

Project title:
Performance and Usability Evaluation of Concurrency in Modern Functional Programming Languages

Subject:
Evaluation of Functional Programming Languages

Project periode:
Autumn 2014

Group name:
dpt908e14

Supervisor:
Lone Leth Thomsen

Group members:
Daniel Rune Jensen
Søren Kejser Jensen
Thomas Stig Jacobsen

Copies: 2

Pages: 104

Appendices: 1

Finished: 13-01-2015

Abstract:

This report evaluates performance and usability of advanced concurrency models in functional programming languages which we have defined as models of higher abstraction than the common Thread model. Clojure, Erlang, F#, Haskell and Scala were chosen based on three evaluation criteria. The theoretical concurrency models were each evaluated using four characteristics. K-Means clustering were implemented to benchmark these languages execution times as a means to calculate their speed-up. A usability evaluation was done by implementing the Santa Claus problem in each language and evaluating on development time and lines of code in addition to a subjective discussion of the languages' implementation of the theoretical model. Based on these evaluations and metrics we end with a discussion of the choices made and results achieved and a conclusion on the state of concurrency models in functional programming languages.

Preface

This report is the result of a three person computer science semester project on the 9^{th} semester. The report is self-contained while serving as the preceding work for the authors master thesis. The source code of the program developed doing the project is available for download at the following URL: `https://github.com/eXeDK/dpt908e14`.

Citation numbers in square brackets are used throughout the rapport as references to existing work. A bibliography at the end of the rapport contains a detailed description of these sources and information on how to obtain them. Two styles of citation are used in this rapport. The first style is when a citation is before a punctuation mark which means the citation is only associated with the current line. The second style is when a citation is after a punctuation mark at the end of a paragraph then it is associated with the entire paragraph.

Source code examples are presented throughout the rapport and may have been formatted differently in order to fit page width and therefore code examples might include indention or other syntax which is not preferable from a readability standpoint. All code references and keywords in this project are written with `teletypefont`.

Contents

CHAPTER 1

Introduction

This chapter introduces the needed background information for the project and argues the needed for a performance and usability evaluation of concurrency in functional programming languages. First incentive for the report is found in Section 1.1, followed by an introduction to the concept of functional programming in Section 1.2. A description of various concurrency models is given in Section 1.3, with related work documented in Section 1.4. Finally the problem statement is formulated providing the direction for the rest of the report in Section 1.5.

1.1 Incentive

Computers Central Processing Units (CPU) have evolved over the course over several years to become smaller and faster, but when research was not able to go further in that direction as easy any more, multi-core CPUs were introduced. A multi-core CPU is designed to enable program threads to execute in parallel. The problem is that the introduction of multi-core CPUs does not increase the execution speed of non-concurrent programs. Programs need to be designed with concurrency in mind to fully utilise the potential power of a multi-core CPU. In order to design concurrent programs, developers must be provided with programming languages which support concurrency and a large number of languages does so already. [1, p. 1]

In addition with the aforementioned stagnation in the increase of clock cycles for a single CPU core, an increase in velocity, volume and variety of data being produced today [2] created a higher need for concurrency models that are simple to use for inexperienced developers that allows for high utilisation of the available CPU cores. The introduction of devices like smartphones have also created a need for languages to support concurrency and parallelism on devices where any additional overhead reduces battery time and reduces the limited set of resources available for other applications.

Many programming languages today only support a basic threading model as support for concurrency and parallelism, leading to complicated systems that are hard to debug and whose correctness is difficult to verify. The

problem is further increased as most programming languages in use today advocate an imperative object-oriented style of programming [3], where objects encapsulates some form of state, this state is then manipulated through the object's public methods. As methods on a object can change the state of the object, and not just manipulate variables declared directly in the method any changes made to a particular method needs to not only ensure the correct returned value, but also that any changes to the objects internal state are done correctly. This makes refactoring and extension of programs an expensive endeavor as the developer not only needs to understand the method itself, but also how it interacts with other methods on the object. If the changes made to how internal state is manipulated it might also be necessary for to objects inherit these changes from the object, leading to an even more complex task.

Alternatively, languages advocating a functional style of programming have seen an increase of usage in recent years, with new languages being compatible with existing platforms backed by commercial support by known vendors such as F# and Scala. In addition to an increased interest in existing functional programming languages such as Erlang and Haskell, making a functional approach to programming with a minimal number of side-effects and abstraction through stateless functions available for all platforms, allowing compatibility with existing codebases.

Multiple models of abstraction have been built on top of the existing Thread model to make use of threads easier by decreasing the required knowledge and experience to make use of it properly. Comparison of these new constructs in the context of functional programming languages have been neglected with existing work being focused on introducing new abstractions or focusing on the performance of the model and not the necessary knowledge needed for using the language construct. This is making the knowledge needed for choosing a suitable functional programming language hard to acquire.

1.2 Functional Programming

Functional programming provides a different paradigm for programming computer systems compared to the more widespread combination of the imperative and object oriented programming paradigms [3]. Higher-order-functions and the absence of assignments are the biggest differences compared to imperative programming. This makes functional programming seem like a less expressive paradigm, as no assignment statements are available, meaning that once a variable is given a value it never changes, for example removing the possibility for a function changing the state of a program as a side-effect. This lack of side-effects makes it easier to understand the effect a function has on the state of a program since the returned data is the only

product of the function.

This leads to referential transparency that allows direct substitution of functions, as long as the signature (as well as the data returned) of the function remains the same. Functional programming can in some circumstances produce programs that are magnitudes shorter than their structured programming counterparts, as well as producing highly modular programs that ease reuse and maintenance [4]. An example of functional programming producing a significantly sorter implementation compared to for example an imperative language is an implementation of Quicksort in Haskell compared C which can be found in [5].

Programming functionally is possible in most languages with ranging enforced strictness from the language itself. From the strictness of purely functional languages like Haskell, mostly functional languages like Clojure, F# and Scala, imperative languages with great functional programming support as C# and Python to imperative languages with minimal functional programming support like C and Java. To what degree the functional programming paradigm should be embraced to receive these benefits is however still up for debate, with some arguing that simply reducing the number of side-effects are without effect [6].

Additionally the libraries a given programming language provides are often an essential part of the reason for choosing a specific language. Most popular imperative and object-oriented languages provide larger libraries than those present in most functional programming languages even though a considerable effort has been made it getting them up on par. Some languages like F# are able to utilise the same libraries as other programming languages, for example the way F# can use the libraries from other .NET based languages like C# [7].

Furthermore, many functional programming languages provide inadequate tooling compared to what is available for object-oriented imperative languages, however corporations have started developing more advanced tools, for example Microsoft and Typesafe who are actively developing tooling for F# and Scala respectively. Software engineering methodologies for functional programming languages have also had less focus, forcing developers to not only rethink how to structure programs, but at the same time leaving them with a smaller community of developers to draw experience from [8].

Concepts of functional programming are used in frameworks for data processing such as Google's MapReduce, where a map and a reduce function is created by the user while the framework distributes data and code to perform the operation on a cluster [9]. Functional constructs like higher-order-function are also added to object-oriented imperative programming languages such as C# and Java [10, 11]. Overall current tendencies indicate a larger use of functional programming languages in the foreseeable future [3], leading to the necessity of simple to use high performance constructs for

parallel computing, due to the widespread availability of multi-core processors in computers today and the need for a higher throughput to handle for example big data processing.

1.3 Concurrency Models

In the following section we will give a short explanation of main concurrency models in use today. A concurrency model is in this context a method allowing multiple threads of execution in a single program, allowing the program to execute concurrently on single core systems and enabling parallelism when multiple cores are available. Being able to scale up the number of cores seamlessly also calls for the model being able to scale down the number of cores seamlessly without an unexpected decrease in performance due to the decrease in the number of execution threads. The majority of the models provide a means for task parallelism, allowing multiple independent threads of execution to exist in the program at the same time. Parallel Collections however allows for the use of data parallelism, running the same operation on multiple data elements in parallel.

1.3.1 Coroutines

Coroutines are extended subroutines that allow for multiple suspending and entry points into the routine. This allows for a routine to suspend and resume doing the execution of a subroutine in multiple places, used for example in generators using `yield`. In this way a coroutine is able to communicate with other coroutines by returning part of a computation instead of waiting for it to be finished. There is not one master program in the use of coroutines, instead all coroutines act as if they were the master program. [12]

1.3.2 Shared Memory Concurrency using threads

Using threads and shared memory is the most known of the concurrency models used today, with a large number of popular programming languages implementing the model, with popularity based on the Tiobe index [3]. It enables the developer to communicate between multiple threads using their shared memory. Using shared memory for concurrency however, often leads to problems with either race conditions or deadlocks. Manuel management of threads, use of mutexes, locks etc. has proven difficult for developers to do correctly. This difficulty leads to the problems with race conditions and deadlocks mentioned. Concurrency using threads is available in a wide variety of languages like C, C#, Java and even languages like PHP which is simply using wrappers of the `pthreads` functionality from C. [13, 14]

1.3.3 Actors

Actors are similar to the objects in object-oriented programming with both encapsulates operation and data. The actor model is inherently asynchronous due to its use of asynchronous message passing for communication, as opposed to the synchronous methods calling used in most object-oriented languages. This combination of asynchronous message passing and encapsulation, allows each actor to operate concurrently without regard for other actors in the system, as no memory is shared and all data are received through messages. [15]

1.3.4 Process Calculus Based Models

Multiple concurrency models have been created from formal process calculus such as Communicating Sequential Processes (CSP) [16], the Join-calculus [17], and the π-calculus [18]. These calculi allow for a formal definition of parallel processes and communication between them. Communication is for most calculi modelled as channels, allowing two processes to communicate through a set of matching channels. The use of channels allows for both synchronous and asynchronous communication, in contrast to the actor model where message passing make communication asynchronous. The programming languages Facile and Concurrent ML already in 1989 and 1991 respectively supported the use of the Calculus of Communicating Systems (CCS) as a native concurrency model [19, 20]. However few mainstream languages, based on the Tiobe index [3], have a calculus as a base for its concurrency model, but libraries are being developed, such as the CSP based PyCSP library for Python.

1.3.5 Software Transactional Memory

As an attempt to simplify concurrent programming transactional memory were introduced. However due to the lack of hardware supported transactional memory, Software Transactional Memory (STM) is available [21]. The idea of transactional memory is in general the same as with transactions in databases, allowing a section to be executed atomically prevents executions of the same section of code to interfere with each other. Being able to define a set of operations which are ensured to be executed together or not at all makes isolating critical sections of a program very simple from a developer's perspective.

1.3.6 Asynchronous Partitioned Global Address Space

The Partitioned Global Address Space (PGAS) model is designed to merge the performance of data locality with a simple way of referring to shared-memory. They do this by providing a local-view programming style that

differentiates between local and remote data partitions. Furthermore they give access to a global address space that is directly accessible by any process and the compiler handles the communication of the remote references. The Asynchronous Partitioned Global Address Space (APGAS) model extends the PGAS model by providing a richer execution framework. The added richness is in the form of *async*. Places are a collection of data together with the threads that operate on that data. Async is the ability to launch a statement as a thread to run in a specific place and stay there until termination. The model is currently being used through the programming language X10. [22]

1.3.7 Parallel Collections

Parallel collections allows operations on collections to be performed in parallel by dividing a collection into multiple parts and running a transformation on each part at the same time. Examples of programming language including supporting parallel collection include Scala and F#, providing a familiar high-level abstraction for processing the languages built-in collections. The idea behind parallel collection is simple as it allows the developer to swap out normal sequential collections for ones that are operated on in parallel. This idea is viable as collections are a well understood and frequently used programming abstraction, however it the model is also more restrictive then the alternatives as only transformations on collection and not just arbitrary statements can be run in parallel. [23, 24]

1.4 Related Work

The authors of [25] compared different programming languages that should ease the process of making programs for a single chip containing multiple processing cores. The tested four languages promised an improved development experience over traditional multi-threaded programming. The tests were based on programs all written by an experienced developer and addition expert reviews of the code. The resulting pool of 96 implementations are used to compare the languages with respect to lines of code, implementation time, execution time and speed-up. Their experiments showed both strengths and weaknesses in all approaches. They concluded their paper by remarking that "there is a need for benchmark to evaluate languages with respect to more general concurrency patterns" because of one of their test examples acted differently based on the used concurrency model. A fully general benchmark of concurrency models should at least be represented in some programming languages from all known programming paradigms to give a proper overview of the overall effects of a given concurrency model compared to a programming paradigm, if such parallels can be drawn at all.

We propose to contribute to this large problem by focusing on concurrency models in modern functional programming languages.

Another paper [26] made a performance and programmability comparison of high-level parallel programming support in three functional programming languages. They focused on high-level constructs to handle parallelism in a pure functional paradigm and two impure functional multi-paradigm languages. They looked at the impact of these languages with respect to program development and performance (execution time) by solving the same problem. They compared their parallel solutions with a sequential solution to get speed-up. They found that languages that use a high level of abstraction is able to produce good performance as well. However additional concurrency models have been implemented in the languages evaluated since the release of the paper, and few comments on the development process and the amount of effort needed to use each concurrency model from a development point of view was given.

Similarly in [27] multiple languages with different programming paradigms were compared to each other in terms of four criteria; implicit or explicit concurrency, fault restricted or expressive model, pessimistic or optimistic model and automatic or manual parallelisation. They compared Java, Cω (a research programming language from Microsoft Research), Fortress and E in regards to these four criteria and concludes on the languages' concurrency abstraction level and its concurrency model. The work finds that a concurrency model should be allowed to grow and that the need for a binding between concurrency and object-orientation is needed. In this paper the Santa Claus Problem [28] is used as a basis of comparison of the concurrency constructs in the chosen languages.

In the paper [29] another problem of synchronisation is implemented in Java using three different concurrency models namely the Actor model, transactional memory and using threads and locks. A sequential implementation of the problem was also completed. The Actor, transactional and sequential implementations were then compared to the one using threads and locks for the aspects of development time, lines of code, execution time and ease of use. All of the implementations were done by seventeen (17) undergraduate and graduate students.

1.5 Problem Statement and Scope

The main purpose of this project is to evaluate the concurrency models found natively in functional programming languages, with respect to both performance and their usability. Our primary focus will be on the performance capabilities of each model, however the model should also simplify parallel programming compared to using threads and locks. Each chosen concurrency model will be evaluated using a set of functional languages in order

to compare both the performance characteristics of each model given multiple different implementations, in addition to a brief evaluation of how each implementation of a given model simplifies the creation of parallel programs compared to threads and locks. We have scoped the report to not include distributed systems but focused on concurrency models for multi-core CPUs.

To summarise, the goal of the project consists of the following:

- Evaluate multiple concurrency models from a theoretical, performance and an usability perspective to give a richer understanding of concurrency models in functional languages.

1.6 Noteworthy remark

In this paper the authors are going to evaluate functional programming languages through implementations they developed themselves to test for both performance and usability. This makes our prior experience with functional programming language relevant because it effects the developed implementation in ways relating to performance and usability, e.g. efficiency, scalability, readability and development time. The authors had some prior knowledge by working with the functional programming languages Haskell, Scala and Scheme in addition to superficial knowledge of Erlang and F#. To make up for this lack of expertise the programming languages community standards of coding style and performance guidelines have been followed but also appropriate development tools such as IDEs and linting software have been used under development.

CHAPTER 2

Choice of Programming Languages

The following chapter details the process of selecting which functional programming languages we will use to evaluate the practical application of the theoretical concurrency models, in addition to a short introduction to each of the chosen languages. First we list our criteria used for the selection of the programming languages in section 2.1, followed by Section 2.2 which gives a short introduction of each language and why it was chosen based on the criteria given in section 2.1.

2.1 Language Criteria

The following criteria is used to select the programming languages we will be using in this report. These criteria have been chosen to restrict the number of languages and concurrency models to match the time frame of this project and to focus our report on functional programming languages used today and currently in active development. Each criteria are accompanied with a more in-depth explanation of the criterion itself and why we decided it was necessary.

2.1.1 Primarily a functional language

As an existing study of concurrency and parallelism in an object-oriented, imperative context already exists [27] and as support for functional programming languages is becoming more widespread, see section 1.2, we have chosen to restrict this study to functional programming languages both with and without support for object-oriented programming. To determine if a language is primarily a functional language and not just supporting constructs from the functional programming paradigm, we will look into the inclusion and ease of use of constructs such as higher-order functions, the availability of currying and lazy evaluation. If the language is not purely functional we will evaluate how well the language prevents the developer from pro-

gramming imperatively, such as providing persistent data structures as the default and discourage use of mutable variables through its type system or other mechanisms.

2.1.2 Actively in use and under development

The language should be actively in use, as we wish to evaluate languages that are mature enough for programming real world applications and not languages designed exclusively as an academic experiment. We define a language as actively in use, if the language is present in the top 100 of the Tiobe Index [3]. The language should also be in active development as new methods and ideas for both language design and implementation is constantly developed. This is to restrict our evaluation to concurrency models and languages developed with the most advanced implementation techniques available and following what is at the time of writing known as best practices in terms of language design. Thereby providing information about weakness in concurrency models for functional programming language, which hopefully can be used for further research in the area of both language design and implementation. We consider a language under active development if its reference implementation was updated in the year 2014.

2.1.3 Supports advanced concurrency models natively

We will purposely only analyse and evaluate concurrency models directly available in these languages without the requirement of additional libraries. Using only concurrency models supplied in the languages ensures that the developers have had access to the language implementation, meaning any optimisations specific for the concurrency models could have been implemented. Furthermore, the languages' native concurrency models are able to be deeper integrated in the language and its constructs. By "advanced concurrency model" we require the concurrency model to provide the developer with abstractions for concurrency control that is not just the use of threads and locks.

2.2 Chosen languages

From the Tiobe Index twelve (12) functional languages were found to still be under active development, these were: Clojure, Common Lisp, Emacs Lisp, Erlang, Factor, F#, Haskell, OCaml, Scala, Scheme, Standard ML and X10. These languages will be evaluated in terms of the criteria presented in Section 2.1. The languages fulfill the criterion of being actively in use as they are present in the Tiobe Index [3].

Common Lisp, Emacs Lisp, Factor, Standard ML and OCaml have all excluded for further study since they only support threads natively which

is not considered an advanced concurrency model. They do have more advanced concurrency models developed by the community as libraries.

X10 supports constructs from the functional paradigm such as higher-order functions and closures. However most X10 programs still needs the use of mutable data. This means that X10 contains some support for functional programming but it is not primarily functional and is therefore excluded from further study. [30]

Finally we have disqualified Scheme from this report based on the wide variety of dialects and implementations of Scheme which have different feature sets [31]. Scheme is standardised in the official IEEE standard "Revised^n Report on the Algorithmic Language Scheme (RnRS)" where the current standard is the R6RS [32] standard. R6RS does not specify any concurrency models which should be implemented in Scheme and is therefore excluded from further study.

This leaves us with the five languages Clojure, F#, Erlang, Haskell and Scala, that all satisfy the criteria mentioned in Section 2.1 by being primarily functional languages as well as supporting advanced concurrency models natively. They fulfill the criterion of being under active development by having had a new release of the language's reference implementation in the year 2014. The mentioned languages will be subject of further study in the following chapters.

2.2.1 Clojure

Clojure is a Lisp style, dynamically strong typed, functional programming language that primarily targets the Java Virtual Machine (JVM), with ports to both .NET and JavaScript. The language was created by Rich Hickey in 2007 and was designed to combine the dynamic nature of scripting languages with a mature and efficient runtime infrastructure necessary for multi-threaded programming. As Clojure is a functional programming language, it first and foremost discourages the use of mutable state, and provide access to a set of immutable data structures. [33]

Support for object-oriented programming has been added to Clojure with the primary abstractions being Records and Protocols, where Protocols defines a signature and Records provides the implementation for the given signature [34]. Also as Clojure is compiled to Java Byte Code and runs on top of the JVM, the language can interoperate with existing Java libraries and take advantage of the mature JVM tooling developed [35].

Clojure was developed with multi-threading in mind and has support for STM in the language itself to allow a controlled mutable state. A mutable state in Clojure is created as Refs and are mutable references to objects that can be shared through the program with STM providing the mechanisms necessary to update references without interference from other threads in execution. [36]

Commercial support for Clojure is provided through the company Cognitect. The company currently provides consulting in Clojure and other open-source projects for companies. [37]

2.2.2 Erlang

The Ericsson and Ellemtel Computer Science Laboratories created Erlang, that is a general-purpose parallel functional language designed for programming concurrent, real-time and distributed fault-tolerant systems. The language contains features such as higher-order functions, function selection through pattern matching, list comprehensions, a dynamic type system and an eager evaluator. Erlang has an explicit process-based concurrency model and the developer can control which computations are performed in parallel and which are not. Message passing between processes are asynchronous. The only way two processes can share information is through message passing, which in turn helps make applications that are easily distributed [38, p. viii, 1].

The language is available as open source and is actively being maintained by the Erlang community [39]. Erlang is still used by Ericsson and it has been used for several large telecommunication systems such as the development of the AXD301 ATM switch [40]. Furthermore Erlang has been used in other fields such as real-time bidding, e-mail systems, SS7 monitoring, VoIP Services, instant-messages for smartphones, etc. Many universities use Erlang at different levels for research and teaching [41].

The Actor model is built in Erlang itself because a process in the language is an actor in itself. Because the language is the model it does not support any other concurrent models [42].

2.2.3 F#

The F# Software Foundation created F# and is maintaining the open-source core F# repository. The language itself is a type-safe, type-inferred functional, imperative, object-oriented programming language. It is somewhat inspired by OCaml and aims to be the self-proclaimed frontier of typed functional programming languages for the .NET framework and .NET's Common Language Infrastructure (CLI) specification [43].

Even though F# is a multi-paradigm programming language it is primarily a functional language, inspired by OCaml a dialect of ML with the addition of objects and object-oriented constructs like class based objects, multiple inheritance [43, 44].

F# is today used in the industry for programming anything from games to high performance applications within Financial Risk [45]. The language is under continuous development and evaluation by the The F# Software

Foundation and Microsoft Research, all source code is open-source and can be found on GitHub [46].

F# has multiple concurrency models natively available to the developer. Besides the traditional and simple Threads known from for example C#, F# also has native support for the Actor model, described in Section 1.3.3, through the MailboxProcessor component.

2.2.4 Haskell

An active community of researches and application developers created Haskell and is still maintaining it. The language takes inspiration from other languages such as Lisp, ML and Miranda. Haskell is a general purpose, purely functional programming language that includes many recent innovations in programming language design. This programming language provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system and a set of primitive datatypes [47, p. 3].

Haskell is used in the context of industry, research and education. It has its footing in the industry sector ranging from aerospace and defence, finance, web start-ups, hardware design companies to lawnmower manufacturers [48]. Since its creation have the development of Haskell been driven in large part by academic efforts, due to its popularity as a implementation language for research into functional programming languages [49]. The language has also been used as an educational tool to teach functional programming to students since at least 1996 [49].

Multiple concurrency models are available natively such as threads (the Glasgow Haskell Compilers implementation of Concurrent Haskell) [50], actors, CSP-style concurrency, nested data parallelism and Intel Concurrent Collections. Messages, regular Haskell variables, *MVar* shared state or Software Transactional Memory [51] are available as tools for synchronisation between tasks [52].

2.2.5 Scala

The programming methods laboratory at École Polytechnique Fédérale de Lausanne (EPFL) created the multi-paradigm language Scala, that fuses object-oriented and functional programming in a statically typed programming language. Scala resembles Java and can seamlessly interact with code written therein, due to it being compiled to Java Byte Code and running on the JVM. Furthermore it uses a pure object-oriented model as every value is an object and every operation is a method call. Higher-order functions, currying, anonymous functions, immutable data structures, flexible symmetric mixin-composition constructs, pattern matching and type inference are some of the features found in Scala [53]. The evaluation of the language is eager

evaluated and non-strict by default, but it is possible to declare variables non-strict with lazy evaluation. [54, p. 44]

Scala is today supported commercially by the company Typesafe developing tooling based on the Eclipse Platform, in addition to providing consulting and training for companies implementing solutions using the Scala programming language. Typesafe has helped a number of companies making this transition from their previous systems. Some of these companies are Twitter, Walmart, CSC, and LinkedIn. The general field of application is broad as it includes subjects such as Big Data, Advertising, E-Commerce and Security [55]. Scala is an open source project under active development by the diverse Scala community and the Scala research group at EPFL as they received a five year European research grant in 2011 to tackle the "Popular Parallel Programming" [56, 57].

There are multiple concurrency models in Scala. Synchronised variables, futures, parallel collections, channels, workers, mailboxes, actors and constructs such as signals, monitors and semaphores to handle threads are present in the programming language as its different models of concurrency [54, p. 141-153,].

2.3 Summary

In this chapter we have chosen a set of modern functional programming languages. These languages all implement advanced concurrency models that provide a simpler method for programming concurrent and parallel applications compared to directly using threads and locks. In the following chapters will we describe these models in greater detail, evaluate their performance characteristics of each implementation, and briefly discuss how well the given implementations simplifies the development of concurrent and parallel programs when compared to using threads and locks.

Programming language	Threads	STM	Actor model	Parallel Collections
Clojure	X	X		X
Erlang			X	
F#	X		X	X
Haskell	X	X		X
Scala	X		X	X

Table 2.1: Natively supported concurrency models in the chosen languages

The Table 2.1 summaries which concurrency models are found natively in the chosen programming languages: F#, Haskell, Erlang and Scala. As documented in Section 1.3 Parallel Collections does provide a simple model for utilising data parallelism. Threads, STM and the Actor model however provides task parallelism. Due to this difference have we chosen to exclude

Parallel Collections for further evaluation, as Parallel Collection by design is restricted to collections while Threads, STM and the Actor model can be utilised for any task.

CHAPTER 3

Evaluation of Concurrency Models

The concurrency models STM and the Actor model that are found in the chosen programming languages in Chapter 2 will in this chapter be presented in greater detail with some of the problems which might occur when using the concurrency model. The description of each modelled is followed by an evaluation of the model based on the criteria presented in Section 3.1.

3.1 Concurrency Models Criteria

The concurrency models found in Section 2.2 are evaluated based on the approach taken in [27]. The approach evaluates the concurrency models by looking at four different characteristics.

- Implicit or Explicit Concurrency
- Fault Restricted or Expressive Model
- Pessimistic or Optimistic Model
- Automatic or Manual Parallelisation

We will be looking at concurrency models that are inherently more implicit compared to the concurrency models in [27] due to the exclusion of languages supporting only a lock based concurrency model. The difference between implicit and explicit concurrency is that in implicit concurrency the constructs are at a higher level of abstraction that does not require the use of explicit locks and semaphores. The use of explicit concurrency constructs are found in the direct use of locks when handling threads which we have already excluded from evaluation since we only want to look at advanced concurrency models. For further details see Section 2.1.

A more restrictive concurrency model can help developers make fewer errors for the cost of some freedom and flexibility. More experienced developers may feel more constrained and less expressive under these conditions

compared to a more expressive concurrency model with less restriction. Less restriction gives more freedom of expression which makes it possible to fine tune a program with greater finesse and allowing for greater control at the cost of leaving more responsibility in the hands of the developer.

Whether the concurrency model is optimistic or pessimistic determines the behaviour of the model. An optimistic concurrency model will try to maximise the number of threads running concurrently, if conflicts happen the model will have to be able to recover from this conflict. In contrast to a pessimistic model that will only utilise the number of threads that can be run safely and does therefore not need to handle conflicts.

The final criterion is regarding automatic or manual parallelisation. If the concurrency model relies on the compiler or the interpreter to automate the parallelisation of the code by inserting concurrency constructs then the parallelisation is deemed automatic. But if the parallelisation of the code has to be done manually it is up to the developer to determine which sections of the code should be run concurrently.[27]

The concurrency models in this chapter are weighted on a five step scale that spans from one side of the contrasting concurrency criteria to the other. Therefore we are able to define a concurrency model somewhere between the two contrasting elements the criterion consists off. An example is if a concurrency model contains both automatic parallelisation and manual parallelisation which would place it somewhere between the two. This allows us to compare the different scores in the summary of this chapter.

3.2 Software Transactional Memory

STM introduces the notion of transactions into concurrency control inspired by the world of databases where transactions are used extensively. This introduction of transactions into concurrency control provides the developer with a way to declare a region of code that should be executed transactionally and letting the system handle the concurrent execution. Using this model of concurrency the developer does not have to manually handle locks, or lock related problems such as deadlocks and race conditions. [58]

The ACID (Atomicity, Consistency, Isolation and Durability) properties are well known in the database community and are partly related to the notion of transactions in concurrency control. What these properties mean in terms of databases is first explained briefly. Atomicity in terms of a database transactions means that the execution of a transaction is always all-or-nothing. Consistency is the promise that a database in a persistent valid state can not end up in a non-valid persistent state because of executing a transaction. Isolation means that transactions does not interfere with each others execution. Finally Durability means that modified data caused by a successful transaction is reflected throughout the system for all intends and

purposes regardless of any system malfunctions. [59]

Concurrency control transactions are different as they do not have all the ACID properties. The reason and main difference is that these transactions are part of a programming language and not a persistent data management system. A programming language in in itself does not handle persistent data and therefore the promise of keeping such a state valid does not make sense which is why the Consistency property is not present. Nor does a programming language handle system malfunctions and thereby make persistent data durable as there is none which is why the Durability property is not present. A concurrency control transaction ensures that an execution of a transaction is all-or-nothing which is why Atomicity is present. Isolation is present as other transactions does not interfere with each other.

Code in programs utilising STM can also exist outside of transactions and there are two approaches to this called either weak or strong atomicity. Weak atomicity is when a transaction is only guaranteed no interference from other transactions but code outside transactions can interfere. Strong isolation is the same as weak isolation but with the addition that non-transactional code is also guaranteed not to interfere. Therefore it is possible for non-transactional code to interfere with transactional code when using only weak atomicity. Whenever we talk about transactions hereafter we refer to concurrency control transactions. An example of a transaction in a programming language can be found in Listing 3.1 where the transaction is made by the instruction `atomic` and a block of code defined by its associated pair of curly brackets. An example of how non-transactional code can interfere with a transaction with weak isolation is shown in Listing 3.2. [60]

```
1 atomic {
2     int i1 = foo.bar;
3     int i2 = bar.foo;
4
5     int i3 = i1 + i2;
6     foo.bar = i3;
7 }
```

Listing 3.1: Example of a transaction code block [60]

```
1  atomic {                          1  int i3 = bar.y; // i3 == 0
2      int i1 = foo.x; // i1 == 0    2
3                                    3  foo.x = 7;
4      int i2 = foo.x; // i2 == 7    4
5      bar.y = i1+i2;                5
6  } // conflict                     6  int i4 = bar.y; // i4 == 7
```

Listing 3.2: Example of a transaction with weak isolation [60]

Following the concept of weak and strong atomicity is yet another concept which is called privatization that is needed since we allow a mixture of both transactional and non-transactional code. Privatization is an operation done by a transaction that modifies the program in such a way that previously shared data is only accessed by one transaction. This is done to prevent non-transactional code from accessing and altering the data. Two problems with privatization exist, the *delayed clean-up* problem and the *doomed transaction* problem. The delayed clean-up problem is when transactional writes interfere with non-transactional reads or writes, which happens if a committed transaction has not yet redone its writes. The doomed transaction problem is when non-transactional writes to privatized data occurs, which leads to inconsistent reads and faulty behaviour of the transaction that has not yet realised it should be aborted. The explicit and transparent privatization are two ways of dealing with privatizing data in transactions. In explicit privatization the developer has to explicitly deal with the privatization in contrast to transparent privatization where the compiler handles the privatization of data. [61]

STM suffers from large overheads caused by instrumentation and privatization. However, these overheads can be reduced by introducing manual instrumentation instead of compiler added ones for non-transactional code. Furthermore, by also letting the developer do the privatization and thereby making it explicit versus transparent in additional overhead reduction can be achieved. To further increase the feasibility of Transactional Memory multiple variations exist besides STM, such as Hardware Transactional Memory (HTM) and hybrids of STM and HTM. It is still being discussed in academia whether or not STM is too heavily influenced by these overheads caused by adding more cores which is important if the concurrency model should have a practical usage for more than research. [58, 62]

3.2.1 Evaluation against criteria

This section evaluates the theoretical STM model according to the criteria described in 3.1. In the evaluation we focus primarily on the model itself and not a specific implementation.

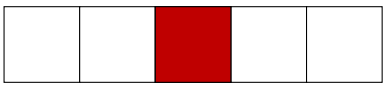
STM removes the lock-based explicit concurrency constructs from threads.

Constructs are available to ensure that expressions, writes and reads are done in a transaction. The transaction block found in STM is an explicit way of defining concurrency. Explicit creation of threads and synchronisation of these are still required. This makes STM an implicit concurrency model by replacing the lock-based explicit concurrency constructs from threads with implicit concurrency. Explicit creation and synchronisation of threads are still needed.

Implicit Concurrency **Explicit Concurrency**

The use of locks and semaphores are automatically handled by the compiler which means that STM removes some of the responsibility from the developer which gives some fault restriction. The use of transactionally safe reads and writes are still optional and therefore letting the developer to use non-transactionally reads and writes. This use of non-transactional code may introduce errors if variables are used outside of the transaction. Furthermore the developer is not able to interact with the isolation mechanisms which makes it fault restrictive. However the fact that the developer has to manage the synchronisation of threads as in a thread based concurrency model which can be faulty and therefore STM is considered a compromise between a fault restricted and an expressive model.

Fault Restricted Model **Expressive Model**



STM is an optimistic concurrency model with the use of retries in case of conflicts. This optimism means that STM will optimistically try to execute multiple transactions concurrently. If a conflict occur then one of transactions will be aborted and retried. There are no restrictions on how many threads that can execute a transaction concurrently.

Pessimistic Model **Optimistic Model**

The developer have to manually specify where transactions must be in the code so the compiler can handle the concurrency in that block. The creation and synchronisation of threads in STM is manual.

This concurrency model looks simpler compared to locks because a transaction block does not need to take into account other transaction blocks because of the Isolation property. The model is both implicit and optimistic which indicates that execution of multiple threads would inevitable lead to a number of conflicts. This number of conflicts would increase as the number of threads increase which in turn also indicates that the model could have problems with scaling.

3.3 The Actor model

The second of the two concurrency models found in the languages chosen in Chapter 2 is the Actor model. An actor is a concurrent, autonomous entity that together with messages forms an actor system. These actors are able to communicate asynchronously with each other through messages, that is the only possibility to affect the state of another actor [63]. Each actor has an immutable name that is needed when passing a message. An overview of an Actor system with three actors can be found in Figure 3.1. The three actors in Figure 3.1 each have their own individual mailbox, set of methods, internal state and execution thread, the only way the three actors are able to communicate is through messages. One of the actors in Figure 3.1 sends out two messages, one with the content *ping* and the other one with the content *create*, these messages are asynchronous and therefore non-blocking.

The Actor model has several features which makes it a compelling concurrency model. Some of these features are: encapsulation, fairness, location transparency and mobility mentioned and explained in [63] and [64]. As mentioned earlier actors are not able to directly access and alter the internal state of another actor. An actor is only able to change the state of another actor indirectly by sending messages to that actor. This encapsulation is a form of safety which prevents for example data races and unsafe object modification. In this encapsulation property two requirements are needed: state encapsulation and safe messaging. State encapsulation is the requirement that the state of an actor should not be accessible by other actors. When message passing between actors should be done using call-by-value semantics in order to prevent memory sharing between actors and thereby preventing access to local memory of an actor from another actor. An example of an actor system which does not respect this encapsulation property can be found in Listing 3.3 which does not respect this encapsulation property as the library used allowed access to the state of an object. The example code in Listing 3.3 is

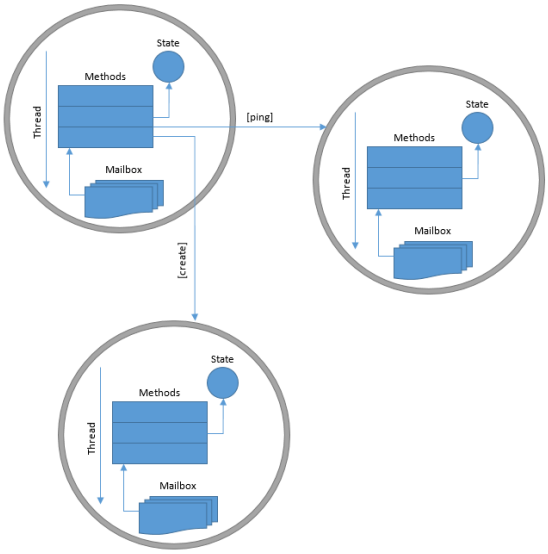


Figure 3.1: An overview of an actor system.

inspired by an example in [63].

```
1  class CountingActor extends Actor {
2      ...
3      function enter() {
4          if (num < MAX) {
5              // This is the critical section
6              num = num + 1;
7          }
8      }
9  }
10
11 function main() {
12     CountingActor counter = new CountingActor();
13     counter.start();
14     counter ! "enter";
15     counter.enter();
16 }
```

Listing 3.3: Example of an actor system not respecting the encapsulation property.

Fairness in the Actor model is the assumption of every message sent from an actor will eventually be delivered to its destination actor. This is assumed unless the destination actor is permanently "disabled". A "disabled" actor is

in an infinite loop or trying to do an illegal operation. The assumption of eventual message delivery to an actor also means that no actor can be permanently starved. An example of an actor system which is not fair is shown in Listing 3.4. This example is not fair because the class will enter the wait method and busy wait for the data from the calculator actor (`calc`), starving the calculator actor. This example is inspired by an example found in [63], the syntax used in Listing 3.4 and 3.3 is somewhat similar to the one in Scala but not correct Scala.

```
1  class FairActor extends Actor {
2      ...
3      function wait() {
4          // Busy-waiting section
5          if (data > 0)
6              print(data);
7          else
8              self ! "wait";
9      }
10
11     function start() {
12         calc ! ("add", 4, 5);
13         self ! "wait";
14     }
15 }
```

Listing 3.4: An example of an actor system not respecting the fairness property.

The name of the actor does not say anything about the actual location of the actor. This means that the developer does not have to worry about the actual physical location of an actor when programming. This property is called location transparency. The physical location and address space of an actor is not known by other actors because of this property that in return helps ensure state encapsulation. The mobility property is about different levels of migration.

Mobility is the ability to move computations across different nodes which includes the ability to migrate an actor to another physical node. There are two different types of mobility, namely strong and weak. It is possible to move both code and execution state of an actor to another node with strong mobility. It is only possible to move code and initial state of an actor to another node with weak mobility.

3.3.1 Evaluation against criteria

This section will evaluate the Actor model according to the criteria described in 3.1.

The Actor model forces the code to be organised using actors. This could be seen as explicit concurrency but in fact it is only the structure of the code and not how the concurrency is handled. The model is implicit in terms of actual use of concurrency constructs.

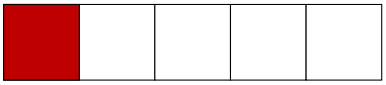
Implicit Concurrency **Explicit Concurrency**

All concurrency constructs of actors are hidden from the developer and the nature of the Actor model makes it very fault restricted. The reason for this is because of the encapsulation, fairness, location transparency and mobility properties explained in Section 3.3 that restrict the developer and thereby helps avoiding concurrency errors.

Fault Restricted Model **Expressive Model**

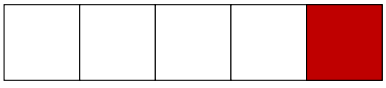
Every actor in the Actor model is isolated with no sharing of memory between them because of the encapsulation property. This makes it safe to run multiple actors concurrently. Based on this one could conclude that the Actor model is pessimistic.

Pessimistic Model **Optimistic Model**



The use of the Actor model forces the developer to manually reorganise his program into actors to enable concurrency. The developer needs to manually specify how many actors should execute concurrently.

Automatic Parallelisation **Manual Parallelisation**



The Actor models encapsulation of data that allows no sharing of data between actors removes the need of explicit access controls such as locks and semaphores. This model requires that the program is structured as actors which makes the addition of actors to an existing program a manual pro-

cess. This process could potentially force the programs architecture to be restructured to accommodate the Actor model. The model simple to work after an actor based code structure is established as each is a single thread of execution. Problems such as dead locks and data races can still be present in an actor based program if the data held by one actors is retrieved and updated differently by two different actors. The communication overhead added by not sharing any information between actors might prevent scalability of programs that spends too little time processing data internally in the actor compared to the time communicating between actors.

3.4 Summary

In this chapter, criteria were introduced and used to evaluate the theoretical concurrency models in the languages in Section 2. The evaluation of the models and the score results will be used as the foundation for the discussing of how each language implementation of the theoretical model fares against the expectation of the theoretical concurrency model in Chapter 5.

CHAPTER 4

Benchmark

In this chapter the selected concurrency models and the functional programming languages that implements them are going to be evaluated on performance and scalability. The benchmarking criteria are defined in Section 4.1. Then the benchmarking problem is described in Section 4.2. Implementation details for the STM based versions can be found in Section 4.3, and for actor based versions in 4.4. Details about the testing procedure which includes the used platform, compilers and implementation verification methods can be found in Section 4.5. The test results can be found in Section 4.6 with an associated analysis. A summary concludes on the benchmarking process and emphasis some of the key finding.

4.1 Criteria

Scalability of a concurrency model is very important and we have chosen to measure it by looking at speed-up and execution time. Speed-up is an indication of how much faster a program can solve a particular problem as the number of cores available to the program is increased, starting from an execution of the implementation using only one core. This is chosen as a criterion as increasing the throughput of a program by adding additional cores are the primary reason for added parallelism to programs, so any language or concurrency model not handling this aspect well is not fit for parallel processing. In relation to speed-up it is worth to take a look at saturation, which is the point at when a program given additional processor cores gains no additional speed-up. Execution time is needed to calculate speed-up and also interesting to look at by itself, as an implementation might have a huge increase in speed per core added to the execution but have such high execution time when only a single core is available that the amount of cores needed for the implementation to be competitive with other solution becomes prohibitive.

- **Speed-up:** How many times faster an implementation is with N cores available compared to using only one core.

- **Execution time:** How fast an implementation is finished executing with N cores available.

4.2 *k*-means

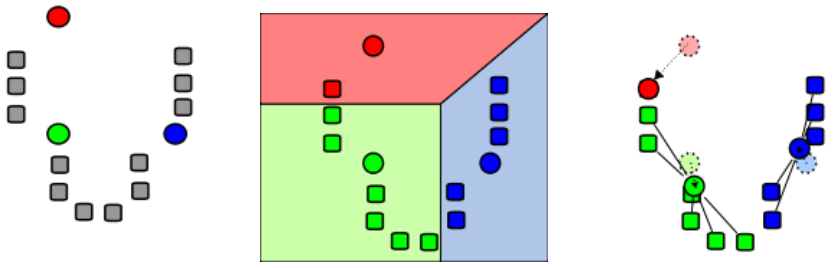
The k-means [65] algorithm clusters n points in a d-dimensional space into k clusters by iteratively minimising the average distance from each point to the nearest centroid. An initial set of centroids is needed which will iteratively be refined until convergence or a predefined maximum number of iterations are reached. The pseudo code is found in Algorithm 1 which gets a list of points and centroids as input. An illustrated example of this can be found in Figure 4.1a. Each iteration consists of two steps. The first step starts at line four where each point is assigned to the centroid with the shortest distance to the point using a distance function. The second step at line nine updates the position of the centroids to the center of each cluster by calculating the average set of features based on each point in the cluster. An illustrated example of step one and two can be seen respectively in Figure 4.1b, 4.1c.

Algorithm 1 Pseudocode K-Means [65]

Input: P = {p_1, p_2, .., p_n} (A list of data points)
C = {c_1, c_2, .., c_k} (A list of cluster centroids)
maxIterations (The number of maximum iterations to run)
Output: P = {p_1, p_2, .., p_n} (A list of data points)
C = {c_1, c_2, .., c_k} (A list of updated cluster centroids)

```
1:  iterations ←0
2:  repeat
3:      isChanged ←False
4:      for p_i ∈ P do
5:          centroid ←ComputeMinDistCentroid(p_i, C)
6:          if centroid ≠PointCluster(p_i) then
7:              UpdatePointCluster(p_i, centroid)
8:              isChanged ←True
9:      for c_i ∈ C do
10:         c_i ←UpdateCentroid(P, c_i)
11:     itearations++
12: until isChanged = False or iterations = maxIterations
```

The k-means was chosen for the performance and scalability benchmark for a number of reasons. First the clustering algorithm has already been used to benchmark parallel programs [67, 68] and is present in the STM benchmarking suite STAMP [69]. The problem exclusively consists of numeric computation simplifying the problem to CPU operations and memory reads, removing the need for tuning the performance of I/O operations in each im-



(a) (b) (c)

Figure 4.1: k-means example from Wikipedia [66]

plementation. Furthermore k-means is simple to understand, implement and is known throughout academia.

4.3 STM

The STM based k-means implementations partitions the dataset of points between a N worker threads in addition to the starting centroids, while the main thread is tasked with distributing new centroids after each iteration in addition to receiving the clustered points from each thread after the algorithm has terminated. All communication between the worker threads and the main thread as shown in Figure 4.2 are performed using three queues that blocks the thread until data is available in the queue, preventing threads from wasting CPU cycles while waiting for data. Three queues are needed as one is used for the threads to send back data for the main thread to distribute the new set of centroids, one is for the main thread to deliver the new centroids, and the last is needed for the worker threads to send back the final clusters when the algorithm terminates. A barrier is used for making sure that no worker thread is able to start the next iteration of the algorithm before all other worker threads have finished their iteration. It is needed because otherwise the same centroid proposal can be submitted multiple times, this is due to the nature of scheduling.

Each iteration of the algorithm consists of the worker threads updating their set of points by assigning each point to the centroid with the shortest distance to the point, the number of points in each cluster is then counted and the features of all the points in a cluster summed as the threads proposals for a new set of centroids. Each worker thread then adds their proposals for the next set of centroids to the first of the queues shared by the worker threads and the main threads, access to the queue is therefore protected by STM. As the queue blocks on reads when empty, the main thread is dormant until each worker thread adds the features and number of points, after data have been received from each thread a new set of centroids are computed by adding the

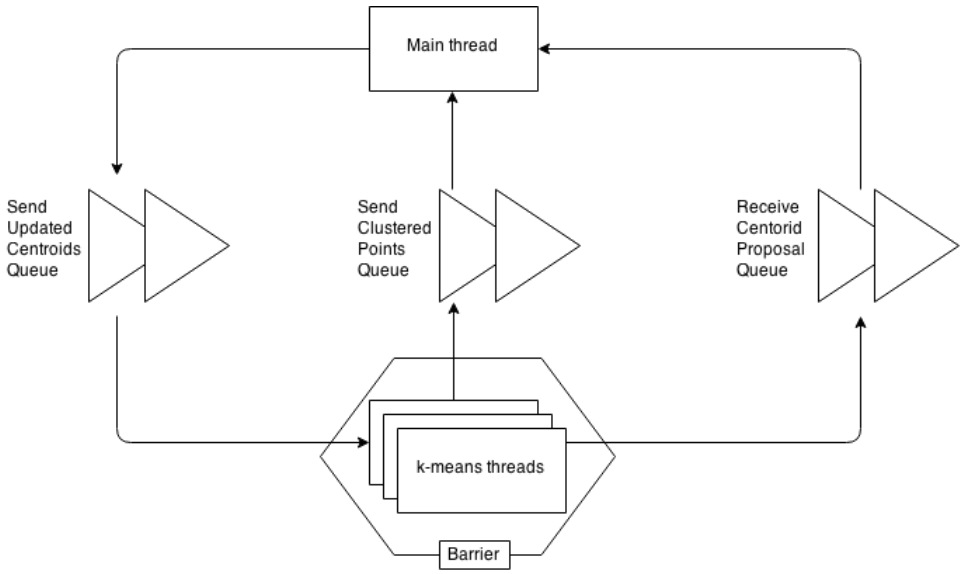


Figure 4.2: STM Implementation Architecture

number of points in each cluster across all threads as well as their summed features, the summed features are then divided by the number of points that cluster contains to provide the center each cluster across all threads as the new centroid. One set of the new centroids per thread is added to the second blocking queue shared by all threads, allowing threads to go dormant while waiting for new centroids due to the queue blocking on reads when empty. To prevent a thread from performing multiple iterations using the same set of centroids due to another thread being starved by the runtime, a barrier is added that requires all threads to have added their proposal for a new set of centroids before the next iteration can begin. When the maximum number of iterations are performed or no points were moved to a new cluster in any of the worker threads, the resulting clusters are added to the third shared blocking queue, allowing the main thread to combine the results from each thread into one final set of clusters and terminate the program.

The amount of communication needed in this implementation is low due to the exchange of centroids being infrequent and the transactions serve only to avoid problems when operations are performed on the three shared queues. This makes the scalability of these implementations highly dependent on the size of data partition each thread operates on as a small data set would give an increased number of new proposals for centroids because of the increased number of threads, since each thread sends a proposals for each centroid. The amount of data that needs to be exchanged can be smaller than the communication overhead which would either saturate the program or forcing a decrease in performance.

4.3.1 Clojure

The Clojure implementation of K-Means uses Clojure's built in `Vector` due to the known problems of locality with using a linked list data structure as implemented by Clojure's `List` data type. With the exception however of the accumulation of centroid proposals where a `List` is used to provide efficient pre-pending. The creation of threads are done using Clojure's `Future` which creates a new thread and uses it to execute the function passed to it, when the function have finished execution the resulting value can be accessed through the `Future`, however as all communication is performed through queues using STM the resulting value of the future are simply discarded.

Communication between worker threads and the main thread distributing new centroids are done using three instances of `java.util.concurrent.LinkedBlockingQueue`, as Clojure's only blocking data structure `seque` uses a fixed size instance of `java.util.concurrent.LinkedBlockingQueue` internally to allow a producer to be a fixed number of items ahead of a consumer before blocking. Such functionality is unnecessary in our implementation of K-Means and the `put` and `take` interface of the `java.util.concurrent.LinkedBlockingQueue` while blocking on and a call to `take` on an empty list, is deemed to be subjectively simpler to use for our implementation. Despite `java.util.concurrent.LinkedBlockingQueue` providing thread-safe operations where each queue are stored in a `ref`. A `ref` is a reference type that can be read both inside and outside a transaction, if read outside the last committed value is returned, but only able to be updated inside a STM transaction.

The Clojure implementation of K-Means primarily consists of two functions each using multiple smaller helper functions, the first is named `run-k-means`, and is shown as Listing 4.1. This function encapsulates the K-Mean algorithm itself and an instance of this function executes for each worker thread. The function has two main branches if the number of iterations the algorithm has performed has reached the maximum allowed number, the branch on line 4 is executed and the main thread is first informed that no more centroids will be needed by simulating that now points had moved last iteration and the clustered points are then added to another queue before the function terminates. The operation of adding elements to the queue are performed using the `stm-put` function, which simply encapsulates dereferencing of the `ref` containing the queue and the call to the `put` function on the queue inside a `dosync` block. If further iterations of the algorithm are allowed the branch on line 8 is taken, where the set of points assigned to this particular thread first is updated by assigning them to the centroid closet to their position, and the number of points in each cluster counted and the features of all points in each cluster summed to create a proposal of new centroids from this thread. These are then sent to main thread in addition to boolean value indicating if any points have changed cluster between this and the previous

iteration on line 15, this is followed by a call to the function `barrier-await` that simply waits on an instance of `java.util.concurrent.CyclicBarrier` instantiated with the number of worker threads, this is on line 19. The use of a barrier prevents a thread from executing the `run-k-means` function twice in one iteration providing a duplicate centroid proposal while another thread is starved and prevented from providing one. On line 21 the new centroids in addition to a boolean indicating if any threads moved any points last iteration is received from the main thread. If no worker thread moved a point to a new cluster the centroids the algorithm terminates by returning the final clusters, if any points did change cluster the number of iterations are decremented and the function is called recursively with the new centroids on line 26.

```
(defn run-k-means [points centroids iteration n_points n_clusters]
  (cond
    ; Terminate the algorithm if maximum number of iterations is reached
    (zero? iteration) (do
                        (stm-put send-centroid-proposal-queue [false '()])
                        (stm-put send-clustered-points-queue points))
    ; The points are updated and the set of centroid proposals are computed
    :else (let [updated-points
                  (update-points points centroids)
                proposed-centroids
                  (update-centroids updated-points n_clusters)]
            ; Sends a boolean indicating if any points was moved by
            ; the thread this iteration, and the number of points and
            ; summed features for each of the clusters
            (stm-put send-centroid-proposal-queue
                     [(points-changed? points updated-points false)
                      proposed-centroids])
            ; Prevents a single thread from running multiple iterations
            (barrier-await)
            ; Receives the new set of centroids from the main thread
            (let [all-finished-centroids
                    (stm-take recieve-updated-centroid-queue)]
              ; Checks if all worker threads have finished
              (if (first all-finished-centroids)
                (stm-put send-clustered-points-queue points)
                (run-k-means updated-points
                             (second all-finished-centroids)
                             (dec iteration)
                             n_points
                             n_clusters))))))
```

Listing 4.1: The main K-Means function in Clojure

The second primary function of the Clojure implementation of K-Means are the function named `compute-updated-centroids-from-proposals` shown as Listing 4.2. This function is executed recursively by the main thread and accumulates the centroid proposals sent by each worker thread, and then compute the new centroid for each cluster before sending a copy of the new centroids to each of the worker threads. The contains a nested function named `compute-centroids` that performs the actual aggregation, while `compute-updated-centroids-from-proposals` sends the final result to each worker thread. The function is nested as the inner function is only used by `compute-updated-centroids-from-proposals`, which starts by ex-

ecuting by calling `compute-centroids` on line 26 with the number of worker threads and an empty list for aggregation as arguments. `compute-centroids` checks if all worker threads have delivered their proposals for the new set of centroids on line 5, and divides the features by the number of points in the cluster to provide the new set of centroids. If some threads still have not added their proposals for new centroids to the queue line 8 is executed. Here the data is extracted from the queue and split into two values; a boolean indicating if the worker thread moved any points to another cluster and the set of centroid proposals. On line 14 the boolean and centroid proposals are aggregated with any values received from any other worker threads and the inner function is called recursively. When the the new set of centroids are returned to `compute-updated-centroids-from-proposals` on line 25, a copy is sent to each worker thread on line 30, ending the function with a check if any thread reported a point was moved on line 35, the function calls itself recursively if a worker did indicate a points was moved.

```
(defn compute-updated-centroids-from-proposals [number-of-threads]
  (defn compute-centroids [threads-running all-finished acc]
    (cond
      ; All threads have returned their centroids proposals
      (zero? threads-running)
          [all-finished (mapv divide-centroid-features acc (range))]
      ; A centroid proposal is received and unpacked
      :else (let [centroid-proposal
                    (stm-take send-centroid-proposal-queue)
                  points-changed (first centroid-proposal)
                  centroids (second centroid-proposal)]
              ; Prevents "map" from returning the empty list
              ; when we receive the first set of centroids
              (if (empty? acc)
                (compute-centroids
                  (dec threads-running)
                  (and (not points-changed) all-finished)
                  centroids)
                (compute-centroids
                  (dec threads-running)
                  (and (not points-changed) all-finished)
                  (map add-centroid-proposal centroids acc))))))
  ; Calls the inner function to aggregate the proposals and
  ; compute the new centroids in the middle of each cluster
  (let [all-finished-centroids
          (compute-centroids number-of-threads true '())]
    ; The final set of centroids are ready and is transmitted to
    ; the threads in addition to a boolean indication if a new
    ; iteration is necessary
    (send-centroids
      (second all-finished-centroids)
      (first all-finished-centroids) number-of-threads)
    ; If any thread changed the cluster of a point are the function
    ; called recursively again with a decremented number of iterations
    (when-not (first all-finished-centroids)
      (compute-updated-centroids-from-proposals number-of-threads))))
```

Listing 4.2: The centroid update function in Clojure

4.3.2 Haskell

The Haskell implementation makes use of the `Vector` library bundled with the Haskell Platform, instead of Haskell's built-in `Data.List` data structures.

This is due to the better use of locality offered by an array style data structure like `Data.Vector`. Haskell's built-in `Data.Array` was not used as it provides a much smaller API compared to `Data.Vector`, which is bundled with The Haskell Platform which we consider the canonical Haskell implementation.

Communication between worker threads and the main thread distributing new centroids are done using three instances of `Control.Concurrent.STM.TBQueue`, which is accessed through two functions `readTBQueue` which reads the next value in the queue and blocks the thread if the queue is empty, and the function `writeTBQueue` which appends a value to the queue. To ensure all STM based operations in Haskell are performed inside a STM block are they encapsulated in the `STM` monad. This allows STM operations to be combined using Haskell's built-in monad operations, and prevents execution of STM operations accidentally interleaved with non monadic code. To execute a chain of STM operations in Haskell are the function `atomically` provided, which executes the operations atomically and returns the result encapsulated in the `IO` monad.

The Haskell implementation of K-Means primarily consists of four functions each using multiple smaller helper functions, the first is named `runK-Means`, and is shown as Listing 4.3. This function encapsulates the K-Mean algorithm itself and an instance of this function is executed for each worker thread. The function has two implementations and pattern matches on the parameters, if the number of iterations the algorithm has performed has reached the maximum allowed number, the first version on line 12 are executed and the main thread is informed that no more centroids will be needed by simulating that no points have moved last iteration and the clustered points are then added to another queue before the function terminates. If further iterations of the algorithm is allowed the version of the function defined on line 17 is used, the set of points assigned to this particular thread is first updated by assigning them to the centroid closet to their position, the number of points in each cluster is then counted and features of all points in each cluster summed to create a proposal of new centroids from this thread. The computation of updated points, centroid proposals, and the check for if any points are changed are all made strict using a GHC syntax extension to Haskell name BangPatterns which uses the `!` symbol placed on a data binding such as a `let`, this forces the computation creating the data being bound is computed strictly. The extension was used instead of Haskell's `seq` function which also forces evaluation to be strict as `seq` must be placed in the function calls instead of at the binding like BangPatterns adding complexity to the processing code, the use of strict evaluation was necessary as Haskell else executed the program in near constant time without regard for the number of system threads it was allowed to use. These are then sent to main thread in addition to boolean value indicating if any points have changed cluster between this and the previous iteration on line 28, this is followed by a call to the function `awaitOnConcreteBarrier` with threads unique index

on line 31, this function is a version of the barrier function shown as Listing 4.6 curried with an instance of our own `Barrier` type created with the number of threads in the system. The use of a barrier prevent a thread from executing the `runKMeans` function twice in one iteration providing a duplicate centroid proposal while another thread is starved and prevented from providing one. On line 33 the new centroids are in addition to a boolean indicating if any threads moved any points last iteration is received from the main thread. If no worker thread moves a point to a new cluster the centroids are the same as last iteration and the algorithm terminates by returning the final clusters, if any points did changed cluster the number of iterations are decremented and the function is called recursively with the new centroids on line 38.

```haskell
runKMeans :: Vector Point
          -> Vector Centroid
          -> Int
          -> Int
          -> Int
          -> TQueue (Vector Centroid, Bool)
          -> TQueue (Vector (Int, Vector Double), Bool)
          -> TQueue (Vector Point)
          -> (Int -> IO())
          -> Int
          -> IO ()
runKMeans points _ 0 _ _ _ sendCentroidProposalQueue
    sendClusteredPointsQueue _ _ =
    - Terminate the algorithm if maximum number of iterations is reached
    writeToQueue sendCentroidProposalQueue (emptyVector, False) »
    writeToQueue sendClusteredPointsQueue points
runKMeans points centroids iterations nPoints nClusters
    recieveUpdatedCentroidQueue sendCentroidProposalQueue
    sendClusteredPointsQueue awaitOnConcreteBarrier threadId = do
        - The points are updated and the set of centroid proposals
        - are computed, the updated points are checked to see if any
        - where moved to a new cluster
        let !updatedPoints = V.map (`updateCluster` centroids) points
        let !centroidProposals = updateCentroids updatedPoints nClusters
        let !isPointsChanged = pointsChanged points updatedPoints False
        - Sends the boolean indicating if any threads have moved and
        - the set of centroid proposals for this thread
        writeToQueue sendCentroidProposalQueue
            (centroidProposals, isPointsChanged)
        - Prevents a single thread from running multiple iterations
        _ <- awaitOnConcreteBarrier threadId
        - Receives the new set of centroids from the main thread
        (updatedCentroids, allFinished) <-
           atomically \$ readTQueue recieveUpdatedCentroidQueue
        - Checks if all worker threads have finished
        if allFinished
           then writeToQueue sendClusteredPointsQueue points
           else runKMeans updatedPoints updatedCentroids
                (iterations - 1) nPoints nClusters
                recieveUpdatedCentroidQueue sendCentroidProposalQueue
                sendClusteredPointsQueue awaitOnConcreteBarrier threadId
```

Listing 4.3: The main K-Means function in Haskell

The second primary function of the Haskell implementation of K-Means is the function named `computeCentroidsFromProposals` shown as Listing 4.4. This function is executed recursively by the main thread to accumulate the centroid proposals sent by each worker thread, and then compute the new centroid for each cluster before sending a copy of the new centroids to each of the worker threads. The function contains two implementations with the right version to execute selected using pattern matching, the first version defined on line 8 is called when the all worker threads reports that no points moved during last iteration, which sent an empty list of centroids and a `True` boolean to terminate the worker threads. The second version starts on line 12 and is called recursively until all worker threads report that no points have moved during the last iteration. This version calls `recieveCentroids` shown as Listing 4.5 on line 16, which receives the set of centroid proposals from the worker threads and computes the new set of centroids from them in addition to a flag indicating if all worker threads have finished. The new centroids and the boolean flag are then sent to each worker thread on line 19 and the function calls itself recursively on line 21.

```
1  computeCentroidsFromProposals :: Bool
2                                -> Int
3                                -> TQueue (Vector (Int, Vector Double),
4                                   Bool)
5                                -> TQueue (Vector Centroid, Bool)
6                                -> IO ()
7  - Exiting function as no more threads are in need of new centroids
8  computeCentroidsFromProposals True nThreads _
9      recieveUpdatedCentroidQueue =
10     writeNToQueue recieveUpdatedCentroidQueue (emptyVector, True)
11         nThreads
12 computeCentroidsFromProposals _ nThreads sendCentroidProposalQueue
13     recieveUpdatedCentroidQueue = do
14     - The centroid proposals are aggregated by a helper function,
15     - and the next set of centroids are computed
16     (centroids, allFinished) <- recieveCentroids nThreads
17         sendCentroidProposalQueue emptyVector True
18     - The new set of centroids are distributed to each thread
19     writeNToQueue recieveUpdatedCentroidQueue
20         (centroids, allFinished) nThreads
21     computeCentroidsFromProposals allFinished nThreads
22         sendCentroidProposalQueue recieveUpdatedCentroidQueue
```

Listing 4.4: The centroid update function in Haskell

The function `recieveCentroids` shown as Listing 4.5 aggregates the cen-

troid proposals sent from the worker threads into a new set of centroids, which is then returned to `computeCentroidsFromProposals` shown in Listing 4.4. Once again is two implementations used, the first on line 6 divides all features for the centroid of each cluster by the number of points in that particular cluster to find the center of the cluster as the new centroid, the set of centroids are then returned. The second implementation defined on line 10 reads the proposal for new centroids from the worker threads on line 13 together with a boolean indicating if the thread moved any points last iteration, the proposals for new centroids are then added feature by feature on line 16 and the accumulated result is used as parameter for the recursive call on line 20.

```
1  recieveCentroids :: Int
2                   -> TQueue (Vector (Int, Vector Double), Bool)
3                   -> Vector (Int, Vector Double)
4                   -> Bool
5                   -> IO (Vector Centroid, Bool)
6  recieveCentroids 0 _ accCentroidProposals allFinished =
7      - All worker threads have returned their proposals for new centroids
8      return (buildCentroidFromProposals accCentroidProposals,
9              allFinished)
10 recieveCentroids threadsRunning sendCentroidProposalQueue
11     accCentroidProposals allFinished = do
12     - Eeach centroid proposal are aggreated recursively
13     (centroidProposal, isPointsChanged) <-
14       atomically \$ readTQueue sendCentroidProposalQueue
15     - Prevents zipWith being called with an empty list ass acc
16     let cp = if V.null accCentroidProposals
17                 then centroidProposal
18                 else V.zipWith addCentroidProposal
19                           centroidProposal accCentroidProposals
20     recieveCentroids (threadsRunning-1) sendCentroidProposalQueue
21         cp (not isPointsChanged && allFinished)
```

Listing 4.5: The centroid proposal receive function in Haskell

Due to The Haskell Platform not containing an implementation of a barrier, a simple barrier is created using STM. The barrier consists of two parts shown as Listing 4.6, first the data type named `Barrier` that contains the number of threads the barrier blocks until they are released, a counter to know how many threads are currently block, and an sequence of boolean flags used to block each thread. The second part is the function `awaitOnBarrier` that takes as parameters an instance of the `Barrier` type and an unique integer used by the barrier to indicate which flag the thread should block

against. The function consist of two blocks each executed atomically the first defined on line 6 increments the counter of how many threads have reached the barrier, and sets all the boolean flags to `False` unblocking all threads if enough threads have reached the barrier. The second block defined on line 20 is to block each thread, here the thread uses its index to check if its flag is set to `True` and uses Haskell `retry` function to block if it is, otherwise the thread continues the execution after setting its flag to `True` to ensure it is blocked when it next calls `awaitOnBarrier` with the same barrier unless it is the last to reach it.

```
1  data Barrier = Barrier (TVar Int) (TVar Int) (TVar (Seq Bool))
2
3  awaitOnBarrier :: Barrier -> Int -> IO ()
4  awaitOnBarrier (Barrier bCount bNThreads bBFlags) threadId = do
5      - Decrements the barriers thread counter atomically
6      atomically (do
7                  count <- readTVar bCount
8                  nThreads <- readTVar bNThreads
9                  let updatedCount = count + 1
10                 if updatedCount /= nThreads
11                    then writeTVar bCount updatedCount
12                    else do
13                        - The expected number of thread have arrived
14                        flagSeq <- readTVar bBFlags
15                        let newFlags =
16                              mapWithIndex (\_ _ -> False) flagSeq
17                        writeTVar bBFlags newFlags
18                        writeTVar bCount 0)
19     - Blocks all threads until the barriers count is zero
20     atomically (do
21                 flagSeq <- readTVar bBFlags
22                 if index flagSeq threadId
23                    then retry
24                    else writeTVar bBFlags
25                          (update threadId True flagSeq))
```

Listing 4.6: STM barrier implementation

As Haskell is lazy evaluated by default the function `deepseq` are used to force the full evaluation of the finished result before the time is recorded as shown in Listing 4.7.

```
1  ...
2  startTime <- getPOSIXTime
3  clusteredPoints <- kMeans dataset centroids maxIterations nThreads
4  endTime <- clusteredPoints `deepseq` getPOSIXTime
5  ...
```

Listing 4.7: K-Means execution

4.4 Actor

In the actor implementation of k-means the set of points is partitioned and a partition is given to each of the actors. Each of the actors are sent a message to start the clustering of the points it holds, each actor will answer with the number of points that changes its centroid. If no points changed centroid the clustering has converged and the program will terminate. If any points changed its centroid the actors is messaged a request for input for new centroids. Each actor then computes a sum of features based on the points held by the actor. The feature sums for each centroid together with the count of data points which is the basis for the feature sum are sent to the main thread which then computes the average of these sums from each actor to get the new centroids. The new centroids are then sent to the actors to be used in the next clustering iteration.

An overview of the communication of the actor implementation can be found in Figure 4.3. The implementations will follow the pseudo code found in Algorithm 1 but will of course have to follow the structure of the Actor model.

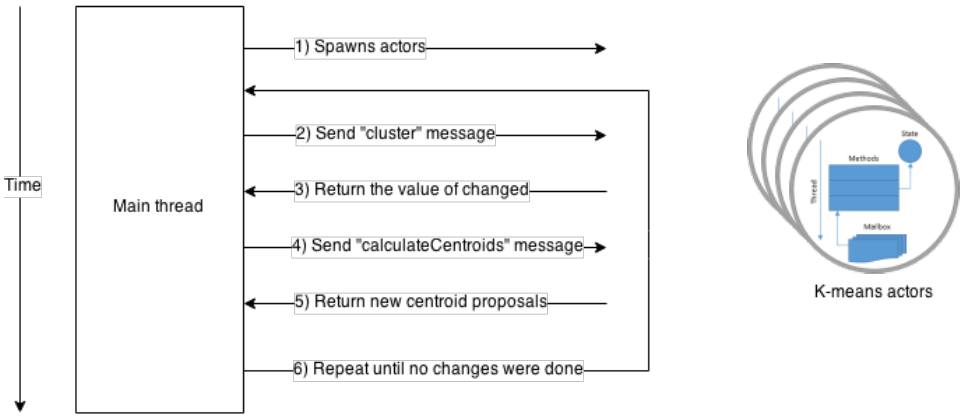


Figure 4.3: Actor Implementation.

The following sections will explain the implementations of k-means using the Actor model written in the languages Erlang, F# and Scala. After the

individual explanations a comparison of the different implementations can be found.

4.4.1 Erlang

The implementation of k-means in Erlang does not follow the general overview of communication of the actor implementations found in Figure 4.3 because of some limitations in the Actor model implementation in Erlang, these limitations and the implementation will be explained in this section.

The limitations of the Actor model implementation lies in a lack of specific functionality. A common application when using any messaging, for example when using actors, is being able to ask for an answer to a message. The Erlang implementation of the Actor model does not support this, therefore some differences can be found in the Erlang implementation of k-means compared to the ones in F# and Scala. The functionality needed is implemented using a simple structure which can be found in Listing 4.8.

An actor in Erlang is a single recursive function which uses the `receive` block to pattern match and handle all incoming messages. The `cluster` message is handled in the `main_actor` from line number 6 to 18. The code checks if the `Iteration` argument is still over 0, if so a `{self(), cluster}` message to each `Actor` in `Actors` where `self()` is the location of the `main_actor` itself. Since Erlang lacks the functionality to ask for an answer when messaging other actors some synchronisation code is needed. In the lines 21 to 34 the number of changes from each actor's clustering is handled. In line number 3 the received number of changes is added to the list `ChangesList` and assigned to the variable `NewChangesList`. The number of elements in `NewChangesList` is then in line number 26 compared to the number of actors in the system. If true the block of code in line number 28 is run, otherwise the `main_actor` will call it self with the new list of changes. The Erlang implementation uses a `list` for its data points as well as its centroids, this is because of the lack of functionality in its array data structure. The `list` data structure in Erlang is a linked list therefore the locality of the data structure is not as good as with the use of arrays.

```
1  main_actor(Actors, Iteration, ChangesList,
2            SuggestionsList, DataPointsList,
3            ClusteringDone, StartingTimestamp) ->
4    receive
5      %%% Start clustering of points
6      cluster ->
7        case Iteration > 0 of
8          true ->
9            % Message all actors to cluster their data
10           lists:map(fun(Actor) -> Actor ! {self(), cluster} end,
11             Actors),
12           main_actor(Actors, Iteration, ChangesList,
13                      SuggestionsList, DataPointsList,
14                      ClusteringDone, StartingTimestamp);
15         false ->
16           ...
17       end;
18     %%% Get clustering result
19     {cluster_result, Changes} ->
20       % Add cluster result
21       NewChangesList = [Changes | ChangesList],
22       % If all results are received proceed
23       case length(NewChangesList) == length(Actors) of
24         true ->
25           ...
26         false ->
27           % All results are not received, wait for more results
28           main_actor(Actors, Iteration, NewChangesList,
29                      SuggestionsList, DataPointsList,
30                      ClusteringDone, StartingTimestamp)
31       end;
```

Listing 4.8: The synchronisation construct created for the Erlang implementation

Mapping, summation and additional functionality on lists, arrays etc. are bundled in modules in Erlang. As an example see Listing 4.9 where several functions from the `lists` and a function in the `math` modules are used to manipulate various lists in the implementation of the DataPoint update in the K-means actors. In line number 6-7 all `dataPoints` the `DataPartition` variable is mapped to its current `centroid` which is then assigned to the `OldCentroids` variable. From line number 9-25 the current `dataPoints` in `DataPartition` is mapped to a new list of `dataPoints` where the `centroid`

of each `dataPoint` is updated to the `centroid` closest to the `dataPoint` based on the Euclidean distance from the `dataPoint` to the `centroid`. To complete Erlang implementation can be found in the source codes for this project.

```
1  %%% The K-means actor
2  kmeans_actor(DataPartition, Centroids) ->
3    receive
4      %%% Tell the actor to cluster its data points
5      {MainActor, cluster} ->
6        OldCentroids = lists:map(fun(DataPoint) ->
7          DataPoint#dataPoint.centroid end, DataPartition),
8
9        NewData = lists:map(fun(DataPoint) ->
10         % Returns list of list of:
11         % [Distance from DataPoint to Centroid, Centroid.Name]
12         Distances = lists:map(fun(Centroid) ->
13           [lists:sum(lists:zipwith(fun(X,Y) ->
14             math:pow(X - Y, 2) end,
15             DataPoint#dataPoint.features,
16             Centroid#centroid.features)),
17           Centroid#centroid.name] end, Centroids),
18
19         % Returns the name of the Centroid with
20         % the minimum distance to the DataPoint
21         MinCentroidName = lists:nth(2, lists:min(Distances)),
22
23         % Return new DataPoint to the map function
24         update_datapoint(DataPoint, MinCentroidName)
25         end, DataPartition),
26       ...
```

Listing 4.9: The clustering implementation in Erlang

4.4.2 F#

The implementation of the Actor model in F# supports commonly used functionality for asking an actor for a response to a message without the need for an additional actor or the need for additional synchronisation constructs. The Actor model implementation in F# can be found in the `MailboxProcessor` class. An actor in F# is a single function

In Listing 4.10 each `actor` is asked to reply to the message `Cluster` which takes the `AsyncReplyChannel<int>` called `replyChannel` as an argument. In line number 2-4 a map over the `actors` in the system is done and each

`actor` is messaged. The reply from each `actor` will be the type of the data inside the `AsyncReplyChannel`, in this case an `int` which is the number of changes from the `actor`. The code in the lines 5-7 makes sure the messages are sent and their responses are done in parallel and that all responses will be awaited before adding together the number of changes from each `actor` into the variable `sumChanged`. The F# implementation uses `Array`s for the data points as well as the centroids to achieve better locality when performing calculations, internally in data points, centroids and centroid proposals a `List` is used for holding the features.

```
1  let sumChanged =
2    (actors |> Array.map(fun actor ->
3      actor.PostAndAsyncReply(fun replyChannel ->
4        Cluster replyChannel)))
5    |> Async.Parallel
6    |> Async.RunSynchronously
7    |> Array.sum
```

Listing 4.10: Example of the synchronisation in F# for getting the sum of changes

The pattern matching of the message `Cluster` with the argument `replyChannel` happens in line number 6. In line 7, the current `dataPoints` is mapped to their current `Centroid` used later for calculating the number of changes of centroids in the `dataPoints`. The `dataPoints` with updated centroids is calculated in the lines 10 to 17. In the lines 19-20 the reply is sent using the `replyChannel`, hereafter the recursive function handling the messages calls itself with the `newData` as they are updated with new centroids. Worth noting is in line 4, `let!` is used to return the message received inside an `Async` type using an computation expression.

```
let rec loop(dataPoints: array<DataPoint>,
             currentCentroids: seq<Centroid>, msg) =
  async {
    let! msg = inbox.Receive()
    match msg with
    | Cluster(replyChannel) ->
      let oldCentroids = dataPoints
                         |> Array.map(fun dataPoint -> dataPoint.Centroid)

      let newData = dataPoints |> Array.map(fun dataPoint ->
        updateCentroid(dataPoint,
          currentCentroids |> Seq.minBy(fun centroid ->
            Seq.sum(Seq.map2(fun f1 f2 -> (f1 - f2) ** 2.0)
            dataPoint.Features centroid.Features)
          )
        )
      )
      ...
      replyChannel.Reply(changedCount)
      return! loop(newData, currentCentroids, msg)
```

Listing 4.11: The clustering implementation in F#

4.4.3 Scala

The Scala implementation of the Actor model supports the commonly used functionality for asking an actor for a response to a received message. When asking for a response the immediate result is a `Future`. The `Future` is a placeholder object for a value yet to be returned by the actor in this case. The Actor model implementation in Scala consists of an `ActorSystem` object which essentially holds the functionality of the entire actor system. This `ActorSystem` is then given all of the created actors in order for them to be scheduled.

Found in Listing 4.12 is the code for retrieving the number of changes from all actors after their clustering and assigning the sum of these results into the variable `sumChanged`. In the lines 3-4 all `actors` is messaged (or in this case `ask`ed) using the case object `KmeansActor.Cluster`. The returned `Future`s is then again mapped over in lines 4-6, awaiting their result and finally adding all of the results together in line 7.

```
val sumChanged =
(
  (actors map
    (actor => actor ask KmeansActor.Cluster)) map
      (future =>
        Await.result(future, timeout.duration).asInstanceOf[Int])
).sum
```

Listing 4.12: Example of the synchronisation in Scala for getting the sum of changes

In Scala an actor is a sub-class of the super-class `Actor`, the k-means actor can be found in Listing 4.13. The class then needs to implement the `receive` method which then pattern matches and handles all incoming messages. A side-effect of this choice of implementation is that the `Actor` class needs to have mutable variables in order to hold the data being worked on. The class is defined and imports the case objects in the lines in 1-3, initial data is assigned to class variables in line 5 and 6. The `Cluster` message is handled in lines 9 to 18.

```
1  class KmeansActor(dataPartition: Array[DataPoint],
2                    initialCentroids: Array[Centroid]) extends Actor {
3    import KmeansActor._
4
5    var data = dataPartition
6    var centroids = initialCentroids
7
8    def receive = {
9      case Cluster =>
10       val oldCentroids = data map(x => x.Centroid)
11
12       data = data map(dataPoint =>
13         updateDatapoint(dataPoint, centroids minBy(centroid =>
14           ((centroid.Features zip dataPoint.Features) map {case (f1, f2) =>
15             pow(f1 - f2, 2)
16           }).sum
17         ))
18       )
19     ...
```

Listing 4.13: Example of the functionality placed on classes and types in Scala

4.5 Procedure

We have chosen an Amazon EC2 instance as platform with the possibility for others to later recreate our benchmarks. The instance is of the type `c3.8xlarge` that has 32 vCPUs on Intel Xeon E5-2680 v2, Ivy Bridge. The vCPUs are running at 2.8GHz and are hyper-threads. The instance has 60GB of memory and two 320GB SSDs are part of the instance as non-persistent memory. The instance also includes persistent memory in the form of an Amazon EBS volume which holds the operating system. An Amazon EBS storage is a durable, block-level storage volume that can be attached to a single Amazon EC2 instance. The Amazon EBS storage is not directly connected to the host computer, however the two non-persistent SSDs are. The operating system of the Amazon EC2 instance is Microsoft Windows Server 2012 R2 Standard edition (64-bit) because of it being the only platform with official support for all languages, primarily of concern was the maturity of the Open Source Mono runtime in comparison to Microsoft .Net runtime [26].

All the implementations are compiled once and executed multiple times on the mentioned platform with four different datasets as input. Each implementation is going to be executed a number of times with respectively 1, 2, 4, 8, 16, 32 cores available which gives us the execution time to give an overview of the implementations scalability. The execution time is measured just before the data partition takes place in the implementations and after they have loaded the datasets into memory and their respective environments have been fully loaded.

The included datasets is found through experimentation by running datasets with ranging sizes from 32 samples up to a dataset that gives a indication of a clear difference in execution time when executed with the six different configurations of cores available. The datasets which results in interesting execution times that shows overhead of the implementation and scalability is then included. The datasets used for benchmarking is generated using the dataset generation feature in the Python machine learning library *scikit-learn*.

Each implementation is executed 10 times and the median is found because usually multi-threaded implementations contains outliers which would affect the average execution time [70].

A sequential implementation have been omitted because it does not give a significant variability compared to a single threaded version [70].

The correctness of the implementations is tested by using the labeled Iris dataset [71] and a set of fixed starting centroids as input. This ensures that all implementations returns the same correct result. The implementations are made as efficient as we found possible while still staying true to the algorithm.

The following programming language distribution is considered to be the

canonical implementation of the language in question and have been used for the benchmark, and the arguments if any used for compilation and execution of each implementation.

Clojure Clojure 1.6.0 (Java 8 Update 25)

Erlang Erlang/OTP 17.3

Execution: erl -eval main:start(). -noshell +A 32 -extra

F# F# 3.1 (.NET 4.5)

Haskell The Haskell Platform 2014.2.0.0

Compile: -threaded -O2

Execution: +RTS -N32

Scala Scala 2.11.4 (Java 8 Update 25)

4.6 Test results and Summary

The results from the benchmarking procedure are presented in this section. We have experimented with different dataset sample sizes and found that 32, 1,000, 10,000 and 50,000 samples sizes provided the most insightful image of how the various implementations react. The dataset with 32 samples is interesting because it shows the overhead of each implementation and the datasets of 1,000, 10,000 and 50,000 samples shows how the implementation scales up with data processing. The number of features per sample did not have significant effect on execution time compared to increasing the number of samples. We chose to keep the amount of features relatively small compared to the amount of samples because of the lack of effect on execution time and therefore not helpful in testing scalability. The amount of features was set to 100. Each dataset was generated to have three clusters and a copy of three samples was given as centroids. This means that the included dataset consists of 32, 1,000, 10,000 and 50,000 samples each of them with 100 features per sample. Visualisations of how the execution times were distributed can be seen in Appendix A.1. An overview of the implementations execution times can be found in Figure 4.4 and an overview of the speed-up can be seen in Figure 4.5. A more detailed graph representation of the execution times and speed-up can be respectively seen in Appendix A.2 and Appendix A.3.

The Clojure implementation shows some speed-up when executed on 32 samples up to 8 threads compared to using 1 thread. After 8 threads the speed-up is lost as the overhead of the additional communication with the addition of more threads. Even with larger datasets the overhead of adding more threads to the execution of the implementation does not seem to give

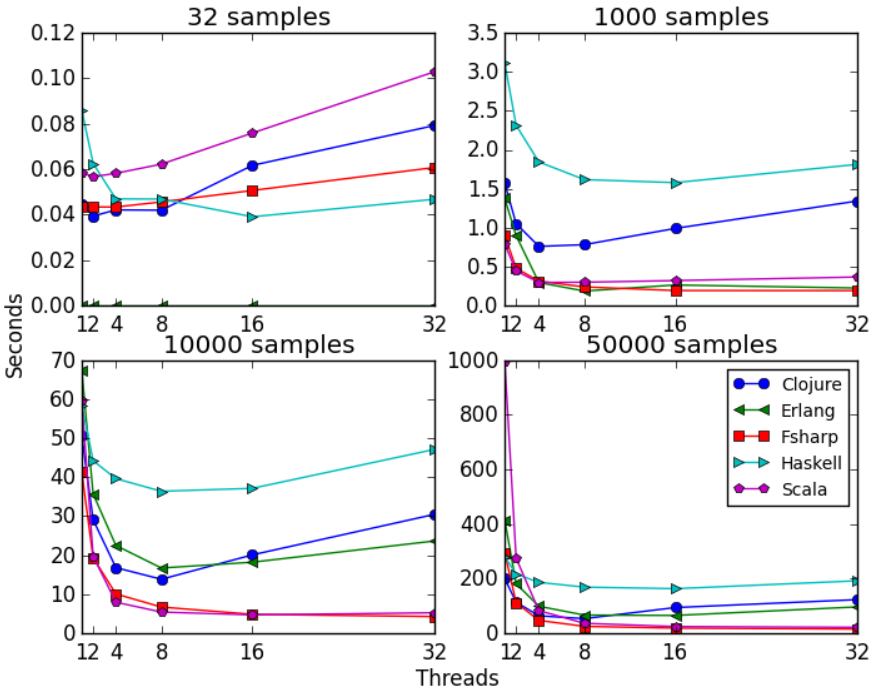


Figure 4.4: Execution times in seconds with the four datasets as input.

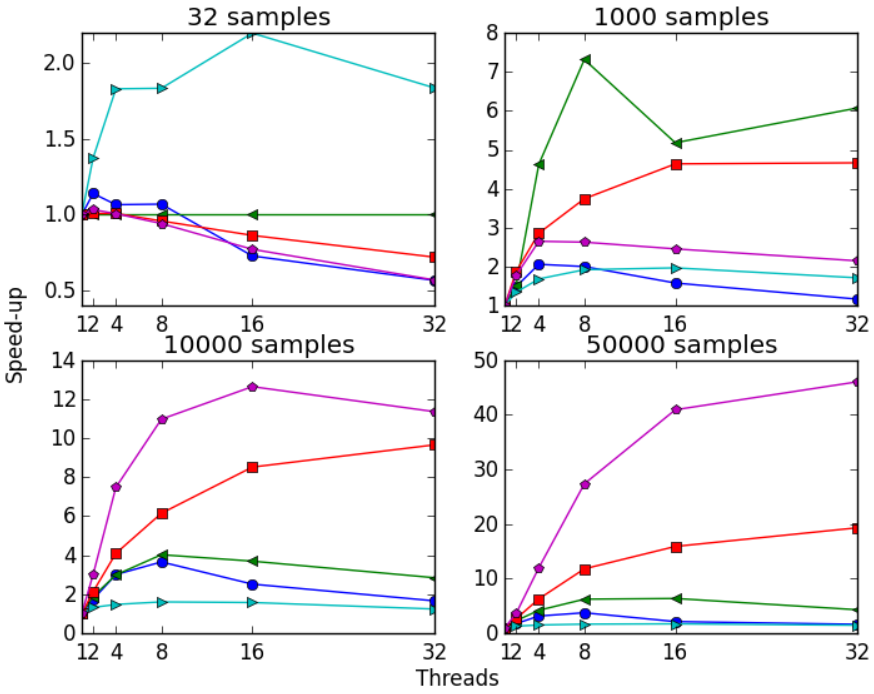


Figure 4.5: Speed-up based on the four datasets as input.

any speed-up at all. In the best case with 50,000 samples the speed-up is close to the same as with 32 threads as with 16 threads.

The Erlang implementation has the shortest execution time on 32 samples but the results look abnormal because they are so close to zero for all number of threads. There is no speed-up because the execution time is the same for all of the tested number of threads. This could indicate that Erlang's actors are very lightweight compared to the other implementations as there is no overhead of adding more actors to the system. As the amount of samples increases to 1,000 the Erlang implementation gain speed-up up until 16 threads where it suddenly drops before regaining speed-up when executing with 32 threads. The reason for this is that the execution time of the implementation with 16 threads have a slight increase in execution time from 8 threads and then a decrease in execution time with 32 threads. The underlaying reason for this behaviour is hard to pinpoint but it seems like the Erlang implementation does not scale well with more than 8 threads because the execution time either stagnates or increases which makes the speed-up decrease. This behaviour is also shown in the benchmarks with 10,000 and 50,000 samples.

The F# implementation has an expected decrease in speed-up with only 32 samples as more actors are added. When the sample size increases the implementation performs very well in terms of both speed-up and execution time as the execution time always decreases with the addition of additional actors. An exception to this is found when the implementation goes from 16 to 32 actors as the execution time stagnates when executed with 1000 samples as input. This indicates that the overhead of adding 16 more actors to this relatively small sample set were greater than the speed-up gained. Besides this one exception in speed-up and execution time, does F# scales favourably as additionally actors are added to the execution.

The execution time of the Haskell implementation is excellent with only 32 samples but with the larger sample sizes the implementation is slower than all of the other implementations. It is noteworthy that the Haskell implementation is the only to gain a speed-up with only 32 samples.

The Scala implementation shows clear overhead when executing on 32 samples that increases with the number of actors. The overhead of the Scala implementation is apparent after 4 actors on 1,000 samples and after 16 actors on 10,000 samples because its speed-up decreases after these points. The Scala implementation does have a very high execution time with only 1 actor with 50,000 samples which is why its speed-up is equally large.

The Erlang and Clojure implementation did not scale well as their speed-up only decreased after 8 threads. The Scala implementation showed good speed-up with 50,000 samples but showed decreased speed-up with 10,000 samples after 16 threads and with 1,000 samples after 2 threads. Furthermore the Scala implementation does have a very high execution time with 50000 samples using only 1 actor and first becomes reasonable after using

four actors. The F# implementations showed excellent speed-up and execution time across all four sample sizes. This means that the implementation that scales across multiple threads are F# and Scala.

CHAPTER 5

Usability

This chapter contains a usability analysis of the different advanced concurrency models implemented in the chosen programmings languages. Section 5.1 presents the criteria for the usability analysis followed by Section 5.2 which justifies the use of and describes the Santa Claus Problem as the base of the usability analysis. The STM based implementations is presented and discussed in Section 5.3, followed by a presentation and discussion of the actor based implementations in Section 5.4. A summary is found in Section 5.5.

5.1 Criteria

An usability analysis of programming languages have a less solid foundation compared to that of performance benchmarking in academia. We want to evaluate how simple, readable and efficient the chosen implementation programming languages are compared to each other. Two metrics where chosen to directly measure aspects of the implementation created in the chosen programming languages.

- **Lines of code (LOC):** How many LOC the implementation consists of while ignoring comments.
- **Development time:** How fast the implementation was created (low, medium, high).

LOC is used as a metric because that it makes our results comparable as it is widespread in academia to indicate both simplicity and readability where it have been used to measure programming languages. On the down side, LOC have been used to represent an even wider range of values than both simplicity and readability and there is no clear coherent understanding on what exactly LOC represent. [72]

By taking this into account we have chosen to use LOC as criterion because the ability to compare with the rest of academia is valuable and this continued use throughout academia could be because of a lack of alternatives

to measure simplicity and readability. Furthermore to reduce the impact of coding style on the result and make simpler and more comparable implementations of the Santa Claus Problem. We have chosen not to count comments and empty lines as LOC because they are not a direct part of the program. We have chosen to format our implementations using each languages canonical style guide and have each solution verified by the linting tools available to ensure that the implementations are respecting each language communities best practices. The style guides and tools where selected based on a Internet search with precedence given to guide and tools from the languages own homepage. With conflicts between recommended practices a choice were made based on the subjective opinion of the authors.

Development time is a measurement that indicates how simple and efficient a programming language is because it captures powerful language constructs that simplifies the implementation [25]. However, it is also highly dependent on the developer's prior knowledge in the implementation language and problem. A solution to a problem takes less time as a developer becomes more familiar with the language and problem. To make development time more comparable it have been generalised into three values indicating the relative development time between the different solutions and representing a half day, a full day, or multiple days of work respectively. This reduces the impact of the developers skill and their prior knowledge about the problem as this is taken into account by determining the measure by comparing it with the other implementations. The developers have gathered prior knowledge about the unfamiliar languages before starting the implementations and had practiced by developing the performance benchmarks.

A subjective discussion of the pros and cons of each languages implementation of the given concurrency model will be presented in addition to the two metrics because existing work have shown that hard metrics are insufficient in giving a complete picture of the usability of a programming language [29]. This is to discuss the result of the metrics, how well the implementation matches the theoretical model and how the implementation was perceived to allow for the development of the Santa Claus problem.

5.2 The Santa Claus Problem

We have chosen to analyse the usability of the chosen programming languages by implementing the Santa Claus problem [28]. The reason for choosing this as a base for our usability test as the small size of the problem allowing implementation of it to be produced in the time frame of the project, while still forcing the solution to include synchronisation between multiple threads of execution and three different tasks. Moreover the problem is well-known and has been implemented in different languages with concurrent capabilities, creating a context for our language evaluation [28, 73, 27]. An overview

of the Santa Claus problem can be seen in Figure 5.1, followed by the full description of the problem from [28].

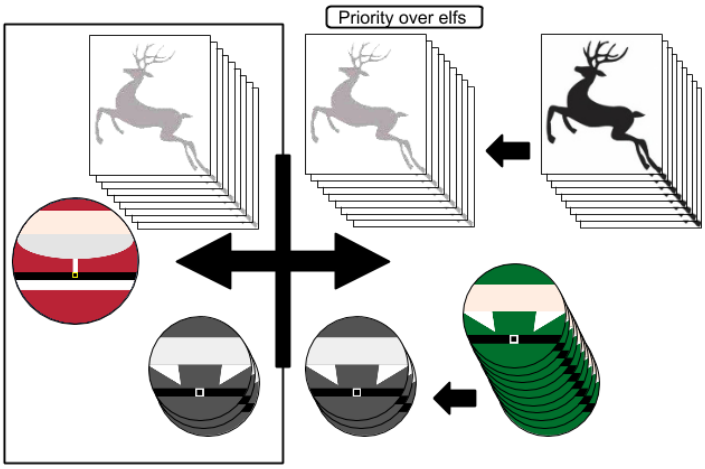


Figure 5.1: Overview of the Santa Claus problem.

"Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh."

5.3 Software Transactional Memory

The STM based implementation works by having both Elfs and Reindeers trying to go to Santa when they need help and then block and wait for Santa to awake if there currently are not enough room. The implementation primarily consists of three functions `worker`, `santa`, and `gotoSanta`, the first function is used to represent both Elfs and Reindeers as they share the same behaviour but with different parameters. The second function is used to represent Santa and spends most of the time blocking the thread executing it and only runs after it have been awoken by either the Elfs or the Reindeers, when this happens help is given to entity that needs it with precedens to the Reindeers and afterwards the thread block. The function `gotoSanta` handles the operation of sending either a Reindeer or Elf to Santa and wake him up if enough have arrived, for the Reindeer or Elf being allowed to go to Santa is the length of the list for the respective type of worker checked to ensure there are room for them, and if there are room then if the Worker already have been added. If either of these checks fail are the transaction forced to fail and the thread in question blocked until Santa awakes as the invariance cannot change before he does.

5.3.1 Clojure

- Line Of Code: 79
- Development Time: High
- Tools: eastwood, kibit
- Styleguides:

  `https://github.com/bbatsov/clojure-style-guide`

Implementing the Santa Claus problem in Clojure was complex due to the lack of constructs for blocking threads until an event has occurred in the language, in addition to some form of control mechanism for prematurely terminating a STM transaction by the developer. The lack of a dedicated abort mechanism for STM transactions means that Java exceptions with `try/catch` blocks were used to abort transactions forcefully if the queue in front of Santas house is full or a worker is already present in the queue. The worker is in the Clojure implementation represented as two functions, first `worker` and `do-work` both shown as Listing 5.1, the use of two functions is to simplify rerunning the STM transaction if any of the invariance checks fails without having the thread go back to sleep again. The function `worker` defined on line 15 makes the thread sleep a random interval before calling the `do-work` function which contains the logic needed for a worker. `do-work` tries to add the Elf or Reindeer it represents to the queue at Santas house

if there are space as shown on line 6, if it adds itself to the queue then the function is called recursively and the thread is then blocked on line 12 until Santa awakes and processes the queue. If the worker is already present in the queue, an exception is thrown to terminate the STM transaction, this is done using Clojure's interoperation with Java as it allows for a program to block on an `Object` until the method `notify` is called using the method `wait` using Java objects built in monitor. The operations in the worker function is encapsulated in both a `locking` function on line 3 which takes the monitor of its argument object and releases it when leaving the function block, and a `dosync` on line 4 defining a STM transaction, both are necessary as Clojure's `ref` type cannot be altered outside of an transaction and the monitor is necessary for preventing the Santa thread from waking before the transaction is finished and the `ref` updated. This is due to the `notify` method call on the Santa thread to wake Santa takes effect immediately and not after the STM transaction commits, depending on how the threads are interleaved this can allow for Santa to be awoken when only two elfs or eight reindeers are in the queue as the last one has yet to commits its STM transaction, forcing the use of both a monitor and STM.

```
1  (defn do-work [wid maxSleep queue-ref maxQueue]
2    (try
3      (locking santa-sleep-lock
4        (dosync
5          (let [current-queue-length (count (deref queue-ref))]
6            (if (< current-queue-length maxQueue)
7              (goto-santa queue-ref wid)
8              (throw (IllegalArgumentException.)))
9            (when (== (inc current-queue-length) maxQueue)
10             (wake-santa)))))
11     (catch IllegalArgumentException iae
12       (block-worker)
13       (do-work wid maxSleep queue-ref maxQueue))))
14
15 (defn worker [id maxSleep queue-ref maxQueue]
16   (sleep-random-interval maxSleep)
17   (do-work id maxSleep queue-ref maxQueue)
18   (recur id maxSleep queue-ref maxQueue))
```

Listing 5.1: The Worker Function

The `sleep-santa` primary function of the function shown as Listing 5.2 is to make the Santa thread sleep using his own thread's object monitor, however due to the lack of support for Java method calls in Clojure's STM

implementation an extract monitor are used on line 6. This is the same monitor that protects the STM transaction in `worker` and prevents Santa from processing the elfs or reindeers before the last worker's transaction is complete and the worker has released the `santa-sleep-lock` monitor.

```
1 (defn sleep-santa []
2   (let [santa-thread (deref santa-thread-ref)]
3     (locking santa-thread
4       (notify-all-workers)
5       (.wait santa-thread))
6     (locking santa-sleep-lock)))
```

Listing 5.2: The Santa Sleep Function

If the queue has space for another worker the function `goto-santa` is executed shown as Listing 5.3, this function checks if that particular worker already is in the queue and if not adds it. This check is necessary as all blocking workers are restarted when Santa awakens, so Reindeers blocked in the Reindeer queue will try to add themselves again when Santa helps a set of Elfs. Alternatively two different `Objects` could be used to block workers, one for Elfs and one for Reindeers, however this would subjectively add more complexity to the program than the current solution with an extra check as the block and notify functions would be added to two different functions instead of containing a single `if` check in the `goto-santa` function. If the element currently does not reside in the queue is it added, and the Santa thread is awoken if enough Elfs or Reindeers are present.

```
1 (defn goto-santa [santas-door id]
2   (let [queue-at-door (deref santas-door)]
3     (if (elem? id queue-at-door)
4       (throw (IllegalArgumentException.))
5       (alter santas-door conj id))))
```

Listing 5.3: The Goto Santa Function

The primary function of the `santa` function shown as Listing 5.4 is to sleep and process the Reindeer and Elf queue once it have been awoken by one of those threads respectively. After each set of workers have interacted with Santa, with precedence to the Reindeers as per the problem definition, the `notify-all-workers` function is called again using Clojure's interoperation with Java to notify all the worker threads blocking after being denied entrance to Santa since he is not asleep.

```
(defn santa []
  (sleep-santa)
  (dosync
    (let [reindeers (deref reindeer-queue-ref)
          elfs (deref elf-queue-ref)]
      (if (== (count reindeers) 9)
        (work-santa reindeers reindeer-queue-ref
                    "delivering presents with reindeers:")
        (work-santa elfs elf-queue-ref "helping elfs:"))))
  (recur))
```

Listing 5.4: The Santa Function

Clojure's implementation of STM is in general very similar to the theoretical model, with the function `dosync` defining the beginning and end of an atomic block in which all or no operations are performed, with transactions being retried if they fail. Clojure's STM implementation however does not provide safe operations across multiple threads for shared data, created using the `var` type in Clojure. Instead a transactional safe variable type have been added named `ref`, which can be shared between multiple threads and accessed safely inside an `dosync` block. In accordance with Clojure's dynamic nature, operations that are required to be in a `dosync` block are not checked at compile time, however runtime exceptions are produced if such functions are used outside an atomic block. Some functions have alternative implementations based on if they are used inside an atomic block or not, such as the function `deref` which are used for accessing the contents of a `ref` variable. Inside a transaction the current transactional value is returned, while outside the last committed value is returned. Such dual nature prevent the checking if functions intended to be part of an atomic block are inside the block or simply operates using their outside of block version.

While Clojure provides constructs for ensuring atomic operations on shared state, it does supply no operations for halting a threads operation either based on time or an external event. This forces developers to use Clojure's ability to interoperate with Java methods such as for example `Thread.sleep()`, `Object.wait()` and `Object.notify()` for time and event based blocking respectively. However while such operations worked as expected and simple to utilise due to some special syntax provided by Clojure, it was still unnatural to use due to nested structure of objects compared to the more flat list based data structures used in Clojure, and forced the use of Java monitors in `synchronized` blocks through the function `locking` instead of an STM block as accustomed in Clojure.

In addition to the lack of support for controlling a thread the implementation does also lack support for controlling the STM transactions, such as

forcing retries based on an invariance in the transaction. And while one can ague that any functionality that allows control over the transaction by the developer goes against the model as the developer's task should be to just mark sections with shared data as STM blocks, forcing a thread to fail a transaction intentionally simplify some function as seen in Listing 5.3, as transactions then not only can be forced to retry until the data is available but available in a state that is usable for the operation consuming it. While such behaviour could be emulated by reading the data in one block, checking the invariance and then entering another atomic block, the data could have been changed between the two blocks. However such a system would put increased pressure on the runtime as simply retrying a transaction that checks a known invariant, before any changes have been performed to set invariance is wasting clock cycles.

5.3.2 Haskell

- Line Of Code: 56
- Development Time: Medium
- Tools: Haskell Style Scanner, hlint, ghc -wall
- Styleguides:

  `https://www.haskell.org/haskellwiki/Programming_guidelines`

  `https://github.com/tibbe/haskell-style-guide`

Implementing the Santa Claus in Haskell was helped by the inclusion of the function `retry` in the language's STM implementation, and the language's type system that prevents accesses to shared transactional variables by encapsulating such operation in the `STM` monad. The `retry` function allows developers to purposely retry a STM transaction, for example based on the current value of a shared variable. This allows STM in Haskell to not only be used for ensuring a block is executed atomically, but also to synchronise operations between threads as a thread can call `retry` if no data is ready for consumption allowing the Haskell runtime to block the thread until a change is detected in one of the shared transactional variables read inside the transaction [74].

The `worker` function in the Haskell implementation of the Santa Claus problem, shown as Listing 5.5 is executed by each one of the Elf and Reindeer threads in an infinite recursion, and is implemented using `retry` to synchronise the worker threads by checking if there is room for another elf or reindeer in the respective queue waiting for Santa. This is done using an if check at line 6 that check if the queue contains less then three individuals for the Elf's or nine individual for the Reindeer are the function `gotoSanta`

called on line 6, else if the queue is already full is the worker forced to retry the transaction created by the use of the function `atomically` with the `do` block as input on line 4, and wait until the queued individuals have received help from Santa. When either enough Reindeers or Elfs have been added to the queue Santa is awoken by calling the `wakeSanta` function with the boolean variable `sleepTVar` that the workers thread's and the Santa thread shares as argument, on line 8.

```
1 worker :: TVar [Int] -> Int -> Int -> Int -> TVar Bool -> IO ()
2 worker queue maxQueue wId maxSleep sleepTVar = do
3     sleepRandomInterval maxSleep
4     atomically (do
5         queueLength <- lengthOfTheQueue queue
6         if queueLength < maxQueue then gotoSanta queue wId else retry
7         when (queueLength + 1 == maxQueue) \$ wakeSanta sleepTVar)
8     worker queue maxQueue wId maxSleep sleepTVar
```

Listing 5.5: The Worker Function

Another necessary check before the worker can enter the queue is performed on line 4 in the `gotoSanta` function shown as Listing 5.6, the check to determine if a worker has already entered the queue. The arises from not blocking workers after they are allowed to enter the queue, as it was subjectively deemed simpler to simply block the worker threads when they tried to enter the queue a second time, compared to introducing an additional shared variable for blocking worker threads added to the queue. While this method has the disadvantage of having these threads awoken and blocked again when they detect that they already are present in the queue each time a new worker thread is added to the queue due to the runtime detecting a change in a variable read before the `retry` function was called, this method of blocking the threads allow for a single function call added in the program instead of function blocking the threads in `worker`, see Listing 5.5, and a function call unblocking the workers in `santa`, Listing 5.7.

```
1 gotoSanta :: TVar [Int] -> Int -> STM ()
2 gotoSanta santasDoor wId = do
3     queueAtDoor <- readTVar santasDoor
4     if wId `elem` queueAtDoor
5         then retry
6         else writeTVar santasDoor (wId : queueAtDoor)
```

Listing 5.6: The Goto Santa Function

The `santa` function shown as Listing 5.7, is executed by a single thread and spends most of the time in a blocked state while waiting for the Elfs and Reindeers to need help. The thread is blocked by a call to `sleepsanta` with the boolean shared transactional variable `sleepTVar` on line 3, the variable is shared with the worker threads to allow for the Elfs and Reindeers to wake Santa by flipping the value of the variable. After Santa have awoken the reindeer and elf queues are read from the shared variables on line 4 and 5, followed by Santa printing a short message and sleeping five seconds to emulate working on a task by calling the function`workSanta` that have been excluded from the rapport due to its simplicity.

```
1  santa :: TVar [Int] -> TVar [Int] -> TVar Bool -> IO ()
2  santa elfQueue reindeerQueue sleepTVar = do
3      atomically \$ sleepSanta sleepTVar
4      elfs <- readTVarIO elfQueue
5      reindeers <- readTVarIO reindeerQueue
6      if length reindeers == 9
7      then workSanta reindeers reindeerQueue
8          "ho ho delivering presents with reindeers: "
9      else workSanta elfs elfQueue
10         "ho ho helping elfs: "
11     santa elfQueue reindeerQueue sleepTVar
```

Listing 5.7: The Santa Function

Forcing the Santa thread to sleep indefinitely by developing Mutex using a single boolean variable named `sleepTVar` shared between the `santa` and `worker` functions, and the `retry` function in Haskell's STM implementation. Two functions were developed to encapsulate the lock / unlock operations namely `sleepSanta` and `wakeSanta` shown in Listing 5.8, the variable `sleepTVar` starts with the value `False` forcing the `sleepSanta` function to block the Santa thread by executing the `retry` function on line 4. Once enough workers have been added to the queue the `wakeSanta` function is called by the last worker, here the shared variable `sleepTVar` is flipped to `True` on line 9 if Santa is asleep, this change to the shared variable unblocks the Santa thread which now can enter the `True` clause of the if statement on line 4, set `sleepTVar` to `False` so the thread will block when it calls `sleepSanta` again. If Santa already had been awoken by the other type of workers the if statement on line 9 would evaluate to False, forcing the worker to try to wake Santa retry the transaction and wait for `sleepTVar` to change when Santa goes to sleep again.

```
1 sleepSanta :: TVar Bool -> STM ()
2 sleepSanta sleepTVar = do
3     sleepBool <- readTVar sleepTVar
4     if sleepBool then writeTVar sleepTVar False else retry
5
6 wakeSanta :: TVar Bool -> STM ()
7 wakeSanta sleepTVar = do
8     sleepBool <- readTVar sleepTVar
9     if sleepBool then retry else writeTVar sleepTVar True
```

Listing 5.8: The Sleep Santa Function

Using STM in Haskell was in our opinion a simple method to ensure atomic reads and writes of shared data, and provided confidence on compile time that all operations on shared transactional data where performed in a transactional block as the system would have prevented it otherwise. In this regard the implementation followed the theoretical model rather close as operations not part of a monad or part of the STM monad could be encapsulated in Haskell's `do` notation and the entire supplied block supplied to the `atomically` function for execution, as shown in Listing 5.5 line 4. In addition the function `retry` and `orElse`, a function for executing an alternative operation if a transaction fails, provided constructs for performing synchronisation between threads at a single point in the program instead of having both a blocking operation at the consumer end and a unblocking operation at the producer.

However while these operations provides some control, reasoning about when the runtime system unblocks a thread and retries a transaction can be very complex. This problem might be worse by having the behaviour be performed by the runtime and not explicitly by the programmer using locks, this lead to cases of threads being mistakenly awaken elsewhere in the program due to a change in a transactional variable used in multiple parts of the program. While the thread would simply be blocked if the invariance that forced the original call to `retry` still holds and the behaviour follows the STM models optimistic method of execution it could lead to wasted CPU cycles if multiple threads were waiting for a set of data only one could consume as one would get the data and the rest would execute only to be forced back to sleep. An extension to Haskell's STM implementation that allowed the developer to specify how many threads where allowed to be awaken for retrying a transaction at the time could be a possible solution.

5.4 Actor

The actor based implementation introduces the `SantasHouse` actor which represents the door into Santa's house. This is needed in order not the have the queue for waking up Santa in the Santa actor. Instead Santa is woken by the `SantasHouse` actor whenever the queues for elfs or reindeers are full. The queue for elfs and reindeers have different maximum sizes and whenever one of the queues are full Santa is messaged. If the `SantasHouse` actor both has messages from reindeers and elfs in its mailbox it will prioritise the messages and handle the messages from the reindeers first. In order to wake up Santa the queue for els must contain three elfs and the queue for reindeers must contain nine reindeers. An elf or reindeer is an instance of a `Worker` actor since their implementation otherwise would be identical except from their identifier and maximum sleep time. When an elf or reindeer needs Santa's help or has come back from vacation they will message the `SantasHouse` actor and be added to the appropriate queue. When Santa is finally awakened from his sleep he will help the reindeers first if their queue is full and after that help the elfs if their queue is full.

5.4.1 Erlang

- Line Of Code: 54
- Development Time: Medium
- Tools: None (erl_lint - not used due to lack of documentation)
- Styleguides:

  `http://www.erlang.se/doc/programming_rules.shtml`

  `https://github.com/inaka/erlang_guidelines`

The simplicity of the Erlang implementation of the Actor model matches the simplicity of Erlang itself. An actor in Erlang is created by `spawn`ing a recursive function where the arguments of the functions serves as the state of the actor. It is worth noting that if the recursive function in not called after handling a received message the actor will stop. This might seem very intuitive but can cause some headache to the developer.

The `Worker` actor implemented in Erlang can be found in Listing 5.9. When the `receive` block is entered the actor will wait for the `leave` message from the `Santa` actor. After that the `Worker` will once again sleep a maximum of `SleepTime` seconds and message `SantasHouse` to get help from Santa.

```
1  worker(Type, ID, SleepTime, SantasHouse) ->
2    sleep_random_interval(SleepTime),
3    SantasHouse ! {Type, ID, self()},
4    receive
5      leave ->
6        worker(Type, ID, SleepTime, SantasHouse)
7    end.
```

Listing 5.9: The `Worker` actor implementation in Erlang

Worth noting about the implementation of the `Santa` actor found in Listing 5.10 is that since the precedence is done by the `SantasHouse` actor and the nature of actors, no precedence is needed in the `Santa` actor itself since the order of messages sent is maintained as the order received. When either the reindeers or elfs have been helped the `Santa` actor will send a `leave` message to all of them, triggering them all to start sleeping again.

```
1  santa() ->
2    receive
3      {reindeers, IDS, PIDS} ->
4        io:format("Santa: ho ho ... with reindeers: ~w\n",
5          [IDS]),
6        sleep_random_interval(5),
7        lists:foreach(fun(PID) -> PID ! leave end, PIDS);
8      {elfs, IDS, PIDS} ->
9        io:format("Santa: ho ho helping elfs: ~w\n", [IDS]),
10       sleep_random_interval(5),
11       lists:foreach(fun(PID) -> PID ! leave end, PIDS);
12   end,
13   santa().
```

Listing 5.10: The `Santa` actor implementation in Erlang

In the `SantasHouse` actor implementation in Listing 5.11 the queues for reindeers and elfs are checked before any message handling is done in lines 2 to 10. First the queue for reindeers and hereafter the one for elfs. If any of the queues are full, the `Santa` actor will be awaken to take care of the problem. The message handling here is quite simple as it just adds the elf or reindeer that messaged the `SantasHouse` actor to the correct queue and calls it self with the new queue, forcing a new check if any of the queues are full.

```
1  santas_house(Santa, Elfs, Reindeers) ->
2    if
3      length(Reindeers) == 9 ->
4        {IDS, PIDS} = lists:unzip(Reindeers),
5        Santa ! {reindeers, IDS, PIDS},
6        santas_house(Santa, Elfs, []);
7      length(Elfs) == 3 ->
8        {IDS, PIDS} = lists:unzip(Elfs),
9        Santa ! {elfs, IDS, PIDS},
10       santas_house(Santa, [], Reindeers);
11     true ->
12       receive
13         {elf, ID, PID} ->
14           santas_house(Santa, [{ID, PID} | Elfs], Reindeers);
15         {reindeer, ID, PID} ->
16           santas_house(Santa, Elfs, [{ID, PID} | Reindeers])
17       end
18   end.
```

Listing 5.11: The `SantasHouse` actor implementation in Erlang

Compared to the theoretic evaluation of the Actor model in Section 3.3.1 the Erlang implementation is very loyal. The model maintains the very implicit concurrency that the Actor model also does by hiding all concurrency constructs from the developer, partly because of the model itself where all communication between actors happens in asynchronous messages from one actor to another, not allowing any sharing of memory between the actors, this encapsulation the Erlang implementation respects.

5.4.2 F#

- Line Of Code: 54
- Development Time: Low
- Tools: FSharpLint
- Styleguides:

  `http://fsharp.org/specs/component-design-guidelines/`

The `MailboxProcessor`s in F# are easy to use this this is also reflected in the low development time. The model is very close to the theoretic model evaluated in Section 3.3.1 and showing the same characteristics which is very implicit concurrency as a result of the encapsulation between actors.

An actor in F# consists of a single recursive function that holds the state of the Actor in its arguments. This simple concept of state respects the concept of functional programming of having only immutable variables. For example when having a list of items, when a change to the list is done, a new list is returned instead of changing the data itself.

The implementation of the `Worker` actor in F# can be found in Listing 5.12. The worker will not be receiving any messages from other actors which is why no message handling is needed. Besides the lack of message handling in this implementation it is worth noticing is the need for the `async` and the `while true do` blocks in order to function as an actor. Instead of blocking when waiting for messages to handle, the F# implementation instead utilises the `PostAndReply` method on the `MailboxProcessor` class which will wait for a response before continuing the execution, ultimately restarting the `while true do` loop.

```
1  let Worker workerType id maxSleepTime =
2    MailboxProcessor.Start(fun inbox ->
3      async {
4        while true do
5          SleepRandomInterval(maxSleepTime)
6          SantasHouse.PostAndReply(fun replyChannel ->
7            (workerType, id, replyChannel))
8        }
9    )
```

Listing 5.12: The `Worker` actor implementation in F#

Compared to the other actor implementations in Scala and Erlang the F# implementation of the Santa actor found in Listing 5.13 is about the same length but the use of `AsyncReplyChannel`s is somewhat confusing to new programmers as it is a concept which exclusive to F#. The pattern matching is used in the Santa actor and the syntax is very clear and understandable but the need for the `async` and `while true do` block still seems like an artifact of having the actor work on top of the .NET platform with no direct support for the actor behaviour.

```
let Santa = MailboxProcessor.Start(fun inbox ->
  async { while true do
    let! (workerType, ids, replyChannels) = inbox.Receive()
    match workerType with
    | Reindeer ->
      printfn
        "Santa: ho ho ... with reindeers: %A" ids
    | Elf -> printfn "Santa: ho ho helping elfs: %A" ids
    SleepRandomInterval(5)
    for (replyChannel : AsyncReplyChannel<_>) in replyChannels
      do replyChannel.Reply()
  }
)
```

Listing 5.13: The `Santa` actor implementation in F#

The F# implementation of `SantasHouse` found in Listing 5.14 takes advantage of the way that the state of the actor is stored. Before handling any messages in its mailbox it first checks whether or not it is time for messaging Santa. These checks are done before the actor is blocked waiting for new messages meaning that as soon as a message is received, it is handled and before handling any other messages it is once again checked to message Santa or not. This solution is quite more elegant than the one found in the Scala implementation of `SantasHouse` in Listing 5.17.

```
let SantasHouse = MailboxProcessor.Start(fun inbox ->
  let rec loop (reindeers : List<_ * _>) (elfs : List<_ * _>) =
    async {
      if reindeers.Length = 9 then
        let (ids, replyChannels) = List.unzip(reindeers)
        Santa.Post(Reindeer, ids, replyChannels)
        return! loop [] elfs
      elif elfs.Length = 3 then
        let (ids, replyChannels) = List.unzip(elfs)
        Santa.Post(Elf, ids, replyChannels)
        return! loop reindeers []
      let! msg = inbox.Receive()
      match msg with
      | (Reindeer, id, replyChannel) ->
        return! loop ((id, replyChannel) :: reindeers) elfs
      | (Elf, id, replyChannel) ->
        return! loop reindeers ((id, replyChannel) :: elfs)}
  loop [] [])
```

Listing 5.14: The `SantasHouse` actor implementation in F#

The F# implementation follows the theoretical Actor model by maintaining by hiding all concurrency constructs from the developer, partly because of the model itself where all communication between actors happens in messages from one actor to another, not allowing any sharing of memory between the actors. However F# provides additional both synchronous message passing and built-in request / reply functionality in their implementation of the Actor model. The addition of synchronous message passing allows for the development of actors which expects messages to arrive in a specific order, which in turn makes it simpler to create a deadlock due to actors waiting for a responds to a synchronous message, a problem less present in a system that only supports asynchronous message parsing as the developer have no control over the order of which messages arrive.

5.4.3 Scala

- Line Of Code: 76
- Development Time: Medium
- Tools: scalastyle, scalac -Xlint
- Styleguides:

  `http://docs.scala-lang.org/style/`

The pattern matching used in the Scala implementation of the Santa Claus problem is a little bit different than the one used in the Scala implementation of K-means. In the Santa Claus problem a tuple with an enum as its first element is used whereas in the K-means implementation case objects are used. Choosing the one over the other is merely a matter of subjective opinion of the syntax as they are equal in their expressive power. More time were used understanding the Actor model implementation in Scala than with the other languages. In Scala 2.11.0 and forward the Actor model implementation is deprecated and replaced with the Akka implementation of the Actor model [75]. The Akka implementation requires some set-up of the actor system prior to starting it, which took some time to understand compared to the more simple approach in for example Erlang.

The implementation of the common `Worker` actor in Scala can be found in Listing 5.15. One thing to note is the `self ! Unit` in the constructor of the actor which is needed for the actor to start sleeping, this is needed in Scala since `receive` is first called when a message is received.

```
1  class Worker(workerType: WorkerType.WorkerType, id: Int,
2    maxSleepTime: Int, santasHouse: ActorRef) extends Actor {
3    //Forces the first work iteration and going to Santa
4    self ! Unit
5
6    def receive = {
7      case Unit =>
8        Main.sleepRandomInterval(maxSleepTime)
9        santasHouse ! (workerType, id)
10   }
11 }
```

Listing 5.15: The `Worker` actor implementation in Scala

In order to force the handling of reindeer messages first the pattern matching of the messages is split up into two different `case` expressions compared to an actor where all messages have the same precedence. The first case expression handles reindeer messages whereas the second handles elf messages.

```
class Santa extends Actor {
  def receive = {
    case (WorkerType.Reindeer, ids: List[Int],
        actorRefs: List[ActorRef]) =>
      println(s"Santa: ho ho ... with reindeers: \$ids")
      Main.sleepRandomInterval(5)
      actorRefs.foreach(ref => ref ! Unit)
    case (WorkerType.Elf, ids: List[Int],
        actorRefs: List[ActorRef]) =>
      println(s"Santa: ho ho helping elfs: \$ids")
      Main.sleepRandomInterval(5)
      actorRefs.foreach(ref => ref ! Unit)
  }
}
```

Listing 5.16: The `Santa` actor implementation in Scala

The Scala implementation of the `SantasHouse` actor can be found Listing 5.17. Since the developer in Scala do not directly recursively call the `receive` function he/she is forced to have mutable variables in the class to change the content of the queues. This is in contrast to the way state is normally handled in actors as a product of calling the `receive` function recursively with changed parameters. Instead the Scala implementation is required to use the method `enterQueue` to check if it is time to message Santa or the elf or reindeer only should be added to the appropriate queue.

```
class SantasHouse(santa: ActorRef) extends Actor {
  var elfs = List[(Int, ActorRef)]()
  var reindeers = List[(Int, ActorRef)]()

  def enterQueue(workerType: WorkerType.WorkerType, id: Int,
    actorRef: ActorRef, queue: List[(Int, ActorRef)],
    maxQueue: Int): List[(Int, ActorRef)] = {
      val updatedQueue = (id, actorRef) +: queue
      if (updatedQueue.length == maxQueue) {
        val (ids, actorRefs) = updatedQueue.unzip
        santa ! (workerType, ids, actorRefs)
        return List[(Int, ActorRef)]()
      }
      updatedQueue
  }

  def receive = {
    case (WorkerType.Reindeer, id: Int) =>
      reindeers = enterQueue(WorkerType.Reindeer, id,
        sender, reindeers, 9)
    case (WorkerType.Elf, id: Int) =>
      elfs = enterQueue(WorkerType.Elf, id, sender,
        elfs, 3)
  }
}
```

Listing 5.17: The `SantasHouse` actor implementation in Scala

The Scala implementation follows the theoretical Actor model but also differs the most from the theoretical model of three different implementations. As it not only provides both synchronous message passing and built-in request / reply functionality in its implementation of the Actor model, but also because it implements actors as `Classes` allowing actors to extract and send their own `this` pointers in a message breaking encapsulation. The addition of synchronous message passing allows for the development of actors which expects messages to arrive in a specific ordering, which in turn makes it simpler to create a deadlock due to actors waiting for a responds to a synchronous message, a problem less present in a system that only supports asynchronous message parsing as the developer have no control over the order of which messages arrive.

5.5 Summary

In summary each different language and the implementation of their respective concurrency models, each have strengths and weaknesses. First the harder metric points LOC and development times are presented in Table 5.1. The table shows that the most complex version to develop, namely the Clojure version, also became the longest and took the longest time. This was due to the lack of constructs for retrying STM transactions and blocking threads in Clojure, forcing the use of Java constructs to archive a complete implementation that blocked Elfs and Reindeers until Santa had finished helping those already in the queue. As Haskell contains such constructs the implementation were simpler to create. Blocking threads is a part of Haskell's own STM retry mechanism as transactions only are retried if changes are done to variables in the transaction. The constructs that Haskell supports but Clojure lacks is the `retry` and the `orElse` which allows the developer to retry the current transaction or try a different transaction if the first transaction fails.

The actor based implementations generally took shorter time to develop and was shorter due to synchronisation of threads are an inherit part of the Actor model. The size of the Scala version is an exception to this due to the verbose syntax used for actors as they are implemented as classes deriding from the superclass `Actor` instead of functions like in Erlang and F#. Another difference between the Actor model implementations was how request-respond communication was handled by each language. Erlang contained no such functionality and two actors could only communicate if the actors explicitly exchanged ids, F# have a function messages without an expected responds named `Post` and four different send functions if the actor wants a reply with the four function named `PostAndReply`, `PostAndAsyncReply`, `PostAndTryAsyncReply`, and `TryPostAndReply`, each function representing a different combination of if the function should block until a reply is received and how the actor should react if a reply is not received before a time-out is reached. Even through the Erlang implementation of the Actor model is very loyal to the theoretic model, the lack of request-respond functionality is by the authors seen as a disadvantage compared to the implementations in F# and Scala.

	Languages				
	Clojure	**Erlang**	**F#**	**Haskell**	**Scala**
LOC	79	54	54	56	76
Time	High	Low	Low	Medium	Medium

Table 5.1: LOC and development time

The Actor model is in general easy to use, the implementations are true to the theoretic model and shows the same characteristics. The use of STM however is not as easily utilised for concurrent programming as the synchronisation and blocking of threads are still something that needs to be handled by the developer him/herself. Comparing the two concurrency models in terms of development time and LOC does not conclude one model superior to the other. However it is worth noting that the only implementation which has a development time of *High* is the Clojure implementation using STM. Furthermore the only implementations to get the development time classification of *Low* is the two actor implementations written in Erlang and F#. In regards to the size of the implementations there are no significant difference, both of the concurrency models have examples of both short and longer implementations of the Santa Claus Problem.

When comparing the characteristics of the theoretic concurrency models found in Sections 3.2.1 and 3.3.1 is very alike the actual implementations found in the languages used in this project. The STM implementations suffers from a lack of implicit and automatic thread synchronisation. The problems experienced when implementing the Santa Claus Problem in Haskell and Clojure is all related to thread synchronisation, therefore it seems that STM in its current form is not the solution to easy concurrent programming. The use of Actors removes the need for explicit thread synchronisation as this synchronisation in an inherent part of the model it self. As mentioned in Section 3.3.1 the code needs to be organised into actor for it to be concurrent but with the Santa Claus Problem this organisation of the code is very straight forward as the different entities and behaviour of the problem is easily split up into actors.

CHAPTER 6

Conclusion

We will in this chapter evaluate the most important choices made throughout the project and conclude on the problem statement. We will then go into future avenues based on the results and end with a short discussion of possible master thesis subjects.

6.1 Evaluation

This section will evaluate the different aspects studied in this project. This includes the choice of languages, the evaluation of the characteristics of the concurrency models in these languages, the benchmarking done of the k-means implementations written in the languages and finally the usability of the languages and their implementation of the concurrency models used to implement The Santa Claus problem.

6.1.1 Choice of languages

The choice of languages were based on the Tiobe index and a set of criteria which can be found in Section 2.1.

The criteria should ensure that we would only select functional languages that are still in development and supports advanced concurrency models. These criteria should also make the results of this project to be of value in the future. The set of languages provided a good variety from the Lisp inspired language Clojure to Java like languages such as Scala. Additional variety were also introduced by different execution platforms used like virtual machines like Erlang's Beam and Scala's JVM to Haskell's native code. This variety gave an interesting perspective for comparing the different concurrency models in the different languages.

As an alternative of having the primary focus on functional programming languages and concurrency models, a focus on tools to make working with regular locks and threads easier could also be considered as it solves some of the same issues the concurrency model abstractions tries to solve.

6.1.2 Concurrency model evaluation

The criteria for evaluating the concurrency models available in the chosen languages were presented in "A Study in Concurrency" [27]. The four criteria covers different characteristics of a concurrency model and we feel that the criteria describes the concurrency models in a fair way and enables us to discuss the concurrency model using a set of defined terms.

We have found very few efforts in the literature regarding the development and study of concurrency models in other terms than just pure performance or metrics for comparing LOC or development time. In fact [27] and the successive work by the same authors in [1] were the only examples we have found with these kind of criteria for evaluating concurrency models. One article showed promising results for using a subjective discussion for evaluating concurrency models as opposed to hard metrics **??**.

6.1.3 Benchmarking for concurrency model implementations

The benchmarking of the concurrency model implementations adds a performance aspect to the collective evaluation of the concurrency models. Benchmark suites exist for STM but not for the Actor model and therefore we chose to implement k-means in the five languages for the benchmarking as it is commonly used for performance benchmarks and it is a part of the STAMP benchmark [69]. The amount of synchronisation in k-means is relatively small but still considered great enough to determine the overhead of adding more threads, thereby adding more communication while also expressing the computational power of each language.

We did 10 executions of each implementation on each dataset. It can be discussed if this amount of executions was large enough as multi-threaded programming is known to have unstable execution times because of the nature of threads. It could be interesting to do some dedicated research into how many executions are needed for experiments to be statistically confident. The study [70] stated that there were no significant difference between a sequential implementation compared to their 1 threaded multi-threaded counterpart but it would be very interesting to test this nonetheless and especially in the case for Scala's 1 threaded executions of the dataset with 50,000 samples. Another thing which could be worth researching is why each implementation reacted as they did when presented with the different datasets because Erlang shows very good speed-up when dealing with the relatively small dataset of 1,000 samples compared to the datasets with 10,000 and 50,000 samples.

6.1.4 Usability evaluation

The choice of using The Santa Claus Problem for our usability evaluation is based on an extensive look at the literature of concurrent programming

toy problems. The only toy problem we found used in other work was the Santa Claus Problem. Furthermore the Santa Claus problem were selected for the usability evaluation in order to be comparable with the other implementations and to implement more of a synchronisation problem than the benchmark problem k-means. Even tough the Santa Claus problem was small it was efficient in showcasing problems in expressing communication and synchronisation between multiple threads of execution.

6.2 Conclusion

This project evaluated concurrency models in five different functional programming languages, the languages were chosen based on the criteria that it must be primarily a functional language, actively in use, under development and must support advanced concurrency models natively. This left us with the following five languages: Clojure, Erlang, F#, Haskell and Scala which all fulfilled the specified criteria. The available advanced concurrency models present in the found languages were the Actor model and STM are then evaluated based on the following characteristics: Implicit or Explicit Concurrency, Fault Restricted or Expressive Model, Pessimistic or Optimistic Model, Automatic or Manual Parallelisation. The evaluation of the STM showed that its characteristics mostly consists of Implicit Concurrency, in between a Fault Restricted Model and Expressive Model, it was a fully Optimistic Model and consist of fully Manual Parallelisation. The Actor models evaluation showed that it consists of fully Implicit concurrency and it was a fully Fault Restricted, Pessimistic Model. Furthermore the Actor model consist of fully Manual Parallelisation. The languages were performance benchmarked on speed-up and execution time as evaluation criteria. A k-means implementation was developed for each language using their available advanced concurrency model. The result of the benchmark was that the Clojure, Erlang and Haskell implementations did not scale well and were unable to match the speed-up and execution of F# and Scala. The execution times of the Scala implementation are higher when running using four threads compared to the F# implementation. In general F# and Scala had excellent speed-up when using larger datasets compared to the three other implementations where they after four threads almost were identical in execution times. A usability evaluation was carried out using Lines Of Code (LOC) and development time as the criteria. A implementation of The Santa Claus problem was developed in each of the five functional languages. The result of this evaluation showed that the constructs of the Clojure implementation of STM was the most complex to develop, took the longest time and had the largest code base of 79 LOC this was due to a lack of constructs for retrying STM transactions and blocking threads. Haskell contained these constructs and therefore the implementation of the Santa

Claus problem was simpler to develop both in terms of development time and LOC. The Actor based implementations in Erlang and F# generally took shorter time to develop and were both shorter programs of 54 LOC because the synchronisation of threads was an inherit part of the Actor model. The Scala implementation was an exception because of the verbose syntax for actors which resulted in 79 LOC. The Actor model of Erlang were loyal to the theoretic model but a lack of request-reply functionality was deemed disadvantageous compared to the Actor model of F# and Scala. Therefore we at the time of writing considers the Actor model as being the superior of the two concurrency models, based on both the performance benchmarks and the usability evaluation.

6.3 Future work

This chapter contains a discussion of possible future avenues of research based on the work presented in this rapport.

6.3.1 In-Depth Performance Benchmarks

The benchmarking performed in Chapter 4 showed some interesting differences between the implementations. The evaluation focused exclusively on the throughput of the K-Means algorithm which is a CPU intensive task. Many computing tasks are bound primarily by the CPU but an evaluation of how each concurrency model and language utilises the entire system, such as memory consumption and locality of built-in data structures could be insightful. Such an in-depth evaluation provide a better overview on strengths and weaknesses of each implementation and give a clear indication of how to utilise each language for high performance computing.

6.3.2 Distributed Parallel Programming

In addition to taking advantage of multi-core systems some concurrency models simplify execution in a distribution environment such as a cluster or a data centre. This allows developers to more easily create programs that scale beyond a single machine by abstracting away the inherit communication and placement problems present in such a context. However as execution on distributed systems where deliberately removed from the evaluation performed is the suitability of functional programming both in terms of performance, communication latency and usability for developing such systems still an open question. The Actor model have been used as inspiration for the creation of the functional programming language Erlang which have been created to make distributed systems. An evaluation to determine if the promises of Erlang holds in practise would make for an interesting project.

6.3.3 Imperative Languages

This project focused on how well functional programming languages provides tools for handling concurrency and multi-core processors, as described in Section 1.5. Functional programming languages have been described as better suited for parallel programming but this project have shown that concrete evidence of this statement are few and far apart. Only a few studies have been conducted towards usability and performance evaluation of multi-core languages in general and they have been covering parallel programming in imperative or functional languages separate. A combined study of the performance characteristics and usability of both imperative and functional programming languages would be useful in determining which of the two paradigms provides the best tools for programming multi-core systems.

6.4 Master Thesis Ideas

The following chapter contains a discussing of possible ideas for the subject of our master thesis.

6.4.1 Development of STM and Actor Extensions

The evaluation of the Actor model and STM implementations, showed that there were a difference in the capability of each language. Primarily Clojure and Erlang where found to have a less expressive models than the other languages. Clojure currently depends heavily on its interoperability with Java for synchronising running threads, currently Clojure's STM implementation do not have the capability to be controlled directly by the programmer using constructors such as `retry` and `orElse` as found in Haskell, forcing the programmer to use a Java mutex or similar construct to block a thread if continued execution is undesirable while waiting for other threads. Erlang also was shown to have a very classical implementation of the Actor model compared to F# and Scala, as only asynchronous messages parsing where possible and actors expecting a response had to add its PID to the message it sends.

6.4.2 Development of Language Extension

Clojure currently cannot optimise tail recursion away due to its use of the Java calling convention and instead provided special constructs to compensate, mainly the `recur` and `loop` functions. The `recur` function must be placed in the tail position of a either a function or a block defined by the `loop` function, using `recur` allows Clojure to perform recursive looping in constant space allowing the development of recursive function that else would have caused the stack to overflow. For mutually recursive functions the function

`trampoline` is provided, allowing for trampolining to be used for creation of mutual recursive functions without additional stack consumption for each call. While the `recur` function is simple to use as for function is a simple case of switching the call to the function with a call to `recur` once a stack overflow have been found or one could use it everywhere, would the addition of tail call optimisation allow tail calls and normal functions to be the same instead of having a special case.

Bibliography

[1] B. Damborg and A.M. Hansen. *Cava: A New Concurrency Model for Java*. Aalborg University. Department of Computer Science, 2007. 1.1, 6.1.2

[2] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, 2001. 1.1

[3] TIOBE Software: Tiobe Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2013. Accessed: 25-10-2014. 1.1, 1.2, 1.3.2, 1.3.4, 2.1.2, 2.2

[4] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989. 1.2

[5] Introduction - haskellwiki. `https://www.haskell.org/haskellwiki/Introduction#Quicksort_in_Haskell`. Accessed: 10-01-2015. 1.2

[6] Erik Meijer. The curse of the excluded middle. *Commun. ACM*, 57(6):50–55, June 2014. 1.2

[7] Overview: Introducing F# and functional programming. `http://msdn.microsoft.com/en-us/library/hh297121`, 2010. Accessed: 27-10-2014. 1.2

[8] Philip Wadler. How enterprises use functional languages, and why they dont. In *The Logic Programming Paradigm*, pages 209–227. Springer, 1999. 1.2

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 1.2

[10] Linq: .net language-integrated query. `http://msdn.microsoft.com/en-us/library/bb308959.aspx1`, 2007. Accessed: 27-10-2014. 1.2

[11] Project lambda. `http://openjdk.java.net/projects/lambda/`, 2014. Accessed: 27-10-2014. 1.2

[12] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. 1.3.1

[13] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006. 1.3.2

[14] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`. 1.3.2

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 1.3.3

[16] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, January 1983. 1.3.4

[17] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM. 1.3.4

[18] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. 1.3.4

[19] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989. 1.3.4

[20] John H. Reppy. Cml: A higher concurrent language. *SIGPLAN Not.*, 26(6):293–305, May 1991. 1.3.4

[21] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997. 1.3.5

[22] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*, 2010. 1.3.6

[23] Microsoft. Array.parallel module (F#). `http://msdn.microsoft.com/en-us/library/ee821135.aspx`, 2014. Accessed: 27-10-2014. 1.3.7

[24] Aleksandar Prokopec and Heather Miller. Parallel collections overview. `http://docs.scala-lang.org/overviews/parallel-collections/overview.html`, 2009. Accessed: 27-10-2014. 1.3.7

[25] Sebastian Nanz, Scott West, Kaue Soares da Silveira, and Bertrand Meyer. Benchmarking usability and performance of multicore languages. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 183–192. IEEE, 2013. 1.4, 5.1

[26] Prabhat Totoo, Pantazis Deligiannis, and Hans-Wolfgang Loidl. 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*, pages 49–60. ACM, 2012. 1.4, 4.5

[27] B. Damborg and A.M. Hansen. *A Study in Concurrency.* Aalborg University. Department of Computer Science, 2006. 1.4, 2.1.1, 3.1, 5.2, 6.1.2

[28] John A Trono. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994. 1.4, 5.2

[29] Meredydd Luff. Empirically investigating parallel programming paradigms: A null result. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009. 1.4, 5.1

[30] X10: Performance and productivity at scale faq. `http://x10-lang.org/home/faq-list.html#FAQ-IsX10afunctionallanguage%3F`, 2014. Accessed: 08-11-2014. 2.2

[31] Scheme steering committee position statement. `http://scheme-reports.org/2009/position-statement.html`, 2009. Accessed: 25-10-2014. 2.2

[32] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. *Revised 6 Report on the Algorithmic Language Scheme.* Cambridge University Press, New York, NY, USA, 1st edition, 2010. 2.2

[33] Clojure. `http://clojure.org/`. Accessed: 09-11-2014. 2.2.1

[34] Clojure oo. `http://clojure.org/datatypes`. Accessed: 09-11-2014. 2.2.1

[35] Clojure jvm. `http://clojure.org/jvm_hosted`. Accessed: 09-11-2014. 2.2.1

[36] Clojure stm. `http://clojure.org/concurrent_programming`. Accessed: 09-11-2014. 2.2.1

[37] Cognitect. `http://cognitect.com/clojure`. Accessed: 09-11-2014. 2.2.1

[38] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.6333, 1993. 2.2.2

[39] Erlang/otp. https://github.com/erlang. Accessed: 30-10-2014. 2.2.2

[40] Magnus Hallenstål, Ulf Thune, and Gert Öster. Engine server network. *Ericsson Review*, 3, 2000. 2.2.2

[41] What is erlang. http://www.erlang.org/faq/introduction.html. Accessed: 30-10-2014. 2.2.2

[42] Learn you some erlang for great good!, introduction. http://learnyousomeerlang.com/introduction#about-this-tutorial. Accessed: 30-10-2014. 2.2.2

[43] Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, et al. The F# 3.0 language specification, 2005. 2.2.3

[44] Didier Rémy and Jérôme Vouillon. Objective ml: A simple object-oriented extension of ml. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 40–53, New York, NY, USA, 1997. ACM. 2.2.3

[45] Testimonials - the F# software foundation. http://fsharp.org/testimonials/, 2014. Accessed: 26-10-2014. 2.2.3

[46] F# compiler and components (open edition). https://github.com/fsharp/, 2014. Accessed: 26-10-2014. 2.2.3

[47] Simon Marlow et al. Haskell 2010 language report. http://www.haskell.org/onlinereport/haskell2010, 2010. 2.2.4

[48] Haskell in industry. http://www.haskell.org/haskellwiki/Haskell_in_industry. Accessed: 29-10-2014. 2.2.4

[49] Haskell in research. http://www.haskell.org/haskellwiki/Haskell_in_research. Accessed: 29-10-2014. 2.2.4

[50] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, volume 96, pages 295–308. Citeseer, 1996. 2.2.4

[51] Software transactional memory. http://www.haskell.org/haskellwiki/Software_transactional_memory. Accessed: 29-10-2014. 2.2.4

[52] Concurrency in haskell. http://www.haskell.org/haskellwiki/Concurrency. Accessed: 29-10-2014. 2.2.4

[53] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004. 2.2.5

[54] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008. 2.2.5

[55] Company / case studies. `https://typesafe.com/company/casestudies`. Accessed: 31-10-2014. 2.2.5

[56] Scala: Contribute. `http://www.scala-lang.org/contribute/`. Accessed: 31-10-2014. 2.2.5

[57] Scala team wins erc grant. `http://www.scala-lang.org/old/node/8579`, 2011. Accessed: 31-10-2014. 2.2.5

[58] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008. 3.2, 3.2

[59] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. 3.2

[60] Adam Welc. Transactional memory: From semantics to silicon. `http://www.youtube.com/watch?v=JVfd1uVUNtQ`, april 2007. Accessed: 09-11-2014. 3.2, 3.1, 3.2

[61] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 67–74, Washington, DC, USA, 2008. IEEE Computer Society. 3.2

[62] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54(4):70–77, April 2011. 3.2

[63] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM. 3.3, 3.3, 3.3

[64] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. 3.3

[65] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. California, USA, 1967. 4.2, 1

[66] k-means clustering. `http://en.wikipedia.org/wiki/K-means_clustering`. Accessed: 17-12-2014. 4.1

[67] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and apgas at petascale. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 53–66. ACM, 2014. 4.2

[68] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM, 2011. 4.2

[69] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008. 4.2, 6.1.3

[70] Abdelhafid Mazouz, S Touati, and Denis Barthou. Study of variations of native program execution times on multi-core architectures. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 919–924. IEEE, 2010. 4.5, 6.1.3

[71] R.A. Fisher. Iris dataset. `http://archive.ics.uci.edu/ml/datasets/Iris`. Accessed: 10-12-2014. 4.5

[72] Shane Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7. ACM, 2010. 5.1

[73] Simon Peyton Jones. Beautiful concurrency. *Beautiful code*, pages 385–406, 2007. 5.2

[74] libraries@haskell.org. Control.monad.stm. `https://hackage.haskell.org/package/stm-2.1.1.2/docs/Control-Monad-STM.html`. Accessed: 10-01-2015. 5.3.2

[75] The scala actors migration guide. `http://docs.scala-lang.org/overviews/core/actors-migration-guide.html`. Accessed: 05-01-2015. 5.4.3

APPENDIX A

Benchmark Results

This chapter contains the distribution of execution times in Section A.1. Furthermore it includes detailed graphs of execution times and speed-up respectively in Section A.2 and A.3.

A.1 Execution times distribution

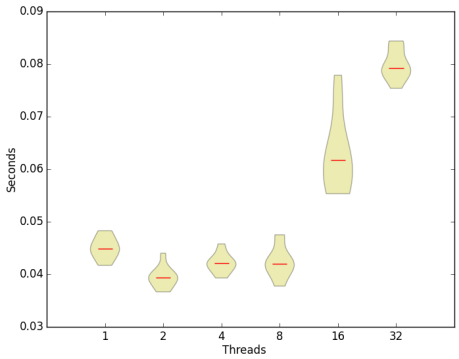
This chapter presents violin plots of the distributions of the execution times for the different datasets for each implementation with red lines that marks the median. The distributions of the Clojure, Erlang, F#, Haskell and Scala implementation can be found respectively in Figure A.1, A.2, A.3, A.4 and A.5.

A.2 Execution time graphs

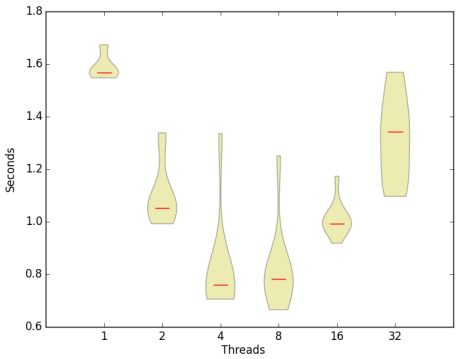
Graphs of the implementations execution times can be found in Figure A.6, A.7, A.8, A.9.

A.3 Speed-up graphs

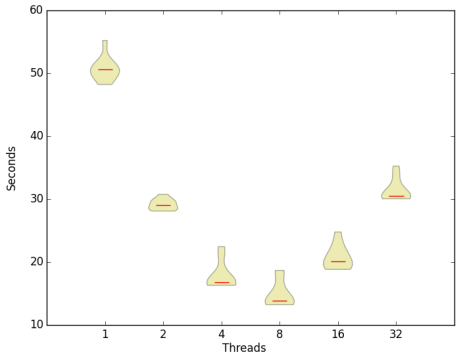
Graphs of speed-up can be found in Figure A.10, A.11, A.12, A.13.



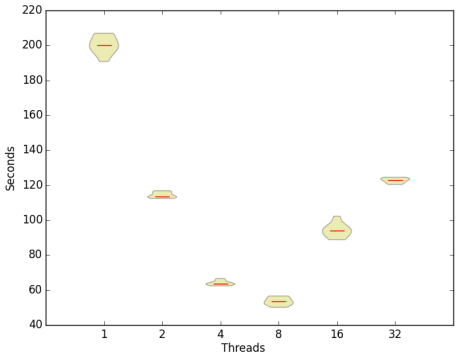
(a) 32 samples



(b) 1000 samples

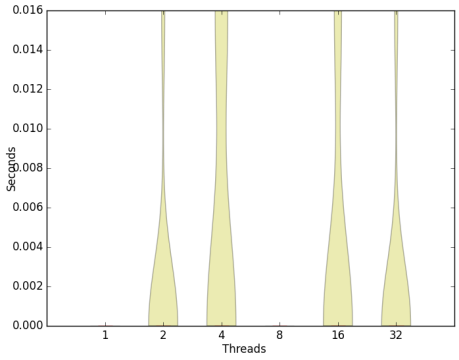


(c) 10000 samples

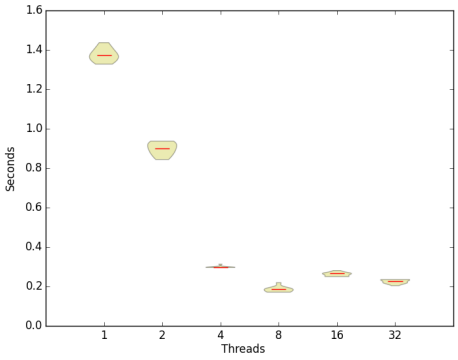


(d) 50000 samples

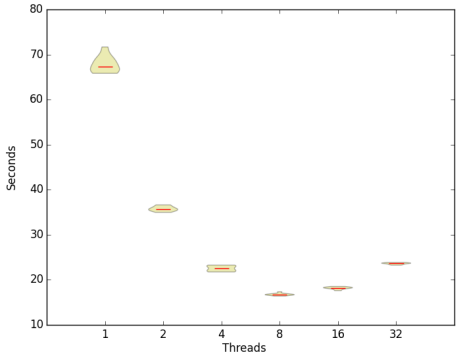
Figure A.1: Execution time distribution of the Clojure implementation.



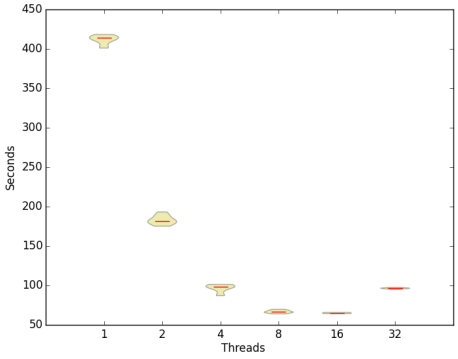
(a) 32 samples



(b) 1000 samples

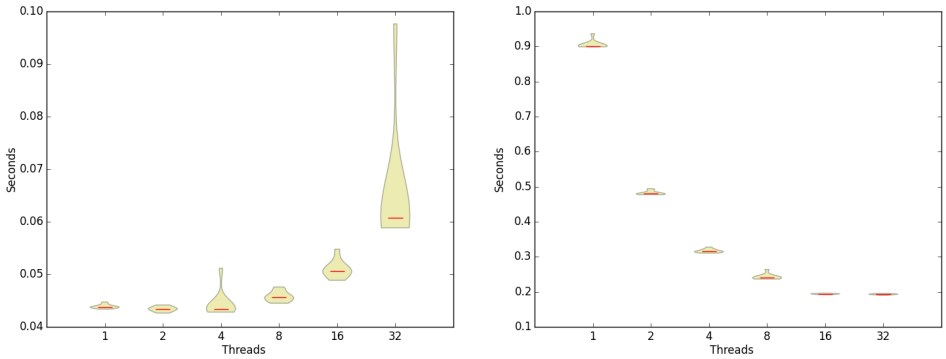


(c) 10000 samples



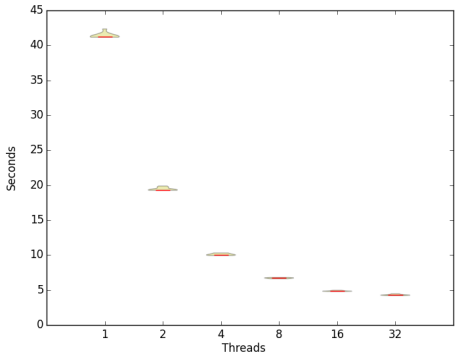
(d) 50000 samples

Figure A.2: Execution time distribution of the Erlang implementation.

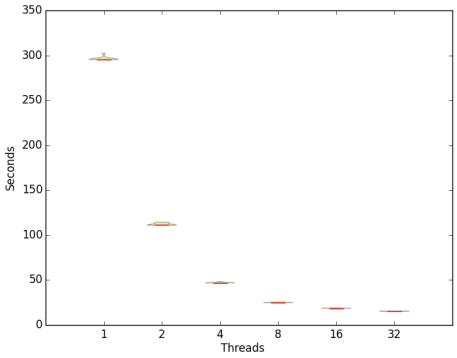


(a) 32 samples

(b) 1000 samples

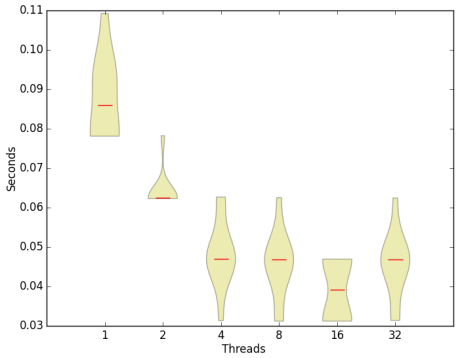


(c) 10000 samples

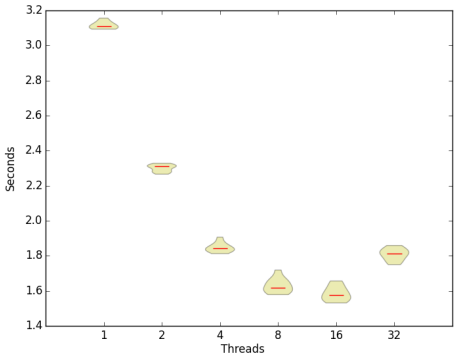


(d) 50000 samples

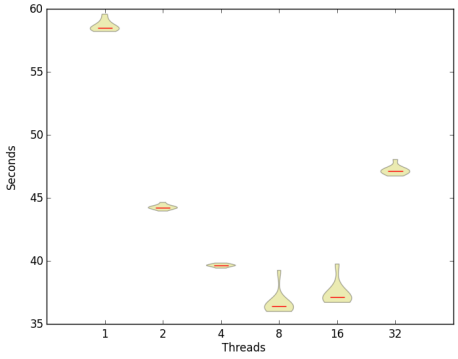
Figure A.3: Execution time distribution of the F# implementation.



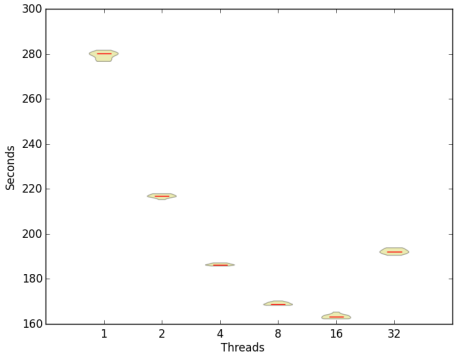
(a) 32 samples



(b) 1000 samples

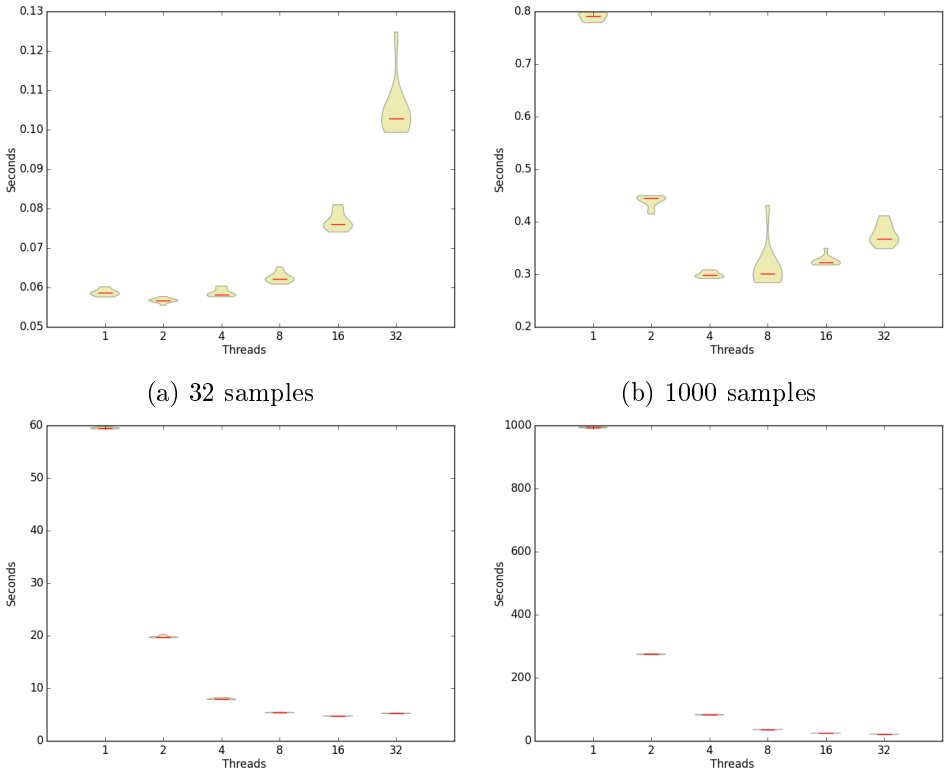


(c) 10000 samples



(d) 50000 samples

Figure A.4: Execution time distribution of the Haskell implementation.



(a) 32 samples

(b) 1000 samples

(c) 10000 samples

(d) 50000 samples

Figure A.5: Execution time distribution of the Scala implementation.

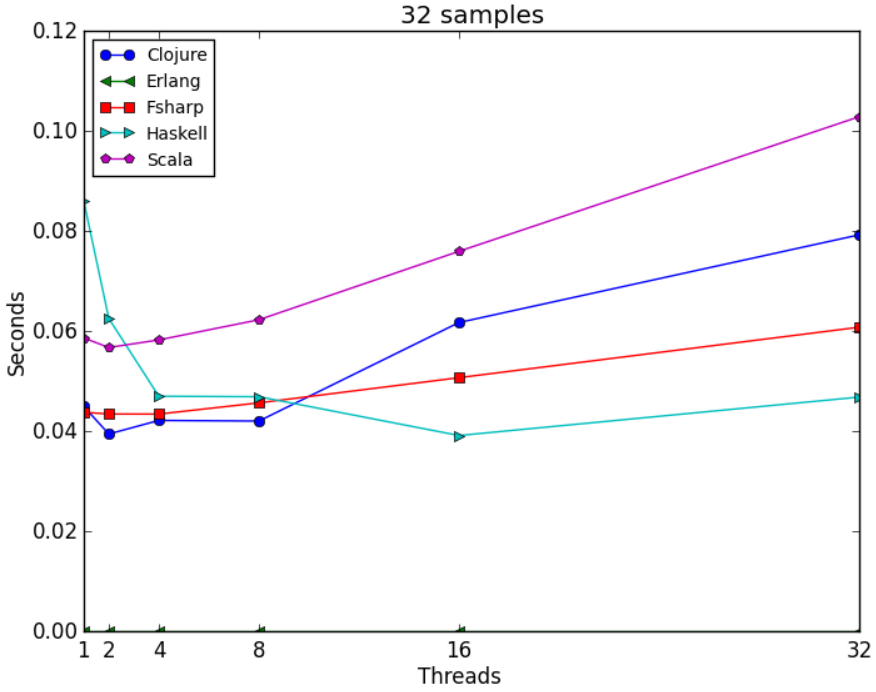


Figure A.6: Execution times in seconds with the four datasets as input.

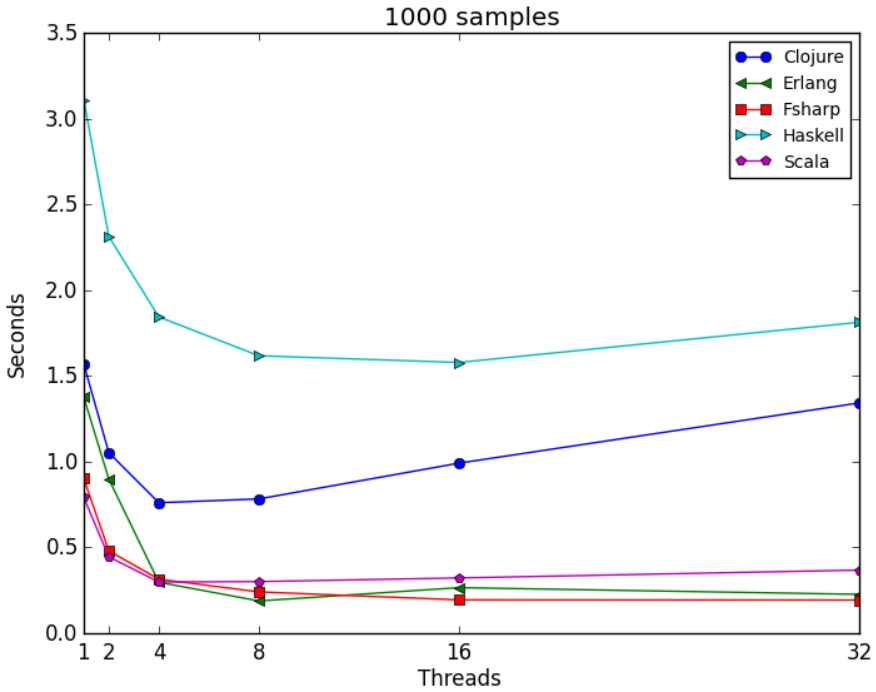


Figure A.7: Execution times in seconds with the four datasets as input.

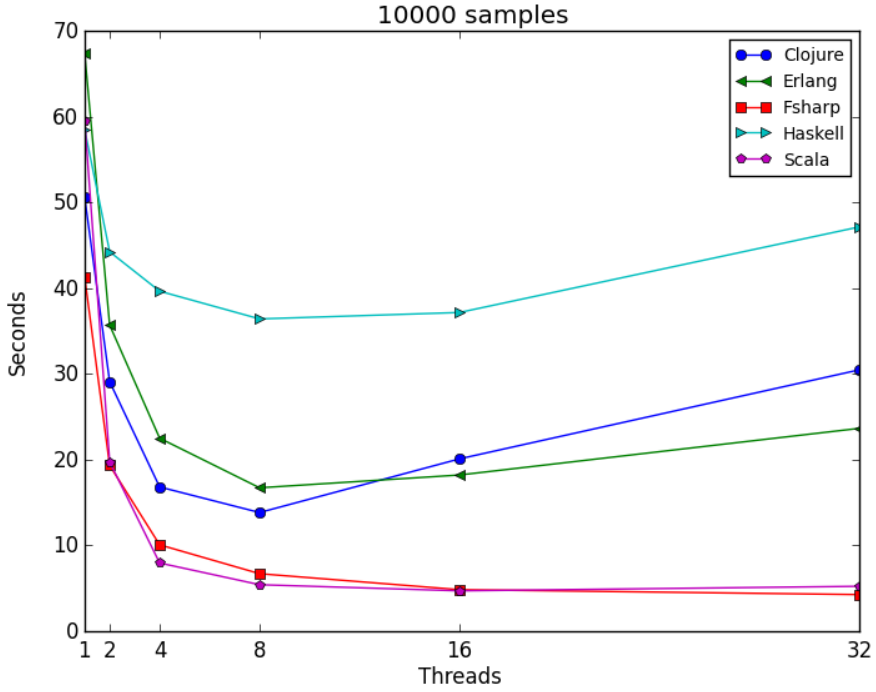


Figure A.8: Execution times in seconds with the four datasets as input.

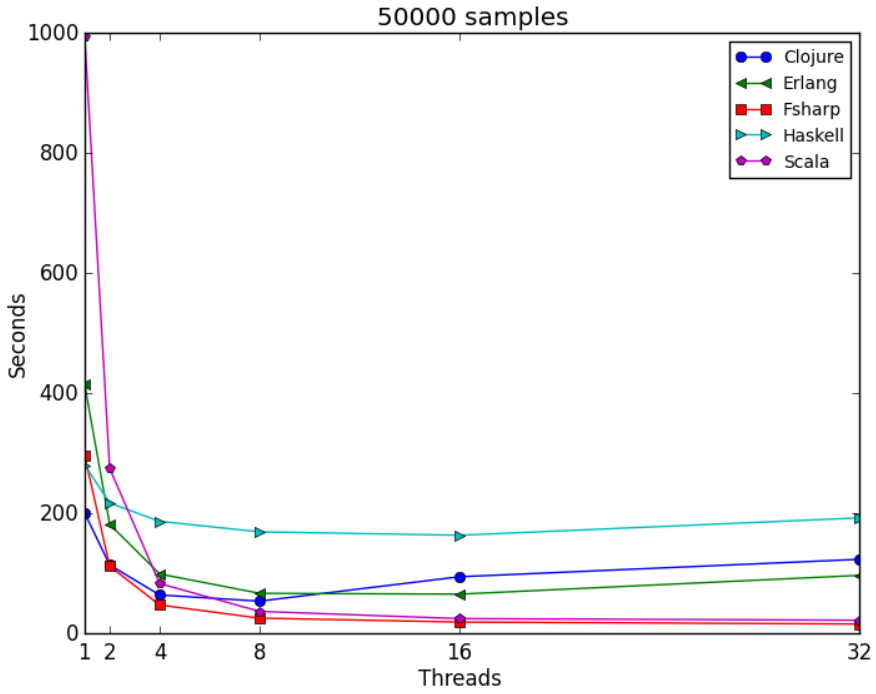


Figure A.9: Execution times in seconds with the four datasets as input.

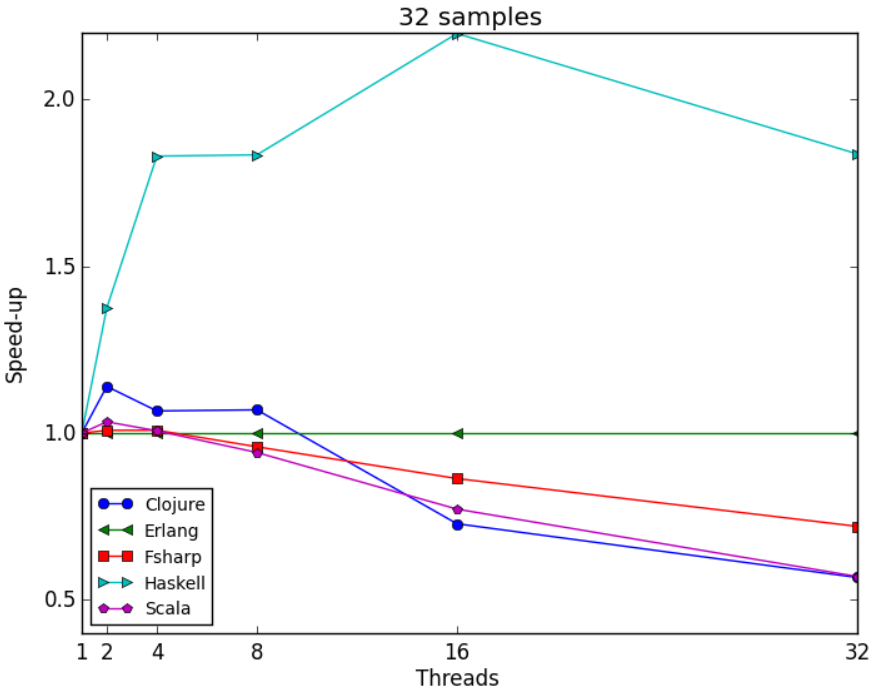


Figure A.10: Speed-up based on the four datasets as input.

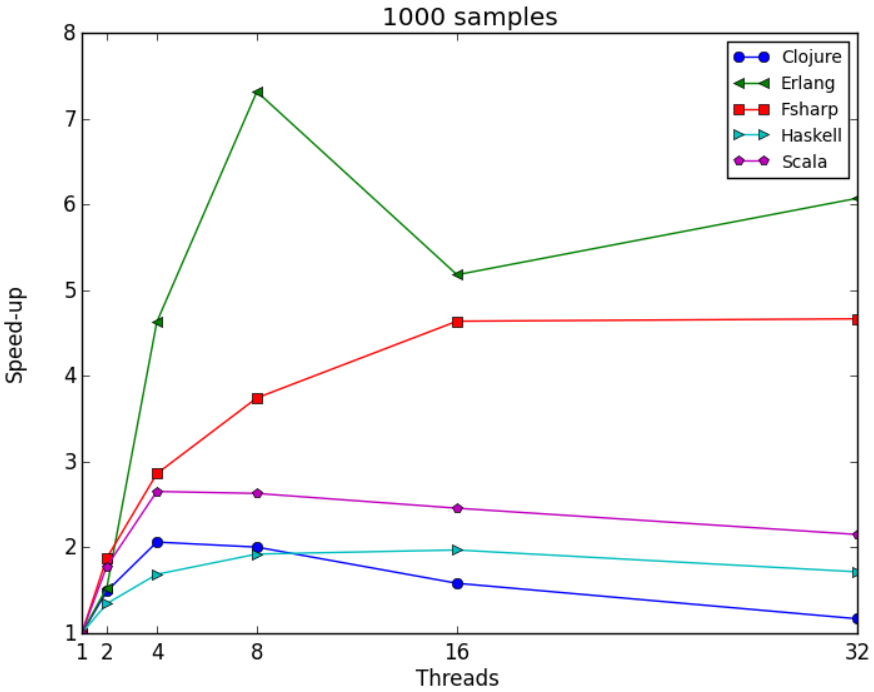


Figure A.11: Speed-up based on the four datasets as input.

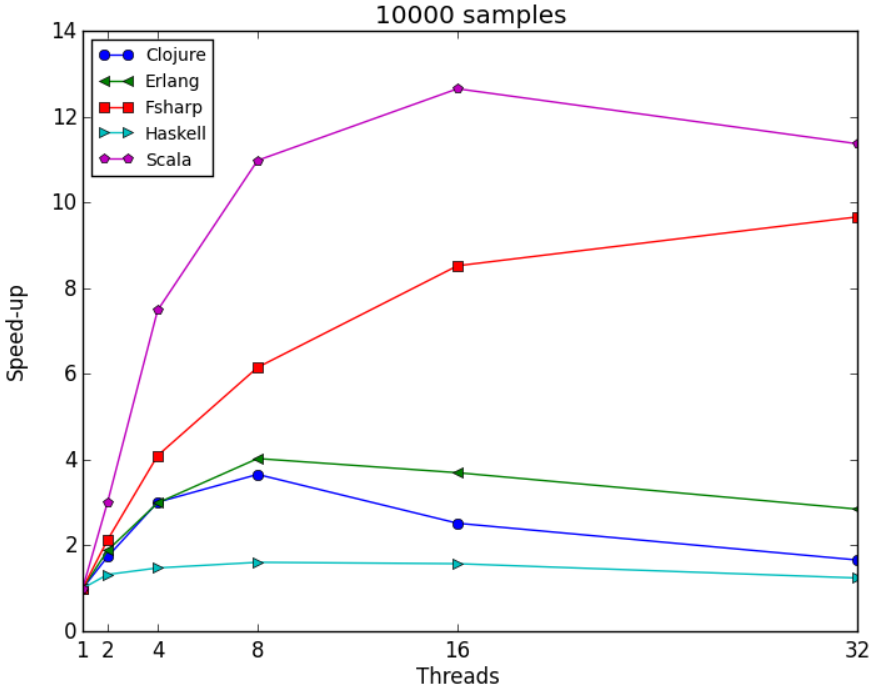


Figure A.12: Speed-up based on the four datasets as input.

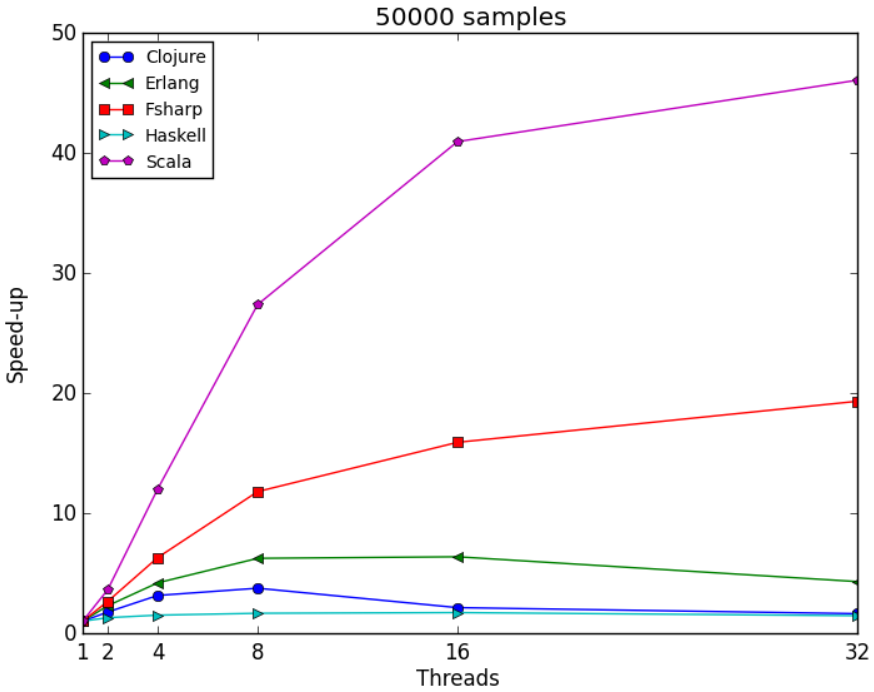


Figure A.13: Speed-up based on the four datasets as input.