Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Ταυτοχρονισμός και Παραλληλία σε Erlang, F# και Scala – Μια συγκριτική μελέτη

Διπλωματική Εργασία

του

Γεώργιου Ψαρόπουλου

Επιβλέπων: Κωστής Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού

Αθήνα, Ιούλιος 2013

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Ταυτοχρονισμός και Παραλληλία σε Erlang, F# και Scala – Μια συγκριτική μελέτη

Διπλωματική Εργασία

του

Γεώργιου Ψαρόπουλου

Επιβλέπων: Κωστής Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου, 2013.

.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Ευστάθιος Ζάχος
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

.....
Γεώργιος Ψαρόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η επικράτηση των πολυπύρηνων αρχιτεκτονικών στο σύγχρονο υπολογιστικό γίγνεσθαι επανάφερε το συναρτησιακό προγραμματισμό στο προσκήνιο ως το προφανές πλαίσιο απλοποίησης και αφαίρεσης. Υπάρχουσες συναρτησιακές γλώσσες, όπως η Erlang και η Haskell, υπέστησαν μετατροπές ή επεκτάθηκαν προς εκμετάλλευση του παράλληλου υλικού, ενώ νέες, όπως η F#, η Scala και η Clojure, δημιουργήθηκαν προς διαχείριση της πολυπλοκότητας σε δημοφιλείς εικονικές μηχανές.

Οι εν λόγω γλώσσες προσφέρουν ένα ευρύ φάσμα χαρακτηριστικών που σχετίζονται με τον ταυτοχρονισμό και την παραλληλία, ωστόσο μια συστηματική παρουσίαση και σύγκριση αυτών απουσιάζει από τη σύγχρονη βιβλιογραφία. Η συνεισφορά μας συνίσταται σε μια συγκριτική μελέτη των γλωσσών Erlang, F# και Scala ως προς την εκφραστικότητα, την ευκολία χρήσης, την επίδοση και την κλιμακωσιμότητα. Συγκεκριμένα, αξιολογούμε τα μέσα που παρέχουν οι γλώσσες αυτές για παράλληλο μετασχηματισμό συλλογών στοιχείων, ταυτοχρονισμό βασιζόμενο σε futures και το μοντέλο των actors, κάνοντας χρήση τους σε υλοποιήσεις που επιλύουν ένα απλό υπολογιστικό πρόβλημα ονόματι Orbit. Η μελέτη μας καταλήγει στα ακόλουθα βασικά συμπεράσματα για τις γλώσσες που εξετάζουμε ως προς τον ταυτοχρονισμό και την παραλληλία: η Erlang αποτελεί μια ώριμη λύση με καλή κλιμακωσιμότητα αλλά χαμηλές επιδόσεις σε αριθμητικούς υπολογισμούς, η F# παρέχει σταθερή βάση για την αντιμετώπιση των εξεταζόμενων ζητημάτων αλλά κλιμακώνει μέτρια, ενώ η Scala, παρά τα προβλήματα που ανακύπτουν από τον επί του παρόντος γρήγορο ρυθμό εξέλιξής της, συνδυάζει πλήθος σχετικών χαρακτηριστικών με πολύ καλές επιδόσεις και κλιμακωσιμότητα.

Λέξεις-Κλειδιά

Ταυτοχρονισμός, Παραλληλία, Συναρτησιακός Προγραμματισμός, Erlang, Scala, F#, Futures, Actors.

Abstract

In the contemporary reign of multicore computing, functional programming has regained attention as the obvious paradigm to simplify and abstract. Existing functional languages, like Erlang and Haskell, have been modified or extended to exploit parallel hardware, while new ones, like F#, Scala and Clojure, have been created to tackle complexity on popular virtual machines.

These languages offer a vast range of features that are related to concurrency and parallelism, yet current literature lacks a systematic demonstration and comparison of those offerings. Our contribution is a comparative investigation of Erlang, F# and Scala in terms of expressiveness, ease-of-use, performance and scalability. In particular we evaluate their language constructs that facilitate parallel collection transformations, future-based concurrency and the actor model, by employing them in implementations that solve a simple computational problem named Orbit. Our study reaches the following conclusions for the examined languages regarding concurrency and parallelism: Erlang is a mature solution with good scalability but low performance in arithmetic computations; F# provides a solid base for handling the examined matters, yet with medium scaling; and Scala, despite the problems that arise from its currently agile evolution pace, combines numerous related features with good performance and scalability.

Keywords

Concurrency, Parallelism, Functional Programming, Erlang, Scala, F#, Future-Based Concurrency, Actor Model.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τους καθηγητές μου, κ. Κωστή Σαγώνα και κ. Νίκο Παπασπύρου για την αγάπη που μου μετέδωσαν μέσω της διδασκαλίας τους για το αντικείμενο των Γλωσσών Προγραμματισμού καθώς και για την συνεργασία μας τα τελευταία χρόνια.

Ιδιαίτερη μνεία αξίζει στον πρώτο εκ των δύο και επιβλέποντα καθηγητή στην παρούσα διπλωματική εργασία, κ. Σαγώνα, για την καθοδήγηση, την στήριξη και την εμπιστοσύνη που μου έδειξε κατά την εκπόνησή της.

Επίσης θα ήθελαν να ευχαριστήσω τους φίλους και συναδέλφους για τα χρόνια που ζήσαμε ως φοιτητές στη σχολή ΗΜΜΥ και τις εμπειρίες – καλές ή κακές – που μοιραστήκαμε.

Τέλος θα ήθελα να εκφράσω ένα μεγάλο ευχαριστώ στην οικογένειά μου, καθώς και σε όλους τους φίλους για τη αμέριστη στήριξη, συμπαράσταση και κατανόηση που μου δείχνουν όλα αυτά τα χρόνια.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

For decades programming was determined by the twofold increase in processor performance every 18 months, a trend first observed by G. E. Moore [1]. Since this trend was synonym to equivalent clock-frequency increases, programmers were accustomed to expect better performance from their programs that required little or no effort. However, around 2003 clock-frequency reached its physical limit; while processor performance kept its ascending course, it would no longer be translated into higher frequency and consequently programmers could not rely on it any more for better execution of their sequential programs.

In "The free lunch is over" [2], H. Sutter observed the era of concurrency arising, as the additional computational power comes in the forms of multithreaded and multicore processors. As a result, programmers need to evolve and employ concurrency to harness performance benefits for their applications – thus concurrency becomes mainstream.

Unfortunately programming concurrency is hard, especially in the established imperative context. Traditional tools require significant cognitive effort to master; yet threads, locks, mutexes, critical sections, synchronized methods and the like are not helpful enough in the struggle against low performance, data races, deadlocks or – even worse – data corruption. Furthermore, as machines gain computational power in the form of additional cores, the requirement for scalability emerges; software should take advantage of extra computational resources without modifications.

As with every complexity in computer science, concurrency is tackled with simplification and abstraction. Numerous languages and frameworks have been proposed over the course of years and claim to provide at least a partial solution to the problem. Among them, functional programming has resurfaced as the obvious paradigm to simplify and abstract; mature languages like Erlang gained support for multicore environments while new hybrid languages have emerged in major ecosystems – Scala on the JVM and F# on the CLI – to entice programmers into the functional paradigm with the promise of better concurrent tools and interoperability with existing investments.

Objectives

In this thesis we examine the concurrent and parallel offerings of Scala, F# and Erlang in an attempt to provide a thorough comparison in terms of expressivity, ease of use, performance and scalability. To that end we provide implementations that follow slightly different approaches to solve a computational problem called Orbit and demonstrate several language features. Nonetheless we only investigate language constructs that support parallel collection transformations, future-based concurrency and the actor model.

Thesis Outline

The rest of the thesis is organized as follows:

- Chapter 2 gives a basic background for the notions mentioned and concurrent/parallel tools used in our study. It introduces definitions for concurrency and parallelism along with a description of future-based concurrency and the actor model, and also presents and compares the features of Scala, F# and Erlang that are employed in our implementations; however this is not a full-fledged presentation of the languages.
- Chapter 3 deals with the implementation part of our study. First a definition of the Orbit problem is given and then some notes that are common for all implementations. The main body of this chapter consists of the analysis of the four implementation approaches we followed and the corresponding code in each of the three languages, along with a brief comparison regarding two code metrics. Finally, the chapter concludes with remarks from our implementation experiences.
- Chapter 4 reflects the experimental evaluation of the implementations. After explaining the benchmarking configurations and describing the execution environment, we examine the results from the execution of those configurations, together with some additional ones. Similarly to the previous chapter, we wrap up with overall remarks.
- Chapter 5 presents some work that is similar to ours, and future directions for continuing the study of concurrency and parallelism in languages that support functional programming.
- Chapter 6 concludes this thesis with an overview of our investigation and some final comments on each language.

Chapter 2

Background

In this chapter we provide a basic background for the notions and constructs that appear in the rest of the thesis. First, we distinguish between concurrency and parallelism by defining those two terms, and describe the concurrency models that are used in this work, namely future-based concurrency and the actor model. Subsequently we demonstrate each of the three languages and particularly present their constructs and libraries related to concurrency and parallelism.

2.1 Concurrency & Parallelism

Concurrency and Parallelism are two frequently occurring terms that are often used with the same meaning. In our study each has a separate meaning, the definition of which relies in the notion of control flow: a *control flow* is the order in which the statements or expressions of an imperative or a declarative program are executed or evaluated.

Based on that notion we define:

- ***Concurrency***: an inherent property of systems that expresses simultaneous progress of several control flows and potential interaction among them.
- ***Parallelism***: a runtime property that refers to simultaneous execution of several similar and mutually independent control flows either on multiple cores or on multiple machines.

By giving these definitions we do not claim universality or indisputability. We want to emphasize nonetheless that parallelism relates to runtime execution whereas concurrency is a property independent from the execution environment; manifestation of parallelism requires multiple computational units to share the workload, while concurrency emerges from only the existence of several control flows and can express situations where they progress interchangeably using time-sharing techniques.

In a manner compatible to the aforementioned definitions and descriptions, parallelism is considered to infer concurrency, but not the opposite.

2.2 Future-based Concurrency

In computer science a *future* is a construct that represents an asynchronous operation. It is a container for the eventual result that will occur when the computation completes and it is used to decompose sequential operations into independent parts that will progress concurrently.

Future-based concurrency can lead to parallelism when used for computations in environments that support parallel execution. However, this is not its sole use case; futures are also used for managing asynchronous I/O operations.

Implementations for concurrency constructs similar to futures have appeared lately in many mainstream languages like Java, C#, C++ and Python, though their functionality differs.

2.3 Actor Model

The actor model is an alternative approach to concurrency that was first proposed by C. Hewitt [3] and improved by G. Agha [4]. It is a form of message-passing concurrency that describes systems and processes in terms of actors and communication between them – hence the aphorism, "everything is an actor".

An actor is a reactive entity that communicates asynchronously with messages. It consists of two basic components: a *behavior* and a *mailbox*. All incoming messages are buffered in the mailbox waiting to be processed – one at a time. For each message, the actor behavior determines, in conjunction with the current state, the *reaction*; the actor may react in the following ways: send a number of messages to other actors, create a number of *children*, or change its state and behavior.

Actors were first popularized by Erlang [5] and lately similar implementations of the actor model have appeared in languages like Scala [6], F# [7] and Haskell [8].

2.4 Scala, F# & Erlang

2.4.1 Scala

Scala [9, 10] fuses the object-oriented and functional paradigms into a flexible statically-typed and general-purpose language that runs on the Java Virtual Machine. It benefits from the almost seamless interoperability with Java and appends the numerous Java libraries to its own ecosystem. One main characteristic of Scala is its extensibility; there are language mechanisms that facilitate custom language constructs to be defined as libraries, thus providing a great infrastructure for adding functionality in the form of expressive and concise internal DSLs.

Scala offers rich support for concurrency and parallelism on top of the Java concurrency model. All related functionality is implemented in libraries that exploit the flexible Scala

syntax as well as other language features that are not specific to concurrency. These libraries include parallel collections, futures and promises, actors and several other tools that are not examined in this study (dataflow concurrency, STM, etc.).

Futures & Promises

Scala supports future-based concurrency with the type `scala.concurrent.Future` in its standard library. Scala Futures are created either by starting an asynchronous computation using the `scala.concurrent.future` function, or from a promise, which is a write-once container that is expected to be filled. Since computations result in either values or exceptions, futures are capable of containing each result. That result can be retrieved in three ways: by blocking while waiting for the future to complete, by specifying callbacks that will be called when the result becomes available, or by creating a chain of futures using future-combinators.

```
import scala.concurrent._, duration._, scala.util._
import ExecutionContext.Implicits.global

def f(x: Int) = future { x + 1 } // Future creation

val res = future(1) flatMap f map (_ + 1) // future tranformations
val res2 = for { x <- future(1); y <- f(x) } yield y + 1 // syntactic sugar

Await.result(res, 1.seconds)            // block waiting
res2 onComplete {                       // callback
  case Success(v) => println(v)
  case Failure(_) => println("error")
}
```

Listing 2.1: Scala Examples with Futures

In Listing 2.1 we give some simple examples of futures in Scala. We define function `f` that takes a number and returns a future with the next number. Afterwards, we use this function in a series of transformations, which is presented in two ways: as a method chain and using the for-comprehension syntactic construct. Finally we present two ways of retrieving the value contained in the future.

Akka Actors

The actor model is supported in Scala through various libraries, most importantly through the Akka toolkit [11, 12], which is an extended set of libraries designed to support concurrent, distributed and fault-tolerant applications. Akka actors are objects with the `Actor` trait; as such, their state is stored in fields and their behavior is encoded in the `receive` method, which is defined only for specific type of messages (partial function). All actors are part of an `ActorSystem` and belong to a supervision hierarchy. They are also identified through an `ActorRef`, the only way referencing them. Moreover, Akka provides also a DSL that significantly reduces the code needed to create simple actors.

```
import akka.actor._, ActorDSL._ // necessary imports
case object Ping // Message types
case object Pong
val system = ActorSystem() // the actor system

// using the actor DSL
// create the ping actor
val ping = actor(system) (new Act {
  become { // define the behavior
    case Pong => sender ! Ping
  }
})

// define the PongActor class
class PongActor extends Actor {
  def receive = { // define the behavior
    case Ping => sender ! Pong
  }
}
val pong = // create the pong actor
  system actorOf Props[PongActor]

ping tell (Pong, pong) // send Pong to ping and make it look like pong did it
```

Listing 2.2: Infinite Ping-Pong in Scala

Listing 2.2 contains a simple example of Akka actors. We use the actor DSL to create a `ping` actor that responds to `Pong` messages by sending `Ping` to the sender. Similarly we define the pong actor that does the opposite, using the main actor API. As we can see, both actors need an actor `system` in which they will be created.

Concurrency Infrastructure

Both futures and actors run on `ExecutionContexts`, which can be viewed as threadpool abstractions. Futures require an `ExecutionContext` to be specified (explicitly or implicitly) inside their lexical scope, while actors use either the default dispatcher of the `ActorSystem` where they belong or the one specified during their creation. There are several `ExecutionContext` implementations provided in the Scala library, each implementation with its own properties and configuration options. Additionally custom `ExecutionContexts` can be creating using the functionality provided by the `ExecutionContext` companion object.

Scala Collections

Scala offers one of the most comprehensive collection libraries. It contains every basic collection and each collection has a vast number of methods that cover most use case scenarios. Aside its vastness, a significant property of the Scala collection library is also its consistency: there is a single hierarchy where all collections belong and all functionality that is shared among collections is provided through the same interfaces. This consistency enables code reuse

Scala collections are organized in four major groups:

Immutable collections never change - new collection for each alteration - and they facilitate functional programming.

Mutable collections are altered in place and they are used for imperative programming.

Parallel collections are transformed in parallel while retaining the usage patterns of their sequential (immutable/mutable) counterparts.

Concurrent collections can be concurrently altered from different threads.

All collections have strict evaluation semantics but they also provide *views*, a way to apply transformations lazily: when a view on a collection is acquired all operations on it are postponed until the elements of the resulting collection are required or the `force` method is called on the result.

Last but not least, the collection library contains functionality for conversion between collections:

- `to[AnotherCollection]`: converts a collection to `AnotherCollection`
- `par`, `seq`: convert a collection to its parallel counterpart and back
- `scala.collection.convert`: provides wrappers for disguising Java collections as Scala ones and the opposite.

2.4.2 F#

F# [13, 14] is a statically-typed, functional-first, general-purpose programming language for the popular .NET and Mono implementations of the Common Language Infrastructure. It facilitates concurrency and parallelism via libraries that provide such functionality but have limited configurability. F# benefits from the CLI in terms of interoperability with other languages and access to a vast library pool. It favors immutability and lack of side-effects but it also supports the CLI object-oriented model and mutation for performance-critical scenarios.

Concurrency and parallelism are not part of the F# language specification. F# uses generic language features and core CLI concurrency primitives to define its concurrent and parallel abstractions, namely asynchronous workflows and the `MailboxProcessor<'T>`. It also retains access to .Net libraries, such as the *Task Parallel Library (TPL)* [15, 16] and *PLINQ* [17].

Asynchronous Workflows

Asynchronous workflows [7] are a form of future-based concurrency. They express non-blocking computations with a syntax similar to their blocking counterparts, without using callbacks. That syntax includes constructs for value binding, error-handling, looping and computation composition that has modified semantics for asynchronous execution. Some of them are showcased in Listing 2.3 which contains simple examples that closely resemble the ones in Listing 2.1.

```
let f x = async { return x + 1 } // Function returning an asynchronous computation
let res = async {
    let x = 1     // Ordinary value binding
    let! y = f x  // Asynchronous value binding: wait for the computation to
                  // complete asynchronously and bind the result to y
    return! f y   // Return the asynchronous computation
}
let bangedResult = Async.RunSynchronously res  // Block to wait for the res value
```

Listing 2.3: Simple Asynchronous Computations in F#

Asynchronous workflows are based on a general syntactic mechanism called *computation expressions* [18, 19]; inside their scope one can use custom syntax with target-specific semantics. Included in F# are computation expressions for sequences, asynchronous computations and queries (Listings Listing 2.3 & Listing 2.4).

```
// a sequence expression that has the argument lst repeated n times
let repeat n lst = seq {
    for i = 0 to n do  // loop for n times
        yield! lst     // returns all values from sequence lst
}

// a query expression for all names of adult customers in the database
let q = query {
    for customer in db.Customers do  // for each customer in db.Customers
    where (customer.Age >= 18)       // if he/she is an adult
    select customer.Name             // select his/her name
}
```

Listing 2.4: Examples of Computation Expressions

Agents

On top of *asynchronous workflows*, `MailboxProcessor<'T>` is defined (usually abbreviated as `Agent<'T>`). This type facilitates actor-based concurrency with a lightweight implementation; it represents an actor that accepts only messages of a specified type; it is a typed actor.

In Listing 2.5 we demonstrate the F# version of the ping-pong example we presented for Scala actors. `Agent` is used for the definition of both actors, which are created by providing the desired actor behaviors to the `Agent.Start` static method.

```
type Agent<'T> = MailboxProcessor<'T> // an alias
type Ping = Ping of Agent<Pong>       // recursive type for ping/pong messages
and Pong = Pong of Agent<Ping>
// ping : Agent<Ping>                    // pong : Agent<Pong>
let ping = Agent.Start(fun inbox ->     let pong = Agent.Start(fun inbox ->
    let rec loop() = async {                let rec loop() = async {
        // wait for Pong                        // wait for Ping
        let! Pong(sender) =                     let! Ping(sender) =
             inbox.Receive()                         inbox.Receive()
        // reply with Ping                      // reply with Pong
        sender.Post <| Ping(inbox)              sender.Post <| Pong(inbox)
        // recursive call                       // recursive call
        return! loop()                          return! loop()
    }                                       }
    loop()                                  loop()
)                                       )
pong.Post <| Ping(ping) // send Ping to Pong
```

Listing 2.5: Infinite Ping-Pong in F# using Agents

Tasks & PLINQ

Beside F#-specific concurrency constructs, Base Class Library (BCL) also offers related tools such as TPL, PLINQ and concurrent collections. Task Parallel Library (TPL) enables another form of future-based concurrency and is optimized for CPU-bound computations. It offers two alternatives for concurrency: the Parallel class that has static methods for operations like parallel for, parallel foreach and parallel execution of functions; and the Task class that facilitates explicit task creation and execution. Built on TPL, Parallel Language Integrated Queries (PLINQ) offer a set of parallel operations for collections that resemble functional operators (such as map, fold, filter, group, etc.) using a naming convention closer to SQL queries. These operations come as extension methods to classes that implement `IEnumerable<'T>`.

Concurrency Infrastructure

All aforementioned libraries and constructs use the CLI threadpool by default. Each CLI instance has a pool of managed threads, each of which resembles an OS thread in current .NET and Mono implementations. The CLI threadpool maintains a request queue where tasks and asynchronous operations are submitted. Each queue entry is dispatched to a pool thread for execution; execution starts immediately using an existing idle thread or a newly and thus created one, or it is deferred until a thread becomes available – the heuristic decision is based on thread availability and assessed work load.

Depending on the library in use, one may be able customize the threadpool usage. Asynchronous workflows and, by extension, agents do not offer particular configuration options; CLI threadpool usage is hardcoded in the implementation and therefore execution relies exclusively on threadpool parameters. TPL, on the contrary, is more configurable: each

task execution is managed by a task scheduler (which submits the tasks to the threadpool) that can be specified for each task. However only one implementation is provided in BCL.

Collections

F# includes basic immutable collections like list, set and map, while arrays are also first class citizens. In addition, BCL contains several collections - sequential and concurrent - that can be used when there is no F# equivalent, when they offer better performance characteristics or when interoperability with other CLI languages is required; BCL collections are no panacea, however, since they are mutable and relatively odd to use.

2.4.3 Erlang

Erlang [20] is a language designed for concurrency. As a language, it provides lightweight primitives for straightforward and succinct concurrent solutions; as a runtime, it supports their efficient and scalable execution.

Processes and Actors

Erlang supports the message-passing type of concurrency, namely the actor model. The so-called processes have all the attributes of an actor: an unbounded mailbox for incoming messages, a behavior for processing a message and reacting accordingly, a private internal state and a well-defined lifetime. The behavior of a process is defined in a recursive function and the state comes in form of function parameters. Process creation and message exchange are primitive operations in Erlang that are supported by non-blocking built-in operators of the language. In Listing 2.6 we demonstrate actors in Erlang using the same ping-pong example we used previously.

```
ping() -> % ping function
  % receive pong message
  receive
    {pong, Pong_PID} ->
      Pong_PID ! {ping, self()},
      ping() % recursive call
  end.
pong() -> % pong function
  receive
    {ping, Ping_PID} ->
      % repond with pong
      Ping_PID ! {pong, self()},
      pong()
  end.
start() ->
  % create ping and pong actors
  Ping_PID = spawn(fun ping/0),
  Pong_PID = spawn(fun pong/0),
  % send ping to pong
  Pong_PID ! {ping, Ping_PID}.
```

Listing 2.6: Infinite Ping-Pong in Erlang

Processes are lightweight. Each process begins its lifetime with a tiny memory space, including a private heap space where it stores its data; this memory is resized according to usage and it is independently garbage collected. Process execution is regulated by the schedulers of the Erlang runtime – by default one scheduler for each core available in the machine –, each of which manages its work queue. Every process is assigned to one of those work queues and its execution is preemptively scheduled: after a specified number of so-called reductions the scheduler pauses the running process, returns it back to the work queue and resumes the next process (if any). That way the execution of every process is ensured.

Data Structures

Erlang comes with a comprehensive standard library: several common functional data structures such as lists, sets and dictionaries; implementations of reusable concurrent abstractions over the aforementioned language primitives; built-in functions (called BIFs) for numerous operations that cannot be expressed or efficiently implemented in Erlang.

The Erlang Term Storage (ETS), a mechanism for key-value store, is a part of the library that deserves special mention. It offers a family of mutable data structures (set, ordered_set, bag and duplicate_bag) that are characterized by efficient lookup, addition and removal operations. More importantly, ETS tables can be configured to allow concurrent access to and mutation of their contents and therefore are invaluable tools for efficient parallel programming.

Another significant part of the library consists of abstractions and patterns for concurrent and distributed programming. Nevertheless, Erlang does not include ready-to-use parallel constructs, neither as primitive functionality nor as part of the standard library, although parallel abstractions can be composed using the provided lightweight concurrency constructs. (See for example Listing 2.7)

```
%% Map function F over list L in parallel.
parallel_map(F, L) ->
  Root = self(),
  [ receive Res -> Res end || _ <- [
    spawn(fun() -> Root ! F(X) end) || X <- L ] ].
```

Listing 2.7: Simple Parallel Map in Erlang

All things considered, Erlang is built for concurrency. The language simplifies development of concurrent solutions and time-tested libraries facilitate productivity and quality. Still, all solutions are required to be modeled using actor model – even when a different approach is more appropriate – and, as a rule, the programmer has to forget the convenience of mutation.

2.4.4 Comparison

In the last three paragraphs we gave a brief description of Scala, F# and Erlang and presented the features of those languages that we use in our study. Here we demonstrate a more detailed comparison of those features in the following tables.

Table 2.1: Future concurrency

	Scala	**F#**	
Relevant Feature	Futures	Async Workflows	Tasks
Execution Mechanism	`ExecutionContext`	`ThreadPool`	
Configurability	High	Limited	Moderate
Composition	Combinators	Custom Syntax	Task API

In Table 2.1 we compare the means to express future-based concurrency in Scala and F# – Erlang has no specific feature, although futures can be implemented using processes.

Scala futures run on an `ExecutionContext` that is specified by the programmer, while both F# asynchronous workflows and tasks are executed on the CLI `ThreadPool` by default. Consequently future execution is highly configurable, considering that two futures can be configured to run on different `ExecutionContexts`, each with its own options and execution policies. Nonetheless, tasks can use a custom `TaskScheduler` instead of the default, while the execution of asynchronous workflows is rather fixed, since the use of the `ThreadPool` is hardcoded in their implementation. Regarding compositionality, on the other hand, futures and async workflows have composable designs, contrary to tasks, whose API provides rather verbose.

Table 2.2: Actor Concurrency

	Scala	**F#**	**Erlang**
Relevant Feature	Akka Actors	`Agent<'T>`	Processes
Support Level	Library	Library	Language
Behavior	Callback	Async Workflow	Recursive Function
Behavior Change	`become`/`unbecome`	Tail-Recursive Call	Tail-Recursive Call
Interface	Dynamic	Static	Dynamic
Execution	`Dispatcher`	`ThreadPool`	Erlang Schedulers
Configurability	High; Execution Mechanism per Actor, Mailbox, Supervision Policy	None	Limited; Parameters of Execution Mechanism

In Table 2.2 we compare the actor implementations of Scala, F# and Erlang. The F# library support for actors consist of only the `Agent` and any actor abstractions (routing, supervision, etc.) have to be implemented by the library user, while Akka and Erlang offer considerable amount of such functionality out-of-the-box. Each of the two latter actor implementations has a different level of configurability: Akka actors offer several configuration options thanks to their implementation as a library, while the configurability of Erlang processes is limited to runtime properties. Moreover, these actors have a communication interface that can change during their lifetime, contrary to the F# `Agent` whose interface is part of its type and thus fixed. Finally, the way the actor behavior is defined resembles properties of the underlying runtime: the Erlang and F# runtimes support the tail-call optimization so the actor behavior is can be defined as a recursive function, contrary to Akka, which defines it as a callback function due to the lack of JVM support for tail-call optimization.

Chapter 3

Implementation

In this chapter we present the first part of our thesis, namely the comparative evaluation of Scala, F# and Erlang regarding the expression of concurrency and parallelism. First we define the problem we use for our purposes and then we compare our implementations; for each language we provide a sequential implementation and some concurrent ones that follow three different approaches of performing the computation. Finally we make an overall assessment about our overall experience of programming in those three languages.

3.1 The Problem: Orbit

For our comparative investigation we needed a problem with a simple parallelizable solution to facilitate the evaluation of the various constructs offered by each language. We have chosen the Orbit problem (orbit_int) from the BenchErl Benchmark Suite [21].

In Listing 3.1 we present the definition used in our study; it differs from the inductive definition given in [21], but it resembles better our implementation approach.

Given:

- a space S
- a set of generators $G = \{g_i : S \to S, i \in \{1 \dots n\}\}$
- a set of initial elements $X_0 = \left\{ x_j \in S, j \in \{1 \dots m\} \right\}$

Let:

- $X_{k+1} = \left\{ g_j(x_i), g_j \in G, x_i \in X_k \right\}$
- $E_u = \bigcup_{k=0}^{u} X_k$

Goal:

- Compute the set $Orb = E_\infty$

Listing 3.1: Orbit Definition

In our study we considered a special case of the Orbit problem where $X \subseteq \mathbb{N}$ and finite.

3.2 Solution Overview

The general algorithm for computing the Orb set uses the above definition with a slight modification that avoids re-computation of the same elements:

$$X_{k+1} = \left\{ g_j(x_i) \notin E_k, g_j \in G, x_i \in X_k \right\}$$

For each k we compute X_{k+1} and insert its elements to E_k which is represented by a set data structure. This procedure is repeated until the k* where X_{k^*+1} is empty; k* is ensured to exist unless the result Orb has infinity elements.

In order to examine the different concurrency models we described we examined the following approaches:

A. Sequential: It is an exact translation of the described algorithm into code.
B. Parallel: The elements of X_k are computed in parallel (Scala Parallel Collections/F# PLINQ).
C. Futures: Each X_k is split into chunks $\left\{X_k^i\right\}$ of a specified size G and computing each X_{k+1}^i is a concurrent computation (future).
D. Persistent Actors: Similar to C, but persistent actors are used to compute X_{k+1}^i instead of futures.

3.3 Common Notes

3.3.1 Representation of the Orbit problem definition and the solver

The orbit problem definition is represented as shown in Table 3.1 and Listing 3.2.

Language	Problem Definition	S	G	X_0
Scala	trait	type member T	method generators	initData
F#	generic record	type parameter T	function generators	initData
Erlang	record	-	function generators	init_data

Table 3.1: Problem Definition Representations

```
Scala:
trait Definition {
  type T
  def generators(x: T): Seq[T]
  val initData: Seq[T]
}
F#:
type Definition<'T> = {
    generators: 'T -> 'T seq
    initData: 'T seq
}
Erlang:
-record(definition, {generators, init_data}).
```

Listing 3.2: Problem Definition Representations

In Scala we can represent S also as a type parameter on a generic definition of `Definition`, however we chose to represent it as type member for better encapsulation.

```
Scala:
def solve(p:Definition)(/* other arguments */) : Set[p.T] = { /* body */ }
F#:
let solve <'T when 'T: equality> (* other arguments *)
  { initData = initData; generators = generators } = (* body *)
Erlang:
solve(#definition{init_data = InitData, generators = Generators} % other arguments
) -> % body
```

Listing 3.3: Problem Definition Representations

Overall, our solvers are designed to be mostly functional and generic. Every implementation is a function from a problem definition to the corresponding *Orb* set (Listing 3.3).

3.3.2 Sets and Concurrent Sets in Scala, F# and Erlang

We need an appropriate data structure to represent the resulting Orbit set. Our implementations use either a simple set (as provided by the standard library of each language) or a concurrent one; the choice depends on the accesses to the data structure, if they are

concurrent or not. To examine the performance characteristics of comparable data structures we have decided to use only set implementations that have practically constant access time.

In Scala we have the option to use either Scala or Java collections. We created a simple abstraction (Listing 3.4) to ease the use of both libraries: we use `ScalaSets` and `JavaSets` as factories of Scala and Java sets respectively. The Java sets can be used like native Scala sets using the thin wrappers provided by the Scala collection library; the `import collection.convert.WrapAsScala._` statement brings in scope the necessary implicit conversions that apply those wrappers.

```
// Import packages with shorter names
import collection.{ immutable => i, mutable => m, concurrent => c }

// Common interface for set providing objects
trait SetProvider {
  def iSet[A]: i.Set[A]
  def mSet[A]: m.Set[A]
  def cMap[A]: c.Map[A, Unit]
}

// A set provider for Scala sets
object ScalaSets extends SetProvider {
  def iSet[A]: i.Set[A] = i.Set[A]()
  def mSet[A]: m.Set[A] = m.Set[A]()
  def cMap[A]: c.Map[A, Unit] = c.TrieMap[A, Unit]()
}

// A set provider for Java sets
object JavaSets extends SetProvider {
  import java.{ util => ju }, java.util.{ concurrent => juc }
  import collection.convert.WrapAsScala._
  def iSet[A]: i.Set[A] = throw new NoSuchElementException("Immutable Java Set")
  def mSet[A]: m.Set[A] = new ju.HashSet[A]
  def cMap[A]: c.Map[A, Unit] = new juc.ConcurrentHashMap[A, Unit]
}
```

Listing 3.4: Set and ConcurrentMap abstraction for Scala

In F# and Erlang the standard immutable sets are implemented as trees and therefore have logarithmic complexity. Thus we use only `System.Collections.Generic.HashSet<'T>` (abbreviated as `MutableSet<'T>`) in F# and ETS tables of type set in Erlang.

For the concurrent set functionality the ETS set of Erlang can be configured to allow concurrent reads and writes. Unfortunately both F# and Scala lack such a data structure, so we use concurrent maps as a workaround; we use the set elements as keys mapped to dummy values:

- In F# we use the BCL `System.Collections.Concurrent.Concurrent-Dictionary <'Key, 'Value>` with `'Key` = `'T` and `'Value=obj` (with `null` as value), abbreviated as `ConcurrentSet<'T >`.
- In Scala we have the option to use either `scala.collection.concurrent.TrieMap[Key, Value]` [22] or `java.util.concurrent.ConcurrentHashMap[Key, Value]` with `Key = T`. The implementation of `scala.collection.concurrent.TrieMap` handles `null`-mapped elements as not included in the `Map` so we cannot use `Value = Null`; instead we use the similar `Value = Unit`.

3.3.3 Partitioning into chunks of specific size

The last two categories of our implementations need to explicitly partition the data of a sequence to chunks of a specified size. Scala collections have such a method: `grouped(size: Int): Iterator[Repr]`. As neither F# nor Erlang have such functionality ready to use, we created functions that behave similarly.

```
let chunked (chunkSize:int) (sq:#IndexedSeq<_>) :seq<_> =
    let index = ref 0
    let length = IndexedSeq.length sq
    seq {
        while !index + chunkSize < length do
            let idx = !index
            yield seq { for i = idx to idx + chunkSize - 1 do yield sq.[i] }
            index := !index + chunkSize
        if !index < length then
            let idx = !index
            yield seq { for i = idx to length - 1 do yield sq.[i] }
    }
```

Listing 3.5: F# partitioning function

In F# we wanted to have on-demand evaluation semantics. We created three partitioning functions:

- one that partitions sequences (`System.Collections.Generic.IEnumerable`) and is defined in terms of `IEnumerator` and `IEnumerable`
- a second that partitions random-access collections (`System.Collections.Generic.IList` abbreviated as `IndexedSeq`), is also defined in terms of `IEnumerator` and `IEnumerable` and is more efficient as it reuses the initial data structure for the chunks
- a third that also partitions random-access and retains the benefits of the second version while being defined using the comfortable computation expression syntax of F#

In our implementations we used only the third of those functions (Listing 3.5).

Contrary to Scala, these functions do not retain the type of the partitioned collection, a fact that highlights the less expressive F# type system.

In Erlang we do not have the tools to create similar functionality so we created a simple function that eagerly splits a list to lists of the specified `ChunkSize` and also returns a count of the created chunks.

```
split(ChunkSize, List) ->
  split_helper(List, ChunkSize, ChunkSize, [], [], 0).

split_helper([], _ChunkSize, _Left, [], Result, Count) ->
  {Count, Result};
split_helper([], _ChunkSize, _Left, Acc, Result, Count) ->
  {Count + 1, [Acc|Result]};
split_helper(List, ChunkSize, 0, Acc, ResultAcc, Count) ->
  split_helper(List, ChunkSize, ChunkSize, [], [Acc|ResultAcc], Count + 1);
split_helper([H|T], ChunkSize, Left, Acc, Result, Count) ->
  split_helper(T, ChunkSize, Left - 1, [H|Acc], Result, Count).
```

Listing 3.6: Erlang partitioning function

3.3.4 F#-specific libraries versus BCL

Contrary to F# libraries, the Base Class Library is designed with object oriented principles and for use in languages like C#. Most functionality is offered as methods on mutable objects and cannot be easily composed. Moreover, using BCL results in non-idiomatic F# code and hinders the benefits of key language features like type inference.

Collections are an area that highlights this remark: operations on F# collections are defined as functions grouped in modules, while BCL collections have the corresponding functionality implemented as methods of each collection instance. Type inference can deduce the type of a collection from the former group, based on the functions that operate on it; for the latter, it needs a type annotation because it cannot deduce object types from method calls.

```
type MutableSet<'T> = System.Collections.Generic.HashSet<'T>

[<RequireQualifiedAccess>] // forbid usage without the module identifier
module MutableSet =
    let unionWith (set:MutableSet<'T>) seq = set.UnionWith seq
    let add (set:MutableSet<'T>) elem = set.Add elem
    let contains (set:MutableSet<'T>) elem = set.Contains elem
    let empty<'T> = MutableSet<'T>()
    let ofSeq (seq:seq<'T>) = MutableSet<'T>(seq)
```

Listing 3.7: `HashSet` *alias and wrapper module*

Aiming for natural F# implementations, we chose to provide wrappers for most of the used functionality as helper functions organized in modules resembling their respective type. For instance, we present the wrapper module for BCL `HashSet` (Listing 3.7) and `ConcurrentDictionary` (Listing 3.8).

```
type ConcurrentSet<'T> =
    System.Collections.Concurrent.ConcurrentDictionary<'T, obj>

[<RequireQualifiedAccess>]
module ConcurrentSet =
    let add (set:ConcurrentSet<'T>) elem = set.TryAdd (elem, null)
    let contains (set:ConcurrentSet<'T>) elem = set.ContainsKey elem
    let empty<'T> = ConcurrentSet<'T,obj>()
    let create<'T> (concurrencyLevel:int) (initialCapacity:int) =
        ConcurrentSet<'T, obj>(concurrencyLevel, initialCapacity)
    let ofSeq (seq:seq<'T>) =
        ConcurrentSet(seq |> Seq.map (
            fun x -> System.Collections.Generic.KeyValuePair(x,null)
        ))
```

Listing 3.8: `ConcurrentDictionary` *alias and wrapper module*

In the following sections we present our implementations for each category. The actual code is slightly different as it also includes code related to configuration and time measurement.

3.4 Approach A - Sequential

This is reference implementation for all languages. X_{k+1} is computed using a slightly different definition: X_k is mapped through G and only the elements that are not contained in E_k are kept.

Each k is a distinct computational step that relies on the previous one. The result set is found at the step k^* where X_{k^*} is empty.

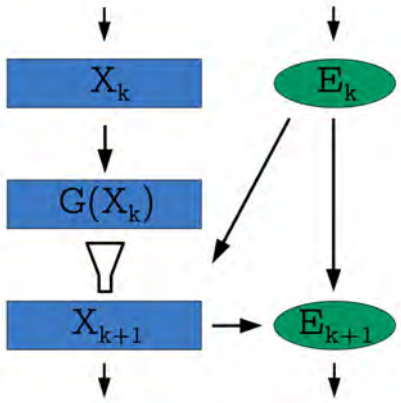


Figure 3.1: Logic of Approach A

3.4.1 Scala

There are two sequential Scala implementations, one using an immutable set (*Scala I*, Listing 3.9) and a second using a mutable set (*Scala M*, Listing 3.10). Both sets have a common superclass: `collection.Set` and therefore share common functionality; yet we cannot reuse code between the two implementations as the mutable set has different usage pattern from the immutable one.

Moreover, the local import statements enable the use the members of `p: Definition` as if they were locally defined.

```
def simpleLogic(p: Definition)
  (seq: GenSeq[p.T], results: Set[p.T]): Set[p.T] = {
  import p._
  def helper(currentSeq: GenSeq[T], results: Set[T]): Set[T] = {
    val nFilteredSeq =
      currentSeq
        .flatMap(generators(_))
        .filterNot(results.contains)
        .distinct
    if (nFilteredSeq.isEmpty) results
    else helper(nFilteredSeq, results ++ nFilteredSeq)
  }
  helper(seq, results)
}

// Immutable Set
def solve(p: Definition): Set[p.T] = {
  simpleLogic(p)(p.initData, iSet ++ p.initData)
}
```

Listing 3.9: Approach A – Scala I

```
def solveMutableSet(p: Definition): (Set[problemDef.T], Long) = {
  import p._
  val results = mSet ++ initData
  def helper(currentSeq: Seq[T]) {
    val nFilteredSeq =
      currentSeq
        .flatMap(generators(_))
        .filterNot(results.contains)
        .distinct
    results ++= nFilteredSeq
    if (!nFilteredSeq.isEmpty) helper(nFilteredSeq)
  }
  helper(initData); results
}
```

Listing 3.10: Approach A - Scala M

3.4.2 F#

The F# sequential implementation (*F# (Sequential)*) in Listing 3.11 uses the `MutableSet.add` function to test for inclusion and add an element to the set, instead of checking and then adding together all the new elements.

```
let solve<'T when 'T: equality>
  { initData = initData; generators = generators } =
    let foundSoFar = MutableSet.ofSeq initData
    let rec helper current =
        if Seq.isEmpty current then
            foundSoFar :> seq<'T>
        else
            let nCurrent =
               current
               |> Seq.collect generators
               |> Seq.filter (MutableSet.add foundSoFar)
               |> Seq.toArray
            helper nCurrent
     helper <| Seq.toArray initData
```

Listing 3.11: Approach A – F#

Note the explicit upcast of `foundSoFar` to `seq<'T>`: we want to hide the actual implementation of the result. Ideally the upcast would be to `System.Collections.Generic.ISet<'T>` but the Keys property `System.Collection.Concurrent.ConcurrentDictionary<'K, 'V>` we use in other implementations does not support it and we wanted to have a common interface for all our implementations.

Also Seq.toArray is essential to avoid computing of `nCurrent` at every enumeration.

3.4.3 Erlang

The Erlang implementation (*Erlang (Sequential)*) in Listing 3.12 is similar to the F# one: each element is atomically inserted in the set if not already present.

```
solve_helper([], _Generators) ->
  ets:match(hashset, '$1');
solve_helper(Current, Generators) ->
  NCurrent = lists:flatmap(fun(C) ->
    [X | X <- Generators(C), ets:insert_new(hashset, {X}) end]
  end, Current),
  solve_helper(NCurrent, Generators).

solve(#definition{init_data = InitData, generators = Generators}) ->
  ets:new(hashset, [set, named_table, public]),
  Result = solve_helper(InitData, Generators),
  ets:delete(hashset),
  Result.
```

Listing 3.12: Approach A – Erlang

We use the ets module to:

- create a `set` that is configured to accept reads and writes from any process (`public`) and have a globally visible name (`named_table`) using the `new` function,
- atomically insert an element in the set if not already present, using the `insert_new` function,
- retrieve all elements from the set using the `match` function, and
- delete the set at the end of the computation.

3.5 Approach B - Parallel

We present two variations: the first is similar to approach A, with X_{k+1} being computed in parallel; the second uses a concurrent set where the elements are inserted during the parallel section. We present implementations only in Scala and F# which support the notion of parallel collection manipulation (we could have implemented a parallel map on lists using Erlang processes but we do not find it of value).

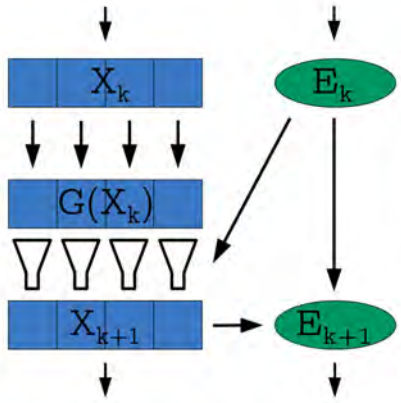


Figure 3.2: Logic of Approach B – Variation A

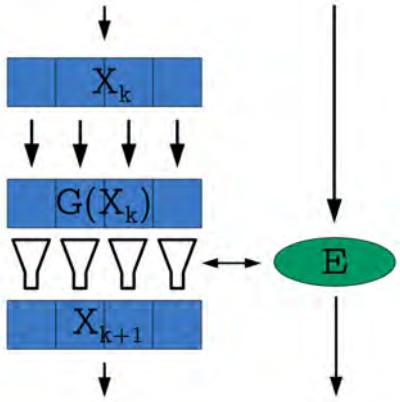


Figure 3.3: Logic of Approach B – Variation B

3.5.1 Scala

The first variation (*Scala (ParSeq)*) in Listing 3.13 shares its logic with approach A. Before passing `initData` to the logic we transform it to a parallel collection (`ParSeq`) by calling its `par` method. This is possible because `simpleLogic` is implemented in terms of `GenSeq`, a common superclass of `Seq` and `ParSeq`.

```
def simpleLogic(p: Definition)
  (seq: GenSeq[p.T], results: Set[p.T]): Set[p.T] = { … }

// Sequential
def solve(p: Definition): (Set[p.T], Long) = {
  simpleLogic(p)(p.initData, p.initData.to[Set])
}

// ParSeq
def solveParSeq(p: Definition): (Set[p.T], Long) = {
  simpleLogic(p)(p.initData.par, p.initData.to[Set])
}
```

Listing 3.13: Approach B – Scala – Variation A and Comparison with Approach A

The second variation in Listing 3.14 (*Scala (ParSeq & ConcurrentMap)*) uses `ConcurrentMap[T, Unit]` as concurrent set. The `putIfAbsent` method behaves similarly to the `ets:insert_new` function of Erlang but returns an `Option` with the old value if there was one in place of just a boolean value. At the end of the computation the `keySet` method is called to return the desired Orbit set.

```
def solveParSeqWithConcurrentMap(p: Definition): Set[p.T] = {
  import p._
  val results = cMap[T]
  def helper(currentSeq: GenSeq[T]) {
    val nFilteredSeq = currentSeq flatMap {
      generators(_) filter (results.putIfAbsent(_, ()).isEmpty)
    }
    if (!nFilteredSeq.isEmpty) helper(nFilteredSeq)
  }
  helper(initData.par); results.keySet
}
```

Listing 3.14: Approach B – Scala – Variation B

3.5.2 F#

The first variation in Listing 3.15 uses PLINQ for parallel processing. Its logic is closer to the first Scala variation of this approach: instead of checking for inclusion and simultaneously adding the elements to the set, we first check for inclusion and then add all unique new elements to the set.

```
let solvePLinq<'T when 'T: equality>
  { initData = initData; generators = generators } =
    let foundSoFar = MutableSet.ofSeq initData
    let rec helper current =
        if Seq.isEmpty (current:seq<'T>) then
            foundSoFar :> seq<'T>
        else
            let nCurrent =
                current
                    .AsParallel()
                    .SelectMany(generators)
                    .Where(not << MutableSet.contains foundSoFar)
                    .Distinct()
                    .ToList()
            MutableSet.unionWith foundSoFar nCurrent
            helper nCurrent
```

Listing 3.15: Approach B – F# – Variation A

It should be noted that PLINQ has different semantics from Scala parallel collections. The `AsParallel()` method does not parallelize a collection; it creates a parallel query that is evaluated upon enumeration of the result. Therefore calling `ToList()` is essential to avoid re-evaluation of `nCurrent`.

There is also a stylistic difference with the sequential version: F# does not provide a predefined wrapper module for PLINQ contrary to the case of the `Seq` module and LINQ; so instead of the idiomatic piping operators and calls to module functions we use the extension methods of PLINQ.

For the second variation we use `ConcurrentSet<T>`. We implemented one version (*F# (PLINQ)*) using PLINQ (Listing 3.16) and a second (*F# (Parallel.ForEach)*) that uses the `Parallel.ForEach` method combined with a `ConcurrentBag` as a way to simulate a parallel collection (Listing 3.17).

```
let solvePLinq2<'T when 'T: equality> M
  { initData = initData; generators = generators } =
    let foundSoFar = ConcurrentSet.create M 1000000
    let rec helper current =
        if Seq.isEmpty (current:seq<'T>) then
            foundSoFar.Keys :> seq<_>
        else
            let nCurrent =
                current
                   .AsParallel()
                   .SelectMany(generators)
                   .Where(ConcurrentSet.add foundSoFar)
                   .ToList()
                helper nCurrent
    for x in initData do
        ConcurrentSet.add foundSoFar x |> ignore
    helper initData
```

Listing 3.16: Approach B – F# – Variation B (PLINQ)

```
let solveParallelForEach<'T when 'T: equality> M
  { initData = initData; generators = generators } =
    let foundSoFar = ConcurrentSet.create M 1000000
    let rec helper current =
        if Seq.isEmpty (current:seq<'T>) then
            foundSoFar.Keys :> seq<_>
        else
            let res = ConcurrentBag<'T>()
            Parallel.ForEach(current,
                generators
                >> Seq.filter (ConcurrentSet.add foundSoFar)
                >> Seq.iter res.Add
            ) |> ignore
            helper res
    for x in initData do
        ConcurrentSet.add foundSoFar x |> ignore
    helper initData
```

Listing 3.17: Approach B – F# – Variation B (Parallel.ForEach)

The `Parallel` class provides library-based data parallel replacements for common operations such as for loops, for each loops, and execution of a set of statements. These replacements are designed to be blocking and consequently are rather restrictive.

Evident in both versions is the limited API of `ConcurrentSet` (which resembles the capabilities of the underlying `ConcurrentDictionary`). For example, there is no function that creates a `ConcurrentSet` with customizable level of concurrency (represented by the `M` parameter) and simultaneously bulk loads it with elements from an existing collection; each element has to be inserted separately.

3.6 Approach C - Futures

In this approach we abandon the stepwise logic of approaches A and B, aiming to remove the bottleneck of waiting for a step to finish before the next step begins. A central role in this approach belongs to the coordinator, an actor that:

- partitions sequences to chunks of a specified size G
- starts an asynchronous job of computing the new elements
- determines when the *Orbit* set has been fully computed

The coordinator handles the following two messages:

Start contains the initial data and a way to return the resulting *Orbit* set

Result contains the data computed by an asynchronous job

This approach has also two variations, distinguished by whether the variation uses a concurrent set or not.

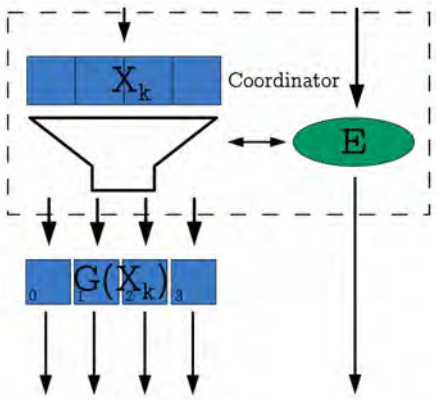


Figure 3.5: Logic of Approach C – Variation A

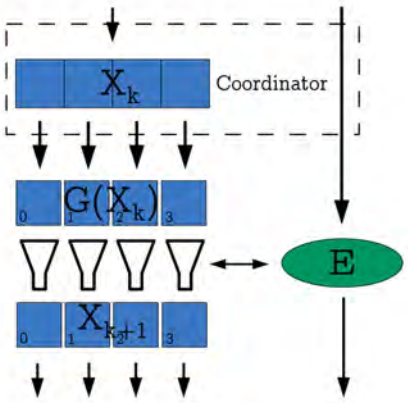


Figure 3.4: Logic of Approach C – Variation B

3.6.1 Scala

The coordinator is defined using the actor domain specific language that is provided by Akka that helps to reduce the amount of code needed for simple actors. The coordinator has an initial behavior that handles `Start` messages. Upon receiving the `Start` message the coordinator assumes the main coordinator logic that reacts to `Result` messages and will eventually store the result to the promise when the computation finishes.

We define the messages as Scala case classes instead of plain classes so that they are immutable and can be used in pattern matching. Case classes also provide structural equality and hashing, as well as an intutitive override of the `toString` method, but we do not use that functionality.

```
// Message definition
case class Result(data: Seq[T])
case class Start(data: Seq[T], promise: Promise[Set[T]])
```

Listing 3.18: Scala Messages

The behavior switch is supported by the `become` method: the argument of `become` determines the actor reaction to messages henceforth. At least one call to this method is needed inside the `Act` body to designate the initial behavior; the reaction logic may have other `become` calls that alter the desired behavior according to the received messages.

```
new Act {
  def chunkAndSend(data: Seq[T], coordinator: ActorRef): Int = {
    …
    future { Result(chunk.flatMap(generators(_)).distinct) } pipeTo coordinator
    …
  }
  var foundSoFar = iSet[T] // or val foundSoFar = mSet[T]
  var remaining = 0

  def loop(replyPromise: Promise[Set[T]]): Receive = {
    case Result(data) =>
      val filteredData = data.filterNot(foundSoFar.contains)
      foundSoFar ++= filteredData
      val jobs = chunkAndSend(filteredData, self)
      if (remaining > 1 || jobs > 0) remaining += jobs - 1
      else replyPromise.success(foundSoFar)
  }
  become {
    case Start(data, promise) =>
      foundSoFar ++= data
      become(loop(promise))
      val jobs = chunkAndSend(data, self); remaining += jobs
  }
}
```

Listing 3.19: Scala Coordinator – Variation A (non-concurrent result set)

The `chunkAndSend` function (Listing 3.20) is used by the coordinator to create the asynchronous jobs. It partitions the input to chunks and creates a job for each of them as a `scala.concurrent.Future`. The result of the Future is then piped to the coordinator using a predefined Akka pattern for future-actor communication.

```
// Partition data to chunks, create jobs and return number of created jobs
def chunkAndSend(data: Seq[T], coordinator: ActorRef): Int = {
  var jobs = 0
  for (chunk <- data.grouped(G)) {
    import akka.pattern.pipe
    future { … } pipeTo coordinator
    jobs += 1
  }
  jobs
}
```

Listing 3.20: Scala partitioning and job-creating function

The Scala implementations are wrappers for the coordinator actor (Listing 3.22). The `resultPromise` is sent along with `initData` to the coordinator and then we wait for its asynchronous completion, when we will be able to access the result contained therein.

```
new Act {
  def chunkAndSend(data: Seq[T], coordinator: ActorRef): Int = {
    …
    future { Result(chunk.flatMap {
      generators(_).filter(foundSoFar.putIfAbsent(_, ()).isEmpty)
    })} pipeTo coordinator
    …
  }

  val foundSoFar = cMap[T]
  var remaining = 0

  def loop(replyPromise: Promise[Set[T]]): Receive = { case Result(data) =>
    val jobs = chunkAndSend(data, self)
    if (remaining > 1 || jobs > 0) remaining += jobs - 1
    else replyPromise.success(foundSoFar.keySet)
  }
  become {
    case Start(data, promise) =>
      foundSoFar ++= data.map((_, ()))
      become(loop(promise))
      val jobs = chunkAndSend(data, self); remaining += jobs
  }
}
```

Listing 3.21: Scala Coordinator – Variation B (concurrent result set)

```
def solveFutures(p: Definition, G: Int): Set[p.T] = {
  import p._, concurrenf.{future, duration, promise}

  implicit val system = ActorSystem("system") // Define the actor system
  val coordinator = actor("coordinator")(new Act { /* coordinator logic */ })

  val resultPromise = promise[Set[T]]

  coordinator ! Start(initData, resultPromise) // Start computation
  val res = // Await asynchronous result
    Await.result(resultPromise.future, duration.Duration.Inf)

  system.shutdown() // Stop all actors and shutdowns the system
  res // Return result
}
```

Listing 3.22: Approach C – Scala

In Listing 3.22 we present the solver of this approach. We can see that even for a simple actor several lines of code are required: to create the actor system, to create the actor itself and to manually shutdown the whole system at the end of the asynchronous computation.

3.6.2 F#

The F# coordinator accepts messages of type `Message<'T>` as defined in Listing 3.23.

```
type Message<'T> =
| Start of array<'T> * AsyncReplyChannel<seq<'T>>
| Result of array<'T>
```

Listing 3.23: F# Messages

We have defined the `chunkAndSend` function to take two parameters in addition to the data:

`sendLogic` is the functionality that creates the asynchronous jobs and is specific for each variation of this implementation approach.

`chunker` is the function that partitions the data.

```
let chunkAndSend sendLogic chunker data =
    let mutable jobs = 0
    for chunk in data |> chunker do
        sendLogic chunk
        jobs <- jobs + 1
    jobs
```

Listing 3.24: F# partitioning and job-creating function

The `sendLogic` parameter is a function that creates the asynchronous jobs. It has a separate implementation for each variation and is defined as a function combination using the `>>` operator of F# (Listing 3.25).

```
let logicMutableSet generators coordinator =
    Seq.collect generators >> Seq.distinct
    >> Seq.toArray
    >> Result
    >> Agent.post coordinator

let logicConcurrentDictionary generators foundSoFar coordinator =
    Seq.collect generators >> Seq.filter (ConcurrentSet.add foundSoFar)
    >> Seq.toArray
    >> Result
    >> Agent.post coordinator
```

Listing 3.25: Job logic of each variation

The `Seq.toArray` function calls force the evaluation of the sequence to happen in the asynchronous jobs; without them, the computation is delayed until the enumeration of the sequence at the coordinator, making the execution practically sequential.

Like in Scala, the coordinator has an initial behavior accepting a `Start` message and a main behavior that reacts to `Result` messages. The behavior change in F# is a simple function call to the new behavior. But, unlike Scala, the messages that the coordinator accepts are part of its type and fixed; hence any possible message needs to be defined as case of a discriminated union, which is the type argument of the coordinator.

The two coordinator variations are shown in Listings Listing 3.26 and Listing 3.27.

```
let coordinatorMutableSet chunkAndSend G inbox =
    let foundSoFar = MutableSet<'T>()
    let rec start() =
        async {
            let! Start(initData, replyChannel) = Agent.receive inbox
            MutableSet.unionWith foundSoFar initData
            let jobs = chunkAndSend inbox (Seq.chunked_opt_2 G) initData
            return! loop replyChannel jobs
        }
    and loop replyChannel remaining = async {
        let! Result data = inbox.Receive()
        let data = data |> Array.filter (not << contains foundSoFar)
        MutableSet.unionWith foundSoFar data
        let jobs = chunkAndSend inbox (Seq.chunked_opt_2 G) data
        if remaining > 1 || jobs > 0 then
            return! loop replyChannel (remaining + jobs - 1)
        else
            AsyncReplyChannel.reply replyChannel <| upcast foundSoFar
    }
    start()
```

Listing 3.26: F# Coordinator – Variation A (non-concurrent result set)

```
let coordinatorConcurrentSet foundSoFar chunkAndSend M G inbox =
    let rec start() =
        async {
            let! Start(initData, replyChannel) = Agent.receive inbox
            let jobs = chunkAndSend inbox (Seq.chunked_opt_2 G) initData
            return! loop replyChannel jobs
        }
   and loop replyChannel remaining = async {
       let! Result data = inbox.Receive()
       let jobs = chunkAndSend inbox (Seq.chunked_opt_2 G)  data
       if remaining > 1 || jobs > 0 then
           return! loop replyChannel (remaining + jobs - 1)
       else
           AsyncReplyChannel.reply replyChannel <| upcast foundSoFar.Keys
   }
   start()
```

Listing 3.27: F# Coordinator – Variation B (concurrent result set)

The coordinator body is implemented as a recursive asynchronous computation. The asynchronous value bindings and function calls are expressed using the F# computation expression syntax and in particular the keywords `let!` and `return!`.

For the first variation there is one version with the jobs being implemented as asynchronous computation and one version using `System.Threading.Tasks.Task`s (Listing 3.28).

```
let solveAsync<'T when 'T: equality> G
    { initData = (initData:seq<'T>); generators = generators } =
    let chunkAndSend inbox =
        let logic = logicWithMutableSet generators inbox
        genericChunkAndSendComputations <|
            fun chunk -> Async.Start <| async { logic chunk }
    let coordinator = Agent.start <| agentLogicMutableSet chunkAndSend G
    run coordinator initData

let solveTask<'T when 'T: equality> G
    { initData = (initData:seq<'T>); generators = generators } =
    let chunkAndSend inbox =
        let logic = logicWithMutableSet generators inbox
        genericChunkAndSendComputations <|
            fun chunk -> Task.Factory.StartNew(fun _ -> logic chunk) |> ignore
    let coordinator = Agent.start <| agentLogicMutableSet chunkAndSend G
    run coordinator initData
```

Listing 3.28: Approach C – F# – Variation A (Async Workflows & Tasks)

The second variation implementation is shown in Listing 3.29.

```
let solveConcurrentDictionary<'T when 'T: equality> M G
    { initData = initData; generators = generators } =
    let chunkAndSend foundSoFar inbox =
        let logic = logicConcurrentDictionary generators foundSoFar inbox
        chunkAndSend <| fun chunk ->
            Task.Factory.StartNew(fun _ -> logic chunk) |> ignore
    let coordinator = Agent.start <| fun inbox ->
        let foundSoFar = ConcurrentSet.create<'T> M 100000
        coordinatorConcurrentSet foundSoFar (chunkAndSend foundSoFar) M G inbox
    run coordinator initData
```

Listing 3.29: Approach C – F# – Variation B (ConcurrentDictionary)

All implementations of this category use a `run` function to "ask" the coordinator for the result (Listing 3.30). The `Agent.postAndAsyncReply` function is used to provide a way of getting back the result set.

```
let run coordinator initData =
    Agent.postAndAsyncReply coordinator <|
        fun channel -> Start(Array.ofSeq initData, channel)
    |> Async.RunSynchronously
```

Listing 3.30: The `run` *function*

3.6.3 Erlang

Both coordinator and asynchronous jobs are implemented as Erlang processes: the coordinator as a persistent actor (with function `solve_conc_helper/4` as body) and the jobs as transient computations (that are spawned by the coordinator).

```
solve_conc(#definition{init_data = InitData, generators = Generators},
  G, Implementation) ->
  ets:new(hashset, [set, named_table, public, {read_concurrency, true},
    {write_concurrency, true}]),
  Master = self(),
  Coordinator = spawn(fun() -> Implementation(Master, Generators, G, 1) end),
  Coordinator ! InitData,
  receive finish -> ok end,
  Result = ets:match(hashset, '$1'),
  ets:delete(hashset),
  Result.
```

Listing 3.31: Approach C – Erlang

In Listing 3.31 we present the wrapper function for both variations; the parameter `Implementation` is the coordinator logic. Notice the creation of the ETS table beside the already described options, the `read_concurrency` and `write_concurrency` options enable concurrent access.

The coordinators uses an altered version of the presented `split:split/2`: `split:split/3` that in addition to partitioning the elements applies a function to each resulting chunk.

```
solve_conc_helper(Master, _Generators, _G, 0) ->
  Master ! finish;
solve_conc_helper(Master, Generators, G, Remaining) ->
  receive Current ->
    FilteredCurrent = [Elem | Elem <- Current, ets:insert_new(hashset, {Elem})],
    Coordinator = self(),
    Count = split:split(G, FilteredCurrent, fun(Chunk) ->
      spawn(fun() ->
        Coordinator ! lists:usort(lists:flatmap(Generators, Chunk))
      end)
    end),
    solve_conc_helper(Master, Generators, G, Remaining + Count - 1)
  end.
```

Listing 3.32: Erlang Coordinator – Variation A (non-concurrent set)

In the first variation (Listing 3.32) the coordinator filters the `Current` elements and creates processes that apply the `Generators` to each `Chunk`, remove duplicate elements using the `lists:usort` function and send the result back to the `Coordinator`.

```
solve_conc_helper(Master, _Generators, _G, 0) ->
  Master ! finish;
solve_conc_helper(Master, Generators, G, Remaining) ->
  receive Current ->
    Coordinator = self(),
    Count = split:split(G, Current, fun(Chunk) ->
      spawn(fun()->
        NCurrent = lists:flatmap(fun(C) ->
            [X | X <- Generators(C), ets:insert_new(hashset, {X}) end]
        end, Chunk),
        Coordinator ! NCurrent
      end)
    end),
    solve_conc_helper(Master, Generators, G, Remaining + Count - 1)
  end.
```

Listing 3.33: Erlang Coordinator – Variation B (concurrent set)

The coordinator in the second variation (Listing 3.33) is limited to creating processes that apply `Generators` to each `Chunk`, insert the new elements to `hashset` and send them back to the `Coordinator`.

3.7 Approach D - Persistent Workers

The implementations of this approach resemble the second variations of the previous approach. The distinctive difference is the use of persistent workers in place of transient asynchronous jobs for computing the new elements. The coordinator partitions the elements to chunks, as before, but each chunk is sent to a worker which is chosen according to a Round-Robin routing policy.

This category mainly aims to examine the actor capabilities of Scala, F# and Erlang, so for each language we present only one version that uses a concurrent set.

3.7.1 Scala

In this implementation we use the full-fledged actor API as it is more flexible than the actor DSL we used previously.

Besides the `Start` and `Result` messages (Listing 3.18), there is also the `Job` message (Listing 3.34) that contains the chunk to be processed by a worker. We could choose to send the chunks without wrapping but due to the erasure of type `T` the chunk received would be a `Seq[Any]`. Therefore we decided to use the `Job` message.

```
case class Job(chunk: Seq[T])
```

Listing 3.34: The `Job` *Message*

In Listing 3.35 we define the `Worker` class: it is an actor that reacts to `Job` messages by computing the new elements and replying to the `sender`, which is the actor who sent the message.

```
class Worker(map: collection.concurrent.Map[T, Unit]) extends Actor {
   def receive = { case Job(chunk) =>
     sender ! Result(chunk.flatMap {
       generators(_).filter(map.putIfAbsent(_, ()).isEmpty)
     })
   }
}
```

Listing 3.35: The `Worker` *class*

In Listing 3.36 we present the Coordinator class:

- `M` workers are created under a `RoundRobinRouter`, whose `ActorRef` is stored in the `workers` value.
- `The chunkAndSend` function partitions the data and sends the chunks to the workers
- The `loop` and `receive` functions resemble the states of the corresponding coordinator of the previous category; again `become` is used for state switching with the difference that, in this implementation, it is a member of the context value instead of a direct member of the actor.

All the actors of this system belong to a single hierarchy: the `context.actorOf` method creates the actor of its arguments as child of the current actor while the `Props(…).withRouter` method creates a router with the routees (W1-W4) as children. As shown in Figure 3.6 the actor hierarchy of this implementation has the coordinator as top-level actor, below it stands the router and at the bottom the workers.

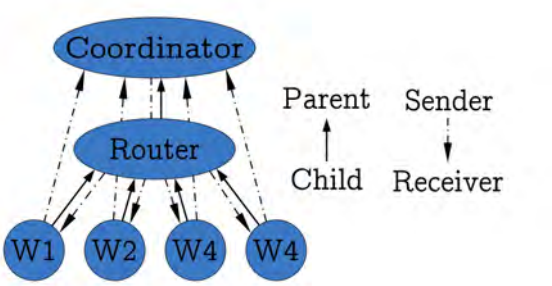


Figure 3.6: Actor Hierarchy & Message Routes

```
class Coordinator extends Actor {
  val foundSoFar = cMap[T]
  val workers = context actorOf {
    Props(new Worker(foundSoFar)) withRouter RoundRobinRouter(nrOfInstances = M)
  }

  var remaining = 0
  def chunkAndSend(data: Seq[T]): Int = {
    var jobs = 0
    for (chunk <- data.grouped(G)) {
      workers ! Job(chunk); jobs += 1
    }
    jobs
  }

  def loop(replyPromise: Promise[Set[T]]): Receive = {
    case Result(data) =>
      val jobs = chunkAndSend(data)
      if (remaining > 1 || jobs > 0)
        remaining += jobs - 1
      else
        replyPromise.success(foundSoFar.keySet)
  }

  def receive = {
    case Start(data, promise) =>
      foundSoFar ++= data.map((_, ()))
      context become loop(promise)
      val jobs = chunkAndSend(data);
      remaining += jobs
  }
}
```

Listing 3.36: The `Coordinator` *class*

In Listing 3.37 we present the implementation structure.

```
def solveActorWorkersConcurrentMap(p: Definition, M: Int, G: Int) = {
  … // Class definitions for messages, coordinator and worker

  // Creates actor system and coordinator
  val system = ActorSystem("system")
  val coordinator = system actorOf Props[Coordinator]

  // Creates, sends and waits for the result of the promise
  val resultPromise = concurrent.promise[Set[T]]
  coordinator ! Start(initData, resultPromise)
  val res = Await.result(resultPromise.future, concurrent.duration.Duration.Inf)

  // Shutdowns the system and returns the result
  system.shutdown()
  res
}
```

Listing 3.37: Approach D – Scala

3.7.2 F#

The F# implementation, shown in Listing 3.38, shares its core logic with the previous approach (Listing 3.25).

```
let solveWorkersAndConcurrentSet<'T when 'T:equality> M G
    { initData = (initData:seq<'T>); generators = generators } =
    let i = ref 0
    let chunkAndSend workers _  = chunkAndSendToWorkers M workers i
    let workPile = Agent.start <| fun inbox ->
        let foundSoFar = ConcurrentSet.create M 100000
        let workers = Array.init M <| fun _ -> Agent.start(fun workerInbox ->
            let rec loop () = async {
                let! chunk = Agent.receive workerInbox
                logicConcurrentDictionary generators foundSoFar inbox chunk
                return! loop()
            }
            loop()
        )
        agentLogicConcurrentSet foundSoFar (chunkAndSend workers) M G inbox
    run workPile initData
```

Listing 3.38: Approach D – F#

The worker actors are stored in an array field of the coordinator and the routing logic is embedded in the `chunkAndSendToWorkers` function (Listing 3.39): the reference `i` stores the index for the to-be-selected actor and is increased at each iteration of the loop, thus providing the desired routing behavior.

```
let chunkAndSendToWorkers M workers i chunker data =
    let mutable jobs = 0
    for chunk in data |> chunker do
        (Array.get workers (!i%M), chunk)  ||> Agent.post
        incr i; jobs <- jobs + 1
    jobs
```

Listing 3.39: The `chunkAndSendToWorkers` *function*

3.7.3 Erlang

The Erlang implementation uses persistent processes for the coordinator and the workers. Similarly to the Scala implementation, the coordinator does not have direct access to the workers. In order to implement the round robin policy we need to keep some state, which should not be exposed to the coordinator logic; therefore we introduces a router that manages the messages for the worker actors.

```
coordinator(Master, Workers, _G, 0) ->
  Master ! stop,
  Workers ! stop;
coordinator(Master, Workers, G, Remaining) ->
  receive
    Current ->
      Count = split:split2(G, Current, fun(Chunk) -> Workers ! Chunk end),
      coordinator(Master, Workers, G, Remaining + Count - 1)
  end.
```

Listing 3.40: Erlang Coordinator

In addition to the domain specific functionality, the Erlang coordinator is also responsible for terminating the router and the workers. When the result is ready, the coordinator sends it back to the wrapper function and sends a `stop` message to the router, which broadcasts `stop` messages to all the workers.

In Listing 3.41 we present the worker actor. It reacts to `Chunk` messages by computing the new elements for each `Chunk` received and to `stop` messages by stopping itself.

```
worker(Coordinator, Generators) ->
  receive
    stop -> ok;
    Chunk ->
      NCurrent = lists:flatmap(fun(C) ->
          [X | X <- Generators(C), ets:insert_new(hashset, {X}) end]
        end, Chunk),
      Coordinator ! NCurrent,
      worker(Coordinator, Generators)
  end.
```

Listing 3.41: Erlang Worker

The absence of mutable variables in Erlang dictated that the router would be a separate actor (Listing 3.42) whose state is the worker that will be chosen next. All actors are in a queue; when a message arrives, a worker is removed from the queue and is put at its end, while the received message is forwarded to it.

```
round_robin_router(Workers) ->
  receive
    stop ->
      lists:foreach(fun(Pid) -> Pid ! stop end, queue:to_list(Workers));
    Msg ->
      {{value, Pid}, NWorkers} = queue:out(Workers),
      Pid ! Msg,
      round_robin_router(queue:in(Pid, NWorkers))
  end.

create_workers_under_rooter(M, Func) ->
  spawn(fun() ->
    Workers = [ spawn(fun() -> Func(I) end) || I <- lists:seq(1, M) ],
    round_robin_router(queue:from_list(Workers))
  end).
```

Listing 3.42: Erlang Round-Robin Router

The wrapper function is presented in Listing 3.41.

```
solve_conc_workers(#definition{init_data = InitData, generators = Gens},
 M, G) ->
  ets:new(hashset, [set, named_table, public, {read_concurrency, true},
   {write_concurrency, true}]),
  Master = self(),
  Coordinator = spawn(fun() ->
    Coord = self(),
    Workers = create_workers_under_rooter(M, fun(_I)-> worker(Coord, Gens) end),
    coordinator(Master, Workers, G, 1)
  end),
  Coordinator ! InitData,
  receive finish -> ok end,
  Result = ets:match(hashset, '$1'),
  ets:delete(hashset),
  Result.
```

Listing 3.43: Approach D – Erlang

3.8 Code metrics

In Table 3.2 we present two metrics for all implementations: lines of code and token count. For each metric we provide a breakdown to a part that is shared among two or more implementations and a part unique for each of them, along with a total number; those numbers contain empty lines and brackets/parentheses.

Table 3.2: Lines of Code and Token Count for each Implementation

Implementation	Total LOC	Total Tokens	Unique LOC	Unique Tokens	Shared LOC	Shared Tokens
F# (Sequential)	14	55	14	55	-	-
F# (PLINQ)	17	55	17	55	-	-
F# (PLINQ & ConcurrentDictionary)	17	63	17	63	-	-
F# (Parallel.Foreach & ConcurrentDictionary)	17	74	17	74	-	-
F# (Persistent Actors)	57	254	15	78	42	176
F# (Persistent Actors & ConcurrentDictionary)	55	248	17	87	38	161
F# (Async)	48	215	7	48	41	167
F# (Task)	48	215	7	48	41	167
F# (Task & ConcurrentDictionary)	46	216	9	64	37	152
Scala I/M (Sequential)	17	57	4	17	13	40
Scala (ParSeq)	20	69	7	29	13	40
Scala (ParSeq & ConcurrentMap)	12	52	12	52	-	-
Scala I/M (Future)	46	148	46	148	-	-
Scala C (Future & ConcurrentMap)	48	147	48	147	-	-
Scala (Persistent Actors)	57	172	57	172	-	-
Scala (Persistent Actors & ConcurrentMap)	58	178	58	178	-	-
Erlang (Sequential)	13	41	13	41	-	-
Erlang (Process)	25	94	13	51	11	43
Erlang (Process & Concurrent ETS)	27	97	15	54	11	43
Erlang (Persistent Actors & Concurrent ETS)	49	154	46	154	-	-

These metrics reflect some interesting aspects:

- Code sharing between implementations that use concurrent and sequential data structures is limited, as the logic differs.
- In Scala and F#, type inference affects the refactoring of common code to a reusable functions. The global type inference algorithm of F# supports that refactoring with relatively few added tokens, while the benefits of code reuse in Scala are often

less important than the incurring token overhead due to mandatory type annotation of function arguments.

- In Scala, the collection library enables code reuse between sequential and parallel implementations by abstracting over the concrete implementation of the collection. Because the core implementation function uses an interface, whether the execution is sequential or parallel depends only on the object that is passed as argument.
- *Scala I/M (Future)* implementations are almost identical; yet the share no code due to semantic differences between `val` and `var`.

3.9 Implementation Remarks

During the implementation period of this thesis we encountered several surprises, oddities and delights that are worth mentioning.

Types
In terms of type systems Scala, F# and Erlang have noticeable differences:

- Scala has a Turing-complete type system with a vast feature set.
- F# has a type system that resembles its ML heritage, though it does not have the expressive module system of its siblings (SML, OCaml).
- Erlang is dynamically typed so there is not much to say.

The abundance of features in the type system of Scala is both a blessing and a curse. It was enticing to experiment with its capabilities in our implementations. First, we hid the element type T from parts of the implementation that did not need it by encapsulating it as a member of the problem definition instead of representing it as type parameter. Then we used type classes (in form of implicit context in the problem definition) to write one implementation that would be used for both `Long` and `BigInteger` elements. Last, we used subtyping in our main method to unify the cases of `Long` and `BigInteger` problem definitions.

During implementation we encountered several intricacies. Early on we came across the beast called type erasure: the element type was erased if we defined auxiliary types using generics. We were forced to redefine auxiliary types (like messages) in every implementation to avoid casts; of course we attempted to extract them as a separate dependency but the result was more convoluted than plain code repetition. In addition we witnessed some bugs: using path depended types caused the appearance of existential types that prohibited a function from passing type checking; and the type inference algorithm would not deduce a type as a subtype of another. These bugs were observed using the version 2.10.1 of the Scala compiler but they were fortunately both fixed in the 2.10.2 update.

The F# experience with types was less adventurous. We used plain generics to represent our problem definitions and we worked around the absence of type classes with statically resolved type parameters that helped us avoid duplicating code between `int64` and `bigint` implementations. On the other hand, global type inference assisted the extraction of common functionality in separate functions as we did not have to declare any argument types. However the type inference algorithm of F# does not work with object oriented features: it does not automatically subsume a type into its supertypes (explicit upcasts are required) and it cannot infer the type of an object based on an access to field or method that this type is known to have.

From our experiences in Scala and F# we were satisfied from their type system capabilities, though we would recommend against combining several exotic features from the Scala type system since there are still rough edges like the aforementioned bugs.

Libraries

In terms of featured libraries, the experience was similar to the above. Scala has the most comprehensive standard library, at least in terms of collection functionality. It offers several different collection categories: mutable and immutable, lazy and eager, concurrent and parallel, along with wrappers for almost every Java collection, which make them appear just like their Scala counterparts. More importantly, all collections belong to a single hierarchy and therefore have uniform usage patterns, while there is an excess of functionality shared among them. In order to benefit from them one has to thoroughly study their behavior to find which is better for one's needs and to be able to reason about their behavior.

Contrary to Scala, F# libraries offer basic functionality and they extensively rely to the Base Class Library. There are only immutable F# collections whose implementation is not very optimized (for instance: F# immutable set), so one has to use the BCL collection that offers the desired functionality. We have to note that BCL collections have an object oriented design so their use in F# feels somewhat unnatural and rules out any assistance from type inference. Furthermore, sequence expressions are a feature to appreciate, since they enable easy sequence manipulation (and every collection is a sequence) and their on-demand evaluation semantics avoids creation of intermediate collections during sequence manipulation; however one has to be careful and most considerate when reasoning about their performance as one may encounter surprising performance characteristics arising from this on-demand behavior.

Erlang, on the other hand, has a variety of functional data structures, and ETS, the de facto choice when high performance access to shared memory is required. On the other hand, it requires plenty of argument forwarding as it does not support global value declarations, and the introduction of additional levels of indirection to implement functionality that could be easily implemented if mutable variables were supported – in our case the addition of the router actor. Consequently, all implementations are plain and functional in nature; which is not necessarily bad, considering that the resulting code is straightforward and easy to reason about.

Feature Maturity and Library Stability

During the prolonged period we were occupied with this study, we developed an overall view regarding the maturity and stability of the language implementations. Erlang and F# had a mostly stable feature set: few new features were added in during our observation period which were mostly compatible with the previous versions, and few bugs were encountered in their libraries. On the contrary, Scala experienced several feature and library additions and revisions that were not always backward compatible like modification of the future library and its transfer from the Akka binaries to the standard library. As a result, Scala is still evolving rapidly and one should be prepared for feature deprecation, functionality modification, implementation bugs and features that do not cooperate well with each other.

Chapter 4

Experimental Evaluation

The second part of this thesis involves the study performance, scalability and other execution characteristics of the examined languages and runtimes. For that purpose we executed the implementations of the previous chapter for an Orbit definition we composed and we measured the effect of several runtime parameters. In the main body of this chapter we comment on the results of that execution and try to explain the observed behavior. Finally we provide an overview of those results along with some other remarks.

4.1 Choosing the right benchmark

We needed a benchmark that:

- requires enough time to complete, in order to minimize the effect of "computational" noise,
- computes elements representable with both 64-bit integers and integers of arbitrary size, to examine the effect of different number representations (in Erlang integers are arbitrary sized so we),
- results in an Orbit set of configurable size, and
- allows the implementations to execute faster in environments with more computational resources

Therefore we composed a benchmark that given a number N computes numbers between *0* and N, using simple functions, which contain additions and multiplications. These functions have trivial computational cost so we chose to introduce a delay to each of them; that delay increases the computation time as needed without affecting the result of the computation.

As simple loop-based delays are optimized away by the language/JIT compilers, we devised a delay function that is more resistant to optimizations (Listing 4.1). This function uses variables `h` and `l`, initialized at `2*d` and `0` respectively; `h` is decreased and `l` is increased until their values become equal after `d` iterations.

```
def delay(d:Int) = {
  var h = 2*d
  var l = 0
  while (h>l) {
    h -= 1; l += 1
  }
  h-l
}
```

Listing 4.1: The `delay` *function*

For most of our measurements we chose the parameters N and d based on trial and error. At first we tried using big values (5-20 million) for N but that resulted in out-of-memory errors in our Scala implementations. We tried to work around those errors by altering the JVM startup parameters. We configured the size of the permanent generation space (*PermGen*) to 256MB instead of the default 64MB, since Scala has high requirements regarding this kind of memory, due to the numerous auxiliary classes it uses to support higher level constructs like closures and nested class definitions. Notwithstanding this compulsory modification, our experimentation with JVM parameters gave unsatisfying results so we decided to follow a different approach: we experimented with different values of parameter d, to artificially increase the computation cost of each generator function so that the execution times for the sequential implementations in the three languages would be similar and long enough for our purposes. This way we managed to increase the parallel proportion of the total computation and achieve higher speedups that reflect better the limits of the concurrent runtime and libraries rather than the scalability of our algorithm and implementation. The final parameter values were:

- $N = 200000$
- $d = 10000$ for F# and Scala and $d = 1000$ for Erlang.

4.2 Runtime Parameters

In order to examine the scalability of our implementations we needed to limit their execution on a specified number of CPU cores. To that end we introduced the following parameters:

- Parameter P: the CPU affinity of the OS process in which each implementation runs. It is specified using the `taskset` command on Linux and the `affinity` parameter on Windows.
- Parameter M: a parallelism or concurrency parameter in the implementation or the libraries used; it may be the level of parallelism of a parallel collection, the level of concurrency of a concurrent data structure or the number of persistent actors in an actor system.
- Parameter G: the chunk size argument of the partitioning function. We need to test several chunk sizes so that we can determine one that is small enough to enable parallelism and large enough to produce chucks that have considerable computational cost.

4.3 Execution environment

For our experimental evaluation we used a Bulldozer-based server and the software described in the following subsections.

4.3.1 Bulldozer Architecture

The machine we used for our benchmarking has 64-core Bulldozer-based server with the following characteristics:

- 4 sockets
- 2 NUMA nodes per socket (8 NUMA nodes in total)
- 8 cores per NUMA node (64 cores in total)
- cache per core
- 64KB L1-instruction cache per 2 cores
- 2MB L2 cache per 2 cores
- 6MB L3 cache per NUMA node
- 16GB RAM per NUMA node (32GB RAM per socket and 128GB in total)

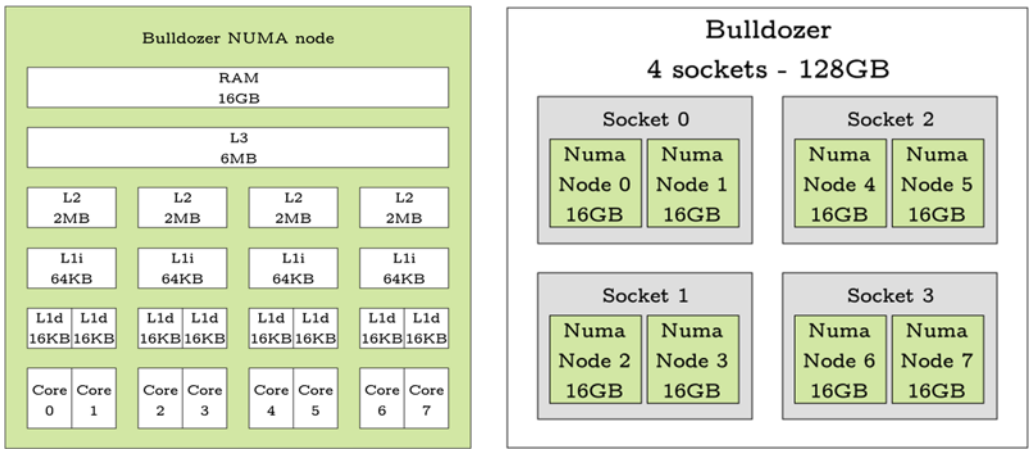


Figure 4.1: Bulldozer Architecture

4.3.2 Language Runtimes, Libraries and Scripting Environment

The language runtimes and libraries we used for benchmarking are the following:

- Scala 2.10.2 + Akka 2.1 on Java HotSpot 64-bit Server VM 1.7.0_21
- F# 3.0 on Mono 3.11 Beta (both Mono and F# were built from the master branch of their respective Git repositories)
- Erlang R16B01 64-bit with natively compiled libraries

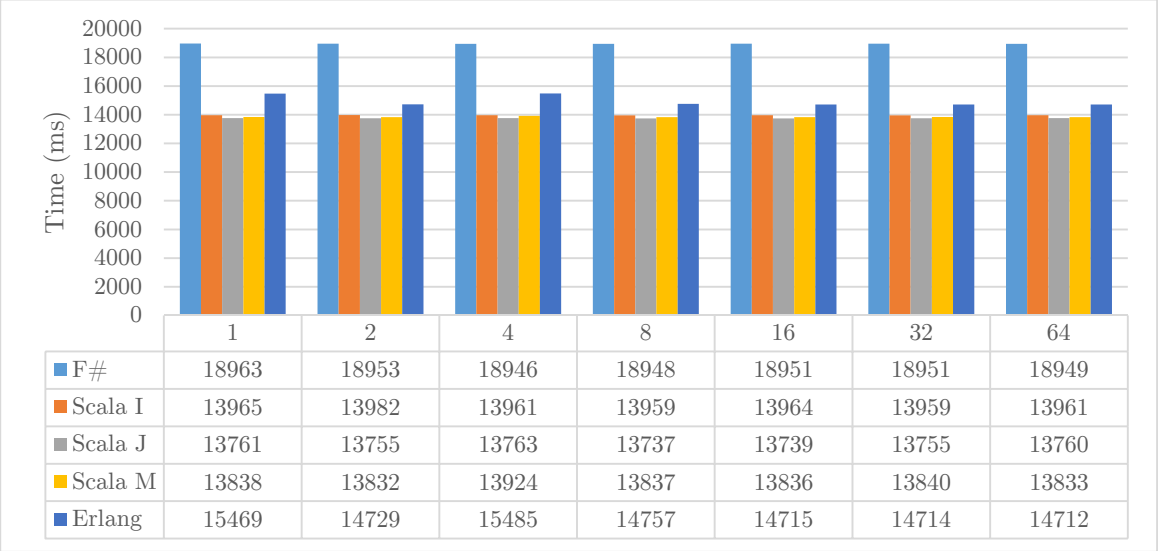
For the execution of the described configurations, we used the F# Interactive (from the above F# installation) as a scripting environment.

4.4 Approach A - Sequential

We ran all implementations of this category for $P = \{1, 2, 4, 8, 16, 32, 64\}$. In Scala we examined implementations with both Scala and Java sets: *Scala I* uses an immutable Scala set, *Scala M* a mutable Scala one and *Scala J* the standard Java `HashSet` (see Table 4.1).

Implementation	Set
Scala I	`scala.collection.immutable.Set`
Scala M	`scala.collection.mutable.Set`
Scala J	`java.util.HashSet`

Table 4.1: Set type for each Scala implementation



	1	2	4	8	16	32	64
F#	18963	18953	18946	18948	18951	18951	18949
Scala I	13965	13982	13961	13959	13964	13959	13961
Scala J	13761	13755	13763	13737	13739	13755	13760
Scala M	13838	13832	13924	13837	13836	13840	13833
Erlang	15469	14729	15485	14757	14715	14714	14712

Figure 4.2: Approach A – Time

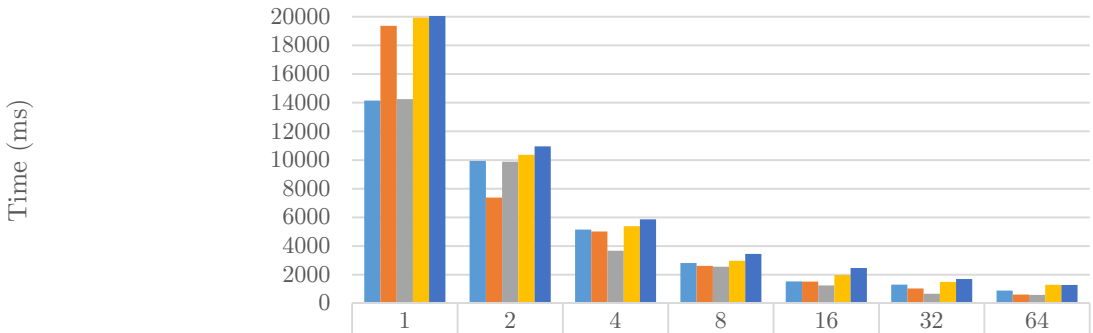
In Figure 4.2 we present the result of the sequential executions. F# exhibited worse performance that Scala, although we expected their corresponding execution times to be similar. This result can be attributed to the different implementation approach (see Section 3.4). Furthermore, Erlang gave times close to Scala but we should keep in mind the order-of-magnitude difference among the parameter d values of the respective configurations.

We should also make some remarks particularly about the three Scala implementations. In terms of performance the Java `HashSet` is fastest, second follows the mutable set and last comes the immutable one. Although the performance differences are small, Java has an apparently more optimized set implementation than Scala. In addition, we noticed that sometimes more than one processors were used in the execution of Scala configurations, a behavior suggesting that that the JVM uses several processors for internal operations like JIT compilation, garbage collection and other maintenance jobs.

4.5 Approach B - Parallel Collections

In this implementation approach we did not follow a universal benchmarking scheme. First of all, we could not run all the implementations; F# Variation A (PLINQ) when run on Mono throws a `NullPointerException` at the `ParallelEnumerable.Distinct()` call, so we were unable to take any measurements for it. We ran the other implementations for $P = \{1, 2, 4, 8, 16, 32, 64\}$.

Also we used the M parameter - differently for each implementation. In Scala it represents the level of parallelism of the `ForkJoinPool` that supports the parallel execution on `ParSeq` and we ran the implementations for $M = \{1, 2, 4, 8, 16, 32, 64\}$. In *F# (PLINQ)*, M is the level of parallelism that we want from the PLINQ library and it took values M = {1, 2, 4, 8, 16, 32, 63}; in *F# (Parallel.ForEach)* M is the level of concurrency of the ConcurrentSet with values $M = \{1, 2, 4, 8, 16, 32, 64\}$.



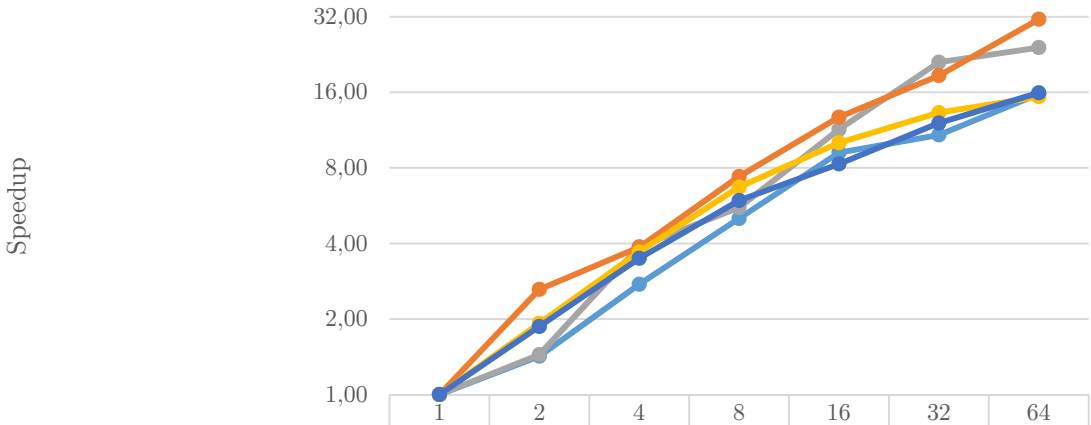
	1	2	4	8	16	32	64
Scala (ParSeq)	14145	9944	5145	2811	1536	1307	898
Scala (ParSeq & juc.ConcurrentMap)	19360	7380	5012	2619	1519	1038	619
Scala (ParSeq & cc.TrieMap)	14251	9879	3673	2564	1252	676	591
F# (PLINQ & Concurrent Dictionary)	19932	10367	5386	2971	1980	1501	1297
F# (Parallel.ForEach & Concurrent Dictionary)	20467	10955	5869	3450	2467	1696	1285

Figure 4.3: Approach B – Time

In Figure 4.3 the effect of parameter P is shown – for the best results regarding parameter M. Relatively to the sequential implementations (*Scala I* and *F#*), the overhead of this approach is tolerable in most cases; only *Scala (ParSeq & juc.ConcurrentMap)* has an overhead of 30% compared with not only the sequential implementation but also *Scala (ParSeq & cc.TrieMap)*. This significant overhead is possibly an effect of JIT-compilation. After repeated execution of both *Scala (ParSeq & cc.TrieMap)* and *Scala (ParSeq & juc.ConcurrentMap)* we observed execution times over 19 seconds in some executions, while in others it dropped to 14 seconds. We assume that there is a heuristic determining whether JIT-compilation would result in faster execution of the running program or not; for these configurations the heuristic result differs between executions.

Except for the *Scala (ParSeq & juc.ConcurrentMap)* and its initial overhead, no implementation speedup is remarkable (Figure 4.4). The cause for the witnessed behavior is the stepwise logic of this category that impedes parallel execution: the elements of each step do not suffice to create enough parallel work to fully utilize the available resources and even if there is ready work for the next step it cannot start because of the stepwise logic.

Nonetheless, the concurrent set in *Scala (ParSeq & cc.TrieMap)* leads to a speedup of over 24 for 64 cores comparing to below 16 of *Scala (ParSeq)*. Contrary to that, both F# implementations resulted in similar speedups, regardless of the set that was used. It seems that Scala features a scalable concurrent set implementation, while F# does not.



	1	2	4	8	16	32	64
Scala (ParSeq)	1,00	1,42	2,75	5,03	9,21	10,82	15,75
Scala (ParSeq & cc.TrieMap)	1,00	1,44	3,88	5,56	11,38	21,08	24,11
Scala (ParSeq & juc.ConcurrentMap)	1,00	2,62	3,86	7,39	12,75	18,65	31,28
F# (PLINQ & Concurrent Dictionary)	1,00	1,92	3,70	6,71	10,07	13,28	15,37
F# (Parallel.ForEach & Concurrent Dictionary)	1,00	1,87	3,49	5,93	8,30	12,07	15,93

Figure 4.4: Approach B – Speedup

4.6 Approach C - Futures

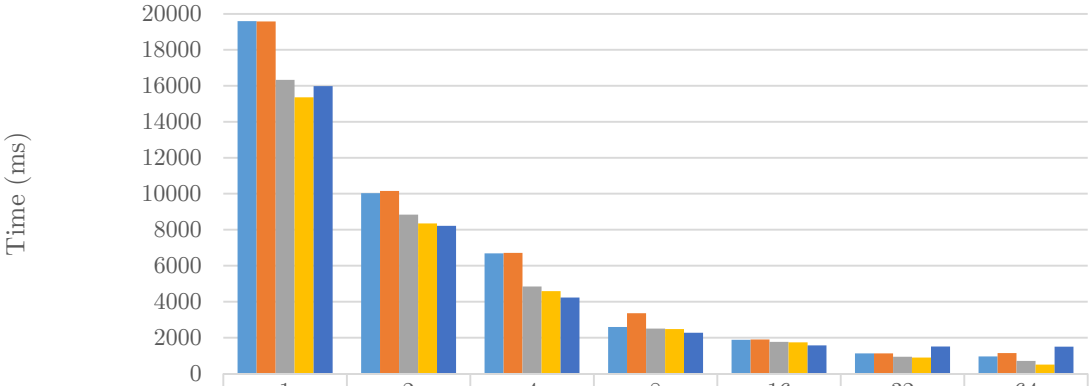
The benchmarking configurations for this category were combinations of the following parameters:

- $P = \{1, 2, 4, 8, 16, 32, 64\}$
- $G = \{1, 10, 100, 500, 1000, 5000, 10000\}$
- $M = \{1, 2, 4, 8, 16, 32, 64\}$ - only for *F# (Task & Concurrent Dictionary)*

Due to the overwhelming amount of possible configurations for each implementation we present mainly the diagrams that show the effect of parameter P on the execution times, for the best values regarding parameters M and G. We present each variation separately to avoid congested diagrams and thereafter we comment the effect of parameters M and G.

Variation A

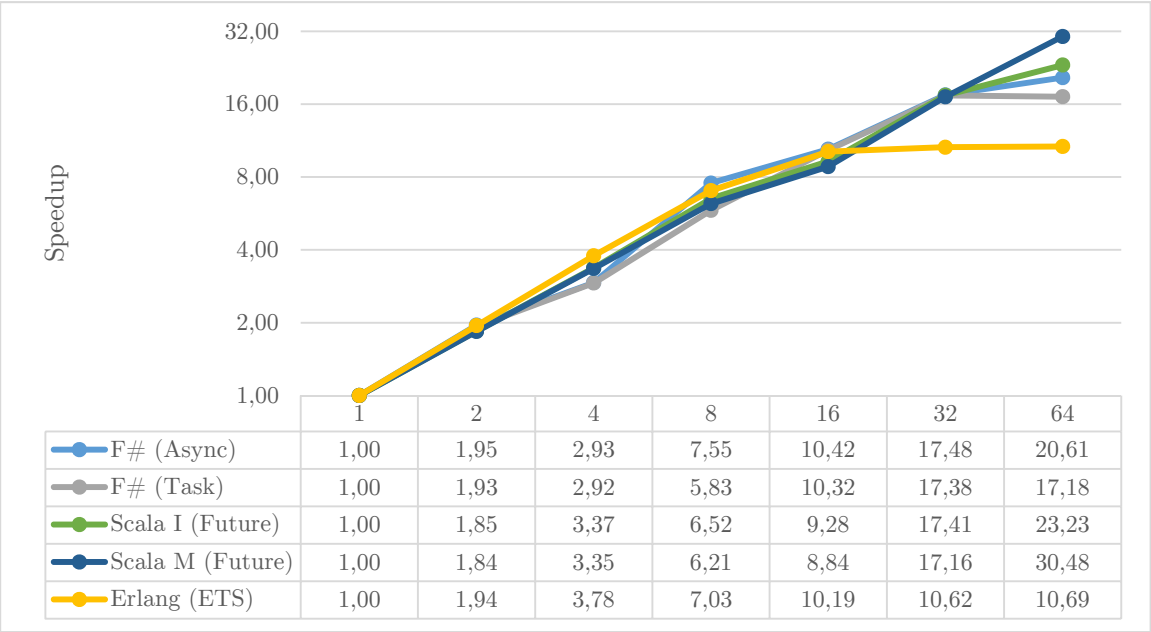
We present the times for Variation A in Figure 4.5. This approach introduces relatively small overheads as we can deduce by comparing the execution times for $P = 1$ and the corresponding sequential times (Figure 4.2).



	1	2	4	8	16	32	64
F# (Async)	19596	10027	6686	2594	1880	1121	951
F# (Task)	19583	10159	6712	3360	1898	1127	1140
Scala I (Future)	16332	8840	4844	2506	1760	938	703
Scala M (Future)	15360	8346	4585	2474	1738	895	504
Erlang (ETS)	15985	8220	4228	2273	1569	1505	1495

Figure 4.5: Approach C – Variation A - Time

By comparing *F# (Task)* with *F# (Async)* we observe that tasks do not offer any performance gains over asynchronous workflows, while they are less idiomatic to use. In Scala, better raw performance is achieved by using a mutable set (*Scala M*) instead of the default immutable one (*Scala I*), a behavior that agrees with the one observed in the sequential implementations.



	1	2	4	8	16	32	64
F# (Async)	1,00	1,95	2,93	7,55	10,42	17,48	20,61
F# (Task)	1,00	1,93	2,92	5,83	10,32	17,38	17,18
Scala I (Future)	1,00	1,85	3,37	6,52	9,28	17,41	23,23
Scala M (Future)	1,00	1,84	3,35	6,21	8,84	17,16	30,48
Erlang (ETS)	1,00	1,94	3,78	7,03	10,19	10,62	10,69

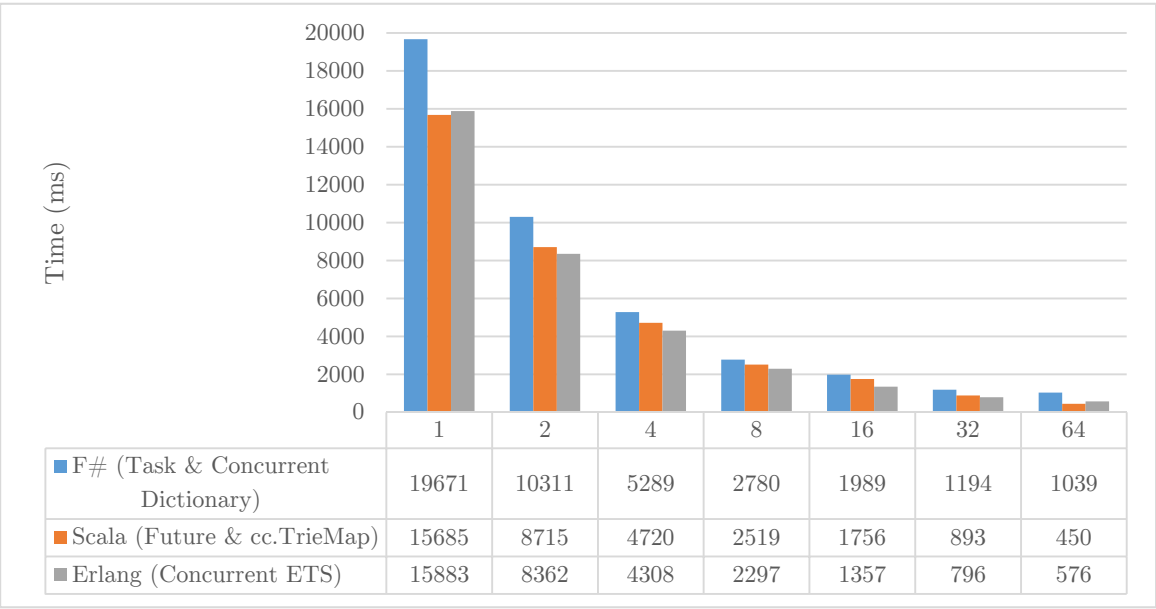
Figure 4.6: Approach C – Variation A - Speedup

In Figure 4.6 we present the corresponding speedups; interestingly all implementations scaled up to 16 cores and then each language behaved differently:

- *Erlang (ETS)* scaled only up to 16 cores after which speedup was fixed.
- F# implementations showed significant speedup increases up to 32 cores; for 64 cores *F# (Tasks)* speedup declined a bit and *F# (Async)* climbed up to 20.
- Scala implementations scaled up to 64 cores: *Scala M (Future)* had speedup over 30 and *Scala I (Future)* over 23.

Next we present the results for Variation B.

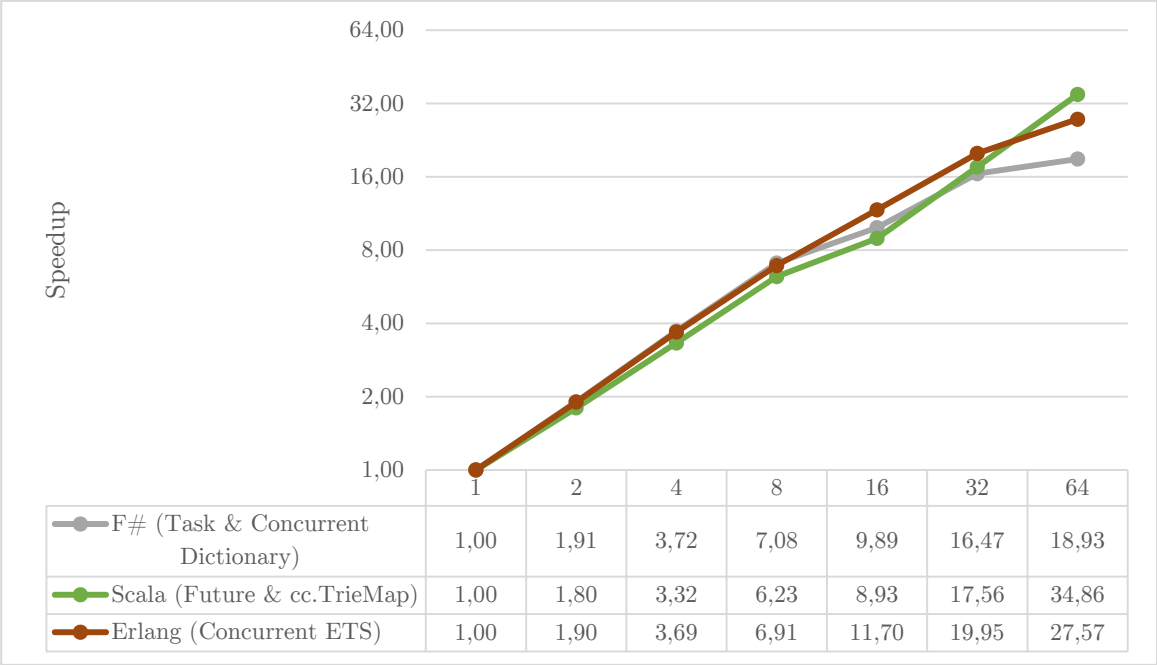
Variation B



	1	2	4	8	16	32	64
F# (Task & Concurrent Dictionary)	19671	10311	5289	2780	1989	1194	1039
Scala (Future & cc.TrieMap)	15685	8715	4720	2519	1756	893	450
Erlang (Concurrent ETS)	15883	8362	4308	2297	1357	796	576

Figure 4.7: Approach C – Variation B – Execution Time

In Figure 4.7 we present execution times. As expected, performance is better than in Variation A because of the reduced sequential part of the implementation logic: the concurrent set enables more work to happen in parallel thus relieving the coordinator from inserting elements to the result set and leading to greater speedup (Figure 4.8).



	1	2	4	8	16	32	64
F# (Task & Concurrent Dictionary)	1,00	1,91	3,72	7,08	9,89	16,47	18,93
Scala (Future & cc.TrieMap)	1,00	1,80	3,32	6,23	8,93	17,56	34,86
Erlang (Concurrent ETS)	1,00	1,90	3,69	6,91	11,70	19,95	27,57

Figure 4.8: Approach C – Variation B - Speedup

As observed, *Scala (Future & cc.TrieMap)* and *Erlang (Concurrent ETS)* scale up to 64 cores with Scala showing the greatest speedup, while *F# (Task & ConcurrentDictionary)* shows only a slight speedup increase after 32 cores. The concurrent set constitutes a key part of this implementation group as it is concurrently accessed by several concurrent threads and undergoes a great deal of contention. Consequently the implementation of each concurrent set is resembled in the observed scaling behavior.

Execution Parameters *M* and *G*

Parameter *M* is used only in *F# (Task & Concurrent Set)* where it represents the level of concurrency of the concurrent set – the Scala and Erlang implementations do not provide any similar configuration options. As presented in Table 3.1 the concurrent set can scale up to 16 concurrent accesses, a number after which its performance decreases.

Parameter *G* on the other hand is used in all implementations of this category and while we have measurements showing its effect for all of them, there is little significance in listing them all. Instead we present a case were its influence is manifested most evidently.

M	Time (ms)
1	1142
2	1049
4	1160
8	1096
16	1039
32	1102
64	1199

Table 4.2: F# (Task & Concurrent Dictionary) – Effect of M (A = 64 and G = 100)

Figure 4.9 shows the effect of parameter G on the execution time. We can estimate the best chunk size for our benchmark to be approximately 100 elements. This size can be associated with chunks that have enough computational work to benefit from parallel execution but are small enough to facilitate use of all available cores.

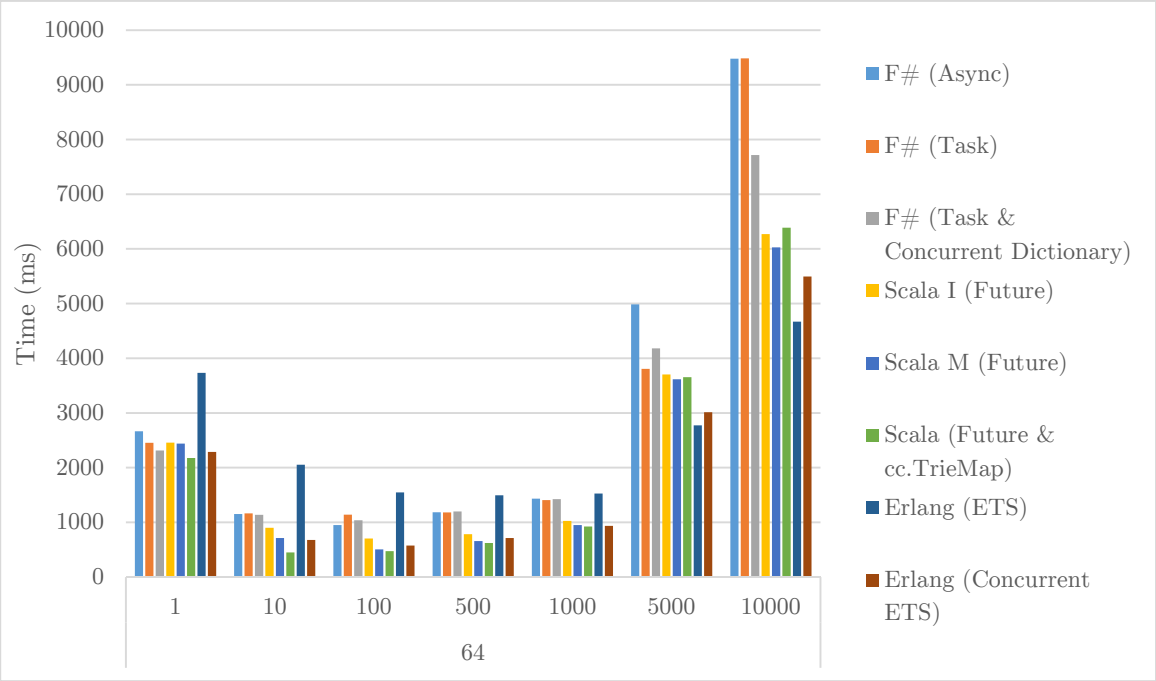


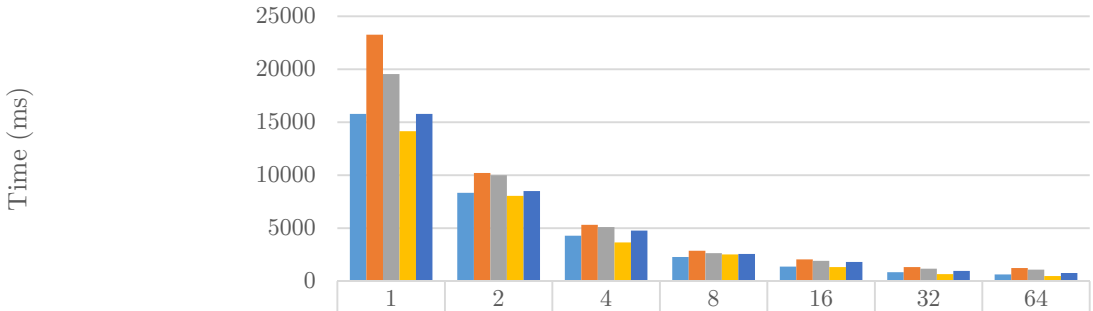
Figure 4.9: Approach C – Variation B – Effect of G (A = 64)

4.7 Approach D - Persistent Actors

We examined the simple actor systems we created for this approach using the same configurations described in the previous section. Parameters A and G are unchanged, while parameter M represents the number of persistent worker actors that were created in each configuration, and the level of concurrency for the concurrent set (where appropriate).

In Figures Figure 4.10 and Figure 4.11 we present the best results regarding parameters M and G for all values of parameter P. Unexpectedly, *F# (Persistent Actors)* exhibited better performance than *F# (Persistent Actors and Concurrent Dictionary)*, which had a significant initial overhead. That result hints towards a bad cooperation between the `ConcurrentDictionary` and the `MailboxProcessor` implementations.

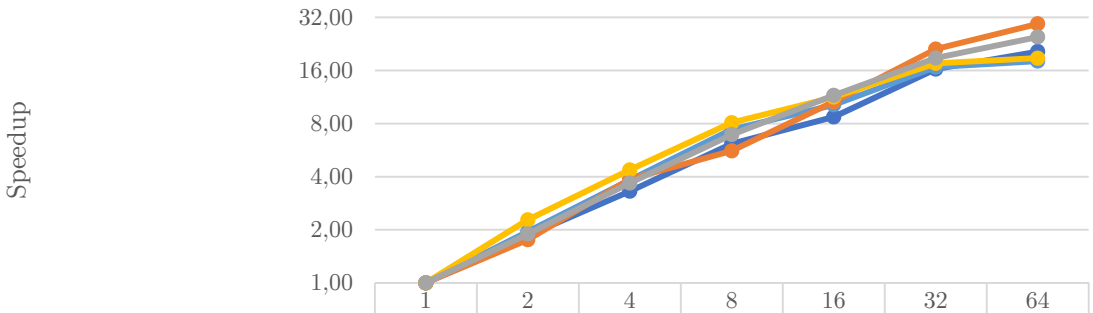
We can observe that all other implementations behaved similarly to their counterparts of the previous category. Considering that the only change was the use of persistent workers in place of the transient ones, it should not be surprising; even the slightly worse performance of some cases can be explained. In Erlang the router implementation is not optimized; it requires message handling which adds an overhead in comparison to the F# and Scala implementations. Moreover it should be mentioned that all implementations use the default configuration for the actor systems; while in F# there are no configuration options of the actor runtime, in Erlang and Scala several related options have to be considered.



	1	2	4	8	16	32	64
Erlang (Persistent Actors & Concurrent ETS)	15781	8333	4283	2277	1364	838	637
F# (Persistent Actors & Concurrent Dictionary)	23250	10203	5318	2873	2058	1325	1238
F# (Persistent Actors)	19542	10000	5103	2642	1912	1169	1081
Scala (Persistent Actors & cc.TrieMap)	14142	8053	3654	2522	1329	668	481
Scala (Persistent Actors)	15777	8496	4769	2566	1807	968	770

Figure 4.10: Approach D – Time

Figure 4.11 presents the corresponding speedups. With the exception of F#, implementations with concurrent sets are the most scalable, as expected. In addition, *F# (Persistent Actors & Concurrent set)* initially shows abnormal speedup (up to 8 cores) – this is not confusing considering the initial overhead.



	1	2	4	8	16	32	64
Scala (Persistent Actors)	1,00	1,86	3,31	6,15	8,73	16,30	20,49
F# (Persistent Actors)	1,00	1,95	3,83	7,40	10,22	16,72	18,08
Scala (Persistent Actors & cc.TrieMap)	1,00	1,76	3,87	5,61	10,64	21,17	29,40
F# (Persistent Actors & Concurrent Dictionary)	1,00	2,28	4,37	8,09	11,30	17,55	18,78
Erlang (Persistent Actors & Concurrent ETS)	1,00	1,89	3,68	6,93	11,57	18,83	24,77

Figure 4.11: Approach D – Speedup

In principle, actors aim for efficient message handling, not heavy computations. The aforementioned results indicate that the examined actor implementations can be used to design systems that perform serious computations with only minor overhead, even though futures are more suitable for this kind of work.

Parameters G and M

As shown in Figure 4.12, the best value for parameter G is around 10 and 100, similarly to the results of Approach D.

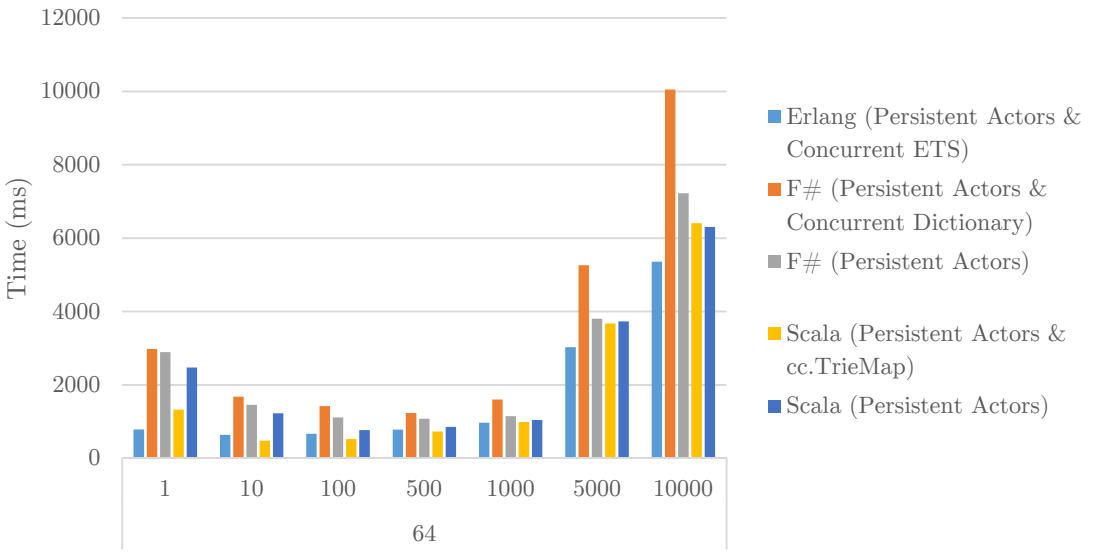


Figure 4.12: Approach D – Effect of G (A = 64)

Regarding parameter M the results are as expected: best is to choose a number of workers that is equal or higher than the number of available cores. As an example, in Figure 4.13 we present the influence of parameter M for $A = 32$. As we can see, peak performance is

achieved for a number of workers that is equal or higher than the number available cores, a result that is consistent with intuition.

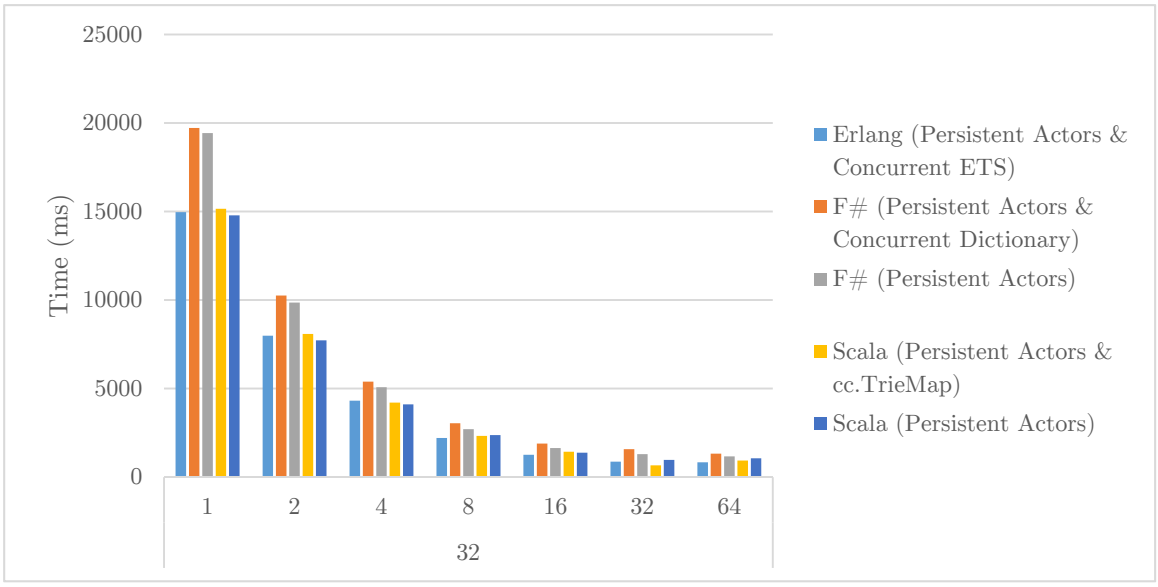


Figure 4.13: Approach D – Effect of M (A = 32)

4.8 Additional Results

4.8.1 Iterations needed for stable execution times

F# implementations had considerably worse performance than the corresponding Scala ones. This remark was quite unexpected considering that high proportion of the measured times involved arithmetic operations; one would expect Mono and JVM to perform likewise under such load.

Trying to explain this behavior we suspected two components of Mono: the Sgen garbage collector and the JIT compiler. Sgen is a recently-introduced generational garbage collector destined to replace the default non-generational Boehm collector. In our benchmarking we used the first which is deemed to offer better performance but is also not as mature as the latter. On the other hand, the JIT compiler of Mono is not as optimized as the JVM one.

Our suspicions were enhanced by the actual results: for F# implementations, performance was still improving at the 10^{th} execution iteration, while for Scala implementations it was stabilized several iterations before. We decided to repeat execution of Approach C implementations for 50 iterations and only for $A = 64$; for F# we used both garbage collectors of Mono and for Scala both mutable and immutable sets.

In Figures Figure 4.14 and Figure 4.15 we present the results for Variation A and B respectively.

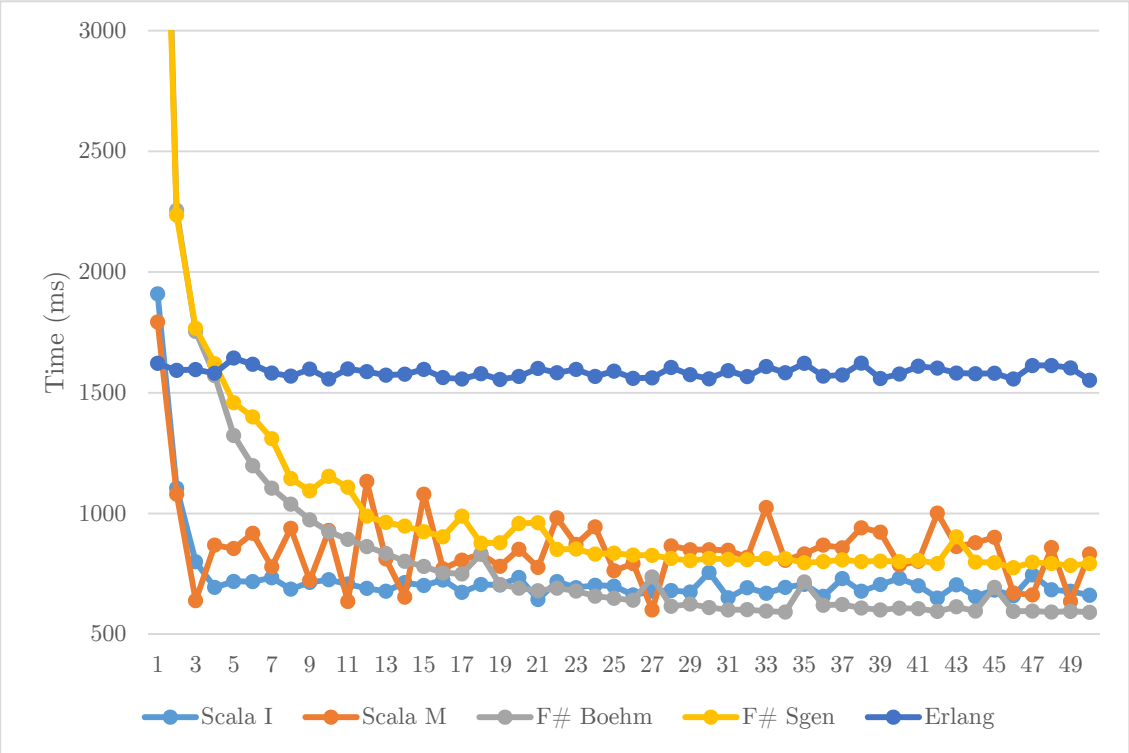


Figure 4.14: Approach C - Variation A - Times for 50 iterations

In Variation A, Scala M times were rather unstable, probably due to reallocations and copying of inner data structures of the mutable set. Furthermore, Scala times reached a

minimum at the 3rd iteration while F# times were still declining after the 40th iteration, a number that is unrealistic for our computation scenario; around the 22nd iteration, however, F# Boehm overcame Scala I in becoming the implementation with the best performance.

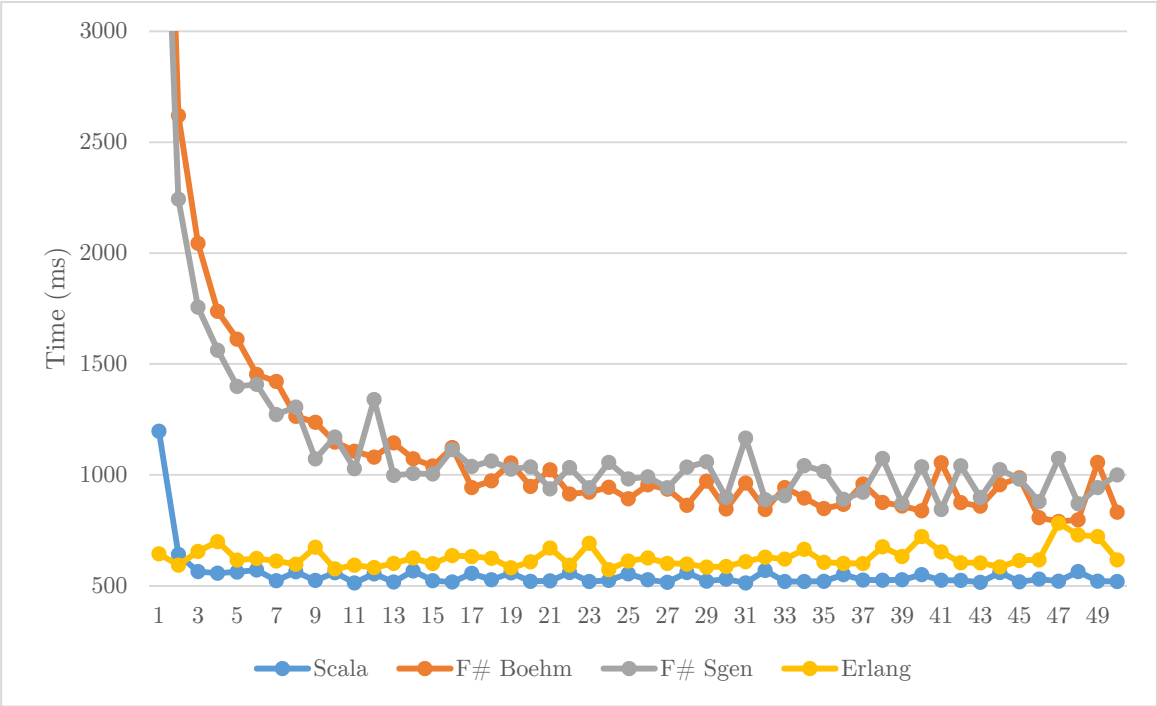


Figure 4.15: Approach C - Variation B - Times for 50 iterations

In Variation B the concurrent set of the F# implementation resulted in highly unstable performance as the nondeterministic access patterns were combined with the effect of garbage collection. Both F# garbage collectors showed similar behavior with Boehm giving slightly better times that Sgen. In any case, both F# configurations were left far behind as Scala was the clear winner in terms of performance.

All in all, each variation behaved differently, yet JIT compilation had evident positive effects on the performance of both Scala and F# configurations (though much more on the latter), while the trend lines of Erlang execution times signified the absence of any JIT optimization. Finally, minor and scarce divergences in the time measurement were witnessed even in stable executions denoting indeterminism in concurrent scheduling, prolonged garbage collection sessions and concurrent execution of other processes running on the same machine.

4.8.2 Integers of arbitrary size

Erlang integers have arbitrary-precision and are subject to garbage collection, contrary to the 64-bit integers we used in Scala and F# implementations. In this section we investigate the performance of integers with arbitrary-precision in Scala and F#.

For A = 64 and d = 10000 we executed the following for 10 iterations:

- *Scala M/I (Sequential)* and *F# (Sequential)* - Figure 4.16

- *Scala M/I (Future)*, *F# (Task)* and *F# (Async)* with M = 64 & G = 100 - Figure 4.17

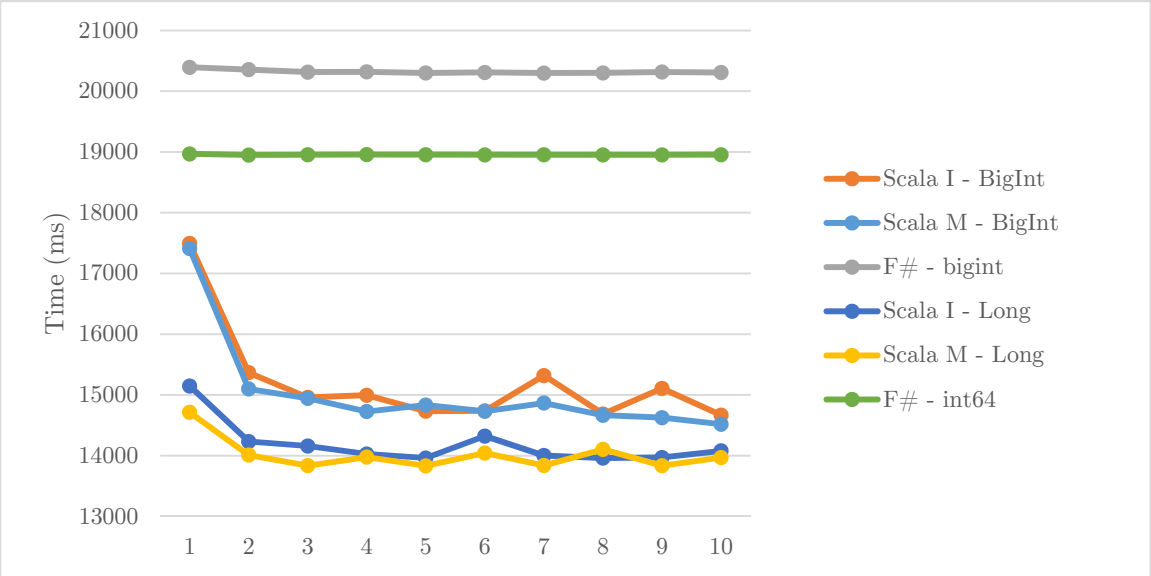


Figure 4.16: Integers of Arbitrary Size – Sequential

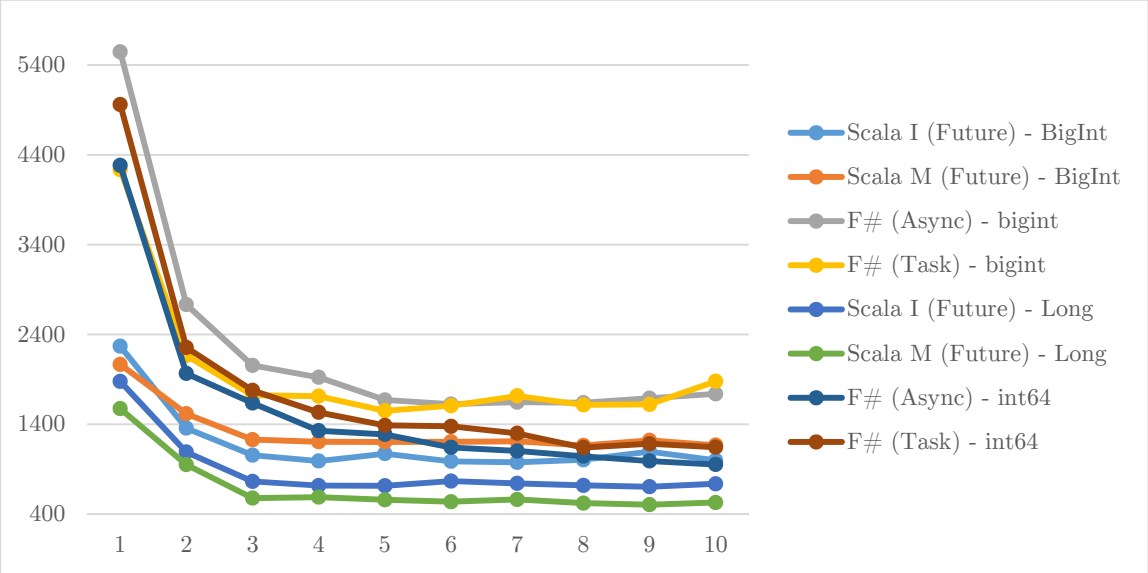


Figure 4.17: Integers of Arbitrary Size - Futures

We see that integers with arbitrary-precision do not imply serious drops of performance compared to the 64-bit integers we used for the main body of our benchmarks – performance is still much better that in Erlang.

As a side-note we should mention that, in addition to the above configurations, we tried implementations with concurrent sets but unfortunately F# implementations run into deadlocks, due to buggy Mono libraries.

4.9 Experimental Evaluation Remarks

For the evaluation part of this study we have the following remarks:

- Erlang and Scala scale up the most, especially in the variations that use concurrent sets. Scala concurrent `TrieMap` facilitates implementations that scaled up the most, while the Erlang concurrent set using ETS tables also exhibits an excellent scaling. On the contrary, `ConcurrentDictionary` shows unremarkable scaling that is minimal after 16-32 cores.
- Scala implementations have the best performance. It is followed by F# implementations which show lower performance contrary to our expectations. Nevertheless, Erlang implementations are the slowest considering the parameter d values for the configurations of each language - Erlang is clearly not suited for arithmetic computations.
- Using arbitrary-precision arithmetic in F# and Scala does not cause significantly slower performance, though it results in deadlocks for F# implementations that use a concurrent set.
- Regarding JIT compilation: JVM provides its most optimized machine code before the 4^{th} execution iteration while Mono needs over 20 iterations to show similar performance, a number which is clearly unacceptable for realistic scenarios.

Chapter 5

Related & Future Work

5.1 Related Work

Similar to our work, P. Totoo, P. Deligiannis and H.-W. Loidl presented a thorough comparison of Haskell, F# and Scala regarding parallelism [23]. For their evaluation they provided several Barnes-Hut implementations of the n-body problem in each of these languages. Their implementations were executed on both Linux and Windows environments and detailed measurements were taken regarding time and memory consumption. Moreover, one of their objectives was to present reusable data parallel patterns so their methodology was mainly limited in experimenting with parallel-map implementations that used appropriate tools of each language and did not contain any actor-based approaches. Their main result: "near best speedups are achieved using the highest level abstraction".

5.2 Future Work

Our investigation is far from complete and can be extended to several directions that include:

- Execution evaluation on Windows. It is a major platform for programming multi-core concurrency and its evaluation is sine qua non for a complete study of concurrency and parallelism on modern systems. Moreover, F# was designed on the .NET framework that has a far more mature implementation on the CLI, so it would be fair to evaluate F# on Windows.
- Detailed memory and execution time measurements. A more comprehensive resource-consumption evaluation of our implementations is important for a conclusive assessment of language and runtime efficiency.
- Experimentation with more configuration parameters. In this thesis we used mostly the default library and runtime settings; it would be interesting to consider different option values, especially for the Akka library that provides numerous configuration options.
- Investigation of additional functional languages like Haskell and Clojure to broaden the scope of the study.

- Comparison with popular imperative tools for concurrency and parallelism, like Cilk++ and Thread Building Blocks. This comparison is essential to assess the performance of functional languages in comparison to imperative solutions.
- Evaluation on concurrent problems that are better suited for the actor model. The computational nature of the Orbit problem does not highlight any actor benefits that are related to error-handling or systems with dynamic topology.

Chapter 6

Conclusion

In this thesis we investigated concurrency and parallelism in languages that support the functional programming paradigm. Aiming to provide a thorough comparison of Scala, F# and Erlang on concurrency and parallelism, we evaluated four different implementation approaches for solving a simple, yet non-trivial computational problem.

Below we provide an overview of our experience along with our final judgment:

Scala exhibits the largest feature set, the broadest library offerings, the best performance and the most scalability. Its rich feature set and comprehensive libraries facilitate high productivity but also may lead to programs whose comprehension requires serious mental effort - with great power comes great responsibility and pain, and one has to be meticulous when deciding which features to use and how. There is also a significant learning curve before being able to benefit from Scala and its currently agile evolution pace implies that one must be ready to modify or even replace code and libraries; this is only balanced by the promises for power, productivity, and performance gains.

F# facilitates concurrency and parallelism by providing a competitive feature set and adequate standard library while retaining access to all CLI libraries. Although it is not as rich in features as Scala, F# facilitates interesting, expressive and concise code with less choices for the programmer to consider. Unfortunately we cannot recommend its usage in Linux for developing concurrent and scalable applications due to several bugs of the Mono implementation.

Erlang is a mature language designed for concurrent programming. It comes with a library that contains all the necessary functionality for highly scalable concurrent solutions. Nonetheless we cannot recommended Erlang for heavy arithmetic computations or when interoperability with other languages is a main requirement. The absence of variable mutation and global variables are irritating traits of Erlang but its functional nature helps in developing code which is correct and easy to reason about.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine,* 1965.

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal,* vol. 30, no. 3, pp. 202-210, 2005.

[3] C. Hewitt, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *IJCAI*, Stanford, California, USA, 1973.

[4] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[5] J. Armstrong, B. O. Dacker, S. Virding and M. Williams, "Implementing a functional language for highly parallel real time applications," in *Proceedings of the 8th International Conference on Software Engineering for Telecommunication Systems and Services*, pp. 157–163, Florence, Italy, Apr 1992.

[6] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.,* vol. 410, no. 2-3, pp. 202-220, Feb 2009.

[7] D. Syme, T. Petricek and D. Lomov, "The F# asynchronous programming model," in *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, pp. 175-189, Berlin, Heidelberg, 2011.

[8] M. Sulzmann, E. S. L. Lam and P. V. Weert, "Actors with Multi-headed Message Receive Patterns," in *Proceedings of the 10th International Conference on Coordination Models and Languages*, pp. 315-330, Oslo, Norway, 2008.

[9] "The Scala Programming Language," École Polytechnique Fédérale de Lausanne (EPFL), 2013. [Online]. Available at: http://www.scala-lang.org/.

[10] M. Odersky, L. Spoon and B. Venners, Programming in Scala, 2nd ed., Artima, 2010.

[11] "Akka," Typesafe Inc., 2011-2013. [Online]. Available at: http://akka.io/.

[12] D. Wyatt, Akka Concurrency, Artima, 2013.

[13] "F# at Microsoft Research," Microsoft, 2013. [Online]. Available at: http://research. microsoft.com/en-us/projects/fsharp/.

[14] D. Syme, A. Granicz and A. Cisternino, Expert F# 3.0, Apress, 2007.

[15] "Task Parallel Library (TPL)," Microsoft, 2013. [Online]. Available at: http://msdn. microsoft.com/en-us/library/dd460717.aspx.

[16] D. Leijen, W. Schulte and S. Burckhardt, "The design of a task parallel library," in *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming, Systems Languages and Applications*, pp. 227-242, New York, NY, USA, 2009.

[17] "Parallel LINQ (PLINQ)," Microsoft, 2013. [Online]. Available at: http://msdn. microsoft.com/en-us/library/dd460688.aspx.

[18] T. Petricek and D. Syme, "Syntax Matters: Writing abstract computations in F#," in *Pre-Proceedings of the 2012 Symposium on Trends in Functional Programming*, 2012.

[19] "Computation Expressions (F#)," Microsoft, [Online]. Available at: http://msdn. microsoft.com/en-us/library/dd233182.aspx.

[20] "Erlang Programming Language," erlang.org , 2011. [Online]. Available at: http:// www.erlang.org/.

[21] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris and I. E. Venetis, "A scalability benchmark suite for Erlang/OTP," in *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, pp. 33-42, New York, NY, USA, 2012.

[22] A. Prokopec, N. G. Bronson, P. Bagwell and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 151-160, New York, NY, USA, 2012.

[23] P. Totoo, P. Deligiannis and H.-W. 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*, pp. 49-60, 2012.