UPPSALA
UNIVERSITET

IT 16 081
Examensarbete 15 hp
Oktober 2016

Benchmarking Parallelism and Concurrency in the Encore Programming Language

Mikael Östlund

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Benchmarking Parallelism and Concurrency in the Encore Programming Language

Mikael Östlund

Due to the limit in speedup of clock speed, most modern computers now sport multicore chips to provide computing power. Currently, programming language support for such machines often consists of extensions to languages that were originally designed to be sequential. New programming models and languages are needed that are parallel by design. One class of such languages are actor- (and active object) based languages and one such language is Encore, a new actor based language currently under development. When developing a programming language it is often important to compare the performance of the language with other well-known and proven languages in the same paradigm. This thesis compares the concurrency and parallelism performance of Encore with Scala and Erlang over a set of benchmarks. The results show that Encore features a very fair scheduling policy and also performs up to par with the languages Scala and Erlang when measuring wall-clock execution time on a desktop computer. Encore currently lag behind a bit in terms of strong scalability when a lot of communication occur, when communication is kept at a minimum however, Encore showcases a strong scalability just as good as the one of Erlang and Scala in the parallelism benchmark Fannkuch. The memory usage of Encore is higher than the memory consumption of Scala and Erlang in one of the three analysed benchmarks and lower on the other two. Overall the results for Encore look promising for the future.

Handledare: Dave Clarke
Ämnesgranskare: Tobias Wrigstad
Examinator: Olle Gällmo
IT 16 081
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to acknowledge my supervisor Dave Clarke and my reviewer Tobias Wrigstad for all help and feedback they have given me in this thesis project. I would also like to thank the Uplang team for providing support when needed. Special thanks goes out to Albert Mingkun Yang, member of the Uplang team, for helping me make sense of the measurements as well as teaching me the inner workings of Encore.

Contents

1 Introduction

As increasing the clock speed of modern processors has become difficult in recent years, most modern computers now utilise multicore chips to gain computing power. The current trend is that more and more parallel processing units are added to computers [4]. Currently, programming language support for such machines often consists of extensions to languages that were originally designed to be sequential. New programming models and languages are needed that are parallel by design in order to make it easier for programmers to better utilise the multicore chips in a computer. One class of such languages are actor- (and active object)-based languages. These offer a notion of actor (or active object) as a unit of parallelism. Actors are conceptually similar to objects, and are thus arguable easy to adopt by programmers used to object-oriented languages.

Encore is a new actor based language currently under development in the context of the European project Upscale [25]. Measuring performance improvements as the language evolves and comparing with others languages, requires support for measuring aspects of run-time behaviour, a benchmarking framework, and appropriate infrastructure to run benchmarks on a range of different platforms.

The purpose of this thesis is to develop a benchmarking framework with the goal of simplifying the process of running and configuring benchmarks for performance measurements of Encore. Said framework is then used to benchmark the concurrency and parallelism performance of Encore. Results of three benchmark programs are produced with focus on measuring strong scalability, execution time, memory usage and CPU utilisation. Measurements of Scala [23] and Erlang [2] versions of the benchmarks are included as a way of comparing Encore with renowned languages in the same paradigm.

Outline The thesis is structured as follows. In Section 2 the measurement techniques and tools used in this thesis are presented followed by a background overview of the programming languages included in the benchmark measurements. The benchmark specifications are then given in Section 3, the measurements of which can be found in Section 4. Following this, in Section 5, the results are analysed and discussed with focus on Encore. Related work is found in Section 6 followed by a reflection over the results in Section 7. Future work for this thesis is covered in Section 8 followed by a conclusion of the thesis in Section 9.

2 Background

This section presents the measurement tools and techniques used in this thesis as well as a brief introduction to the programming languages used in the comparisons in Section 4.

2.1 Measuring Wall-Clock Time of a Program

The most basic measurement of a running program is the wall-clock time, the time span between the start of the program and its termination. The approach taken to measure wall-clock time of a program is using the GNU `time` utility, available on Unix systems [17]. This utility was chosen because it is commonly used and is included by default on Unix systems. GNU `time` also does not add notable overhead to the program execution time, which is beneficial when the goal is only to measure the wall-clock time. However, due to its lightweight nature, no profiling is done which makes us lose information about what procedure takes up most time.

2.2 Measuring CPU Utilisation

When measuring the performance of different languages it is important to observe how different parameter configurations affect the CPU utilisation. The `time` utility can measure more things than just wall-clock time (called real time in the `time` tool), `time` can also measure system time and user time, which are used in conjunction with the wall-clock time to infer the CPU utilisation. This is done by summing up the system- and user-time and dividing the summarisation result by the measured the wall-clock time.

System time is the amount of CPU time the process spends in the system kernel. That is, the actual time spent executing system calls within the kernel. This differs from user time in the way that user time measures the time the process spends outside the kernel, including calls to library code, running in user-space. Both the measurements of user and system time only include the time the specified process uses with no regard of other parallel processes that may be running on the system.

Note that the measured system time or user time may be higher than the actual wall-clock time. Thus, the CPU utilisation can measure above 100% on multicore systems when a program utilises several cores during execution. E.g. when running on a two core machine, if both cores are utilised fully, the measured CPU utilisation will be 200%. High CPU utilisation does not necessarily correlate with high usefulness however. Different languages and applications often utilise the CPU in different ways where some languages may allocate more CPU time into garbage collection tasks, busy-waiting or other tasks

that are not directly involved with the program execution. Keep this in mind when reviewing the CPU utilisation measurement data.

2.3 Measuring Memory Usage

The same GNU utility `time` is used for measuring memory usage of executed programs. `time` measures the maximum amount of allocated memory for a process and reports this value as *maximum resident set size*. It happens that programming languages allocate more memory from the operating system than they actually use to avoid allocating memory in smaller chunks throughout the execution of a program. Thus, memory measurements using `time` need to be analysed with a grain of salt. However, they suffice to give an overview of how memory usage scales over different ranges of parameters as well as giving a hint of the size of the program's actual memory space.

2.4 Measuring Scalability

When discussing scalability, two notions of scalability are often used; *strong* and *weak*.

Strong scalability measure the scenario where the problem size is constant but the number of processing elements varies. Weak scalability measures how performance change when the problem size is fixed for each processing element but where additional processing elements are added in order to efficiently solve bigger problem instances, e.g. by adding more cores to execute the program on.

This thesis explores strong scalability. In Encore, this is done by varying the amount of cores available to the runtime system of Encore while keeping the problem size fixed. At the time of writing this thesis Encore has support for selecting the amount of threads that are available to the runtime using a command line argument, namely the `ponythreads` flag, which was the method of choice. Scalability comparisons between languages are done by running the same benchmarks on one machine while varying the amount of available processor cores.

The Linux utility `taskset` was used for constraining the amount of threads available for the Scala and Erlang versions of the benchmark programs [18]. `Taskset` enables configuration of the CPU affinity of a process, that is, which processor cores that are made available to the process during execution.

2.5 Languages

In this subsection, a brief overview is given of the programming languages that were chosen for the comparison part of the thesis. It is assumed that the reader has some

familiarity with actor paradigms and object-oriented programming.

This thesis explores two different ways to implement concurrency in the three included languages. Namely the message passing and shared memory models. Message passing communication is commonly realised in the actor model, where each actor acts as an enclosed state with private fields which are only accessible atomically. This makes it easier to avoid concurrency hazards such as data races and deadlocks, in contrast to the shared memory concurrency model which requires synchronisation mechanisms such as semaphores, monitors and locks to avoid such hazards [12]. Under the shared memory concurrency model, communication is performed through different threads communicating with each other by sharing memory resources, memory resources which the communicating threads have direct access to. Thus the need for synchronisation. However, implementing concurrency in an application using the shared memory model can sometimes be made faster compared to using message passing in said application, since message passing overhead can be avoided [12] by simply accessing the shared data directly, without the need of message passing. One caveat being that shared memory models are often more difficult to design and make scale well. Threads in the shared memory model often use a one-to-one mapping to hardware threads, making context-switching between threads a heavy task for the runtime system, much more so than in the case of actors. Each thread also requires more memory compared with an actor, further limiting the scalability of the system [12].

Two languages apart from Encore are explored in this thesis, namely the renowned languages Erlang and Scala. All three languages feature message passing as a concurrency construct, making them suitable for concurrency and parallelism performance comparisons. Scala and Erlang were picked over other languages using message passing due to myself being more experienced in Erlang and Scala compared to other languages.

2.5.1 Encore

Encore, currently under development within the Upscale project, is an actor based programming language that is designed with concurrency in mind [5]. Encore uses the notion of *active objects* to express concurrency and parallelism. Active objects are closely related to actors and are similar to regular objects that can be found in object-oriented languages such as Java. They contain fields and methods just like a regular Java object, but under the shell a concurrency model is implemented. One thread of control is associated with each active object and this thread is the only thread that has direct access to its fields, as in the message passing concurrency model.

Encore also supports the notion of *passive objects*, very much alike regular objects

in other object-oriented languages where access to fields and methods of those objects returns the result immediately, i.e. synchronously.

The asynchronous communication between active objects is done by sending messages triggered by method invocations on other active object. Each active object maintains a message queue and processes one message from the queue at a time.

A method call returns immediately by returning a future, the actual result of the method call will later be stored when it has been calculated. Encore method calls also features a *fire and forget* semantics through the `!` (bang) operator, this is useful when the return value of a method invocation is not used by the caller. By invoking another active object's method using this operator, the process of creating a data structure storing the future is avoided, allowing for reduction in execution time as well as memory consumption [5].

Active objects are scheduled in a round-robin fashion in Encore, where each active object process up to 100 messages in its message queue when it is scheduled.[1] If its message queue is empty after having its go in the scheduler, it is marked as inactive and is not placed in any scheduler queue, otherwise it is placed at the end of the current scheduler queue. Encore features one scheduler per thread which is responsible for executing work residing in its queue. Here, a unit of work is defined as an active object with at least one message (method invocation) in its message queue. That is, all active objects are executed in the context of a scheduler. When a scheduler runs out of work in its queue, it will work steal from another scheduler in a preemptive fashion by default. Under the preemptive work stealing approach, work is outright stolen from the beginning of other scheduler queues if the *thief*'s work queue is currently empty.

When a method is invoked on an active object o_1 by another active object o_2, two things may occur. If o_1 is inactive, i.e. its message queue is empty and its not scheduled onto a scheduler, o_1 is appended at the end of the scheduler which is currently executing o_2 and the method invocation (message) is appended to o_1's message queue. If o_1 is already residing in a scheduler queue, the method invocation (message) is simply appended to o_1's message queue.

2.5.2 Scala

The multiparadigm language Scala combines functional, imperative and object-oriented programming and was developed with the goal of making a scalable language [23]. Scalable in the sense that the language can be used for everything between writing small scripts to writing enterprise software, thus scaling with the need of the programmer.

[1] 100 messages is the default setting in the current compiler version.

Scala applications run on a virtual machine, namely the Java Virtual Machine (JVM). By running on the same virtual machine as Java applications, this grants Scala the ability to use Java libraries and classes in Scala applications. By running on a virtual machine, Scala code is also platform independent.

Scala supports both the message passing concurrency model as well as the shared memory model. The message passing concurrency model is in Scala realised through Scala's actor library, where each actor is an enclosed state that communicates with other actors through message passing. Several other actor libraries exist in Scala, such as the Akka library [12]. In this thesis however, only the Scala actor library is explored, mainly because the second best Chameneos implementation for Scala in The Computer Language Benchmarks Game used the Scala actor library at the time of writing this thesis. The implementation included in Savina [14] was also based on Scala actors and translating the two implementations to Akka compatible code was not a priority. The shared memory model is implemented using Java threads in Scala. Java threads are supplied through the `java.util.concurrent` package and use different synchronisation techniques to obtain concurrent execution, such as semaphores, monitors and locks [12].

2.5.3 Erlang

Erlang is a functional programming language thatcreated by Ericsson (Erlang is an acronym for *Ericsson Language*) with focus on concurrent programming [2]. The process-oriented concurrency model used in Erlang is based on the message passing model. Process-oriented in the sense that one or more processes are spawned and managed by the Erlang Run-Time System (ERTS) and each process is isolated from all other processes, much like Scala's actor model and Encore's active objects. Each Erlang process is designed to be as lightweight as possible and is scheduled by Erlangs virtual machine Bodgan's Erlang Abstract Machine (BEAM), avoiding the overhead of using operating system processes and its scheduler.

In this thesis however, the HiPE compiler is used for compiling the Erlang benchmarks which enables sections of an Erlang program to be executed natively, without the use of the BEAM virtual machine. This opens up for potential performance improvements compared to exclusively depending on BEAM [24]. Communication between processes uses message passing. Each process maintains a mailbox that stores messages that are sent to it asynchronously, the messages are then processed one at a time after being selected from the mailbox by the process using pattern matching. In contrast to Encore, where up to *100* messages are processed in a batch each time an active object is scheduled.

3 Defining Benchmarks

This section introduces the benchmarks included in this thesis. It presents the core concepts of the benchmarks in text as well as pseudo-code where appropriate, followed by implementation details for each respective language implementation.

Pseudo-code Syntax. The syntax of the pseudo-code included is similar to the syntax of both Encore and Scala, but not to the one of Erlang. The pseudo-code syntax defines methods to provide message-passing between objects in a similar fashion to Encore. An object communicates with another object by invoking its methods, in the pseudo-code examples included, only the **!** (bang) operator is used for method invocations. Implying that we do not care about the result of the (asynchronous) method invocation, asynchronous in the sense that the invocation may be executed some time in the future. View this as the caller sends off a message to the receiving object and the receiving object will handle this message whenever possible.

The pseudo-code is inspired by the object-oriented paradigm and each class has some private fields, the initialisation of such fields is not shown in the code examples. However, to ease identification of such fields, the declaration of class specific fields are written in `teletypefont` and are found at the beginning of each class. All classes are to be viewed as actors unless specified otherwise.

It is assumed that one thread of control is associated with each class instance in the pseudo-code examples and only one thread can access a method of an object at a time.

Values are assigned through the $A \longleftarrow B$ operator, which assigns the value of B to A. Equality and comparison checks are provided through the $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$ operators. The last line of a method is always the return value of the method.

The type of parameters is specified like so:

$$a : type$$

where a is the parameter and *type* is its type.

If statements are indentation sensitive, illustrated in Algorithm 1. If *expr* evaluates to `true`, then block_A is executed, otherwise block_B is executed.

Three benchmark scenarios are explored in total. The first one, Big, features a lot of concurrent communication between actors. Big explores how the performance is affected by a change in the amount of concurrent communication. The second one, Chameneos, performs lots of communication on a single receiver. This benchmark measures how each language handles a congested communication system. The last benchmark scenario

if *expr*:
    block_A
else:
    block_B

Algorithm 1: Example of an if-else statement.

explored is Fannkuch, a parallelism benchmark where each actor has a high workload and where communication is kept at a minimum.

3.1 Big benchmark

This benchmark implements a many-to-many message passing scenario consisting of N workers alongside a supervisor. All workers are part of a neighbourhood, where all workers in a neighbourhood are aware of its neighbours. A worker may communicate with other workers in its neighbourhood, but since its not aware of workers outside its own neighbourhood, it can not communicate with workers outside its neighbourhood, with the exception of the supervisor.

Two different kinds of messages are used in this benchmark, a `ping` message and a `pong` message. Each worker starts out by sending a message containing its own id to one random worker in its neighbourhood, this is called a `ping` message. The worker on the receiving end of such a message responds back to the worker that sent the `ping` message with a `pong` message. All workers are divided into neighbourhoods with a maximal size of 2000 workers, if possible, as is the case of all parameter configurations explored in this thesis (excluding the configuration with 500 workers), all neighbourhoods will have a size of 1000. This neighbourhood structure implies that each worker only know of other workers in its own neighbourhood, it contains no references to workers outside this neighbourhood structure.

The ping-pong procedure is repeated 16 000 times for each worker. When a worker has repeated the procedure 16 000 times, it reports to the supervisor informing that it is done. The supervisor waits for all the workers to finish and the benchmark is concluded when all workers have reported to the supervisor.

This benchmark was chosen with the goal to measure the concurrency performance of Encore by focusing on the effects of performing a lot of concurrent communication between a lot of actors/active objects/processes. This benchmark that has been used for showcasing Erlang's strong scalability performance in a communication heavy scenario, it is therefore especially interesting to compare the strong scalability performance between Encore and Erlang in this benchmark [3]. This benchmark perform much more message-

passing than computation heavy tasks, the opposite of the Fannkuch benchmark.

The benchmark algorithm is illustrated in Figure 2. Arrows represents messages that are waiting to be processed by the receiver. A filled arrow indicates a `ping` message while a dashed line indicates a `pong` message.

Step 1) `ping` message is sent to a random worker.

Step 2) `pong` message is sent back to the sender.

Figure 1: Illustration of a `ping` and `pong` message being exchanged between two workers in the Big benchmark.

- Step 1 shows the process of worker W_x sending a `ping` message to another randomly selected worker within its neighbourhood.
- Step 2 shows the worker W_y responding to the sender W_x of the received `ping` message by sending a `pong` message back to W_x.

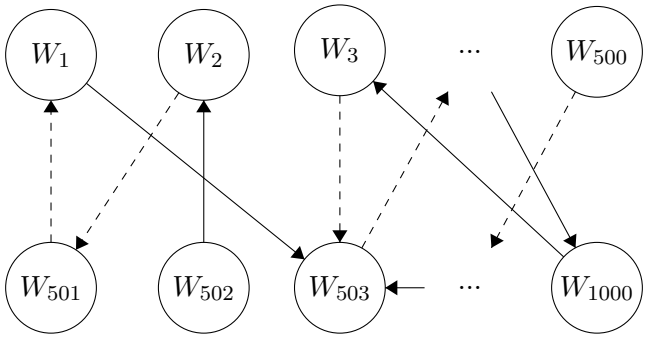


Figure 2: Illustration of the Big benchmark in action with a neighbourhood of 1000 workers.

Figure 2 illustrates the whole benchmark in action, with a neighbourhood of size 1000. Several neighbourhoods may be present at the same time.

The internal workings of the worker actors are found in Algorithm 3 and the internals of the supervisor actor are found in Algorithm 2. Each worker is initialised with a ping limit, specifying how many `ping` messages that actor will send. A worker will continue to generate new `ping` messages to random targets in its neighbourhood until the ping limit has been reached. The workers are all initialised to expect a `pong` from id *start* and the benchmark begins with all workers receiving a `pong` message marked with this id.

The Scala version considered is a slightly modified version of the *Big* benchmark featured in Savina [14]. Modifications include limiting the number of pings from each actor to 16 000 and removing some Savina related code that was not necessary for this scenario. Scala actors are used as the concurrency construct.

As for the Erlang version of this benchmark, several modifications to the *Big* benchmark featured in Bencherl [3] were made, where the most relevant change in terms of this benchmark was limiting the number of pings per process to 16 000. Each worker is implemented as an Erlang process.

The Encore version uses active objects to represent the workers and the supervisor. It uses fire and forget semantics through the usage of the bang operator.

```
/* Method for receiving pong messages                                    */
class SinkActor:
    numWorkers : int
    numMessages : int

    def exit():
        numMessages ⟵ numMessages + 1
        if numMessages = numWorkers:
            print "All workers are done, exiting."
```

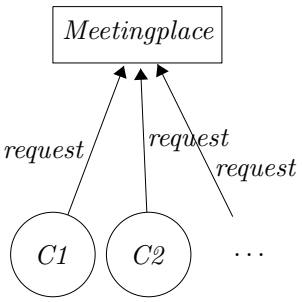
Algorithm 2: Internals of the supervisor (sink) actor.

```
/* Method for receiving pong messages                                    */
class Worker:
    myID : int
    expectedPonger : int
    neighbours : List<Worker>
    sinkActor : SinkActor
    numPings : int

    def pong(id : int):
        if id ≠ expectedPonger:
            throw WrongPinger(myID + " received a ping from the unexpected id
            " + id)
        if pingLimitReached():
            sinkActor ! exit()
        else:
            sendPing()
            numPings ⟵ numPings + 1 // Increment number of pings sent

    /* Method for receiving ping messages from id                        */
    def ping(id : int):
        neighbours[id] ! pong(myID)

    /* Method for sending a ping message to a random neighbour           */
    def sendPing():
        targetIdx ⟵ random.uniform(0, |neighbours|)
        target ⟵ neighbours[targetIdx]
        expectedPonger ⟵ targetIdx
        target ! ping(myID) // invoke ping method of target
```

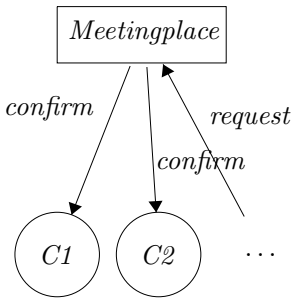
Algorithm 3: Internals of a worker actor.

3.2 Chameneos benchmark

The Chameneos benchmark scenario [16] consists of C Chamaneos creatures, each sporting a personal colour [22]. The goal of a such a creature is to meet as many other creatures as possible at a meeting place. After meeting another creature, a creature might mutate by changing its colour before leaving the meeting place depending on the colour of the other creature. The creatures in the benchmark continue to request meetings with other creatures until a total of M meetings have taken place at the meeting place. This process is illustrated by Figure 3 where the arrows represent messages that are waiting to be processed by the receiver.



Step 1) Creatures requesting a meeting.



Step 2) Meeting place coordinating a meeting between the creatures $C1$ and $C2$.

Figure 3: Illustration of the Chameneos benchmark. C_i represent a Chameneos creature.

- Step 1 illustrates the creatures requesting a meeting with another creature at the meeting place, every creature will request meetings until a total amount of M meetings has taken place at the meeting place. However, the creatures only send one meeting request message for each `meet` message that is received from the meeting place actor.
- Step 2 illustrates that the meeting place actor decided to let $C1$ and $C2$ meet at the meeting place by sending a `meet` message to the creatures participating in the meeting, confirming that they have met each other.

This benchmark serves the purpose of measuring the effects of contention on shared resources while processing messages that are sent back and forth. The shared resource is the actor/process/active object that is acting as the meeting place object. Fairness between actors is also measured through this benchmark, where fairness is a measure of how evenly work is divided between the actors representing the Chameneos creatures.

This benchmark is included in the benchmark suite Savina, where it is used to measure how performance scales when the amount of meetings is increased [14]. The benchmark has also been used for measuring fairness between actors in actor- and agent-based programming languages by Cardoso et al. [9]. On this basis it was selected as one of the three benchmarks included in this thesis.

The internal workings of the creature and meeting place actor(s) is found in Algorithm 4 and Algorithm 5 respectively. The benchmark begins with initialising the `firstColour` field in the meetingplace actor to *None*, followed by all Chameneos creatures requesting a meeting at the meeting place by sending a *meet* message to the *meetingplace* actor. The benchmark then continues until M meetings have taken place at the meeting place. No meetings are accepted before all creatures have had a shot at requesting a meeting.

```
/* Method for meeting a chameneos with id                              */
class Chameneos:
    meetingCount : int
    myID : int
    myColour : Colour
    meetingPlace : MeetingPlace

    def meet(id : int, colour : Colour):
        meetingCount ⟵ meetingCount + 1
        myColour ⟵ colour
        meetingPlace ! meet(this, myID, myColour) // request a meeting

    /* Method for sending summary of amount of meetings of current
       actor to supervisor                                             */
    def stop():
        meetingPlace ! sumMeetings(meetingCount)

    /* Method for starting the meeting loop behaviour                  */
    def run():
        meetingPlace ! meet(this, myID, myColour) // request a meeting
```

Algorithm 4: Internals of a chameneos actor.

```
/* Method for arranging meetings for each chameneos creatures     */
class MeetingPlace:
    firstChameneos : Chameneos
    firstColour : Colour
    firstID : int
    totalMeetings : int
    creaturesDone : int
    remainingMeetings : int
    amountOfCreatures : int

    def meet(chameneos : Creature, id : int, colour : Colour):
        if meetingLimitReached():
            chameneos ! stop()
        else:
            if firstColour = None:
                firstChameneos ⟵ chameneos
                firstColour ⟵ colour
                firstID ⟵ id
            else:
                // calculate complement colour of the two colours
                newColour ⟵ complement(colour, firstColour)
                firstColour ⟵ None
                remainingMeetings ⟵ remainingMeetings − 1
                chameneos ! meet(firstID, newColour)
                firstChameneos ! meet(id, newColour)

    /* Method for summarising meeting counts for each chameneos actor
       */
    def sumMeetings(meetingAmount : int):
        totalMeetings ⟵ totalMeetings + meetingAmount
        creaturesDone ⟵ creaturesDone + 1
        if amountOfCreatures = creaturesDone:
            prettyPrint(totalMeetings) // Outputs amount of total meetings
```

Algorithm 5: Internals of a meetingplace actor.

Both the Erlang and Scala code are modified versions of the ones found at The Computer Language Benchmarks Game website[2]. The Scala version of the benchmark was contributed by Eric Willigers and the Erlang version was contributed by Christian von Roques originally and modified by Jiri Isa. Both implementations were further modified by myself for added support for varying the chameneos amount.

The experiment was performed using five configurations of C creatures, $C = 7$; $C = 70$; $C = 700$; $C = 7\ 000$ and $C = 70\ 000$ with the amount of meetings (M) set to $M = 8\ 000\ 000$.

Two Scala versions are included. One uses Java threads as concurrency concept and one uses Scala actors. The reason that two different versions were included is that the version built on Java threads (from now on called the *threads version*) was the fastest Scala implementation in The Computer Language Benchmarks Game at the time of writing this thesis. An actor version of the Scala implementation is included since it is the one that is conceptually closest to the Encore implementation.

In the thread version of the Scala implementation, each creature runs on a unique Java thread. The meeting place object uses Scala's *synchronized* method in order to ensure that only one thread can access the meet method at a time.

The actor version of the Scala implementation uses Scala actors to represent the meeting place as well as the Chameneos creatures. One actor per creature and one actor for representing the meeting place. The actors communicate using message passing through Scala's *bang pattern* [12].

In the Erlang version of this benchmark, each Chameneos creature is implemented as an Erlang process. The meeting place, denoted as *Broker* in the implementation, is also implemented as an Erlang process.

The Encore version uses active objects to represent the chameneos and the meeting place. It uses fire and forget semantics through the usage of the *bang operator*. Thus, some optimisations become available because there's no need to use the generated future object for the caller.

3.3 Fannkuch benchmark

Fannkuch, an abbrevation for the German word Pfannkuchen (German for pancakes), is a benchmark scenario that is defined as the following in The Computer Language Benchmarks Game:

- Take a permutation of $\{1, ..., n\}$, e.g. $\{4, 2, 1, 5, 3\}$ if $n = 5$.

[2] `http://benchmarksgame.alioth.debian.org/`

- Inspect the first element, here 4, and reverse the order of the first 4 elements, resulting in $\{5, 1, 2, 4, 3\}$.
- Repeat until the first element is 1. Example run: $\{4, 2, 1, 5, 3\} \rightarrow \{5, 1, 2, 4, 3\} \rightarrow \{3, 4, 2, 1, 5\} \rightarrow \{2, 4, 3, 1, 5\} \rightarrow \{4, 2, 3, 1, 5\} \rightarrow \{1, 3, 2, 4, 5\}$.
- Count the amount of flips (5 in the example above).
- Maintain a checksum such that: [3]
  - $checksum = checksum + sign\_function(permutation\_index) * flip\_count$
- Do this for all $n!$ permutations. Record the maximum amount of flips needed for any permutation.

The original Fannkuch benchmark mainly perform a lot of array manipulation and was originally used to compare the performance of Lisp and C in a sequential setting [1]. The versions of the Fannkuch benchmark used in this thesis are rewritten and designed to gain performance when executed on multiple cores in parallel and measuring scaling and parallelism performance thus become the focus of this version of the benchmark. The parallelization technique used is based on the approach that Oleg Mazurov took [19]. The first step is dividing all the work ($n!$ permutations) into chunks whose size should be small enough to support a good load balance but large enough too keep overhead low. Potential overhead include scheduling overhead and some minor communication overhead. The chunks are then converted to tasks (which might be different from the amount of chunks due to rounding). The tasks are then divided among *Fannkuch workers*, each of which works on a range of permutation indices derived by the task number and the chunk size. Workers perform their work in parallel.

The Fannkuch benchmark was chosen as the third and final benchmark examined in this thesis with the goal of measuring parallelism performance. The algorithm is by design computationally heavy and while message passing does occur in the rewritten versions of the algorithm, message passing makes up a insignificant part of the CPU time. This benchmark thus focuses on strong scalability performance in the programming languages compared. It was chosen on this basis and because it is featured in The Computer Language Benchmarks Game, where it has been shown to scale well in terms of strong scalability in most of the included languages, including Scala and Erlang.

Both the Erlang and Scala implementations used are the ones that scored best on The Computer Language Benchmarks Game at the time of writing this thesis. The Scala

[3] $sign\_function(x)$ is a function that returns 1 if x is an even number and -1 if x is uneven.

version was contributed by Rex Kerr and the Erlang version was contributed by Alkis Gotovos and Maria Christakis.

The Scala version of the Fannkuch benchmark uses threads to parallelise the calculation and spawns the same amount of threads that the computer has available for the process. By default four threads will be used during execution, when not restricted with `taskset`. Otherwise the program will use the amount of threads made available by `taskset`. Each thread obtains a new task atomically when it is done with its current task and the threads are joined at the end of the calculation and their maximum flip count as well as checksum values are summarised.

The Erlang implementation spawns n processes when executing Pfannkuchen(n), e.g. Pfannkuchen(10) $\implies$ 10 spawned worker processes. Each process calculates the maximum amount of flips and the checksum for an evenly divided subset of all permutations. When finished they report their calculated max flip count and checksum to the parent process, the process which is in charge of spawning the workers and summarising the results.

Two Encore implementations are included in the experiments, based on two different work stealing techniques, preemptive and cooperative work stealing. The first being denoted as the *preemptive version* and is used by default in Encore. The other denoted as the *cooperative version*. Both implementations use actors to represent Fannkuch workers along with one actor that acts as a manager. The workers execute the algorithm on a subset of permutations of the initial set $\{1, ..., n\}$, divided into Fannkuch tasks as explained previously. The manager is responsible for summarising the calculation results of all the actors after they have reported their calculation results to it. One actor per Fannkuch task is created.

As switching between the two work stealing techniques is not supported by default and is not something done with ease, an older version of the Encore compiler from July 2015 was used as the cooperative work stealing version. This choice was made since the older version of the Encore compiler uses cooperative work stealing by default. Optimisations apart from the swap of work stealing technique have been done during the time span between the July 2015 version and the current version of the compiler. Thus, the reader is urged to focus on the strong scalability measurements when comparing the two techniques, since the conceptual difference between the two work stealing techniques will likely affect this performance aspect.

In cooperative work stealing schedulers are not allowed to steal work from a busy scheduler. A busy scheduler is a scheduler which is currently executing some actor's invoked method, i.e. a scheduler that is processing a message. To detect whether or

not a scheduler is busy a certain protocol is used involving busy-waiting followed by a possible sleep depending on the outcome of the busy-waiting.

The two different Encore implementations were included mainly to investigate the parallelism performance difference between the two work stealing techniques.

4 Experimental Results

This section presents the measurement results of the benchmarks defined in Section 3. See Section 5 for analysis and discussion about these results.

The experiments were performed on a machine equipped with an Intel Core i5 2500K (4 cores, 6MB L3 Cache) processor running at 4 Ghz combined with 16GB of RAM. It runs on Ubuntu 14.04 LTS (3.16.0-30-generic).

The Scala version used was 2.9.2 and the Erlang version used was R16B03. The Encore compiler version used was released 2016-05-31 and has the commit id `45a8325f8e35045334b486798b7d7157ddb8484a`.

A benchmarking framework was used to run and configure the benchmarks. Each benchmark implementation was measured 25 times for each parameter configuration using `time` in combination with the benchmarking framework. The lowest wall-clock times are included in the results among with the CPU utilisation measured from the run with the lowest wall-clock time. The average memory usage between all 25 runs was used to indicate the memory usage for a benchmark. For more information about the benchmarking framework used, see Appendix A.

4.1 Big Benchmark

Figure 4 shows the measurements of elapsed wall-clock time for the benchmark. The graph is based on Table 1. Figure 6 along with Table 2 shows the execution time speedup over four different CPU core configurations and Figure 5 shows the measured CPU load. Memory usage measurements are found in Table 3.

Speedup in Figure 6 is normalised against the execution time of each language's performance when running on one core.

Workers	500	1000	2000	4000	8000
Encore	1.42	2.91	3.20	8.25	27.52
Scala	7.39	14.27	29.15	56.89	123.74
Erlang	3.66	7.30	15.64	40.90	90.20

Table 1: Elapsed time in seconds for the Big benchmark.

The results show that Encore performs best out of the three languages in all configurations execution time wise. All languages seem to handle the increase in workers reasonably well and when the amount of workers is doubled, execution time is about doubled. This is not the case with Encore between 1000 and 2000 workers. The execution time is expected to rise since the amount of total work to be executed is increased when the worker amount is increased. With each worker added to the benchmark, 16 000 more `ping` messages must be processed, and with that, 16 000 `pong` messages as well.

Cores	1	2	3	4
Encore	5.35	3.16	5.67	3.23
Scala	66.32	47.88	33.08	28.02
Erlang	44.20	27.51	19.70	15.72

Table 2: Execution time in seconds for different core configurations for the Big benchmark with 2000 workers.

In terms of strong scalability, Scala and Erlang scale in a similar way and both obtain a roughly 2.5x performance increase when running on 4 cores compared to running on one core. Encore does scale very well when running the benchmark on 2 cores while running on 3 cores result in a performance loss. Running Encore with 4 cores measures about the same wall-clock time as using 2 cores.

Workers	500	1000	2000	4000	8000
Encore	0.47GB	1.09GB	1.42GB	3.26GB	8.02GB
Scala	0.13GB	0.13GB	0.13GB	0.14GB	0.14GB
Erlang	0.03GB	0.08GB	0.14GB	0.25GB	0.46GB

Table 3: Memory consumption for the Big benchmark.

The memory measurements found in Table 3 show that Scala is more or less not

affected by an increase in the worker amount in terms of memory consumption. Though it is uncertain whether or not Scala allocates more memory than Scala actually uses in this benchmark and thus the memory usage might increase but nevertheless, the memory usage never exceeds 0.14GB in the measurements. Erlang's and Encore's memory usage scale similarly and increase with the amount of workers. Encore's memory usage is increased by 2.09x for each successive increase of the amount of workers in the table on average, while Erlang's memory usage is increased by 2.01x for each tick. The magnitude of Encore's memory usage is higher than the one of Erlang however.

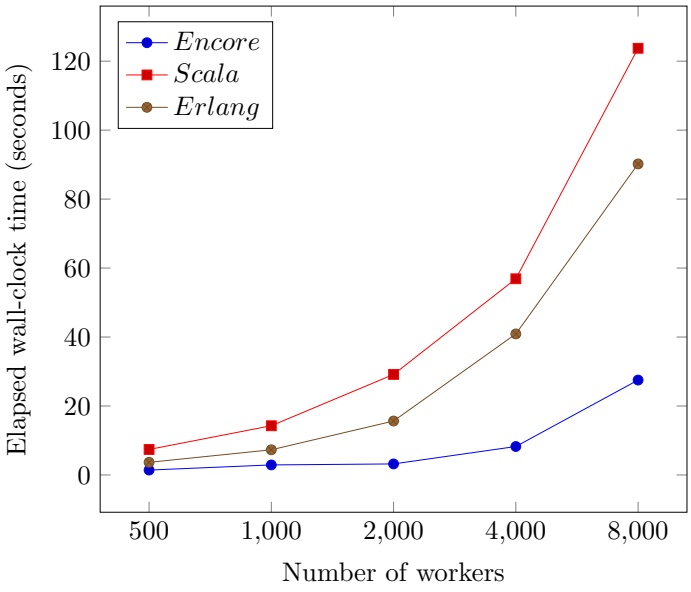


Figure 4: Elapsed time in seconds for the Big benchmark.

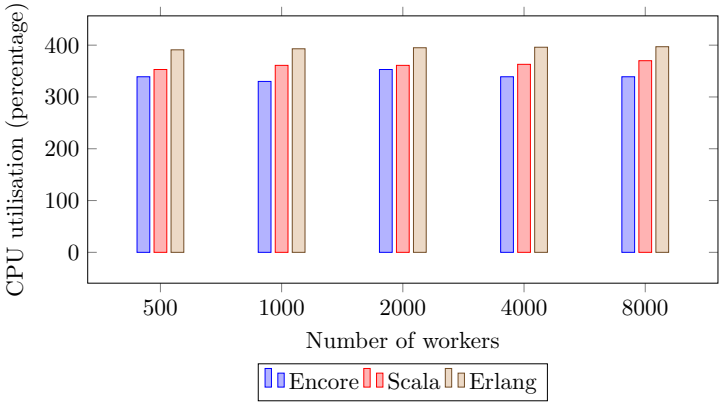


Figure 5: CPU usage percentage for the Big benchmark.

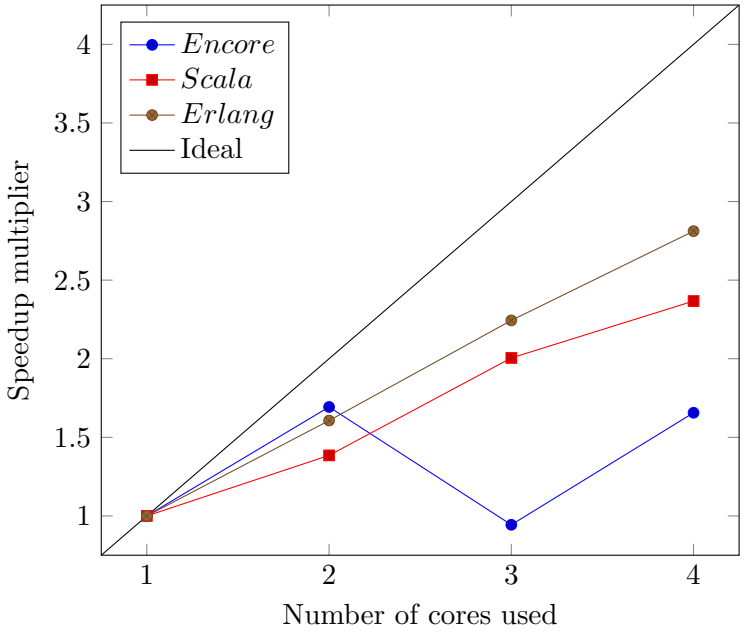


Figure 6: Speedup for the Big benchmark compared to performance with one core (2000 workers).

4.2 Chameneos Benchmark

The results of running the Chameneos benchmark with $M = 8\ 000\ 000$ meetings are found in the following figures. Figure 7 shows the measurements of elapsed wall-clock time for the benchmark. The graph is based on the values in Table 5. Figure 9, based on the values in Table 6, shows the measured speedup when running on different core configurations, normalised against the execution time of each language's implementation when running on one core. Figure 8 shows the measured CPU load when running the benchmark over the range of $\{7, 70, 700, 7000, 70000\}$ creatures on four cores. Table 4 shows the memory usage for the benchmark. This benchmark scenario is also used to measure the scheduling fairness between the languages by measuring the workload distribution between the Chameneos creatures in a benchmark instance. The workload of a creature is represented by the number of meetings that a creature was involved in at the end of the benchmark. This data is represented as boxplots in Figures 10 and 11 where outliers have been hidden. The corresponding standard deviations of the creature meetings for each language is found in Table 7.

Creatures	7	70	700	7000	70000
Encore	0.01GB	0.01GB	0.01GB	0.03GB	0.20GB
Scala (threads)	0.11GB	0.12GB	0.13GB	0.75GB	1.92GB
Erlang	0.02GB	0.03GB	0.04GB	0.15GB	0.62GB
Scala (actors)	0.13GB	0.13GB	0.13GB	0.17GB	0.24GB

Table 4: Memory consumption for the Chameneos benchmark.

While neither of the Scala implementations see an increase in execution time when increasing the creature amount from 7 up to 700 creatures, the thread version of the Scala implementation could not handle 70 000 creatures well and measured a roughly $6x$ increase in execution time compared to running with 7 000 creatures. Both implementations measured a slight increase in execution time when increasing from 700 to 7 000 creatures. While further experiments could have been performed for the Encore, Erlang and Scala (actors) implementations, this was not performed since I deemed the included configurations to be capable of showing the performance differences between the different implementations.

Due to memory constraints in the Scala thread version a test with more than 70 000 creatures was not possible since the execution would sometimes fail due to the JVM system running out of memory, causing the Scala program to be unable to create more threads and therefore all creatures. This is caused by the Java threads being executed as native operating system threads on the test machine,[4] with each operating system thread requiring a relative big amount of memory. The other three implementations included in the measurements showed no issues with memory consumption as witnessed in Table 4.

Chameneos	7	70	700	7000	70000
Encore	4.29	6.30	10.79	13.06	15.06
Scala (threads)	2.72	2.16	2.49	3.58	22.13
Erlang	9.12	8.74	8.03	12.19	13.17
Scala (actors)	24.13	23.45	23.43	31.11	32.43

Table 5: Elapsed time in seconds for the Chameneos benchmark.

When reviewing the elapsed time measurements in Table 5 and Figure 7 it can be seen that the wall-clock times for both the Scala actor version and the Erlang implementation

[4]The utility `top` was used to verify the one-to-one mapping between creatures and native threads.

scale in a similar way. They both gain performance when increasing the creature amount up to 700, then a decrease in performance is seen when increasing the creature amount further. The Encore implementation sees quite small increases in wall-clock time when increasing the amount of creatures in all configurations except when increasing C from 70 to 700. The Scala thread implementation behaves similarly to the Scala actor version and the Erlang implementation for $C \in \{7, 70, 700, 7000\}$ but unlike the case with the other implementations, a big performance hit is seen when increasing C from 7 000 to 70 000.

Cores	1	2	3	4
Encore	12.32	9.84	10.31	10.77
Scala (threads)	4.50	2.33	2.10	2.76
Erlang	8.37	7.82	7.07	7.99
Scala (actors)	33.81	23.03	20.51	22.98

Table 6: Execution time in seconds for different core configurations for the Chameneos benchmark with 700 creatures.

The measurements of strong scalability in Figure 9 with its corresponding values from Table 6 show that the Scala thread version of the benchmark scales well when increasing from one core to two cores, resulting in almost 2x speedup. Neither the Encore nor the Erlang implementation of the benchmark see any major performance gains from increasing the amount of available threads. The Encore implementation even suffers from running on three and four cores compared to using two cores. All implementations except the Encore implementation measures their lowest wall-clock times when using three cores, whereas Encore sees its lowest reading when using two cores. A more in-depth discussion about the reasons behind the Chameneos benchmark performing best around two to three threads is found in Section 5.

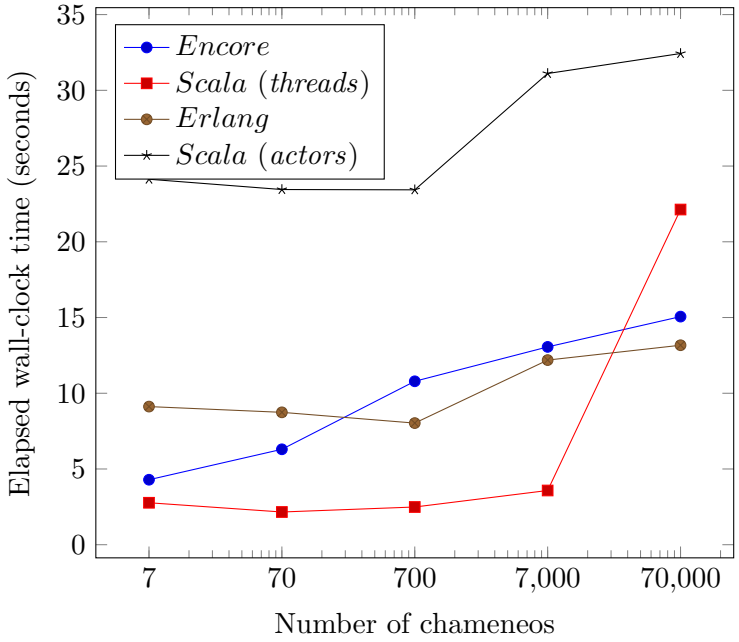


Figure 7: Elapsed time for the chameneos benchmark with $M = 8$ *million* using 4 cores.

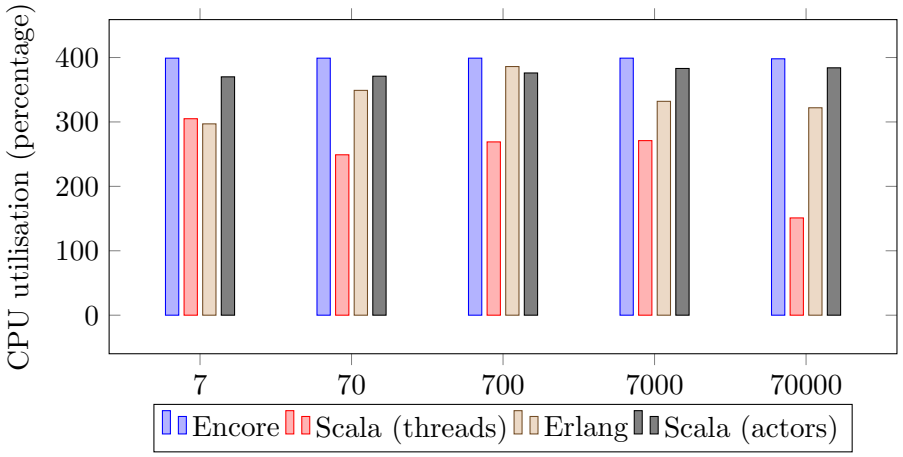


Figure 8: CPU usage percentage for the Chameneos benchmark with $M = 8$ *million*.

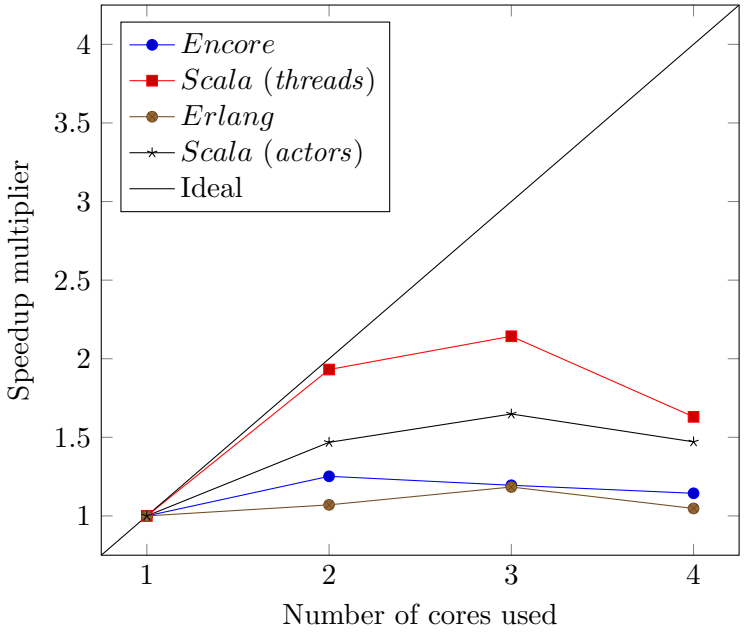
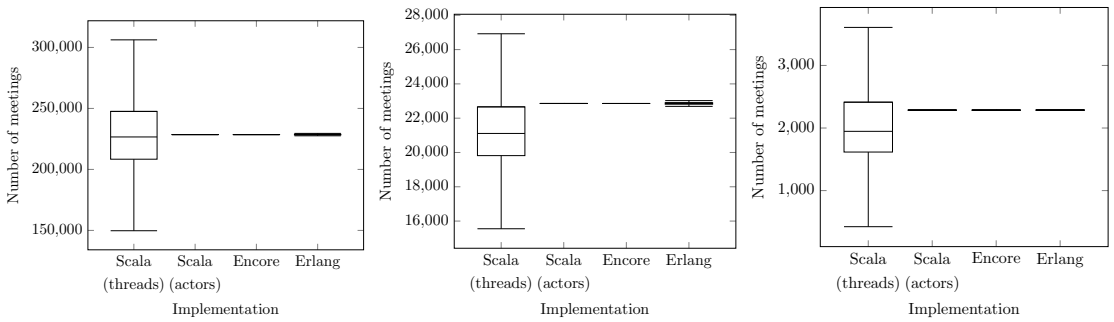
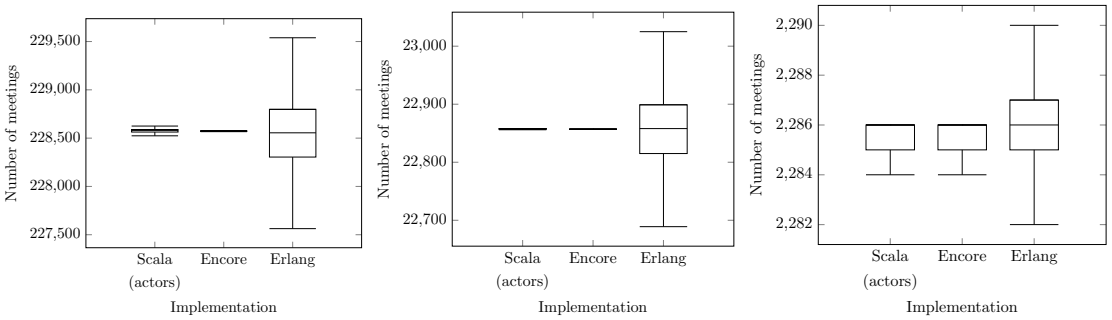


Figure 9: Speedup for different core configurations on the Chameneos benchmark using 700 creatures.



(a) $C = 70$ (b) $C = 700$ (c) $C = 7000$

Figure 10: Boxplot over the number of meetings for each creature in the Chameneos benchmark.



(a) $C = 70$ (b) $C = 700$ (c) $C = 7000$

Figure 11: Boxplot over the number of meetings for each creature in the Chameneos benchmark, excluding the Scala (threads) implementation.

Creatures	70	700	7000
Encore	6.18	0.37	0.45
Scala (threads)	32 452.54	7348.19	2347.61
Scala (actors)	30.99	0.92	0.45
Erlang	841.95	67.06	1.32

Table 7: Standard deviations for the number of meetings in the Chameneos benchmark.

Lastly, when inspecting the fairness measurements found in the Figures 10 and 11 as well as in Table 7, it can be seen that the Scala thread implementation has a very high variance on the number of meetings per creature. In fact, the meeting standard deviation of the Scala thread implementation differ in magnitude compared with the other benchmark implementations for the first configurations, and continues to do so for the other two configurations. The Scala implementation based on the actor model showcases a much more fair meeting distribution, with almost no meeting variance at all between creatures, achieving near perfect fairness between creatures. Encore excels in the fairness aspect of the benchmark, where it measures a more or less perfect fairness throughout the three configurations. The meeting variance of Erlang however is high and symmetrical for the first configuration but shows promising results for the second and third configurations with a rather mediocre measurement on the second configuration but a very low standard deviation on the third. Something that can be observed is that standard deviation for the number of meetings decreases as the amount of creatures is increased in all implementations.

It should be said that the standard deviations are expected to reach smaller numeric values when C is increased, as the ideal number of meetings per creature is expressed through

$$M_c = \frac{2 \cdot M}{C}$$

where M_c is the number of meetings for creature c, M is the input parameter M to the benchmark (amount of meetings taking place in total) and C is the creature amount. An implementation that is identically fair for all configurations should therefore see its standard deviation lowered when the number of creatures increase in practice.

4.3 Fannkuch Benchmark

Figure 12 shows the measurements of elapsed wall-clock time for the benchmark during execution, the graph of which is based on the values in Table 8. Figure 14 shows the measured speedup when running on different core configurations with parameter N = 12, normalised against the execution time of each language's measured execution time when running on one core. Figure 13 shows the measured CPU load when running the Fannkuch benchmark on the parameters 10, 11 and 12 on four cores. Table 9 shows the memory usage.

N	10	11	12
Encore (preemptive)	0.11	1.49	20.92
Encore (cooperative)	0.23	2.87	40.53
Scala	0.25	0.77	7.41
Erlang	0.41	4.16	54.83

Table 8: Elapsed time in seconds for the Fannkuch benchmark.

As seen in Figure 12 and from the values in Table 8, Encore performs the best when $N = 10$ but is beat by Scala when running with the parameters $N = 11$ and 12. Scala sees only a 3.08x slowdown when increasing the parameter from 10 to 11 whereas both Erlang and both versions of the Encore benchmark have their execution time increased by at least a tenfold. Scala also handles the increase from $N = 11$ to $N = 12$ better than it counterparts with a 9.62x slowdown. Both the preemptive and cooperative Encore implementations are hit by a 14.04x and 14.12x slowdown respectively while Erlang sees a slowdown of 13.18x when increasing N from 11 to 12.

N	10	11	12
Encore (preemptive)	4.18MB	3.83MB	4.48MB
Encore (cooperative)	9.68MB	12.53MB	6.49MB
Scala	42.79MB	40.91MB	42.48MB
Erlang	22.31MB	16.92MB	18.82MB

Table 9: Memory consumption for the Fannkuch benchmark.

The memory usage is not changing much when varying the parameter to the Fannkuch benchmark and no implementation uses a significant amount of memory on any configuration. Encore's cooperative version measures its lowest memory usage at $N = 12$ however.

Cores	1	2	3	4
Encore (preemptive)	82.76	41.45	27.76	20.92
Encore (cooperative)	83.51	46.59	45.35	40.53
Scala	28.96	14.45	10.01	7.74
Erlang	217.30	108.90	72.87	55.15

Table 10: Execution time in seconds for different core configurations for the Fannkuch benchmark with N = 12.

From Figure 14 and Table 10 it is apparent that the cooperative Encore version is lagging behind in terms of strong scalability in the Fannkuch benchmark. All other implementations scale near the ideal. Erlang and the preemptive version of the Encore implementation scale in a more or less identical way and thus overlap in the figure. While the Encore implementations used in the measurements uses one actor per Fannkuch task, some initial experiments were made with different configurations of the number of actors in the Encore implementation. Two other configurations were explored to investigate the scaling performance. One which created the same amount of actors as available processor cores, as done in the Scala implementation. The other configuration created N actors, as done in the Erlang implementation. The active objects in both of these alternative configurations atomically grabbed Fannkuch tasks to process from a work pool when each respective actor was done with its current Fannkuch task. However, these two explored configurations had similar, if not identical, performance of the implementation included in the measurements, which uses the same amount of active objects as the input parameter to Fannkuch. Instead of using the same number of actors/act-

ive objects/processes across all language implementations, the respective best version in The Benchmark Shootout Game was chosen for each language.

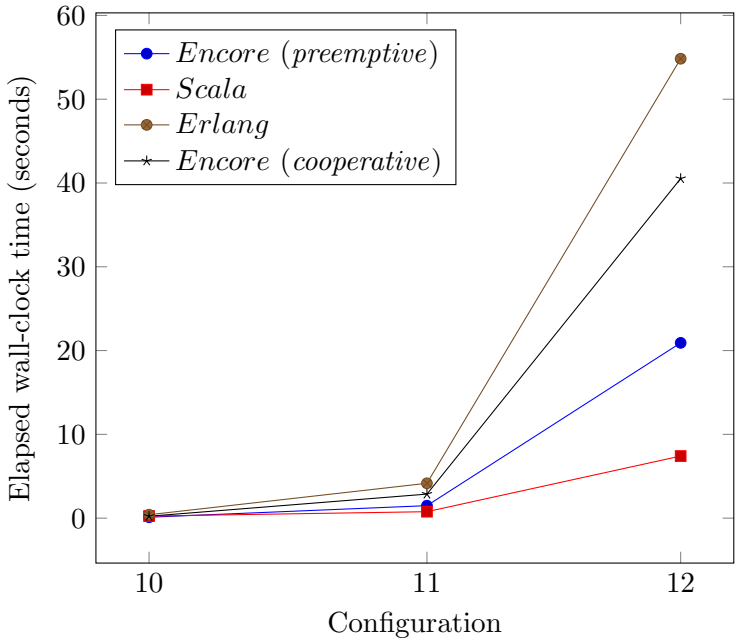


Figure 12: Elapsed time in seconds for the Fannkuch benchmark.

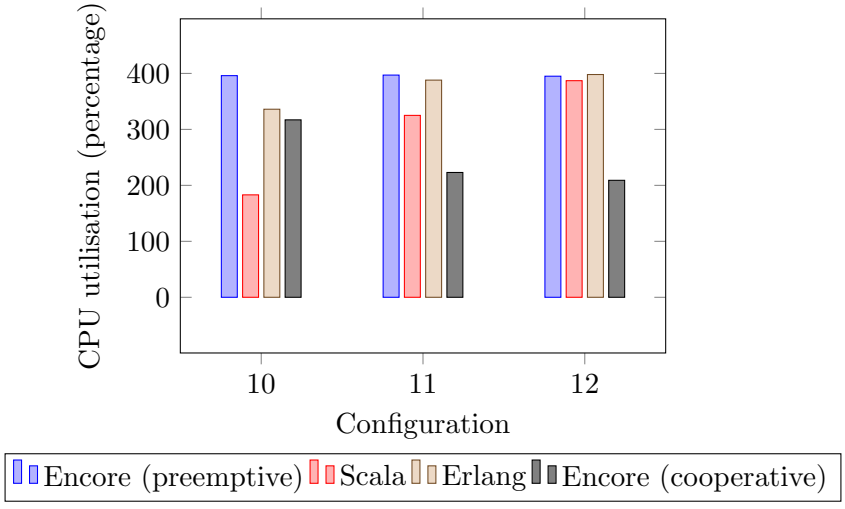


Figure 13: CPU usage percentage for the Fannkuch benchmark.

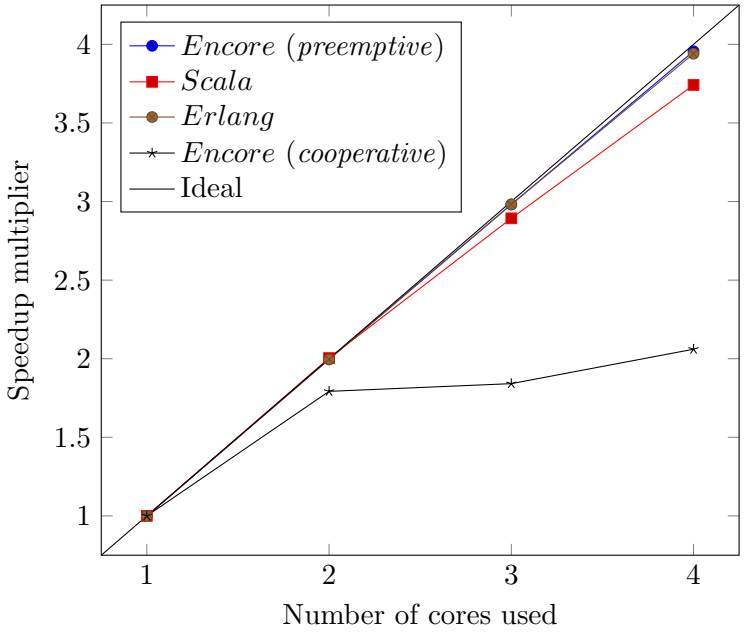


Figure 14: Speedup for different core configurations on the Fannkuch benchmark (using N=12).

5 Analysis of the Results

This section discusses and analyse the results from Section 4 and draws some conclusions based on them.

As seen in Section 4, Encore performs up to par with and sometimes better than Scala and Erlang on these three specific benchmarks when inspecting the wall-clock time measurements in isolation. Encore uses a lot of memory compared with the other languages in the Big benchmark but lower on the other two benchmarks as witnessed in the Tables 3, 4 and 9. In terms of strong scalability Encore does not scale as well as Scala and Erlang in the Big and Chameneos benchmarks. In the third benchmark Fannkuch, using the preemptive version, Encore scales close to ideal. The ideal being 4 cores $\implies$ 4x speedup. Encore does not scale well when using cooperative work stealing however.

5.1 Analysing the Big Results

From the measurements of the Big benchmark it can be seen in Figure 4 that all three implementations are affected by an increase of the number of workers in a similar way.

However, as seen in Figure 6, Encore is outperformed by both Scala and Erlang when strong scalability is measured except when using two cores. Encore see a big increase in performance when going from one to two threads but increasing the amount of cores/-threads from that point is shown to be detrimental to the benchmark's performance. An anomaly is observed when using three threads, with worse performance compared to running on one thread. Since the measurements were done close to time of concluding the thesis, this anomaly was unfortunately not analysed due to time constraints.

Not only does Encore perform the worst in the strong scalability measurements, it also observes the highest memory usage across the three languages. One of the causes of this is Encore's *cycle detector*. This mechanism generates messages as a means to determine whether an actor can be garbage collected or not. When an actor's message queue becomes empty, i.e. it has just consumed a message and no other messages are available in the queue, it sends a `block` message to the cycle detector. This message notifies the cycle detector that the actor may be ready to be collected. After the cycle detector has gone through appropriate steps in its algorithm to detect a dependency between the actor and other actors, it will then decide whether the actor can be collected. However, an actor can be incorrect in its assumption that it is ready to be collected. When an actor realises that the `block` message was based on a faulty assumption, it sends an `unblock` message to notify the cycle detector that it is not ready to be collected. An actor is forced to send a `unblock` message if it has previously sent a `block` message and then received a new message in its message queue.

In the Big benchmark it is likely that an actor repeats the sequence of sending a `block` message to the cycle detector shortly followed by an `unblock` message. This is due to the design of the Big benchmark, where most actors receive one message in its previously empty message queue, process the message and then discovers that its queue is now empty and thus sends a `block` message to the cycle detector. However, each message that a worker (actor) process in the Big benchmark will, depending on the received message, cause two scenarios to unfold. The first scenario is where the received message is a `pong` message, causing the receiver to send a `ping` message to a random worker in its neighbourhood. The other scenario is if the received message is a `ping` message, the receiver will then send a `pong` message back to the sender. Both scenarios will likely cause the `block`→`unblock` sequence to occur. All these `block` and `unblock` messages are generated in vain for all received messages except the last message that a worker receives, after which it will actually be ready to be garbage collected. This is assuming that no two (or more) workers send `ping` messages to one single worker, i.e. the random distribution of targets being completely uniform. This is obviously not the

case in practice, the main takeaway however is that a lot of, in this case, unnecessary messages are generated, causing a very high memory measurement.

What is the case in practice however is that this `block→unblock` behaviour indeed causes a high memory usage. Table 11 shows the effect of disabling the `block` and `unblock` messages in the Encore runtime system.

Workers	500	1000	2000	4000	8000
Encore	0.47GB	1.09GB	1.42GB	3.26GB	8.02GB
Encore (no-block)	0.06GB	0.20GB	0.47GB	0.85GB	1.57GB

Table 11: Memory consumption for the Big benchmark with and without collection of actors during runtime.

As seen in the table, the memory usage is significantly reduced, where it has been reduced down to $\frac{1}{5}$ of the regular values on average. While turning off the `block` and `unblock` messages in the runtime system does disable garbage collection of actors until the end of the program execution, this still helps immensely with the memory consumption. The actual garbage collection of actors in the Big benchmark (and also the Chameneos benchmark) actually makes no difference in the memory measurements as all actors will sometime during execution be alive at once. Remember that the memory measurement is a measure of the *maximum resident set size*. That is, the highest recorded amount of allocated memory at a single point of time during execution. The memory usage for Encore is still high on this benchmark however, and this is likely caused by how the references to neighbours are handled by the Encore runtime system according to an UPLANG team member.

The `block→unblock` behaviour also exists in the Chameneos benchmark. The number of occurrences are much fewer than in the Big benchmark however, which is why the memory usage of the Chameneos benchmark stays at a reasonable level.

5.2 Analysing the Chameneos Results

To understand potential reasons behind the strong-scalability measurements of the Chameneos benchmark showed in Figure 9, a more in-depth analysis of the benchmark itself is required.

First we note that running the Chameneos benchmark on two threads results in the fastest execution time using Encore. There is also a difference in memory usage between different thread configurations as seen in Table 12.

Number of threads	1	2	3	4
Memory usage	736.58MB	8.78MB	10.49MB	14.62MB
Wall-clock time	12.32s	9.84s	10.31s	10.77s

Table 12: Memory consumption and wall-clock time for different ponythread configurations for the Encore implementation of the Chameneos benchmark, using $C = 700$ and $M = 8$ *million*.

The Encore version of the Chameneos benchmark starts by all creatures requesting a meeting at the meeting place by sending it a `meet` message. In the case where $C = 700$, 700 such messages are sent at the beginning of the benchmark, one meeting request by each Chameneos creature. At this time 700 meeting request messages await in the meeting place's message queue while all creatures are idle. Each time the meeting place actor has processed a sequence of two `meet` messages from two different creatures, it invokes the `meet` method of the two creatures participating in the meeting. The two creatures will then change their colour and immediately request a new meeting. Thus, for each meeting that occur at the meeting place, only two threads can be fully utilised to execute the invoked `meet` methods of the participating creatures, one thread per participating creature. In this scenario all threads can not be utilised to full effect, more or less wasting time context-switching and work stealing from each other for close to no benefit. This is also evident in Table 12 where it can be noted that when running the benchmark on two pony threads, both the wall-clock time and memory consumption measure their lowest readings in the table. This can be viewed as the two messages produced from a meeting request are handled before generating new messages. Thus, not clogging up the message queue of the meeting place.

The actual work performed in a creature's `meet` method is also very lightweight. It involves incrementing and modifying some class attributes and then sending away a message to the meeting place actor. Considering this along with the fact that when the meeting place actor invokes the `meet` method on the two creatures participating in a meeting, the work load is appended to the scheduler currently executing the meeting place actor. Other available schedulers will need to work steal these two method invocations, adding some overhead before the actual work can be run in parallel.

As seen from the results in Section 4 when measuring fairness of the four implementations, scheduling in Encore is a lot more fair compared to the scheduling in the Scala thread implementation, but so is every other implementation included. The unfairness of the Scala thread implementation is explained by each creature being executed

in the context of an operating system thread. That is, Scala's runtime system has no control of this scheduling and it is up to the operating system to schedule the threads (creatures). A Scala thread implementation without the one-to-one thread to creature mapping, e.g. an implementation featuring a thread pool, would likely be more fair than the implementation included.

When comparing Encore to Erlang and the Scala actor version, it is apparent that the Encore implementation outperforms the other two implementations in terms of fairness, though only by a slight margin over the Scala actor implementation. However, the Scala actor implementation is also the slowest implementation of the three. This is in contrast to the Scala thread version that is the most unfair implementation by far but also the implementation that is fastest for smaller numbers on C. The thread implementation of Scala also performs the best in the strong scalability measurements. Overall the fairness of Encore is very low across the three configurations.

Cores	1	2	3	4
Std	0.0	4.32	0.91	1.29

Table 13: Standard deviation for the Encore implementation of the Chameneos benchmark with $M = 8$ million and $C = 800$.

It is impossible to reach a standard deviation of 0 for the configurations included in the results due to 8 not being divisible by 7. Measurements of one additional configuration with such a property was included in Table 13 to showcase the effect schedulers have on fairness in Encore, where one thread renders a standard deviation of 0. This is to be expected since each creature is scheduled in a round-robin fashion and no work stealing occur.

5.3 Analysing the Fannkuch Results

To understand the performance of the cooperative work stealing version of the Encore implementation, both wall-clock time and strong scalability wise, one first need to first understand some of Encore's inner workings.

As briefly mentioned in Section 3, cooperative work stealing is not permitted to steal work from other schedulers during message processing. To detect if a scheduler is busy, the work stealing thread that attempt uses a certain protocol for attaining information about whether or not a scheduler is inactive. To do this, synchronisation between schedulers is required. To synchronise with the other scheduler, busy-waiting

will first be used. If the busy-waiting renders unsuccessful then sleep will be initiated, causing a context-switch.

The consequence of this is shown when analysing `time`'s measurements of both versions of the Fannkuch Encore implementation, the measured context-switch amount of the actor version is very high compared with all other implementations, Scala and Erlang included.

To analyse what caused the high context-switch amount, the Linux tool `perf` [20] was used to compare the two Encore implementations of the Fannkuch benchmark. The perf tool can be used to analyse performance events at both hardware and software levels and it can be used to measure only the performance counters related to a specific program instead of measuring performance counters system wide [21]. In this case, `perf` was used in combination with the *-cs* (cs for context-switch) flag to measure the context-switch software event.

The results from `perf`'s measurements shows that the 99.95% of the cooperative version context-switches are caused by the synchronisation mechanism during work stealing. `perf` reports a higher amount of *cs events* (context-switching events) for the cooperative version compared with the preemptive version, measuring 17 000 and 557 respectively. However, the cooperative version also executes for a longer time compared to the preemptive version, so it is also more likely to record a higher amount of events.

Based on the profiling, the frequency of context-switches seems to differ between work stealing techniques used. Thus, the different work stealing techniques likely explain the difference in strong scalability between the two versions. Where the cooperative implementation is held back by the need to synchronise before work stealing. To further validate this claim, we also note in Figure 13 that the CPU utilisation of the cooperative implementation is compared with the other implementations, hinting that a lot of time is spent waiting.

6 Related Work

The methodology of benchmarking concurrency and parallelism in this thesis is inspired by the work of Cardoso et al. [9]. They benchmark a range of Actor and Agent languages, including Erlang and Scala, over three different benchmark programs. Two benchmarks, *thread ring* and *chameneos redux* are explored by Cardoso et al., which are concurrency adapted versions of the ones from the Computer Language Benchmarks Game website. A Fibonacci benchmark is also explored, where a worker acts as a calculation server for C clients. Their measurements of the Chameneos benchmark show that Erlang was the

most fair language in terms of scheduling the creatures, more fair than Scala. While Cardoso et al. do feature Scala in their measurements, their implementation is based on the Akka actor library, while the implementation explored in this thesis is built upon Scala Actors. Their measurements further show that Erlang had a lower execution time as well as a lower memory consumption for all creature configurations compared to Scala. This thesis also explore the *chameneos redux* benchmark by measuring memory usage, execution time and fairness over a set of creature configurations, but focus is also put on analysing the strong scalability of the benchmark.

Cardoso et al. also explore the performance of three variations of the *thread ring* benchmark, included in The Computer Language Benchmarks Game [8]. The thread ring benchmark features a set of workers that pass around tokens to each other. Three different variations of the benchmark are explored and the results show that Scala scales best when considering the increases in wall-clock time when the amount of token passes increased. Three languages were explored, including Scala and Erlang. Erlang stood out as the language with the lowest wall-clock execution time as well as the lowest memory measurements of the three explored languages. While this thesis does not explore the *thread ring* benchmark, Cardoso et al. compares Scala and Erlang in respect to concurrency and the way strong scalability is evaluated is similar to the approach taken in this thesis.

The Savina benchmark suite presented by Imam and Sarkar is a benchmark suite for actor oriented programs [14]. Two of the benchmarks included in Savina are also included in this thesis, namely the *Big* and *Chameneos* benchmarks. No measurements of these two specific benchmarks are performed by Imam et al. but measurements of a few other benchmarks are performed over nine different actor libraries in Scala.

7 Discussion

7.1 Benchmarking Framework

Some flaws exist in the benchmarking framework software. One of which is the `CONFIG` module which is cluttered and hard to grasp at first glance. Improvements could be performed in this aspect. The benchmarking framework also lacks a graph generation component, a feature that would greatly reduce the overhead of manually creating the plots. One way to generate graphs would be using `gnuplot` [15].

Lastly, a lot of trust is given to the programmer of the benchmarks. In order to add a benchmark to the suite, the programmer must write corresponding ruby scripts that the benchmark suite assumes are correct. A potential solution to this would be to not

let the programmer specify how to execute each specific program but instead execute all programs written in a specific language in a generic way specified by the suite. Only letting the programmer specify the path to the program and the language it is written in (the benchmark suite supports benchmarks written in any language since it is the programmers responsibility to execute the programs).

7.2 The Benchmark Implementations

The benchmark implementations might not be implemented in the optimal way, including the Erlang and Scala implementations. Some precautions have been taken while developing the Encore implementations to avoid currently known pitfalls by primarily avoiding the usage of operators with at the time unsatisfactory properties. However, due to Encore's young age and continual development, it is often unknown what is the best practice to follow during development of Encore applications.

Fire and forget semantics has been used throughout the benchmarks to communicate between active objects. This was primarily done because it was a good fit for the scenarios explored in these benchmarks but also because the developers of Encore advised against using `get` operators at the time of writing this thesis. The operator which enables a caller to block and wait for a result of a method invocation on an active object.

The Scala implementation for the Big benchmark was based on the version included in Savina [14], with the removal of Savina specific code included to make the implementation compatible with their benchmarking framework. The Erlang version is a modified version of the one featured in `bencherl` [3]. Both implementations were modified the addition of neighbourhoods. I believe that there are more modern ways of writing the Scala version by using the Akka library instead of using the Scala actor library that was used in the included implementation, this might result in faster execution times for the Scala version of the benchmark. The modifications of the Erlang version involves a lacklustre and subpar implementation of list traversing. However, through some modest profiling through `perf`, it was observed that little to no time was spent in these added steps.

7.3 The Measurement Tools

Using the `GNU time` utility for measuring was deemed adequate for this thesis by myself due to its ease of use and it provided most of the timing statistics needed for this thesis. However, while `time` is a good utility for measuring wall-clock time, CPU utilisation and context-switch amount, it is not the most fitting tool for measuring memory

utilisation in the benchmarks explored in this thesis. This utility measures the highest amount of memory allocated by the process from the operating system, but it occurs that programming languages allocates more memory than they actually use and Encore is such a language. However, the amount of memory used and the amount of memory allocated from the system should not differ by very big amounts. Thus, the memory usage measurements can be used as a way of illustrating how the memory usage increases when input parameters are adjusted. Other approaches of measuring the amount of used memory in an Encore program includes instrumenting the Encore compiler to log or trace the amount of memory allocated and released during the execution. Similar approaches would be needed for the other languages that are included in this thesis as well, a lot of overhead which was unfortunately outside the time budget of this thesis.

7.4 The Measurement Method

It can be argued that reporting the average wall-clock execution time over several runs for the different benchmark implementations is more representable of their real performance. By measuring the average wall-clock time no benchmark implementation can *"get lucky"* by achieving an abnormally low execution time during the measurement process. However, manual examination of the measurements show the wall-clock execution time measurements do not differ much between runs in most configurations. The approach of reporting the lowest wall-clock execution time over several runs was chosen on the basis that this was the approach taken by Cardoso et al. [9].

8 Future Work

While this thesis shows how Encore performs on a small set of benchmarks it could give more insight into why Encore performs the way it does. During the time span of this thesis work, a fair bit of time was spent on configuration of both a FreeBSD and a Solaris system in order to trace Encore's inner workings using DTrace, a tool that is not available on Linux systems [10]. Due to compatibility issues between different software and operating systems, no result came of this. Future work involves getting DTrace or SystemTap up and running and instrumenting the Encore compiler to allow for tracing the scheduler and other points of interest while running the benchmarks. This could open up for possibilities to get a deeper understanding of Encore's concurrency and parallelism performance.

It would also be interesting to add more benchmarks to the comparison to get a broader view of the performance of Encore. This would help answering the question

whether the results in this thesis are representative of the overall performance of Encore. Running the benchmarks on a machine with more than 4 cores would also help a bit in answering the same question.

At the time of writing this thesis a way of switching to preemptive work stealing for active objects was not fully supported. This is why a detailed comparison between the two is not included apart from the Fannkuch benchmark. Future work involves comparing the two work stealing techniques in terms of strong scalability and parallelism performance.

Further future work includes improving the benchmarking framework by adding support for OSX by detecting the operating system and changing the path for the `time` utility[5]. The framework could also benefit from auto generation of plots, potentially using `gnuplot`. The process of executing a benchmark is highly coupled at this time, future work involve modifying the configuration files for the benchmarks located in the *rbfiles* folder to remove their need to execute the benchmark through the `helper` module. The configuration files should simply return the path to the program as well as the execution command so that the `runner` module can execute the benchmark.

9 Conclusion

In this thesis a benchmarking framework for benchmarking four performance aspects over the three languages Encore, Erlang and Scala was introduced and used to produce measurements of memory usage, execution time, strong scalability as well as fairness. Three benchmarks were explored: the Big benchmark, focusing on communication performance through many-to-many message passing; the Chameneos benchmark, a concurrency benchmark which allowed for measuring the fairness between schedulers as well as the performance impact of heavily congesting a shared resource with messages; followed by the Fannkuch benchmark, a benchmark with focus on parallelism performance.

The measurements of these benchmarks show that the performance of Encore is up to par with Scala and Erlang when examining the execution time of the benchmark instances. In fact, Encore greatly outperform the other two languages in terms of wall-clock time performance in one of the three benchmarks, namely Big. However, Encore uses up a lot of memory in the same benchmark and the measurements of strong scalability show that Encore does not scale as well as Scala and Erlang in the two concurrency benchmarks Big and Chameneos. When communication is minimised however, as in the Fannkuch benchmark, Encore wins the strong scalability battle by a small margin,

[5]On OSX GNU time is denoted as `gtime` and can be installed through `brew install gnu-time`.

with a $3.95x$ performance increase when using four cores compared to using one core. Encore also sports the most fair scheduling policy between the three explored languages, achieving a near ideal fairness in the Chameneos benchmark.

Considering it being such a young language, Encore shows promise for the future with fast execution time and good parallelism performance. In time, when the minor quirks with memory consumption as well as the strong scalability performance in communication heavy scenarios have been addressed, Encore could become a real challenger to other languages in the same paradigm.

References

[1] K. R. Anderson & D. Rettig. *Performing Lisp analysis of the FANNKUCH benchmark*, SIGPLAN Lisp Pointers VII, 4 (October 1994), Pages 2-12. ACM, 1994.

[2] J. Armstrong. *Erlang*, volume 53 (no. 9) of *Communications of the ACM*, pages 68-75. ACM, 2010.

[3] 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 workshop (Erlang '12)*, pages 33-42. ACM, 2012.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, et al. *A view of the parallel computing landscape*, volume 52 (no. 10) of *Communications of the ACM*, pages 56-67. ACM, 2009.

[5] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, A. M. Yang. *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*, volume 9104 of *Lecture Notes in Computer Science*, pages 1-56. Springer, 2015.

[6] J. Britt, Neurogami. M. R+D. *Open3*, Ruby-doc.org. `URL: http://ruby-doc.org/stdlib-1.9.3/libdoc/open3/rdoc/Open3.html` (Visited 2015-11-04)

[7] J. Britt, Neurogami. M. R+D. *OptionParser*, Ruby-doc.org. `URL: http://ruby-doc.org/stdlib-1.9.3/libdoc/optparse/rdoc/OptionParser.html` (Visited 2016-04-28)

[8] R. C. Cardoso, J. F. Hübner & R. H. Bordini. *Benchmarking Communication in Actor- and Agent-Based Languages*, Volume 8245 of *Lecture Notes in Computer Science*, pages 58-77. Springer, 2013.

[9] R. C. Cardoso, M. R. Zatelli, J. F. Hübner & R. H. Bordini. *Towards Benchmarking Actor- and Agent-Based Programming Languages*, Proceeding of 2013 workshop on Programming based on actors, agents, and decentralized control, AGERE! '13, pages 115-126, ACM New York, 2013

[10] dtrace.org. *About DTrace*, dtrace.org. `URL: http://dtrace.org/blogs/about/` (Visited 2015-11-04)

[11] D. Flanagan & Y. Matsumoto. *The Ruby Programming Language*, Beijing, O'Reilly, 2008.

[12] P. Haller & F. Sommers. *Actors in Scala*, Artima incorporation, 2012.

[13] P. Haller & M. Odersky. *Event-based programming without inversion of control*, In Proc. *Joint Modular Languages Conference*. Springer LNCS, 2006.

[14] S. Imam & V. Sarkar. *Savina — An Actor Benchmark Suite Enabling Empirical Evaluation of Actor Libraries*, In Proceedings of the *4th International Workshop on Programming based on Actors Agents & Decentralized Control AGERE14*, pages 67-80. ACM, 2014.

[15] P. K. Janert. *Gnuplot in action*, Manning Publications Co, Shelter Island, NY, USA, 2016.

[16] C. Kaiser & Jean-Francois Pradat-Peyre. *Chamneoes, a Concurrency Game for Java, Ada and Others*, ACS/IEEE International Conference on Computer Systems and Applications, 2003.

[17] Linux man-pages project. *TIME(1) Linux User's Manual*, Linux man-pages project. `URL: http://man7.org/linux/man-pages/man1/time.1.html` (Visited 2015-11-04)

[18] Linux Man page. *taskset(1) - Linux man page*, Linux Man Page. `URL: http://linux.die.net/man/1/taskset` (Visited 2015-11-04)

[19] Oleg Mazurov, *fannkuch-redux Java program*, The Computer Language Benchmarks Game. `URL: https://benchmarksgame.alioth.debian.org/u64q/program.php?test=fannkuchredux&lang=java&id=1` (Visited 2015-10-02)

[20] A. C. de Melo. *The new linux 'perf' tools*, Linux Kongress. `URL: http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf`, 2010. (Visited 2015-10-02)

[21] A. Z. Netto, R. S. Arnold. *Evaluate performance for Linux on POWER*, IBM developerWorks. `URL:http://www.ibm.com/developerworks/linux/library/l-evaluatelinuxonpower/` (visited on 2015-10-02)

[22] T. Norgay, *Tensing's New World Fauna Handbook, Third Edition*. Tengboche, 1987.

[23] M. Odersky et al. *An overview of the scala programming language*, Technical Report IC/2004/64. EPFL Lausanne, Switzerland, 2004.

[24] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl. *All you wanted to know about the HiPE compiler: (but might have been afraid to ask)*, In Proceedings of the 2003 ACM SIGPLAN workshop on Erlang (ERLANG '03). ACM, New York, NY, USA, 36-42, 2003.

[25] Upscale. *Welcome to Upscale*, Upscale. `URL: https://upscale.project.cwi.nl/` (Visited 2015-10-18)

Appendices

A Benchmarking Framework

When creating a set of benchmarks one also needs a way to conveniently run and gather data from these benchmarks. This section presents the benchmarking framework which was developed in the context of this thesis to fulfil this need.

The benchmarking framework was developed with the goal of making it easier for benchmark authors to configure, run and gather measurements of benchmark applications for Encore, but also for other programming languages. The framework allows benchmark authors to specify what benchmarks to run, how many times they should be executed and also what parameters that should be used during execution. The framework can extract information such as lowest wall-clock execution time and the average memory utilisation over N number of runs. Other information such as context-switch amount and CPU utilisation can also be collected during execution of a benchmark.

The framework is written in the programming language `Ruby` (version 1.9.3). Ruby is an object-oriented scripting language, where a default Ruby program is a list of instructions that are executed sequentially and executed through an interpreter [11]. Ruby was chosen as the implementation language for the framework due to being quickly adaptable by most programmers and due to its comprehensive standard library, featuring Open3 [6] for executing shell commands while capturing the output of the command. Ruby also has well-documented regular expression support, useful when parsing the time measurement output.

The framework consists of 5 modules, as illustrated in Figure 15.

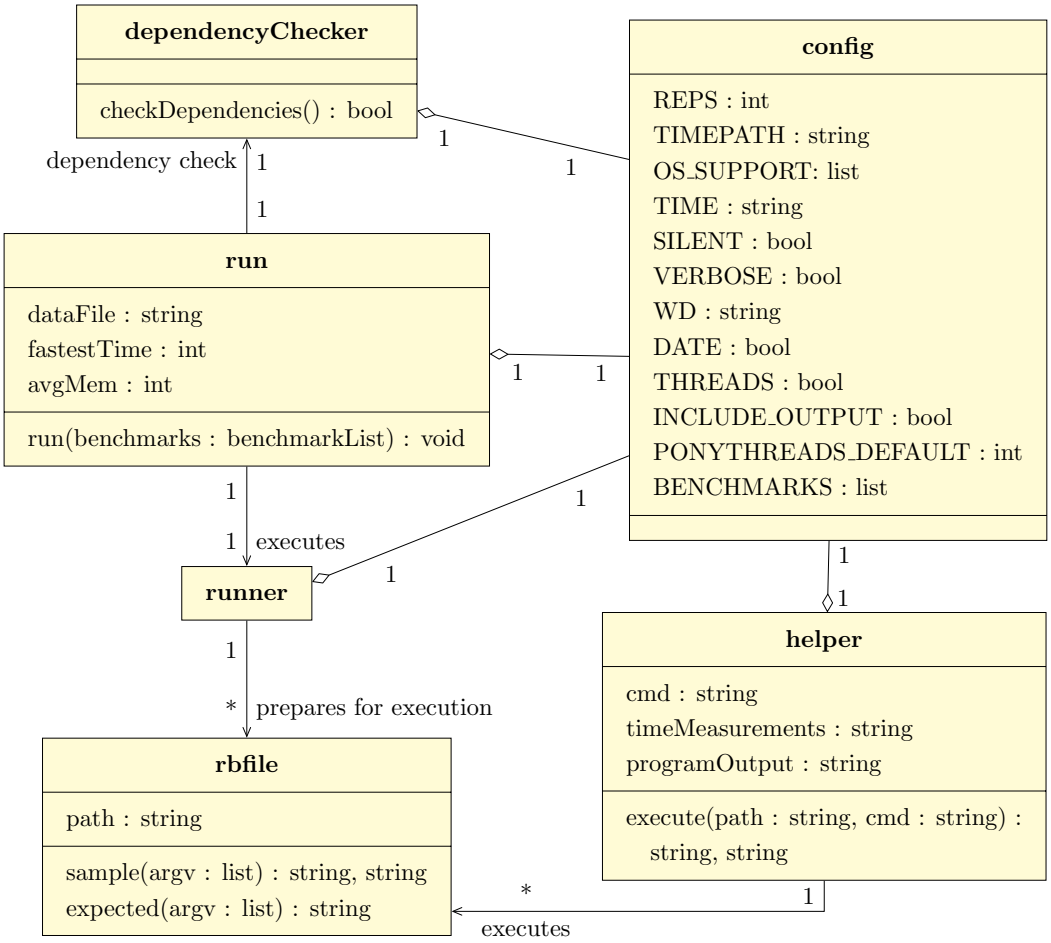


Figure 15: An overview of the benchmarking framework.

The end users of the benchmarking framework configures the framework through the `config` module. This module offers the possibility to configure the path to the measurement tool (by default `time` is used), configuring dependencies of the framework, supported operating systems as well as configuring output options such as whether to include date and time or the amount of threads in the output file. One can also decide whether the program output should be included in the output file. The most important configuration option in this file however is specifying what benchmarks to run, how many times to run them and with what parameters.

The user specifies what benchmarks to run and their parameters using the following format for each benchmark in the `config` module:

```
[benchName: string, confList: list]
```

That is, each benchmark is defined as a list[6] of two elements, the name, which must be identical to the name of the configuration file and the configuration list. Each element of the configuration list for each respective benchmark has the following format:

```
[description: string, [arg1, arg2, ..., argn]: list]
```

where description is a short description of the parameters which will be included in the name of the results file for that parameter setting. The list that follows is the list of input parameters to be given to the benchmark executable.

def *sample(args)*
  path = "*path_to_program*"
  threads =ENV['*ponythreads*'] `# Sets # of threads to value in config`
  $arg_0, arg_1 = *args$ `# Unpacks arguments from input list`
  command = "./name_of_executable –ponythreads #{*threads*} #{arg_0}
  #{arg_1}"
  execute(*path*, *command*) `# Calls execute method in helper module and returns the result`
end

Algorithm 6: Example of a rbfile for a benchmark with two input arguments.

To enable the framework to execute and take measurements of a benchmark the benchmark author must supply a corresponding configuration file in a *rbfiles* folder (The `rbfile` module in Figure 15 corresponds to such a configuration file). This configuration file should have a relevant name in context of the benchmark and must supply one function, which must be named `sample`. This function takes one argument as input which is a vector containing the parameters specified in the `config` module. The sample function must then execute the benchmark with the help of the `helper` module by returning the result of calling `helper`'s `execute` function, a function which takes a path to an executable of the benchmark as well as a shell command in string form that specifies how to execute the executable. An example of a benchmark configuration file for a benchmark with two input arguments is given in Algorithm 6.

The `run` module is to be viewed as the core module of the framework and thus the entrance point of the program. This module parses measurement data through string-matching using regular expressions. This module also performs calculations on

[6]Ruby 1.9.3 does not have support for tuple types, which would be the data type of choice for this application otherwise.

the parsed measurement data, e.g. calculating the average memory consumption. It is also the module responsible for storing the results of the benchmarks to disk. As this is the entrance point of the benchmarking framework software, this module also features some input flags which are processed using OptionParser [7]. The available input flags to the framework can be found in Table 14.

-p [THREADS]	Specifies the default value for amount of threads to use in Encore
-t [TAG]	Prepends TAG to all generated data files
-s	Runs all benchmarks in silent mode (suppresses some of the output to shell while running)
-v	Runs all benchmarks in verbose mode (prints debug output to shell while executing)

Table 14: Input flags for the benchmarking framework software.

The `runner` module executes the `sample` function in the configuration file for each parameter configuration specified in `config` *REPS* times, where *REPS* is the amount of times to execute each parameter configuration for each benchmark. This module outputs the measurement (and program output if specified) data which `run` then parses.

As briefly mentioned previously, the `helper` module executes benchmarks based on the path and execute command given in the benchmarks' configuration files. It does this through the Open3 module included in Ruby, the exact function used is `popen3` which given a shell command executes the command using the child thread and blocks while awaiting the result of the command. This function allows a capture of the output streams stdin, stdout and stderr (GNU `time`, which is used for measurements in the framework, outputs to stderr). After executing a benchmark, the output of `time` as well as the program output (captured from stdout) are returned and later output by the `runner` module, the same output which is later parsed and processed by the `run` module.

The `dependencyChecker` module contain logic for checking for missing dependencies and whether the user runs a operating system supported by the benchmarking framework, based on the stated dependencies in the `config` module. At the moment the framework software only works on Linux systems due to `GNU time` not being available on other Unix-based systems such as OSX by default. Apart from the default measurement tool used, the framework is fully compatible with UNIX-based systems by default. However, to run the framework on systems such as OSX, one can simply download GNU Time (`gtime`) and set the path accordingly in the `config` module.