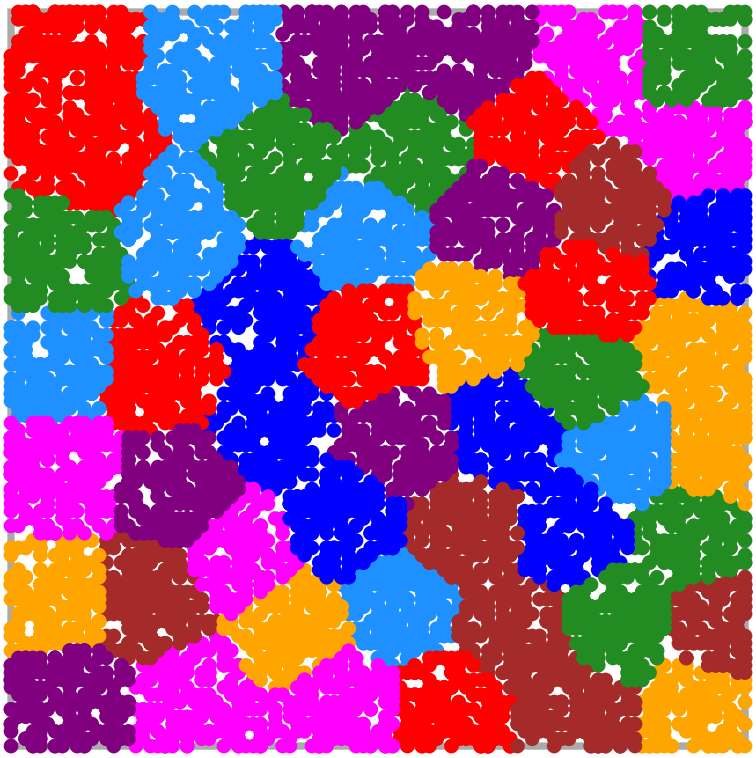

Comparing Functional Programming Languages for Parallel Applications

Haskell, Erlang, Scala and Manticore on 12 cores in two examples of data-parallel problems

Master's thesis in Computer Science and Engineering

Simon Alling

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Comparing Functional Programming Languages for Parallel Applications

Haskell, Erlang, Scala and Manticore on 12 cores in two examples of data-parallel problems

Simon Alling

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Comparing Functional Programming Languages for Parallel Applications
Haskell, Erlang, Scala and Manticore on 12 cores in two examples of data-parallel problems
Simon Alling

Supervisor: John Hughes, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visual representation of k-means clustering with $k = 50$.

Typeset in LaTeX
Gothenburg, Sweden 2019

Comparing Functional Programming Languages for Parallel Applications
Haskell, Erlang, Scala and Manticore on 12 cores in two examples of data-parallel problems
Simon Alling
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

I compare Haskell, Erlang, Scala and Manticore in the context of parallel functional programming using two data-parallel problems: mapping a function over a list, and k-means clustering. I give an overview of the syntax and parallel idioms of each language, observe both expected and unexpected performance figures, and finally analyze each language from a programmer's perspective.

I conclude that functional languages can be quite fast, not very far behind C; that the studied languages scale well in general, especially Erlang; that Manticore is not ready for production use; and that none of the three other languages is a clear overall winner.

Keywords: Parallelism, functional programming, benchmarking, performance, Haskell, Scala, Erlang, Manticore, k-means

Acknowledgements

I would like to express my gratitude to everyone who has supported and helped me throughout this project.

To Professor John Hughes, my supervisor, for the initial idea, as well as suggestions, advice and interesting discussions over the past 20 weeks.

To Professor Carl-Johan Seger, for providing a suitable machine for benchmarking and helping me when I had problems using it.

To Elias, for being my opponent.

To my friends from the Computer Science Division and elsewhere at Chalmers, for practical advice, inspiration and emotional support. Special thanks to Andreas for helping me cross the finish line at the end.

To Anna-Lena, for providing formidable quantities of delicious food for the last frantic weeks, and for helping me back on my feet when I was down for the count.

To Anna, for believing and making me believe.

Simon Alling, Gothenburg, June 2019

Contents

1 Introduction

Functional programming has been claimed to be very well suited for parallelism [1]. Take this Haskell example:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)

mapFib :: [Int] -> [Int]
mapFib xs = map fib xs
```

One way to parallelize this program would be as follows:

```
mapFib :: [Int] -> [Int]
mapFib xs = map fib xs `using` parList rseq
```

With this simple modification, my experiments show a speedup of a factor of 10 when `mapFib (replicate 1000 30)` is evaluated on 12 threads on a 12-core machine.

Now, one might ask: How well does this work in practice? Can more complex programs also be easily parallelized? How do different functional languages compare with respect to performance and parallel idioms? How can one even perform such a comparison? In this report, I try to give insight into those questions:

- I describe two parallelizable functional programs — a trivial example of mapping a function over a list and a more realistic k-means clustering example — and benchmark parallel implementations of these programs in the functional languages Haskell, Erlang, Scala and (partially) Manticore.
- I show that, depending on the language used, relatively small changes to a functional program can result in large parallel speedups.
- I discuss the different languages from a programmer's perspective.
- I show that conducting this kind of research can be more difficult than it sounds, and identify some of the obstacles to be avoided.

1.1 Background

From the conception of microprocessors in the 1960s and for the remaining decades of the twentieth century, single-threaded CPU performance increased at a rate of approximately a factor of two every 18–24 months [2]. This trend largely came

to an end in the first decade of the twenty-first century, due to excessive power consumption, heat development and the difficulty of improving already very complex processor architectures. ARM-based architectures such as the one used in Apple's A-series are demonstrating potential to outgrow the aged x86 [3], but the days of doubled performance every other year are long gone.

As a consequence, programmers have moved towards multi-core computing for performance-intensive applications. However, going parallel is not easy in general. Amdahl's law [4] puts a theoretical cap on parallel speedup, based on the proportion of a program that can be parallelized. And when there is some substantial speedup to be gained, programmers generally need solid knowledge of how to write parallel code, and how to make it produce the correct result deterministically.

The difficulty of writing parallel code can be affected by the choice of programming paradigm and language. For example, purely functional languages typically simplify the construction of correct parallel programs, because they have no side effects, such as mutable state. However, they do not give the programmer as explicit control as an imperative language does (for example over order of execution or memory management), which can make performance harder to predict.

1.1.1 Related work

The primary inspiration of this thesis is the 2003 paper *Comparing parallel functional languages: Programming and performance* by Loidl et al. [5], which compares "PMLS, a system for implicitly parallel execution of ML programs; GPH, a mainly implicit parallel extension of Haskell; and Eden, a more explicit parallel extension of Haskell designed for both distributed and parallel execution". As far as I know, out of those three, GPH (Glasgow Parallel Haskell) is the only one in wide use today; it has since been integrated into the Glasgow Haskell Compiler (GHC), the *de facto* reference Haskell implementation.

It should be noted that in 2003, there did not exist even dual-core mainstream processors. Since the launch of AMD Athlon X2 in 2005, multi-core processors have grown to become ubiquitous in everything from smartphones up to data centers; I think this fact in itself is a good reason to conduct another similar study. Loidl et al. ran their benchmarks on a 32-node Beowulf cluster (consisting of regular workstation computers) connected by a 100 Mbit/s ethernet switch with a latency of 142 μs [5]; communication overhead should be smaller in a system with multiple cores on the same chip.

In the 2009 paper *Comparing and Optimising Parallel Haskell Implementations for Multicore Machines*, Berthold et al. compare GPH and Eden, and improve GPH performance by up to 30 % on 8 cores by applying "a number of optimisations [...], for example [...] a work-stealing approach to task allocation" [6].

Trinder et al. describe *strategies*, a technique that can be used to parallelize Haskell code [1]. It is, to my knowledge, the most used technique for writing parallel Haskell at the time of writing this report.

Marlow's book *Parallel and Concurrent Programming in Haskell* is a very comprehensive guide on not only parallel but also concurrent programming in Haskell.

2

Languages

In this report, I compare some aspects of parallelism, namely performance, scaling and programming experience, in four different high-level functional programming languages. I have chosen such languages because I am interested in the functional style of programming, and because I want to focus on parallelism without having to deal with segmentation faults and memory leaks.

These languages serve as the subjects of this study:

- Haskell [7] – perhaps *the* purely functional language of the last decades, characterized by a powerful type system and lazy evaluation. Its purity suggests good potential for parallelism; this, together with its popularity, makes it an obvious choice for this study. It is also my primary language of choice.
- Erlang [8] – an impure functional language with strong focus on robustness and distributed systems. It has seen great success in the field of telecommunications, for which it was originally designed [9].
- Scala [10] – a functional and object-oriented language that puts great emphasis on being multi-paradigm. It runs on the Java Virtual Machine (JVM) and interoperates with Java. Scala promotes itself as a language with good support for parallelism, which makes it interesting, not least as a representative of somewhat more mainstream languages.
- Manticore [11] – an extension of Standard ML, developed at the University of Chicago. It is specifically designed for both parallelism and concurrency, making it an interesting candidate for this study.

The report is limited to multi-core CPU programming; it does not consider graphics processing units (GPUs) or distributed clusters.

2.1 Haskell

Haskell is a purely functional, strongly statically typed, non-strict language. Being purely functional, it relieves the programmer of having to think about synchronization, race conditions, non-determinism and other issues traditionally associated with parallel programming. Thus, introducing parallelism into a Haskell program should not affect the likelihood of it producing the correct result.

However, achieving good parallel scaling is not trivial in general. Blindly throwing parallelism into a program often results in *worse* performance compared to the sequential version. It is up to the programmer to identify potential for parallelism and exploit it properly.

2.1.1 Evaluation

Thanks to Haskell's non-strictness, programs like this one are perfectly valid — and even ubiquitous:

```
ones :: [Int]
ones = 1 : ones

main :: IO ()
main = print (take 10 ones)
```

In a strict language, this program would never terminate, because `ones` would mean "`1` followed by (`1` followed by (`1` followed by ...))" and so on. But in a non-strict language, `ones` means simply "`1` followed by `ones`" — no further evaluation is performed unless actually needed.

The Haskell Language Report specifies only non-strictness [7], but in practice, Haskell implementations use a special case of non-strictness known as *lazy evaluation*. The difference is somewhat subtle: Whereas non-strictness means that expressions are not evaluated unless their values are needed, laziness adds the benefit that bound (named) expressions are evaluated *at most once*. That is, when a bound expression is needed, it is only evaluated once, no matter how many times it is used. Consider these definitions:

```
zero_slow = ack 3 10 - ack 3 10

zero_fast = let y = ack 3 10 in y - y

-- The Ackermann function:
ack m n
  | m == 0           = n + 1
  | m > 0 && n == 0 = ack (m - 1) 1
  | m > 0 && n > 0   = ack (m - 1) (ack m (n - 1))
```

The evaluation of `zero_slow` requires that the computationally intensive expression `ack 3 10` be evaluated twice, whereas `zero_fast` only requires one such evaluation, the result of which is then reused. Thus, `zero_slow` takes about twice as long to evaluate as `zero_fast`.

2.1.2 Parallelism

The most basic way to express parallelism in Haskell is demonstrated in this example:

```
main = print result
  where
    a = fib 35
    b = fib 36
    result = a `par` b `pseq` a + b

fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

The `result` expression (which can be equivalently written as `a `par` (b `pseq` (a + b))`[1]) essentially says "start evaluating `a`, preferably on another thread, then first evaluate `b` and, after that, evaluate `a + b`". By "preferably", I mean that from there on, it is up to the Haskell runtime system to schedule evaluation on different threads. The programmer can only express that parallel evaluation might be suitable.

Neither `par` nor `pseq` is actually primitive to Haskell; they are defined as follows (in the Glasgow Haskell Compiler, GHC):

```
pseq :: a -> b -> b
pseq x y = x `seq` lazy y

par :: a -> b -> b
par x y = case (par# x) of { _ -> lazy y }
```

However, `seq :: a -> b -> b` is a special function, specified to be strict in both its arguments. That is, `seq a b` forces the evaluation of `a`, even if `a` is not used at all anywhere else in the program. Furthermore, the identity function `lazy :: a -> a` is treated specially by GHC's strictness analyzer. Together, they make parallel evaluation possible.

In practice, a Haskell programmer seldom has to use `par` and `pseq` explicitly, but can instead use less verbose abstractions, for example

- *strategies* from the `Control.Parallel` module in the `parallel` package [12],
- the `Par` monad from the `monad-par` package [13],
- the Repa library [14], and
- the Data Parallel Haskell extension [15].

This report focuses on the strategies approach. A strategy is "a function that embodies a parallel evaluation strategy" [12]:

```
type Strategy a = a -> Eval a
```

where `Eval` is a strict identity monad. Being an identity monad, it does not encapsulate any side effects — it only controls evaluation order. Some basic predefined strategies are

- `r0 :: Strategy a`, which performs no evaluation at all;
- `rseq :: Strategy a`, which performs evaluation to *weak head normal form* (WHNF)[2];

[1] `a `f` b` is syntactic sugar for `f a b`.

- `rdeepseq :: NFData a => Strategy a`, which performs evaluation to *normal form* (for types that support evaluation to normal form, including `Integer` and `[Maybe Double]`, but not function types); and
- `rpar :: Strategy a`, which sparks its argument for parallel evaluation.

One key strength of strategies is that they are composable. The `parallel` package exposes for example the higher-order strategy

```
parList :: Strategy a -> Strategy [a]
```

which, given a strategy for `a`'s, returns a strategy for evaluating a list of `a`'s in parallel, using the given strategy for each element. `parList` is in turn defined in terms of more basic strategies, including `rpar`.

Strategies can be applied using the function `using`; for example, `xs `using` parList rseq` has the same meaning as `xs`, but expresses that each element in `xs` should be evaluated to WHNF (`rseq`) in parallel (`parList`).

2.2 Erlang

Erlang is a functional, dynamically typed, strict language for Bogdan's Erlang Abstract Machine (BEAM), a virtual machine that runs in a single OS process, with a maximum of one OS thread per logical core. It puts great effort into being extremely reliable, especially in a distributed setting, having been credited with an infamous *nine nines* (99.9999999 %) uptime of Ericsson's AXD301 ATM switching system [9].

2.2.1 Syntax

Erlang's syntax differs somewhat compared to other popular languages, both imperative and functional ones. Whitespace has no significance, and commas, semicolons and periods have to be used as separators and terminators for different syntactic entities. The advantage of such a syntax is that it is easy to parse.

Variables cannot start with a lowercase letter, because such an identifier is interpreted as an *atom* instead. An atom is "a literal, a constant with name" [16]; one could view the "atom type" as a virtually infinite sum type. In practice, atoms semantically resemble string literals in other languages. Consider this example:

```
f(x) -> x.
```

`f` above is not the identity function; it is a function defined only for the atom `x`, in which case it returns that atom. Conceptually, it is very much like this Haskell definition:

```
f :: String -> String
f "x" = "x"
```

Or this Java definition:

[2]Roughly, an expression is in WHNF if it is either a data constructor, possibly with arguments (e.g. `True` or `Just (head ones)`), or a function.

```
public static String f(String s) {
  if ("x".equals(s)) return s;
  else throw new Error();
}
```

2.2.2 Multiple function arities

Erlang considers functions with the same name different if they do not take the same number of arguments:

```
-module(community).
-export([join/1, join/2]).

join(Age) -> join(Age, 18).

join(Age, RequiredAge) ->
  if
    Age < RequiredAge -> "Sorry!";
    true              -> "Welcome!"
  end.
```

In this case, `join/1` and `join/2` are used to identify the two functions (as seen in the export declaration at the top).

2.2.3 Immutability and impurity

While Erlang is not purely functional like Haskell, it does feature some aspects of immutability, including variables being read-only. However, due to how pattern matching works in Erlang, the mistake of trying to reassign a variable is not necessarily caught at compile time. Take for example this program:

```
-module(buggy).
-export([f/1]).

f(X) ->
  Y = X + X,
  Y = Y * Y,
  Y.
```

This is what happens if it is compiled and used in `erl`, Erlang's REPL:

```
$ erl
1> c(buggy).
{ok,buggy}
2> buggy:f(5).
** exception error: no match of right hand side value 100
     in function  buggy:f/1 (buggy.erl, line 6)
```

However, `f(0)` and `f(0.5)` are evaluated successfully (to `0` and `1.0`, respectively), because in those cases, the left and right hand sides in `Y = Y * Y` do match.

2.2.4 Parallelism

Erlang allows and encourages the programmer to create a very large number of lightweight internal *processes* (not to be confused with OS processes). These can then be distributed over multiple cores/threads or even geographically separated nodes, and share no data with each other. This means that an Erlang process crashing is not considered a very serious problem; programs are usually written to detect the crash of a process and take appropriate action, for example restarting the crashed process. Since no data is shared, no other process is directly affected.

Parallelism is explicit in Erlang in that the programmer explicitly spawns processes and passes messages between them. Here is a basic example:

```
-module(processes).
-export([main/0]).

main() ->
  Me = self(),
  Worker = spawn(fun() -> workFor(Me) end),
  waitFor(Worker).

waitFor(Worker) ->
  receive
    { done, Worker, Answer } ->
      println("Answer delivered."),
      Answer;
    _ ->
      println("Unrecognized message!"),
      waitFor(Worker)
  end.

workFor(Boss) ->
    timer:sleep(500),
    Boss ! { done, self(), 42 }.

println(L) ->
  io:fwrite(L ++ "\n").
```

The built-in function `self` returns the *process ID* (PID) of the current Erlang process. `spawn`, given a function, spawns a process that evaluates a call to that function. In the rather artificial example above, the spawned process just performs some dummy work and then sends a message to its "boss" process with the result. (`Pid ! Msg` means "send the message `Msg` to the process with process ID `Pid`".)

Any messages received by a process are queued up in the *mailbox* of that process. In `waitFor`, incoming messages are matched against two possible patterns. If the message is a triple whose first element is the atom `done` and whose second element is the PID of the spawned worker process, then the third element, `Answer`, is returned. Otherwise, `waitFor` is called recursively and checks the mailbox again. This means that any messages from another process than the spawned worker will be ignored.

Note that the use of the variable `Worker` in the first `receive` pattern (`{ done,`

`Worker, Answer }`) is *not* a shadowing — it is a pattern match against the value of the `Worker` parameter already in scope. None of the other languages in this report has such semantics: Haskell, Scala and Manticore can pattern match only against data constructors, and would bind a new (shadowing) variable in corresponding scenarios. I do not know of any other language resembling Erlang in this aspect.

A similar technique to the one above is used to introduce parallelism in the real example programs described in section 3.3.

2.3 Scala

Scala is a functional, statically typed, strict language for the Java Virtual Machine, intended to interoperate with Java. It is not *purely* functional, and not as strongly typed as Haskell and Erlang; for example, `"foo"+5` is a valid Scala expression with the value `"foo5"`. Nevertheless, Scala does position itself as a language well suited for functional, type-driven programming:

- It is less statement-oriented and more expression-oriented than Java and similar languages. For example, blocks are expressions, and the value of a block is the value of its last expression:

  ```
  def f(x: Int): Int = {
    val square = x*x
    val sum = x+x
    square - sum
  }
  ```

  In this (somewhat verbose, but demonstrative) example, `f(5)` evaluates to `15`.
- It encourages the use of immutable variables and is more explicit about mutability than many traditional languages. Mutable variables require the `var` keyword:

  ```
  val i = 0 // immutable
  var m = 0 // mutable
  ```

- Laziness can be explicitly introduced when needed:

  ```
  def f(x: Int): Int = {
    lazy val bottom = throw new Error()
    x
  }
  ```

  The code above works fine, with `f(5) == 5`. But without the `lazy` keyword, `f(5)` is $\bot$ (i.e. not defined). There are also lazy collections such as `Stream`, which is a lazy version of `List`.
- The traditional dot syntax for member access can be used in a functional manner:

  ```
  case class Person(name: String)
  val people = List(Person("John"), Person("Simon"))
  val names = people.map(_.name)
  ```

In this example, `names` evaluates to `List("John", "Simon")`.

- Algebraic data types and case expressions are ubiquitous:

```
def f(s: String): Option[Int] = { ... }
val y = f("foo") match {
  case Some(x) => x
  case None    => 0
}
```

2.3.1 Parallelism

This example demonstrates basic use of data parallelism in Scala:

```
def slowId(x: Int): Int = {
  Thread.sleep(1000)
  x
}
val xs = List.tabulate(10)(i => i)
val ys = xs.par.map(slowId)
```

This program runs in about 1 second on a machine with ten or more cores. If `.par` is removed so that the last line is instead `val ys = xs.map(slowId)`, the program becomes entirely sequential and takes around 10 seconds to finish.

The idea is that Scala takes care of implementing parallelism behind the scenes when the programmer writes code such as above. A parallel version can be obtained of any collection implementing the `Parallelizable` trait, using the `par` method.

Notably, if `c` is a parallelizable collection, `c.par` does not in general have the same type as `c`: For example, while `xs` above has type `List[Int]`, `xs.par` has type `ParSeq[Int]` — as does `ys`.

2.4 Manticore

Manticore is based on Standard ML, a strongly typed, strict, functional imperative language known for being the first to implement Hindley–Milner type inference and for its great influence on functional programming as a whole, including Haskell, Erlang and Scala.

The Manticore project extends Standard ML of New Jersey with parallelism and concurrency. The explicit intention is to provide a language that does both, not just one of them [11]. Some parts of Standard ML have been omitted for simplicity, including "mutable reference and array types".

Being an extension of Standard ML, Manticore is similar to Haskell in many aspects, with function application through juxtaposition (`f x`, rather than `f(x)`), `if ... then ... else`- and `let ... in`-expressions, and type inference. However, a complete program can be written much like a script, with no `main` function:

```
fun fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n - 2) + fib (n - 1)

val x = 20
val y = Int.toString (fib x)
val _ = Print.printLn ("fib " ^ Int.toString x ^ " is " ^ y)
```

2.4.1 Parallelism

Data parallelism in Manticore can be achieved using the included `PArray` module:

```
val w = 10
val d = 32
val xs = PArray.fromRope (Rope.tabulate (fn _ => d) (1, w))
val ys = PArray.map (fn x => fib x) xs
```

Note that the keyword `fun` is used for function definitions, whereas `fn` is used for lambda expressions.

3

Benchmarks

3.1 Example programs

To assess performance, scaling and programming experience, I use two computational problems: mapping a function over a list in parallel, and the k-means clustering algorithm. Both of these are described in detail below, including their language-agnostic definitions, implementation details in each language, and why they were chosen.

See Appendix A for details on how to run the benchmarks.

3.1.1 Parallel map

One of the most basic forms of data parallelism is mapping a function over a list, processing the items in parallel. This computational problem has no inherent data dependencies, so it should, in theory, scale perfectly over cores — it is said to be *embarrassingly parallel*.

One advantage of using an embarrassingly parallel problem is that it becomes easy to tell that "parallelism works", i.e. that the implementation is parallel at all, that the compiler is properly configured and so on. Another advantage is that each language gets a chance to show its potential in a best-case scenario. The main disadvantage is that most real-world problems are not embarrassingly parallel, so such a problem is not very representative for parallelism in general.

The parallel map benchmark is very simple. A function that does a lot of work is defined, and then mapped over a list of identical elements. My first choice of the function to map was a recursive factorial function:

$$
\begin{aligned}
&fac : \mathbb{N} \to \mathbb{N}_+ \\
&fac(0) = 1 \\
&fac(n) = n \cdot fac(n-1)
\end{aligned}
$$

However, for reasons outlined in subsection 5.1.6, it was replaced by a similar recursive function fib, which calculates the nth fibonacci number for a given n. The function is not memoized, so it is very inefficient. This is desired in this context, because it is intended to represent a heavy computation that could be mapped over a list.

This is the general, language-agnostic definition of fib:

$$
\begin{aligned}
&fib : \mathbb{N} \to \mathbb{N} \\
&fib(0) = 0 \\
&fib(1) = 1 \\
&fib(n) = fib(n-2) + fib(n-1)
\end{aligned}
$$

Then, I measure the time it takes to map fib over a list of identical elements, namely the number 30 repeated 1 000 times. From now on, I will refer to the length of the list as the *width* and the repeated number as the *depth*. (For the record, $fib(30)$ is 832 040.)

3.1.2 k-means clustering

k-means clustering is a less contrived, more realistic example than the parallel map problem described above. It is known to offer good potential for parallelism, without being embarrassingly parallel [17].

Example

Intuitively, k-means arranges a set of points into a given number of clusters.

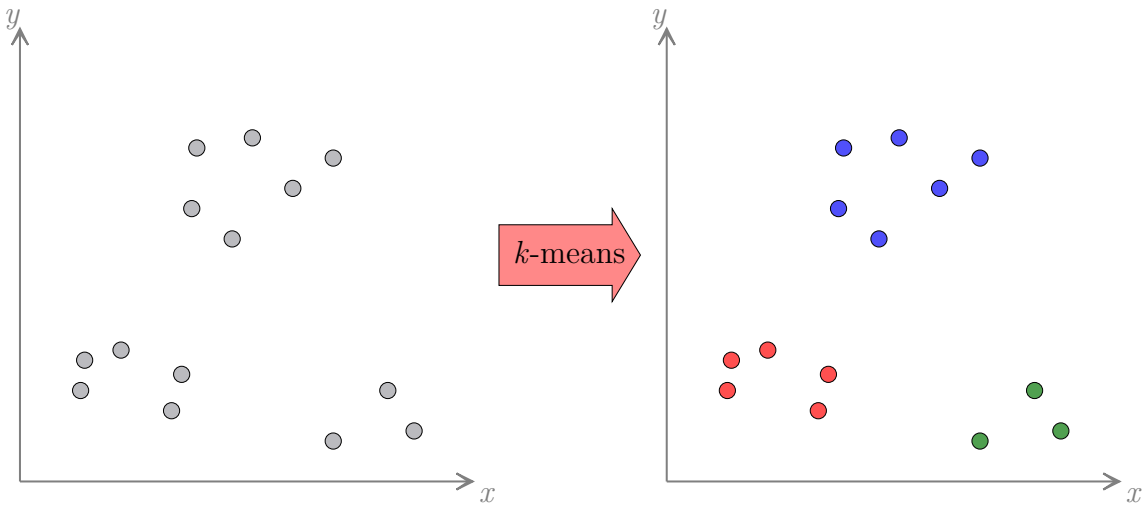

Figure 3.1: k-means example with $k = 3$.

Problem definition

For any positive integer d, given

- a set of data points, $P \subset \mathbb{R}^d$;
- a function $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$, which calculates the distance between any two points in $\mathbb{R}^d$; and
- a positive integer k,

arrange the points in P into k clusters such that no point is closer to the center of another cluster than to the center of its own cluster. The center of a cluster is usually referred to as a *centroid*, and is defined as the "arithmetic mean" of the points in the cluster. For example, the centroid of the cluster $\{(x_1, y_1, z_1), ..., (x_n, y_n, z_n)\}$ is

$$\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}, \frac{\sum_{i=1}^{n} z_i}{n} \right).$$

Algorithm description

k-means clustering works as follows:

1. Make sure that each point is assigned to some cluster and that each cluster has at least one point assigned to it.
2. For each cluster, calculate its centroid.
3. For each point, assign it to the cluster closest to it — i.e. the one with the closest centroid, based on the calculations in step 2.
4. Repeat steps 2–3 until no point is reassigned to another cluster anymore.

The algorithm is not guaranteed to converge. Hence, an implementation may allow the user to specify a maximum number of refinement iterations, after which it simply returns whatever clustering it has reached at that point. In a practical scenario, this is often good enough.

Input data

The test data used is a set of deterministically generated, uniformly distributed points in 3D space, all from the set $[0, 100]^3$ — a cube with its sides parallel to the cartesian planes and two of its corners at $(0, 0, 0)$ and $(100, 100, 100)$. k-means works with any number of dimensions, and there is no particular reason behind the choice of the described subset of 3D space; it was chosen arbitrarily.

The points are generated by the Haskell package `random-fu`. Only integer coordinates (e.g. $(42, 13, 37)$) are generated; they are then converted to double-precision floating-point before being passed to k-means. The benchmark can be run with different problem sizes by taking from the test data set the desired number of points. For the comparisons between languages, I use 20 000 points and $k = 200$. I also came to use only `3` as the random seed, having observed no noticeable difference between seeds.

Figures 3.2 and 3.3 show sample results of the algorithm. (3D points are used for the actual benchmarks, but 2D points are used here for illustrational purposes.)

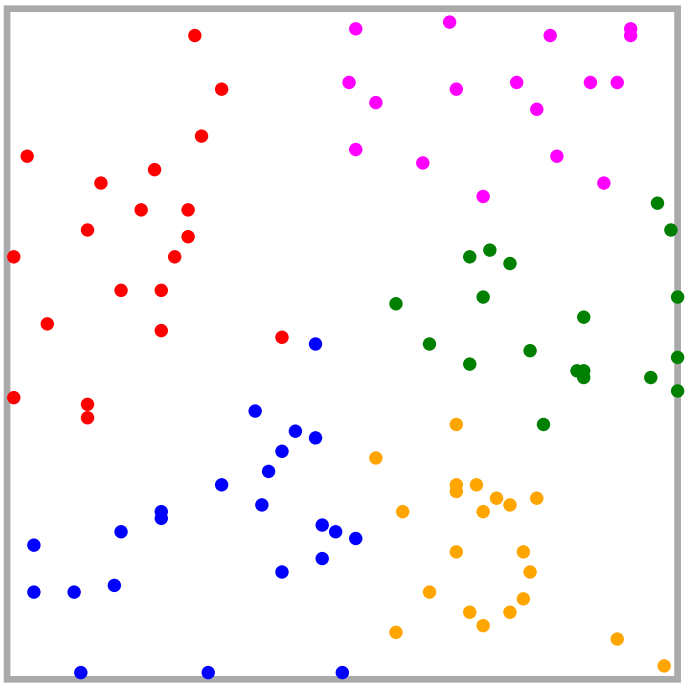

Figure 3.2: k-means clustering with 2D points, $n = 100$, $k = 5$.

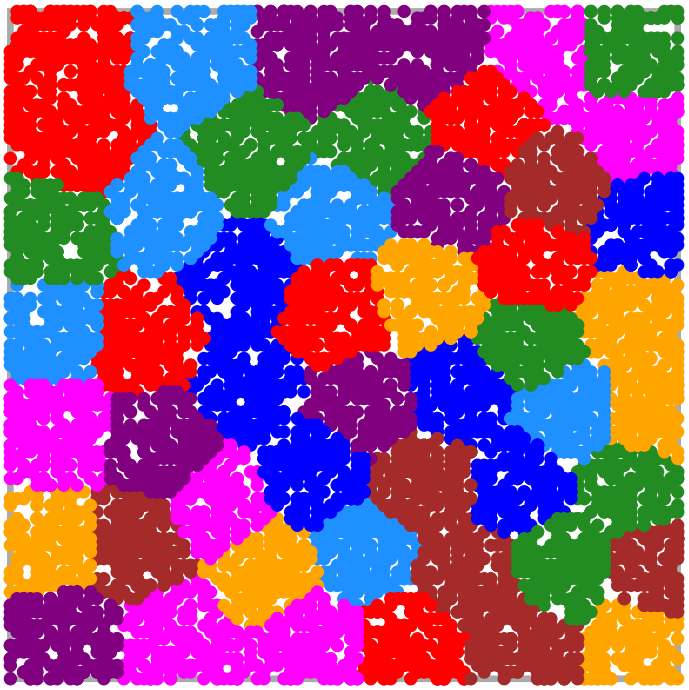

Figure 3.3: k-means clustering with 2D points, $n = 10\,000$, $k = 50$.

3.2 Haskell

I use the benchmarking library Criterion [18] to run benchmarks in Haskell. It is designed to try to automatically produce statistically reliable figures; I let it do this as it sees fit.

3.2.1 Parallel map

The program I use to benchmark the parallel map example in Haskell contains this definition of the *fib* function:

```
-- Boxed ("regular") ints:
fib :: Int -> Int
fib (I# n) = I# (fib' n)

-- Unboxed ints:
fib' :: Int# -> Int#
fib' 0# = 0#
fib' 1# = 1#
fib' n = fib' (n -# 2#) +# fib' (n -# 1#)
```

The difference between `fib :: Int -> Int` and `fib' :: Int# -> Int#` is that the latter explicitly uses *unboxed integers*. While the regular `Int` type represents a pointer to an integer on the heap, its unboxed counterpart `Int#` involves no heap allocation and no pointers — values of type `Int#` are always compiled to native machine integers. (A Haskell compiler may optimize regular `Int`s to native machine integers as well, but I did not know that when I settled on the program above.)

The actual mapping looks as follows:

```
fibonaccis_seq :: [Int] -> [Int]
fibonaccis_seq xs =
  map fib xs

fibonaccis_par :: [Int] -> [Int]
fibonaccis_par xs =
  map fib xs `using` parList rdeepseq

fibonaccis_parChunk :: Int -> [Int] -> [Int]
fibonaccis_parChunk chunkSize xs =
  map fib xs `using` parListChunk chunkSize rdeepseq
```

The only parallelism present in `fibonaccis_par` stems from the addition of `` `using` parList rdeepseq ``. It does not change the denotational semantics at all, but makes sure that the runtime system (RTS) creates a *spark* for each element in the resulting list, so that they can be evaluated in parallel. (There is no guarantee that they will be; it is up to the RTS to distribute tasks among the available threads as best it can.)

Notably, both `using` and `parList` are regular functions, defined in the `parallel` package. The first argument to `parList` must be a *strategy* that describes how each

element should be evaluated. If `s` is a strategy for evaluating values of type `a`, then `parList s` is in turn a strategy for evaluating lists of `a`'s. Finally, `e `using` s` means `e` evaluated with the strategy `s`.

`rdeepseq` means evaluation to normal form, from where no further evaluation is possible. The expression `5` is in normal form; `3 + 2` is not. Not all expressions can be evaluated to normal form, functions being a notable example.

`fibonaccis_parChunk` is a somewhat more elaborate alternative; instead of `parList`, it uses `parListChunk`. While the former sparks every single list element for parallel evaluation, the latter requires that a *chunk size* be specified. If the chunk size is n, then `parListChunk` splits the list up into chunks of n elements each, and sparks each of those chunks for parallel evaluation. The chunk size can be calculated at runtime so that the number of chunks matches the number of available threads (although this is not necessary):

```
main :: IO ()
main = do
  threads <- getNumCapabilities
  let chunkSize = length xs `div` threads
  print (fibonaccis_parChunk chunkSize xs)

xs :: [Int]
xs = replicate 1000 30
```

Note how the parallel versions use the exact same code as the sequential one, namely `map fib xs`. We could even have reused the name `fibonaccis_seq`. The reason why this works is Haskell's non-strict semantics: For all intents and purposes, `map fib xs` denotes only a *description* of a value that *could* be computed — not the actual computed value. Therefore, using `using`, we can describe that whenever it *is* computed, it should be done with the strategy of our choice, e.g. `parList rdeepseq`.

The separation of computation ("what") and coordination ("how") is a key strength of the strategies approach, according to its inventors [1].

3.2.2 k-means clustering

A modified version of the existing `kmeans-par` package from Hackage[1] is used for the k-means benchmarks in this report. It includes both an entirely sequential implementation and one that does assignment/classification in parallel. In fact, the code is very similar to Marlow's sequential and parallel implementations from his 2013 book on parallel programming in Haskell [19]. My modifications to the package are detailed in section 4.1, and the code in its entirety is published on GitHub [20].

The parallel ("Strategies") implementation re-uses almost all code from the sequential one. The only real changes are made in `step` and `computeClusters`, whose sequential versions look like this:

[1] Haskell's package repository.

```
step :: Metric a
     => (Vector Double -> a)
     -> Vector Cluster
     -> Vector Point
     -> Vector Cluster
step = makeNewClusters ...: assignPS

computeClusters :: Metric a
                => ExpectDivergent
                -> (Vector Double -> a)
                -> Vector Point
                -> Vector Cluster
                -> Vector Cluster
computeClusters (ExpectDivergent e) metric
  = computeClusters' e metric 0
```

`(...:)` (above) and `(..:)` (below), defined in the `kmeans-par` package, are not necessary; they only make the code less verbose. `(..:)` basically means function composition, but whereas regular function composition works only with unary functions, `(..:)` composes a unary and a binary function into a binary one. `(...:)` works analogously with ternary functions. For example:

```
absOfDifference :: Int -> Int -> Int
absOfDifference = abs ..: (-)
```

In the parallel implementation, `step` and `computeClusters` instead look like this:

```
(<><>) :: Semigroup a => Vector a -> Vector a -> Vector a
(<><>) = zipWith (<>)

step :: Metric a
     => (Vector Double -> a)
     -> Vector Cluster
     -> Vector (Vector Point)
     -> Vector Cluster
step = makeNewClusters . foldr1 (<><>) . with s ..: fmap ..: assignPS
  where s = parTraversable rseq

computeClusters :: Metric a
                => ExpectDivergent
                -> (Vector Double -> a)
                -> Partitions
                -> Vector Point
                -> Vector Cluster
                -> Vector Cluster
computeClusters (ExpectDivergent e) metric
  = computeClusters' e metric 0  ..: chunkInto . partitions
```

These changes have the effect that the set of points is split up into a number of smaller sets, or *partitions*, each of which can be processed in parallel. The parallelism

stems from `with s`, which means `` `using` s``. The processing in question is basically assignment of points to clusters. At the end, the results for each partition are combined by `foldr1 (<><>)` to obtain the new cluster centroids.

It is up to the user of the library to choose the number of partitions (possibly based on the number of available threads).

3.2.3 Compilation and runtime system

The example programs are compiled with the Glasgow Haskell Compiler (GHC), version 8.6.4, with these flags:

- `-threaded` – enable parallelism.
- `-rtsopts` – make it possible to specify certain parameters for the runtime system when running the compiled program.
- `-O2` – the most aggressive optimization level.
- `-feager-blackholing` – prevent threads from doing work which is already being done on another thread. Recommended for parallel code [21].

Haskell programs run with the help of "a non-trivial runtime system (RTS), which handles storage management, thread scheduling, profiling, and so on" [22]. If a program is compiled with the `-rtsopts` flag, several options for the RTS can be specified when running the program. For example, to run the program on 4 threads:

```
$ ./HelloWorld +RTS -N4 -RTS
```

These RTS options are used for the benchmarks:

- `-H1G` – set the "'suggested heap size' for the garbage collector" to 1 GB[2].
- `-A100M` – set "the allocation area size used by the garbage collector" to 100 MB[2].
- `-Nt` – run the program on t threads.

I picked the first two options based on my experience with parallel Haskell from a course in parallel functional programming.

3.3 Erlang

First of all, I would like to mention that I use a non-conventional coding style for my Erlang code. There are two reasons for that: First, with the conventional style, I often forget to add separators such as commas, resulting in syntax errors. Second, conventionally written Erlang programs are unstable under line addition and removal. Consider this example:

```
-module(example).

println(Line) -> io:fwrite(Line ++ "\n").

greetings() ->
  println("Hello!"),
  println("Goodbye!").
```

[2]Decimal prefixes; G = 10^9 etc.

Now, if we decide that `greetings` should only say `"Hello!"`, the intuitive action might be to remove the last line:

```
greetings() ->
  println("Hello!"),
```

This results in a syntax error:

```
1> c(example).
example.erl:7: syntax error before:
```

The same principle applies for semicolons as separators in `if`/`case`/`receive` expressions and so on.

To avoid these repeated syntax errors, and to simplify Git diffs, I write Erlang like this instead:

```
greetings() -> do
  , println("Hello!")
  , println("Goodbye!")
  .
```

While not as readable, this style makes it almost impossible to forget a comma, and it allows lines to be removed, added or swapped without problems. Note that `do` is just an atom, much like a string literal; one could just as well use any other atom, or even any pure expression.

3.3.1 Parallel map

The *fib* function and the sequential mapping of it are fairly straightforward:

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

fibonaccis_seq(Xs) -> lists:map(fun fib/1, Xs).
```

The parallel mapping is written by Jesper L. Andersen [23], and slightly adapted by me:

```
fibonaccis_par(Xs) -> pmap(fun fib/1, Xs).

pmap(F, Xs) -> do
  , Parent = self()
  , Timeout = 60000 % milliseconds
  , Processes =
      [
        spawn_monitor(fun() -> Parent ! {self(), F(X)} end)
      ||
        X <- Xs
      ]
  , collect(Processes, Timeout)
  .

collect([], _) -> [];
collect([{WorkerPid, MRef} | Rest], Timeout) -> do
  , receive dummy -> dummy
      ; {WorkerPid, Result} -> do
          , erlang:demonitor(MRef, [flush])
          , [{ok, Result} | collect(Rest, Timeout)]
      ; {'DOWN', MRef, process, WorkerPid, Reason} -> do
          , [{error, Reason} | collect(Rest, Timeout)]
    after Timeout -> do
      , exit(pmap_timeout)
    end
  .
```

In `pmap`, the PID of the current process is assigned to the `Parent` variable to be used later, in the anonymous nullary function, to send the result, `F(X)`, back to the process that called `pmap`. The anonymous function is called in a new process spawned using `spawn_monitor`. The latter returns a tuple containing a PID and a reference to a *process monitor* for the new process; these tuples are stored in a list named `Processes` and passed to `collect`.

`collect` consists only of a `receive` expression which looks for two different patterns in the mailbox: either a message from the worker with PID `WorkerPid`, containing a result, or a notification that said worker has crashed. Note that `collect` is called with `Processes` as its first argument and only looks for a message from a specific worker, `WorkerPid`, before it calls itself recursively; this is necessary to preserve the order of the list being mapped over.

3.3.2 k-means clustering

My supervisor helped me write a sequential and a parallel k-means implementation, where the latter is based on the typical Erlang principle of spawning worker threads and collecting their results. It has been omitted here due to its size, but can be seen in Appendix A. The sequential version looks like this:

```
kmeans_seq(0, _, Clusters) -> Clusters;
kmeans_seq(MaxIterations, Points, Clusters) -> do
  , NewClusters = recenter(Points, Clusters)
  , if dummy -> dummy
      ; Clusters == NewClusters -> Clusters
      ; true -> kmeans_seq(MaxIterations-1, Points, NewClusters)
    end
  .
```

In contrast to the Haskell and Scala implementations, this one does not return the actual clustering, only the centroids of the calculated clusters. Hence, it still has to perform the same amount of work.

3.3.3 Benchmarking

All results in this report are from Erlang/OTP 21.0.

Erlang comes with a `timer` module with a function `tc`, which can be used to measure the execution time of a function. The function defined below evaluates a nullary function a specified number of times and returns the median execution time:

```
measure(Repetitions, F) -> do
  , Sequence = lists:seq(1, Repetitions)
  , TimesAndResults = [ timer:tc(F) || _ <- Sequence ]
  , Times = [ Time || { Time, _ } <- TimesAndResults ]
  % Median (if Repetitions is odd):
  , lists:nth(1 + Repetitions div 2, lists:sort(Times))
  .
```

I chose to report the median, and not the mean, to minimize the impact of outliers. As far as I could tell, the individual execution times tended to group up tightly around the median in general.

`main.erl` exposes an interface for running the benchmarks with parameters provided on the command line:

```
$ erl +S 8:8 -noinput -run main fibBenchmark 1000 30 9
```

In this example, the parallel map benchmark is run on 8 threads with $width = 1\,000$ and $depth = 30$. It is repeated 9 times, after which the median execution time is printed.

To automatically run a benchmark with different number of threads, two simple scripts are used, one for the parallel map and one for k-means, as described in subsection A.3.4. The number of repetitions is 9 throughout all benchmarks.

3.4 Scala

I use `sbt` (Scala's included build tool) [24] and the benchmarking library Scalameter to run benchmarks. Scalameter is responsible for warmup, repeating tests, statistical analysis and so on.

3.4.1 Parallel map

The parallel map program in Scala is very similar to the Haskell version described above. The deliberately inefficient implementation of fib looks like this:

```
def fib(n: Int): Int =
  if (n == 0) 0 else if (n == 1) 1 else fib (n - 2) + fib (n - 1)
```

Then we have the mapping:

```
def fibonaccis_seq(xs: Seq[Int]) =
  xs.map(fib)

def fibonaccis_par(xs: ParSeq[Int]) =
  xs.map(fib)

def fibonaccis_parChunk(n: Int, xs: Vector[Int]) = {
  val chunks = chunkInto(n, xs)
  val chunks_par = chunks.par
  val pool = new ForkJoinPool(n)
  chunks_par.tasksupport = new ForkJoinTaskSupport(pool)
  val mappedChunks = chunks_par.map(_.map(fib))
  pool.shutdown()
  mappedChunks.flatten
}
```

The first thing to note is that the body is the same for both `fibonaccis_seq` and `fibonaccis_par`, but their types are different. The reason why `fibonaccis_par` is parallel is because its argument is a `ParSeq` collection; it accepts only parallel collections, whereas `fibonaccis_seq` only works with sequential collections.

So what is a parallel collection? It is a collection very much like a regular (sequential) one, except that it is implicitly parallel. That is, the backing library code takes care of parallel execution without further actions required from the programmer.

The easiest way to obtain a parallel collection is to convert a corresponding sequential collection:

```
val pList = List(1, 2, 3).par

val pVector = Vector(1, 2, 3).par
```

If more control over the parallel behavior is desired, the `tasksupport` property of the parallel collection can be modified:

```
pList.tasksupport = new ForkJoinTaskSupport(new ForkJoinPool(12))
```

In this case, parallel operations on `pList` will be able to use up to 12 threads.

Finally, we have `fibonaccis_parChunk`, which closely resembles the Haskell `parListChunk` implementation, except for two aspects:

- The Haskell implementation uses almost identical chunking code, but it is included in the `parallel` package and called automatically behind the scenes by the `parListChunk` strategy.

- In Scala, we explicitly create a thread pool and assign it to the parallel collection. In Haskell, the number of available threads is set for the program as a whole, not for a specific collection.

The actual benchmark code uses different numbers of threads for the `ForkJoinPool` to test the scaling of `fibonaccis_par`:

```
val width: Int = Config.getParam(Config.REGEX_FIB_WIDTH)
val depth: Int = Config.getParam(Config.REGEX_FIB_DEPTH)
val maxThreads: Int = Config.getParam(Config.REGEX_MAX_THREADS)

val xs = List.tabulate(width)(_ => depth)
val xs_par = xs.par
val xs_vec = xs.toVector

def withThreads(t: Int) =
  new ForkJoinTaskSupport(new ForkJoinPool(t))

val threads = Gen.range("threads")(2, maxThreads, 1)

measure method "fibonaccis_seq" in {
  using (unit) in (_ => fibonaccis_seq(xs))
}

measure method "fibonaccis_par" in {
  using (threads) in (t => {
    xs_par.tasksupport = withThreads(t)
    fibonaccis_par(xs_par)
  })
}

measure method "fibonaccis_parChunk" in {
  using (threads) in (t => fibonaccis_parChunk(t, xs_vec))
}
```

This benchmark suite runs `fibonaccis_par` and `fibonaccis_parChunk` on two threads, three threads and so on up to the specified maximum number of threads.

3.4.2 k-means clustering

The Scala k-means implementation is a direct translation of my modified `kmeans-par` Haskell package, with some minor adaptations. Hence, the parallel version is called "Strategies" even though evaluation strategies do not exist in Scala. The main difference between the sequential and the parallel version is, again, in the `step` and `computeClusters` functions. Their sequential versions follow:

```
def step(
  metric   : Metric[Point],
  clusters : Vector[Cluster],
  points   : Vector[Point],
): Vector[Cluster] =
  makeNewClusters(assignPS(metric, clusters, points))

def computeClusters(
  expectDivergent: ExpectDivergent,
  metric   : Metric[Point],
  points   : Vector[Point],
  clusters : Vector[Cluster],
): Vector[Cluster] =
  computeClusters_(expectDivergent, metric, 0, points, clusters)
```

(Note that `computeClusters_` with an underscore at the end is another function, corresponding to `computeClusters'` in Haskell; Scala identifiers cannot contain apostrophes.)

The parallel versions look like this:

```
def step(
  metric      : Metric[Point],
  clusters    : Vector[Cluster],
  pointChunks : Vector[Vector[Point]],
): Vector[Cluster] = {
  val pointChunks_par = pointChunks.par
  val assigned = pointChunks_par.map(assignPS(metric, clusters, _))
  makeNewClusters(
    assigned.tail.fold(assigned.head)(combineVectors)
  )
}

def computeClusters(
  expectDivergent : ExpectDivergent,
  metric          : Metric[Point],
  partitions      : Int,
  points          : Vector[Point],
  clusters        : Vector[Cluster],
): Vector[Cluster] = {
  val chunks = chunkInto(partitions, points)
  computeClusters_(expectDivergent, metric, 0, chunks, clusters)
}
```

The code in its entirety can be seen in Appendix A. The sequential version returns the exact same clustering as the Haskell reference implementation for the input data described in subsection 3.1.2. However, I have discovered a correctness issue related to parallelism; see section 6.3.

3.5 Manticore

To measure performance in Manticore, I run 9 repetitions, measure the time taken with the `time` command, and present the minimum, maximum and median.

3.5.1 Parallel map

This is the parallel map program in Manticore:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n - 2) + fib (n - 1)

fun fibonaccis_seq width depth =
  let
    val xs = Array.tabulate (width, (fn _ => depth))
  in
    Array.map (fn x => fib x) xs
  end

fun fibonaccis_par width depth =
  let
    val rope = Rope.tabulateSequential (fn _ => depth) (1, width)
    val xs = PArray.fromRope rope
  in
    PArray.map (fn x => fib x) xs
  end
```

(`Rope.tabulateSequential` and `PArray.fromRope` are used because I did not know how to create a parallel array; I searched in the included regression tests and found that method.)

One might wonder why we cannot just map `fib` over `xs`. The reason for this seemingly redundant eta-abstraction is described in detail in section 4.2. (The eta-abstraction is actually only necessary in the parallel case, but I went with it across the board for symmetry.)

The program in its entirety can take from the command line a width, a depth and a number indicating sequential or parallel execution, and evaluate the appropriate expression. For example:

```
$ pmlc -o fib fib.pml
$ ./fib 1000 30 1        # sequential
$ ./fib 1000 30 2        # parallel
```

This will evaluate `fibonaccis_seq 1000 30` and `fibonaccis_par 1000 30`.

Automated benchmarking is done by a script that runs 9 repetitions of each version, sorts the reported execution times and saves them to a file.

3.5.2 k-means clustering

k-means is not included due to lack of time and unfamiliarity with Manticore.

4

Discoveries

This chapter contains discoveries not directly related to my research question, but nevertheless interesting and related to parallel functional programming in general.

4.1 The `kmeans-par` Haskell package

When I started experimenting with the `kmeans-par` package from Hackage, I soon started seeing peculiar results, including "impossible" super-linear parallel speedups. Further investigation unveiled two important discoveries.

4.1.1 The `partitions` parameter

The k-means implementation in `kmeans-par` exposes a `partitions` parameter, described as follows in the documentation:

> This version of k-means takes an additional arguments – the number of partitions the set of points'll be divided into. This needn't equal the number of processors: [...]

However, having consistently observed optimal performance with `partitions` set to n/t for n points and t threads, I examined the source code and discovered that `partitions` is actually the number of points per partition, not the number of partitions. That is, if the algorithm is run on 1 000 points with `partitions` set to 4, the points are split into 250 small partitions for parallel classification. In most cases, this results in bad performance due to overhead related to sparking and scheduling parallel tasks.

Because I felt it was more intuitive to specify the number of partitions than the partition size, I decided to modify the implementation so that, in the aforementioned example, the points are instead split into 4 large partitions.

4.1.2 Time complexity

Having seen very confusing parallel scaling figures, such as a 7× speedup on 4 threads, I examined the `kmeans-par` code to find out what was going on. The reason turned out to be the innermost loop of the sequential implementation (also used in the parallel one). It has been slightly adapted for readability here:

```
assign :: Vector Cluster -> Vector Point -> Vector (Vector Point)
assign clusters points = V.create $ do
  newClusters <- MV.replicate (length clusters) empty
  points `forM_` \point -> do
    let index = identifier (closestCluster clusters point)
    newFriends <- MV.read newClusters index
    MV.write newClusters index (point `cons` newFriends)
  return newClusters
```

In summary:

1. Create a vector representing the "new" set of clusters, all of which are initialized to the empty cluster.
2. For each point in `points`:
   (a) Find the cluster closest to it among the "old" clusters.
   (b) Add it to the set of points already selected for assignment to that cluster.

The problem is that `cons`, from `Data.Vector`, is linear in the size of its second argument. In this case, the second argument is `newFriends`, whose size will often be around n/k, where n is the length of `points`. So each of the n loop iterations performs an $\mathcal{O}(n/k)$ operation, resulting in an overall complexity of $\mathcal{O}(n^2/k)$ for the loop.

The remedy is simple but powerful:

```
assign :: Vector Cluster -> Vector Point -> Vector (Vector Point)
assign clusters points = V.map V.fromList $ V.create $ do
  newClusters <- MV.replicate (length clusters) []
  points `forM_` \point -> do
    let index = identifier (closestCluster clusters point)
    newFriends <- MV.read newClusters index
    MV.write newClusters index (point : newFriends)
  return newClusters
```

The use of regular cons lists instead of vectors means that the cons operation is $\mathcal{O}(1)$ instead of $\mathcal{O}(n/k)$, making the overall complexity of the loop $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2/k)$.

Figures 4.1 through 4.3 show the time taken by the first 10 refinement iterations (i.e. recursive calls of `computeClusters'`) of the sequential implementation before and after my modification, for $k \in \{2, 10, 100\}$.

Notably, the difference is much less dramatic for larger values of k, since the length of `newFriends` is generally around n/k.

Also notably, and as can be seen in Figures 4.4 and 4.5, the parallel version is not as heavily impacted by the loop complexity, because it does not run the loop on the entire set of points — only on smaller subsets. This explains the "impossible" speedups observed before my modifications.

`kmeans-par` before and after my modification

(sequential version, first 10 refinement iterations)

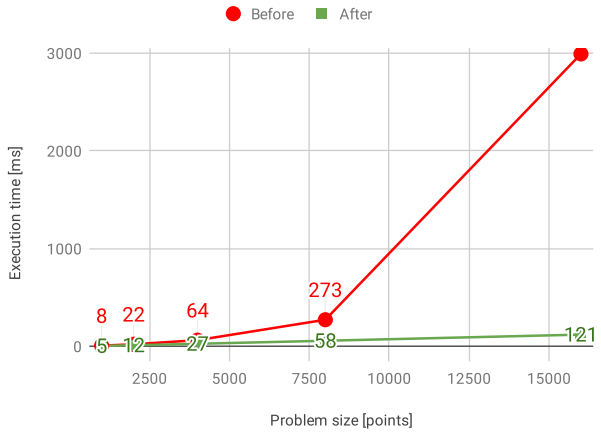

Figure 4.1: $k = 2$.

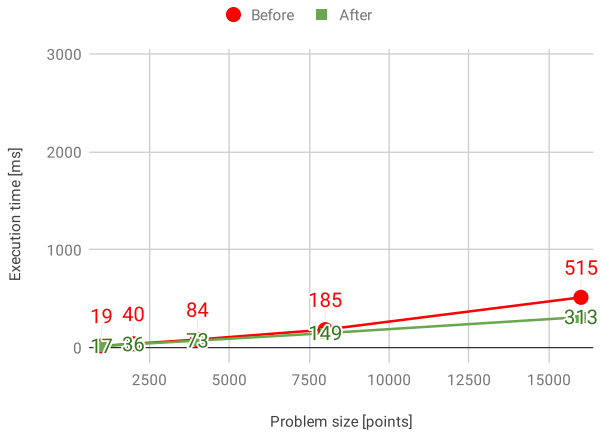

Figure 4.2: $k = 10$.

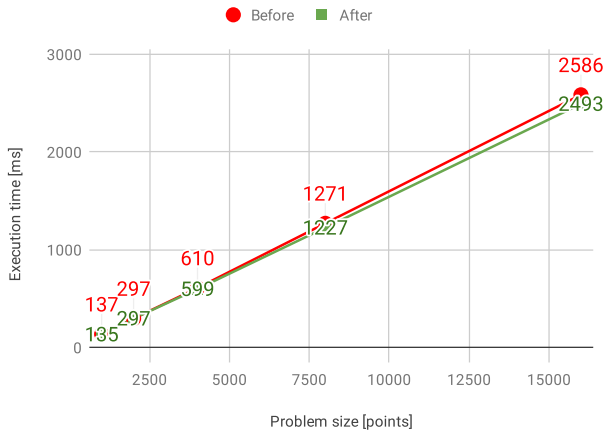

Figure 4.3: $k = 100$.

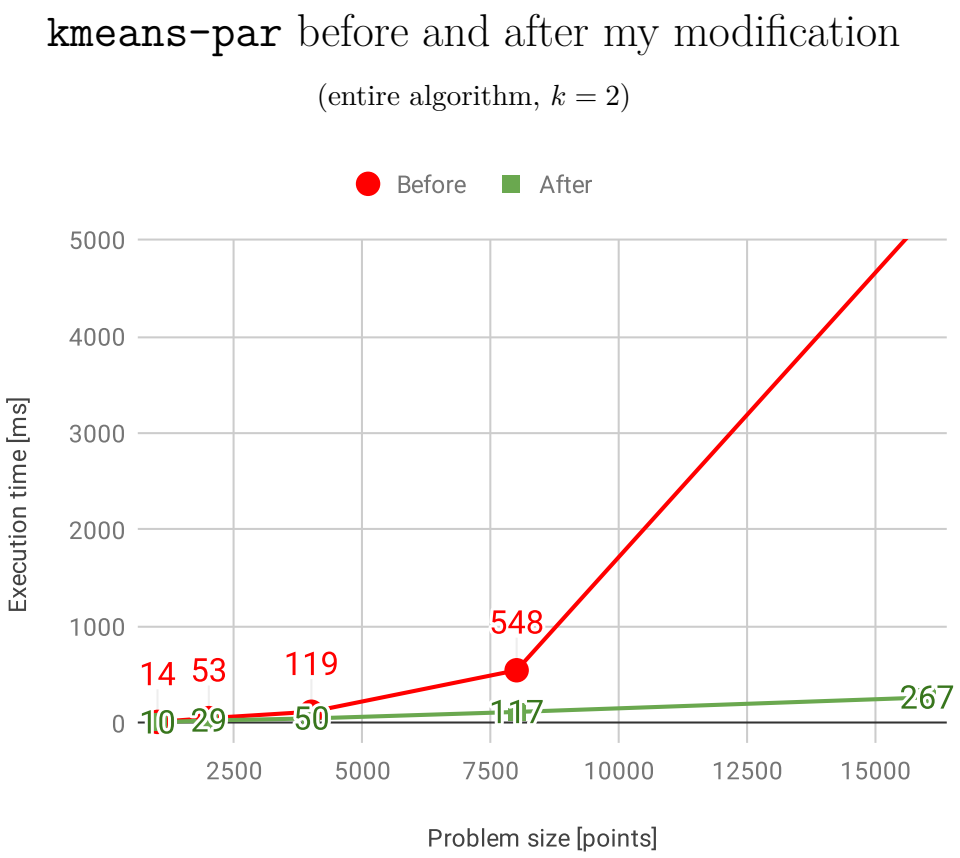

Figure 4.4: Sequential version.

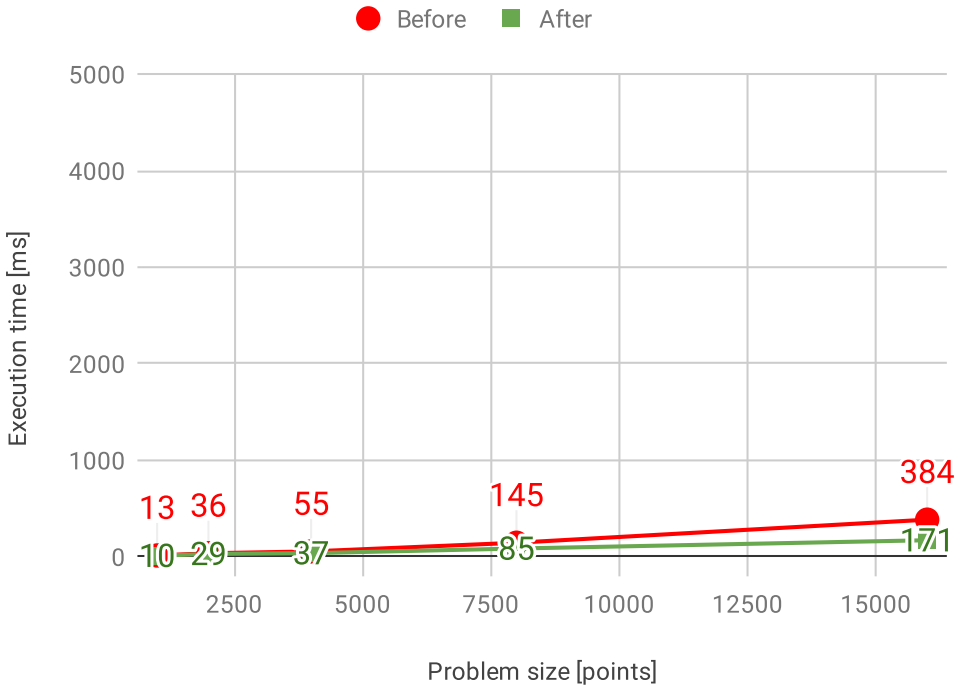

Figure 4.5: Parallel version, 20 threads, 20 partitions.

4.2 The Manticore compiler

During the initial phase of the project, when I was exploring different languages, I discovered an intriguing phenomenon in the Manticore compiler: A recursive function cannot be mapped over a parallel array, unless it is eta-abstracted. Non-recursive functions work fine, as does mapping a recursive function over a regular list.

```
(* Define one non-recursive and one recursive function: *)
fun f x = x + 1
fun g n = if n <= 0 then 0 else 1 + g (n - 1)

(* Define a regular list and a parallel array: *)
val xs = [ 1, 2, 3 ]
val pxs = PArray.fromRope (Rope.tabulateSequential id (1, 3))

(* Map the functions over the list: *)
val _ = map f xs                        (* OK *)
val _ = map g xs                        (* OK *)

(* Map the functions over the parallel array: *)
val _ = PArray.map f pxs                (* OK *)
val _ = PArray.map g pxs                (* Does not compile! *)

(* But with eta-abstraction: *)
val _ = PArray.map (fn x => g x) pxs    (* OK *)
```

When faced with the code above, the Manticore compiler crashes with this error message:

```
***** Bogus CPS in Main after inline *****
** unbound variable g<11431>#4.4
** unbound variable g<11431>#4.4 in Apply
broken CPS dumped to broken-CPS
```

I have submitted an issue to the Manticore repository [25], and according to Kavon Farvardin of the Manticore project, the problem seems to be caused by "some bug in the expansive inliner that runs on the CPS IR[1]."

[1] continuation-passing style internal representation

5
Results

This chapter presents the main results of the experiments. Focus is placed on performance and scaling over cores/threads; more subjective verdicts on programming experience follow in chapter 6.

These metrics are used for the scaling results:

- Execution time – the total time taken to run the program in question.
- Speedup – the sequential execution time divided by the execution time in question.
- Thread efficiency – the speedup divided by the number of threads. A thread efficiency of 1 means perfect scaling.

Execution time and speedup are used by Loidl et al. [5] as well as Marlow in his 2012 book [26]. I have not seen thread efficiency graphs being presented explicitly elsewhere, but Loidl et al. include in their speedup graphs a curve representing linear speedup, which essentially conveys the same information.

Some things that should be kept in mind when looking at the graphs:

- The test system processor has only 12 cores, but it can run up to 24 threads simultaneously.
- "1 thread" in the charts always means the sequential version, not the parallel version run on one thread.
- The speedup and thread efficiency charts have the same scale on the y-axis across all benchmarks and languages.

5.1 Parallel map

This section contains the results from the benchmark of mapping the fib function over a list containing 1 000 copies of the number 30 — a task completed in 2.95 seconds by a sequential C program on the same computer.

5.1.1 Haskell

The scaling over threads of both implementations (`parList` and `parListChunk`) can be seen in Figures 5.1 through 5.3. For the `parListChunk` benchmarks, the `partitions` parameter is always equal to the number of available threads.

Note that the `parList` version demonstrates the same trend as the `parListChunk` one, but delayed with one thread. In particular, it exhibits no improvement whatsoever from one to two threads, but then a speedup of 2 from two to three threads and so on.

5.1.2 Erlang

The Erlang results can be seen in Figures 5.4 through 5.6.

Note that Erlang displays almost perfectly linear scaling up until SMT (Hyper-Threading) takes over as the source of more threads.

5.1.3 Scala

The Scala results can be seen in Figures 5.7 through 5.9.

Note that there is no difference at all between 2 and 3 threads, as well as between 4 and 6 threads.

5.1.4 Manticore

Unfortunately, I found no way to limit the number of available threads for a Manticore program. (I tried different tricks with `taskset` to no avail.) Due to this, I cannot present a chart like the ones for the other languages.

To include some results at all, I decided to compare the entirely sequential version with the parallel one, which can use as many threads as it wants. Keep in mind that the benchmarking machine supports 24 simultaneous hardware threads on its 12 cores. The results can be seen in Table 5.1.

	Min	**Median**	**Max**	**Speedup**[1]	**Thread efficiency**[1]
Sequential	17.586 s	17.607 s	17.635 s	1.00	1.00
Parallel	1.084 s	1.466 s	1.576 s	12.01	0.50

Table 5.1: Mapping the fib function over 1000 copies of 30 in Manticore.

5.1.5 Comparison

Figures 5.10–5.12 show parallel map results from all languages except Manticore side by side. Both Haskell and Scala are represented by their chunk-based versions, because that version was fastest in both cases. Erlang's execution time is literally off the charts for both 1 and 2 threads; I deemed this necessary to make the chart readable at all.

Note that both Haskell and Erlang exhibit very regular scaling behavior up to the number of physical cores in the machine. Scala does not scale quite as well, but is fastest across the board.

[1] Based on median execution time only.

Haskell parallel map ($width = 1\,000$, $depth = 30$)

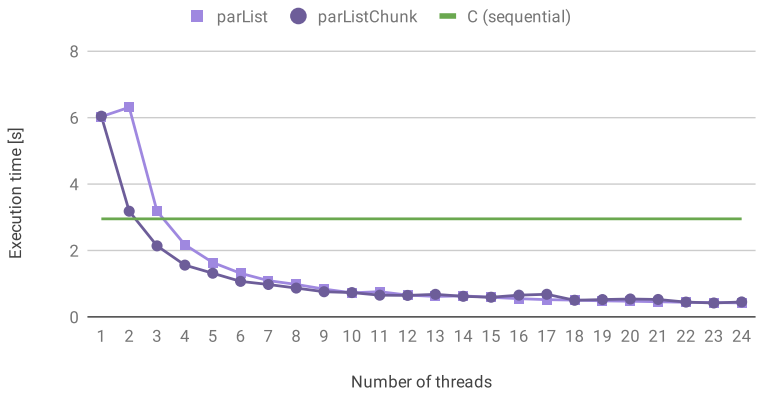

Figure 5.1: Execution time (lower is better).

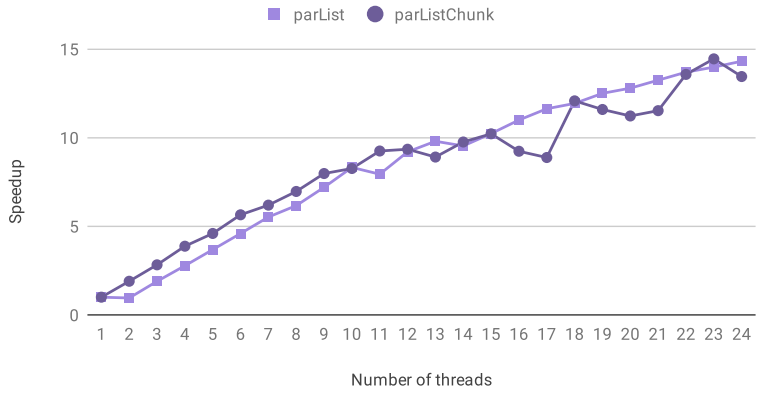

Figure 5.2: Speedup (higher is better).

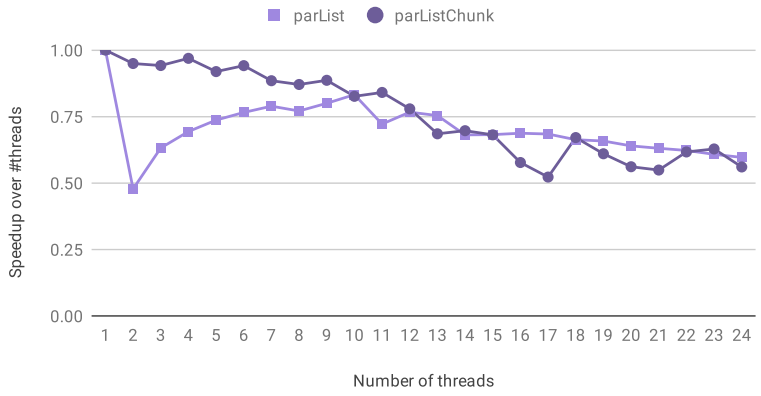

Figure 5.3: Thread efficiency (higher is better; 1 is perfect).

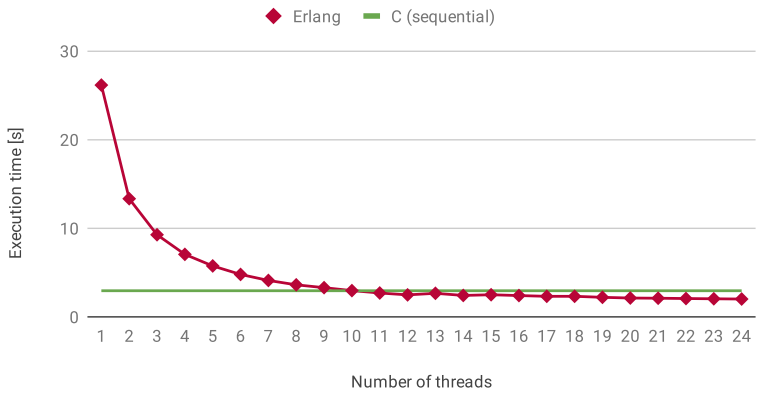

Figure 5.4: Execution time (lower is better).

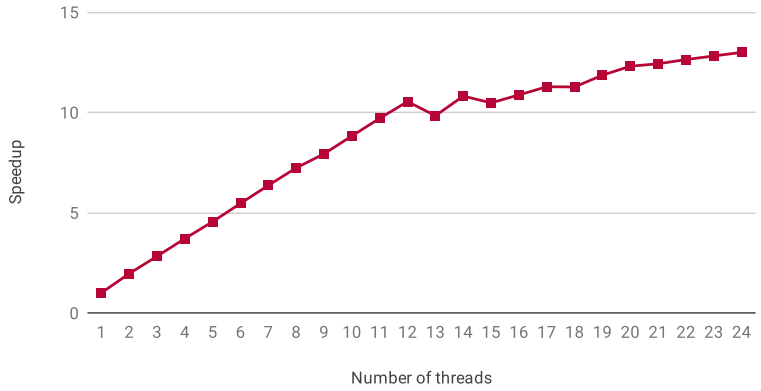

Figure 5.5: Speedup (higher is better).

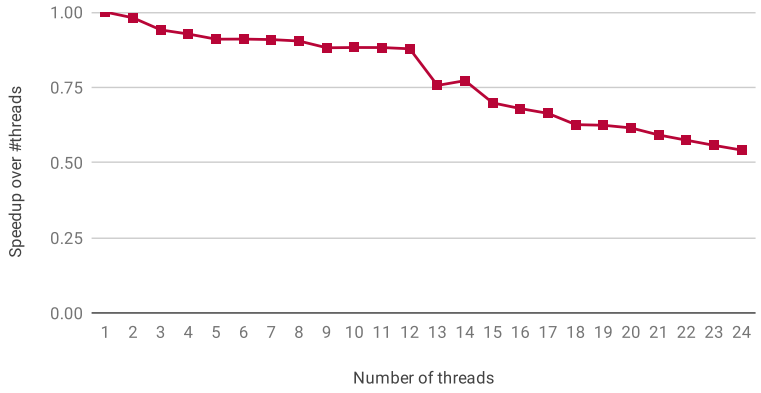

Figure 5.6: Thread efficiency (higher is better; 1 is perfect).

Scala parallel map ($width = 1\,000$, $depth = 30$)

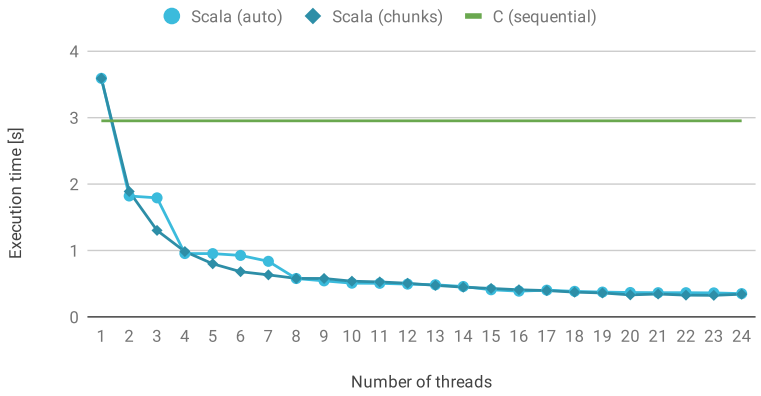

Figure 5.7: Execution time (lower is better).

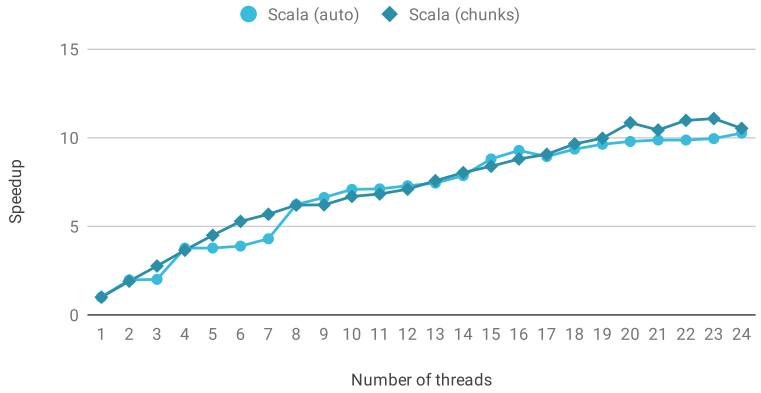

Figure 5.8: Speedup (higher is better).

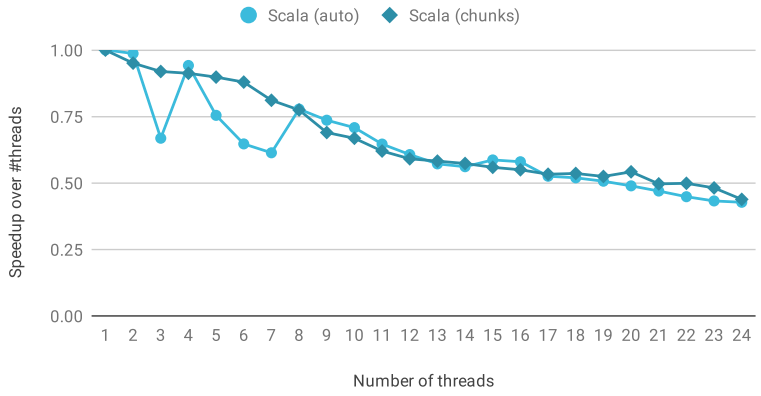

Figure 5.9: Thread efficiency (higher is better; 1 is perfect).

Parallel map comparison ($width = 1\,000$, $depth = 30$)

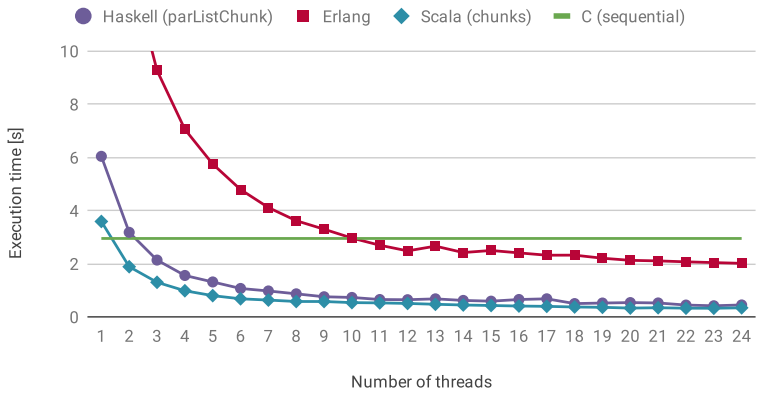

Figure 5.10: Execution time (lower is better).

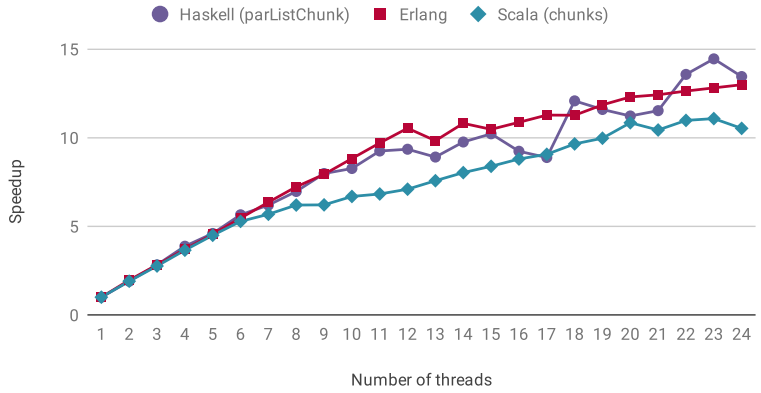

Figure 5.11: Speedup (higher is better).

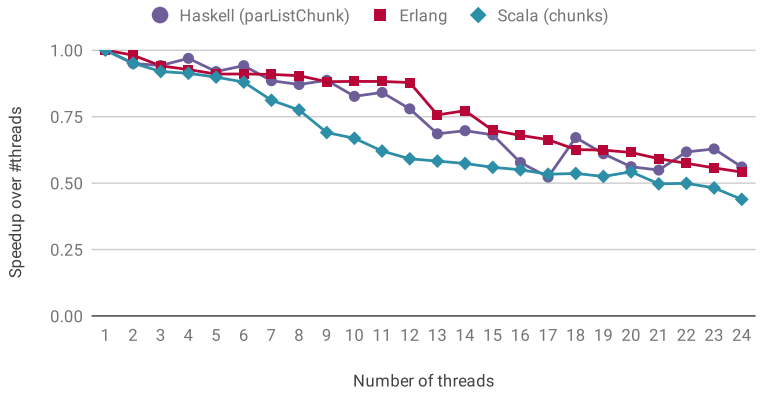

Figure 5.12: Thread efficiency (higher is better; 1 is perfect).

5.1.6 The factorial function

As mentioned in subsection 3.1.1, I at first used a recursive factorial function for the parallel map benchmark, but replaced it with fib because it exhibited very strange scaling behavior on my laptop. (This was before I even had access to the 12-core Xeon machine.) These are its Haskell and Erlang implementations:

```haskell
-- Haskell
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)
```

```erlang
% Erlang
fac(0) -> 1;
fac(N) -> N * fac (N - 1).
```

The Scala implementation is different:

```scala
def fac(n: Int): BigInt = {
  @tailrec
  def withAcc(acc: BigInt, x: BigInt): BigInt =
    if (x == 0) acc else withAcc(acc * x, x - 1)
  withAcc(1, n)
}
```

In Scala, the `fac` function cannot simply be written as in the other languages, because then it crashes with a `java.lang.StackOverflowError` exception for sufficiently large values of n. Instead, a tail-recursive inner function is defined and used. (The `@tailrec` annotation makes the compiler enforce tail-recursion.)

At the end of the project, I recalled the factorial benchmark and decided to give it another try. I at first got very strange results, with Scala absolutely crushing Haskell. This turned out to be because I had used the type `Int`, resulting in integer overflow.

(Specifically, the factorial of any number $n > 1$ has at least $2^{\lfloor log_2 n \rfloor} - 1$ twos among its prime factors. This means that with 32-bit fixed precision integer arithmetic, the factorial of 100 is 0, because $fac(100)$ contains at least 63 twos as prime factors, and $2^{63} = 2^{32} \cdot 2^{31} \equiv 0 \pmod{2^{32}}$.)

Having fixed that by replacing `Int` with `BigInt`, I obtained the results in Figure 5.13. Note that $fac(5000)$ is an extremely large number — it has over 16 000 digits — so this is quite different from the fixed-precision fib benchmark.

I have not found any way to do arbitrary-precision integer arithmetic in Manticore. It is not possible to use e.g. `LargeInt.int` from Standard ML, because Manticore features only "a fixed set of numeric types — `int`, `long`, `integer`, `float`, and `double` — instead of SML's families of numeric modules" [27].

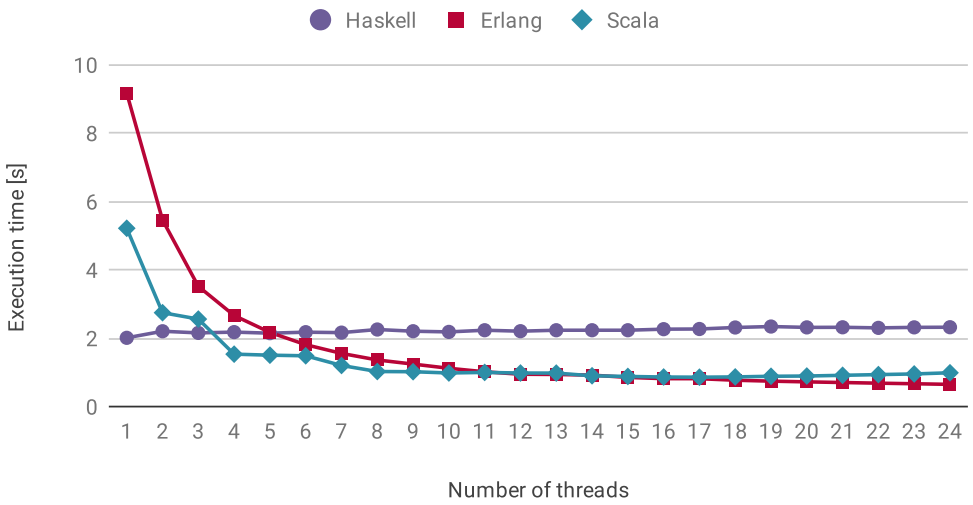

Figure 5.13: Mapping the factorial function ($width = 1\,000$, $depth = 5\,000$). Execution time (lower is better).

5.2 k-means clustering

This section contains the results from the benchmark of clustering 20 000 uniformly distributed points in a subset of 3D space into 200 clusters.

5.2.1 Haskell

The scaling of my modified k-means implementation can be seen in Figures 5.14–5.16.

5.2.2 Erlang

The Erlang results can be seen in Figures 5.17–5.19.

5.2.3 Scala

The Scala results can be seen in Figures 5.20–5.22.

5.2.4 Comparison

Figures 5.23–5.25 show k-means results from all languages except Manticore side by side.

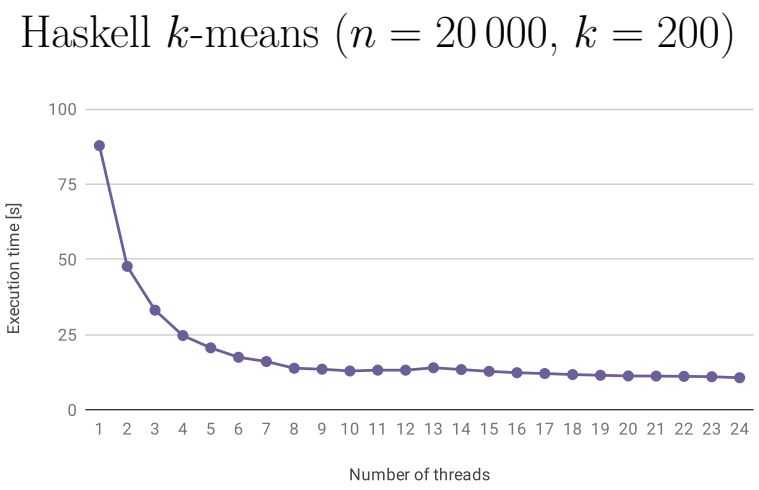

Figure 5.14: Execution time (lower is better).

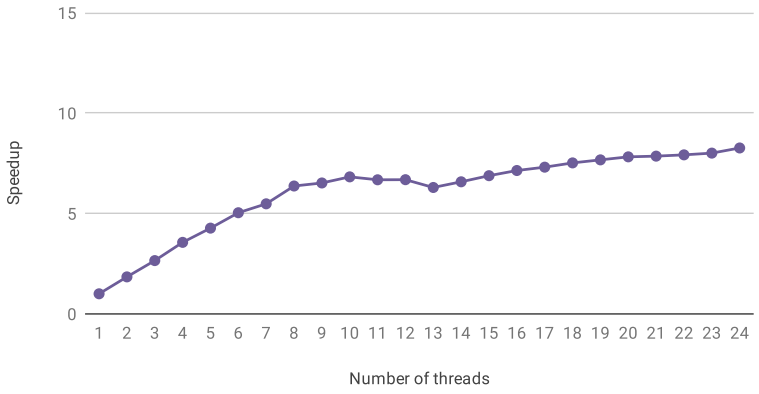

Figure 5.15: Speedup (higher is better).

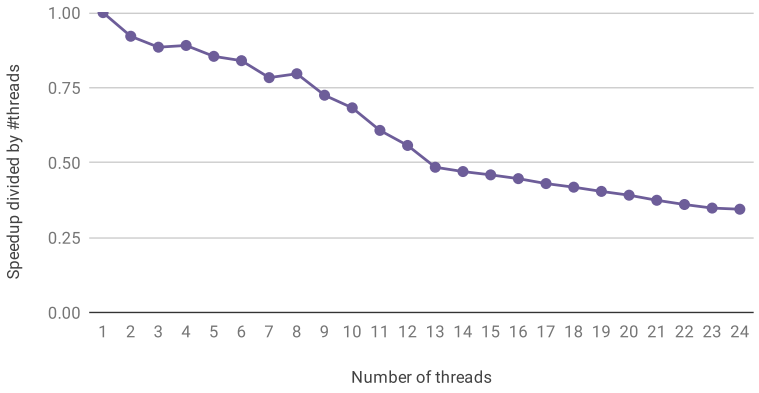

Figure 5.16: Thread efficiency (higher is better; 1 is perfect).

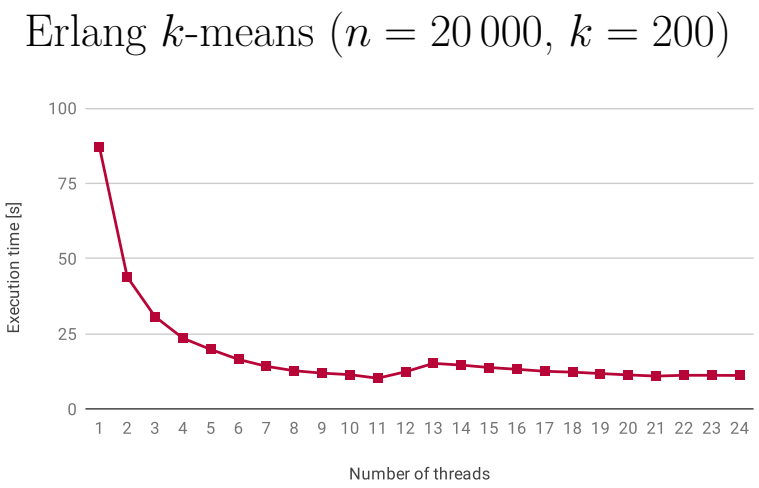

Figure 5.17: Execution time (lower is better).

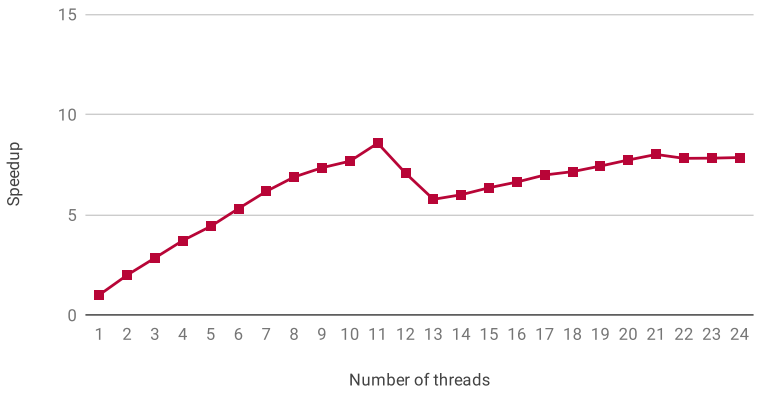

Figure 5.18: Speedup (higher is better).

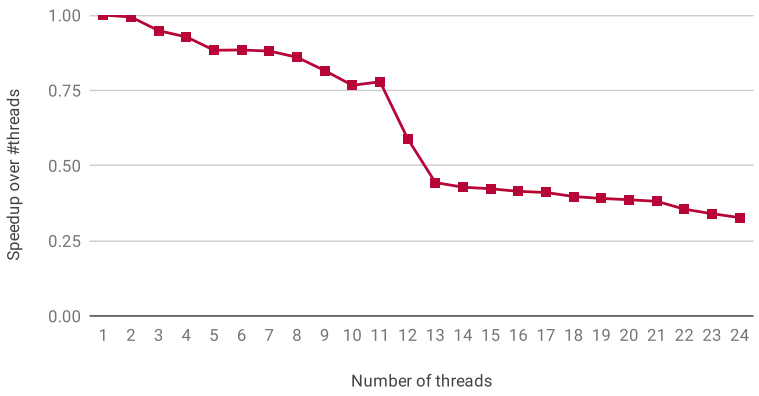

Figure 5.19: Thread efficiency (higher is better; 1 is perfect).

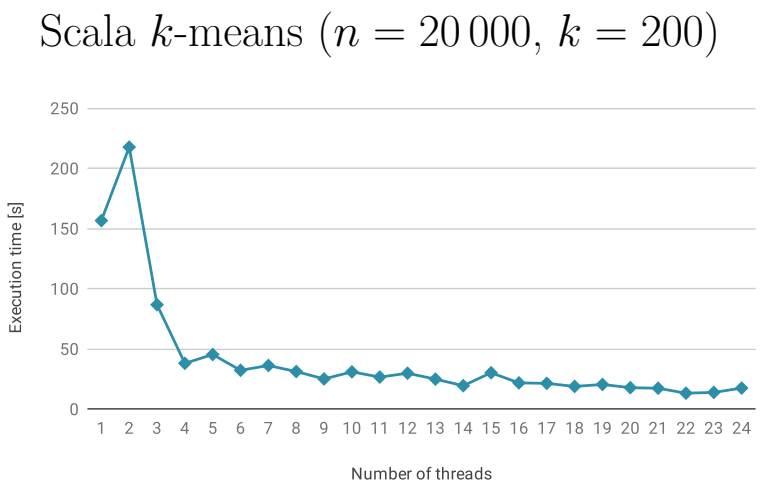

Figure 5.20: Execution time (lower is better).

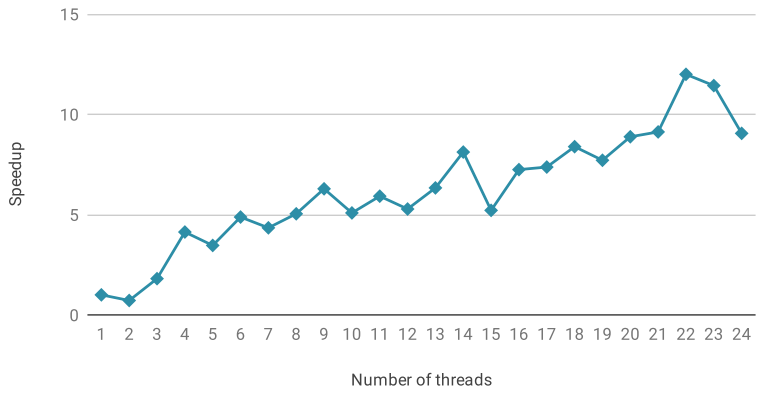

Figure 5.21: Speedup (higher is better).

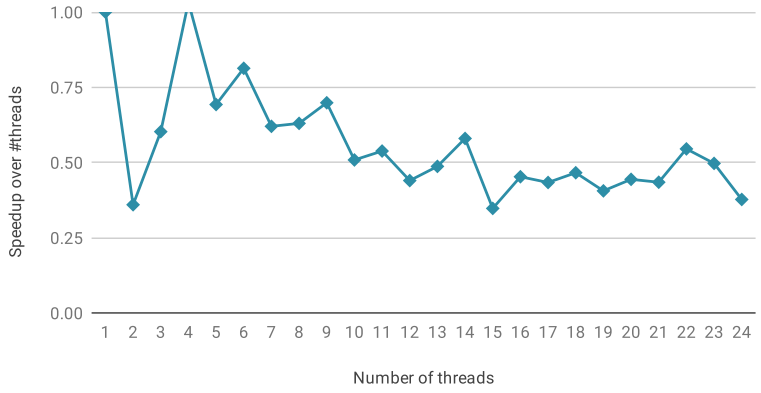

Figure 5.22: Thread efficiency (higher is better; 1 is perfect).

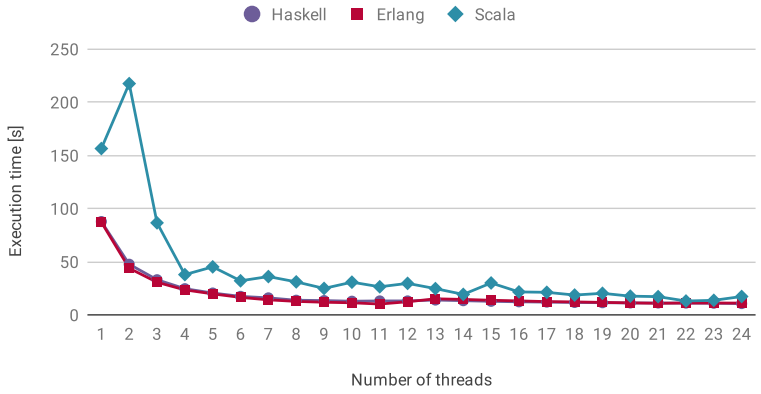

Figure 5.23: Execution time (lower is better).

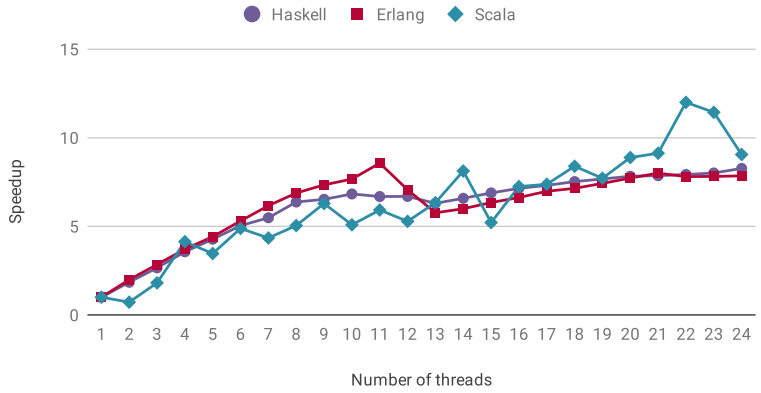

Figure 5.24: Speedup (higher is better).

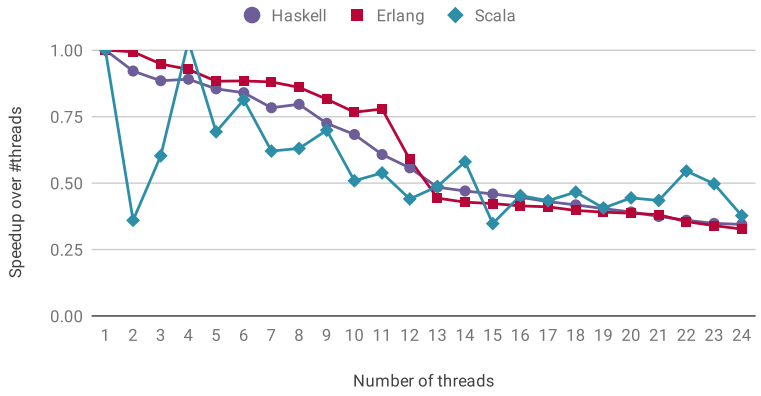

Figure 5.25: Thread efficiency (higher is better; 1 is perfect).

6
Discussion

6.1 Performance

There seem to be a few question marks surrounding the performance measurements presented in chapter 5. Even the embarrassingly parallel example of mapping a function over a list delivers surprising results.

6.1.1 Parallel map

The parallel map program with the fib function (figures 5.1–5.12) displays almost perfect scaling over cores for Haskell and Erlang, both of which stay close to a thread efficiency of 0.9 all the way up to 12 threads on our 12-core machine. Scala seems to be falling off somewhat at 6–7 threads.

However, Scala is faster than Haskell regardless of the number of threads, and especially so up to approximately 10 threads, where it performs around 60–70 % better. Erlang, on the other hand, is trailing *way* behind, being more than 4 times slower than Haskell across the board. Scala being so far ahead of Haskell does not surprise me, but Erlang being so far behind does — especially since Erlang and Haskell are almost impossible to distinguish from each other in the k-means graph.

In terms of absolute performance, Scala is fast enough to outperform a sequential C implementation on 2 cores/threads, while Haskell needs 3 and Erlang about 10.

SMT (Hyper-Threading)

Simultaneous multithreading (SMT) allows a CPU to execute two simultaneous threads on a single core, but it should not be misinterpreted as some magical way to get twice the performance out of the same piece of silicon. From my personal experience, I would expect somewhere around 30 % extra performance from Intel's implementation of SMT (Hyper-Threading).

An optimal scheduler will not put two threads doing a lot of work on the same core unless the number of threads exceeds the number of cores, so I would expect performance to increase steadily up to 12 threads, and then taper off and/or become less regular.

Looking at the results, especially Haskell and Erlang demonstrate similar, and not very surprising, trends with respect to SMT: almost perfect linear scaling up to 12 threads, then more erratic scaling up to 24 threads.

Scala does not display any break in the trend whatsoever as the number of threads increases beyond 12. It has already leveled off at about 6 threads, and sticks to

the same increasing trend almost up to 20 threads. It is difficult to say why this happens, but the complete absence of any dip at 12–13 threads suggests it *might* have something to do with how Scala's parallel libraries interact with the OS scheduler.

The factorial function

I find the factorial function results (Figure 5.13) quite interesting. The Erlang curve is absolutely perfect, to the point of appearing to be artificially constructed. Scala outperforms Erlang on few cores but does not scale as well.

Then there is Haskell. It seems to give up on scaling completely as soon as the fac function is involved. I suspect there might be something wrong with the benchmark implementation.

6.1.2 k-means clustering

The k-means graphs (figures 5.14–5.25) tell quite a different story. Erlang and Haskell have virtually identical execution times all the way up to 24 threads. Both display fairly good scaling with a thread efficiency of around or above 0.75 up to about 10 threads.

Scala, however, behaves completely differently from its two competitors. Its sequential version takes almost 80 % longer than Haskell and Erlang, and it is even slower on 2 threads. After that, it starts scaling fairly well and actually closes in on Haskell and Erlang as the number of threads increases.

The discrepancy between the two benchmark programs with respect to Scala's performance raises suspicions that the Scala k-means implementation may not be optimal. After all, it is almost a direct translation from Haskell, a completely different language. I would not be surprised at all if it turns out that the same algorithm can be implemented more efficiently in Scala.

One should note that the problem size is well beyond what is necessary for parallelism to yield any gains. For "small" problem instances, such as $n = 2\,000$ and $k = 20$, the parallel Haskell version is no faster than the sequential one at all, in my experience. Ideally, different parameter sets should be tried to see how problem size affects performance and parallelism.

6.2 Programming experience

Performance and scaling are not the only important aspects when discussing parallel programming. Just as interesting is the difficulty of writing parallel code — both in terms of how much time it takes and how confident one can be about program correctness.

I like the purely functional style of programming even without involving parallelism, because in my experience, mutable state often makes programming harder with respect to correctness. Issues related to mutable state can be magnified when the state is shared between concurrent processes, so I like functional programming especially much when writing parallel code.

6.2.1 Haskell

My initial impression of parallelism in Haskell was that there seemed to be many different ways to do it. Which one should be used? I ended up using the strategies approach, because I was most familiar with it. In this project, I have barely used a fraction of what strategies can accomplish. It is, in my opinion, an ingenious concept, relying on non-strictness to exist and on composability to excel. If Haskell were strict, the evaluation of an expression like `map f xs` could not be controlled by a strategy, and if strategies were not composable, it would be very cumbersome to define them for any but the simplest data types.

Depending on the problem at hand, however, strategies can require a great deal of knowledge about Haskell's inner workings to be used efficiently. This report has not really brought up any such problems, but whoever wants to write parallel Haskell code should read up on weak-head normal form (WHNF), normal form and how evaluation works in relation to parallelism. I highly recommend Marlow's book [26] for that purpose.

6.2.2 Erlang

Erlang has an entirely different take on parallelism compared to Haskell. Now, from my perspective, these languages are not very similar at all, so that does not come as a major surprise. In comparison, Erlang is more of a concurrent language, and I must say that both data parallelism and more coarse-grained concurrency are fairly accessible even for a relative novice.

In particular, it seems somewhat easier to find out "how to write parallel code", because there seems to be only one idiomatic way to do it: spawn worker processes, send them some work, and collect the results they send back.

There are a couple of aspects of Erlang that I find somewhat frustrating:

- We do not get very much help from the compiler. It does warn about some obvious mistakes, but it does not perform typechecking. This can be argued for and against, and Erlang probably could not be Erlang if it were statically typed, but I am personally fond of static typing.
- Its syntax, being easy to parse, repeatedly gives rise to syntax errors caused by a forgotten comma or deleting, adding or moving a line.

The interaction between variables, functions and atoms is also a possible cause of confusion. I was surprised to learn that a (unary) function named `f` is referenced by `fun f/1`, not `f` (which is an atom). Being used to lowercase variable names, I also sometimes accidentally wrote code conceptually like this:

```
f(x) -> x.
```

Evaluating `f(5)` yields this error:

```
** exception error: no function clause matching call to f/1
```

In a larger program, it may be difficult to understand what the problem is.

All in all, I think the Erlang/OTP architecture is very impressive and certainly an appreciated contribution to parallel/concurrent functional programming, even if

I do not feel entirely comfortable with the language itself.

6.2.3 Scala

In my opinion, Scala is an inspiring language, promoting functional programming while still allowing a more traditional imperative style. Its syntax is surprisingly flexible, to the point where I sometimes thought I was not looking at Scala code at all:

```
measure method "fibs" in {
  using (threads) in { t => fibs(t, xs) }
}
```

This did not look like Scala to me. But here is the same code with parentheses indicating precedence:

```
(measure method "fibs") in ({
  (using (threads)) in ({ t => fibs(t, xs) })
})
```

`method` and `in` are actually member accessors; their infix application is just syntactic sugar:

```
measure.method("fibs").in({
  using(threads).in({ t => fibs(t, xs) })
})
```

And since every block has the value of its last expression, the curly braces can be removed in this case:

```
measure.method("fibs").in(
  using(threads).in(t => fibs(t, xs))
)
```

We could even simplify further:

```
measure.method("fibs").in(
  using(threads).in(fibs(_, xs))
)
```

I cannot deny the beauty in this.

At other times, the syntax felt strange and not very typical for functional programming: A function cannot, in general, be referenced by name alone; one must usually type `f _` or `f(_)` for the function `f`. On the other hand, member accessors can be thought of as functions: `people.map(_.age)` means the same as `people.map(person => person.age)`. I certainly appreciate such a treat from a language that takes pride in being both object-oriented and functional.

When it comes to parallelism, Scala offers a fairly implicit approach that does not require much understanding at all for the most basic cases. On the flip side, it is entirely up to the programmer to keep track of state mutations. This stands in stark contrast with Haskell, which does not have side effects at all unless the

programmer asks really nicely, and Erlang, which keeps data strictly isolated to its owner process.

6.2.4 Manticore

I did not spend enough time with Manticore to get a good grasp of it and its parallelism constructs. It is an extension of Standard ML, so the syntax is similar to other functional languages like Haskell and Elm [28]. However, it is relatively difficult to find examples of how to actually use Manticore, since almost no one uses it. I would strongly advise anyone attempting to use it to study the regression tests included with the compiler.

One thing in particular seems to be outdated or implicit in the 2009 paper *Programming in Manticore* [27]: Functions like `send`, `recv` and `channel` are used without further ado, but they were not in scope for me; I had to use qualified names like `PrimChan.send`.

I also felt that the error messages from the compiler were sometimes confusing. Now, Manticore is a research project used by very few people in the world, so one cannot reasonably expect it to compete with mature, widespread languages like Haskell, Erlang and Scala with respect to usability. But I would like to mention some of the error messages I found confusing. Take for instance this (invalid) program:

```
structure Main = struct

end

main
```

The only output when running `pmlc example.pml` is this:

```
[example.pml:6.1] Error: syntax error; try deleting "ID"
mlb/mlb.sml:217.11-217.16: Error
```

Or spot the mistake in my intended benchmark program:

```
val w = 1000
val d = 30

fun fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n - 2) + fib (n - 1)

val xs =
  PArray.fromRope (Rope.tabulateSequential (fn _ => d) (1, w))

val _ = PArray.map fib xs
```

In this case, the error message does at least point out an existing token in the program:

```
***** Bogus CPS in Main after inline *****
** unbound variable fib<11476>#7.7
** unbound variable fib<11476>#7.7 in Apply
broken CPS dumped to broken-CPS
```

As described in section 4.2, the compiler seems to contain some sort of bug making it unable to handle recursive functions in combination with parallel arrays — unless the function is eta-abstracted. So the last line must instead be:

```
val _ = PArray.map (fn x => fib x) xs
```

Finally, at several points late in the project, when I wanted to run benchmarks, the compiler simply would not compile any valid programs at all. This appeared to happen randomly, and resulted in errors like this one:

```
Fatal error -- bogus fault not in ML:
sig = 11, code = 0x8058bfd, pc = 0x8058bfd)
```

The error may have been connected in some way to Nix shells. (A Nix shell is a temporary shell with a custom environment, spawned by the Nix package manager. I used the feature to have dependencies such as `pmlc` available when working on the project.) Sometimes, for example, entering a *nested* Nix shell would make `pmlc` stop working.

I do think that Manticore is an interesting project, especially since it provides both parallelism and concurrency, but I likely would not use it for real-world applications until it has reached a more mature stage.

6.3 Correctness and parallelism

As stated in chapter 2, the scope of this project is parallelism in functional programming, where correctness should not be jeopardized by traditional concurrency issues related to mutable state. That is, one would expect introduction of parallelism never to affect the correctness of the program.

However, to my surprise, this is not the case for my Scala implementation. Specifically, the number of chunks affects the output. For example, with $n = 100$ and $k = 2$, the parallel version returns the same result as the sequential one when run with `partitions` $= 2$, but not with `partitions` $= 3$. Figures 6.1 and 6.2 show how these results differ.

With the input data used for the benchmarks (3D points, $n = 20000$, $k = 200$; see item 3.1.2), the clustering is performed correctly on 2 threads, but not on 20 threads. (Determining the number of points assigned to the wrong cluster is not trivial due to how the result is represented.)

Further investigation has led me to the conclusion that this is due to floating-point rounding errors: in particular, floating-point addition is not associative. In my k-means implementations, the points are implemented as vectors of double-precision floating point numbers. For each partition, its points are combined (in `assignPS`) into one `PointSum` for each cluster. These results from the different partitions are then combined to create the final result of that refinement iteration.

Scala k-means with 2D points ($n = 100$, $k = 2$)

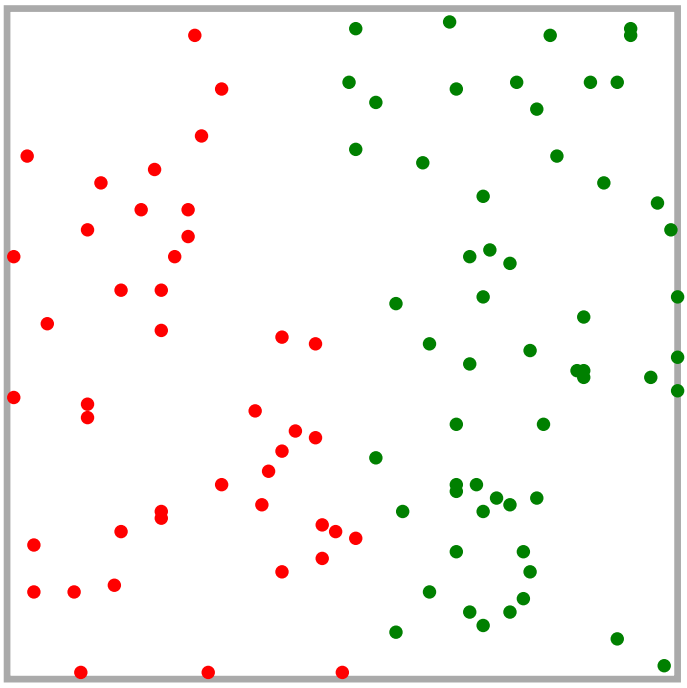

Figure 6.1: `partitions` = 2 (identical to sequential result).

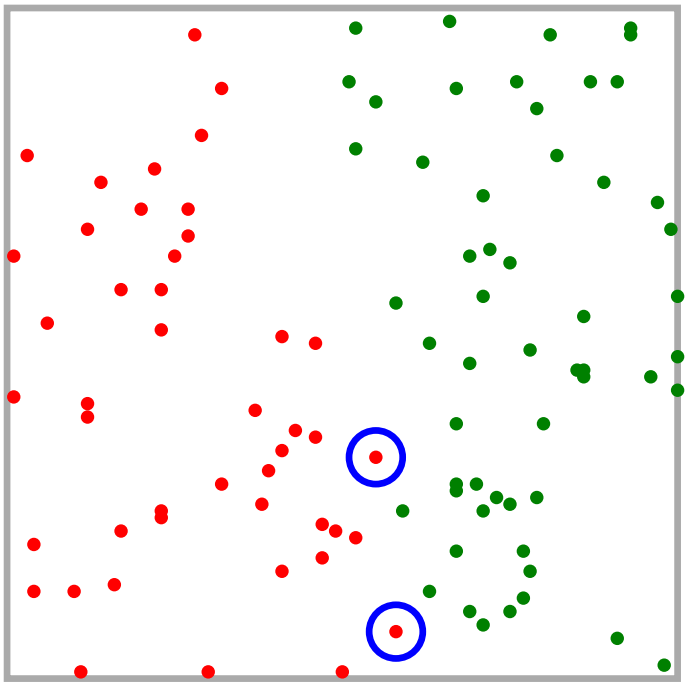

Figure 6.2: `partitions` = 3. The circled points have erroneously been assigned to the red (left-hand side) cluster.

For a given partition and cluster, its points, represented as a vector, are combined by a regular fold, accumulating a larger and larger sum as it traverses the vector. As this sum grows, so does the likelihood of rounding errors, since the difference in size between the accumulated sum and the values being added grows. If there is only one partition (containing all points), there is no difference between the sequential and the parallel behavior. But if there are more than one, floating-point rounding errors may cause a different clustering.

Here is a proof-of-concept Haskell program demonstrating this phenomenon:

```haskell
import Test.QuickCheck (Property, (===))

sumBalanced :: Num a => [a] -> a
sumBalanced xs = go (length xs) xs
  where
    go :: Num a => Int -> [a] -> a
    go 0 _ = 0
    go 1 (a:_) = a
    go n as = go n' as + go (n - n') (drop n' as)
      where n' = n `div` 2

prop_sumBalancedInt :: [Int] -> Property
prop_sumBalancedInt xs = sumBalanced xs === sum xs

prop_sumBalancedDouble :: [Double] -> Property
prop_sumBalancedDouble xs = sumBalanced xs === sum xs
```

`sumBalanced` sums up a list by recursively splitting it in two, then adding the sums of the two parts. Intuitively, it should have the same denotational semantics as `sum`, which just folds the given list using `(+)`. The semantics are indeed identical when summing integers, but, using QuickCheck [29], we can easily find a counterexample for doubles:

```
> quickCheck prop_sumBalancedInt
+++ OK, passed 100 tests.
> quickCheck prop_sumBalancedDouble
*** Failed! Falsifiable (after 4 tests and 8 shrinks):
[-0.1,-0.1,0.4]
0.20000000000000004 /= 0.2
```

This could certainly be a surprise for anyone exploring parallelism in functional programming. I would say it is a very dangerous bug insofar that it can produce errors subtle enough not to be noticed by a human in many scenarios.

One possible remedy would be to replace the naive (folding) summation of point sums with Kahan summation, devised by and named after the creator of the IEEE 754 floating-point standard [30]. Whereas naive summation results in a worst-case rounding error proportional to the number of terms n, with Kahan summation the error is independent of n and depends only on the floating-point precision. However, Kahan summation requires four times as many arithmetic operations; a faster option would be pairwise summation, with a worst-case error proportional to $\log n$.

6.4 Benchmarking is difficult

I have seen very strange results several times in this project. I would even say that the difficulty of benchmarking is one of the main insights I bring with me.

First, there can be an enormous amount of parameter combinations one might want to try. For Haskell alone, examples include problem size (*width* and *depth*; n and k), `parList` versus `parListChunk`, unboxed versus boxed integers, different values for the `-A` and `-H` RTS flags, and `-feager-blackholing`. The overall problem quickly explodes in complexity.

Second, managing collected data proved to be cumbersome. Looking at a file with results from, say, a week ago, I often found myself wondering what implementation details and parameters were used in that particular run.

Third, what constitutes a fair comparison between different programming languages? Is a direct translation of an algorithm fair? Maybe the Right Way to implement it is actually completely different in the target language. But maybe one does not know the correct way to implement a given algorithm in every language one wants to benchmark.

Fourth, it can be hard to visualize the results. The large number of possible parameter combinations certainly contributes to this, and so does the possibly large number of data points to be visualized.

Fifth, things just never stop going wrong. Everything from forgetting to upload a file to the benchmarking machine and accidentally testing the wrong code, to having a GCC linker error appear from nowhere, to forgetting to undo some change intended for debugging, can invalidate the results, sometimes without one even noticing.

6.5 Future work

Looking forward, I see several ways to improve or build upon the work described in this report:

- A better summation algorithm (see section 6.3) should be used to minimize floating-point errors in k-means.
- k-means should be run with different parameters to see how that affects performance and scaling. For example, how large problem sizes are necessary to achieve good scaling in different languages?
- There might be a more idiomatic and efficient way to implement k-means in Scala.
- Finding the reason behind Haskell's strange behavior in subsection 5.1.6 would be nice (and I would like to hear about it).
- A comparison with a lower-level imperative language such as C or C++ would be interesting. The same can be said for the higher-level (but still imperative) popular language Java 8.
- One could investigate how CPU architecture aspects such as cache, SMT implementation and inter-core communication affect performance.

6.6 Conclusion

As mentioned above, the difficulty of conducting this kind of research is a striking observation for me. It is, however, not the only one. I believe I can draw some conclusions about the studied languages as well:

- Parallelism is accomplished through different idioms, depending on the language being used. Erlang, with its explicit creation of lightweight internal processes, and Haskell, with its generic, composable strategies and separation of "what" and "how", seem to represent clearly distinct philosophies, at least from the programmer's point of view.
- Substantial performance gains are sometimes achievable with little difficulty in all languages, for example by adding `` `using` parList rdeepseq `` in Haskell or `.par` in Scala.
- Out of the three languages I have tested thoroughly, Erlang displayed the most regular and predictable scaling behavior overall.
- When mapping a function in Haskell and Scala, splitting the list into chunks seems to be a better choice than having each element processed in parallel, at least when each element can be computed fairly quickly (on the order of 10 milliseconds in my experiments). In light of this, Erlang's delightfully linear scaling, despite no chunking being done, is truly impressive.
- Manticore is an interesting language, but not yet ready for production use.
- Functional languages need not be completely demolished by C performance-wise, and may outperform a sequential C implementation on as few as two cores.
- Simultaneous multithreading can provide a performance boost of almost 50 % in some scenarios, and no improvement at all in others.
- Parallelism *can* affect correctness even in a functional program, presumably due to floating-point rounding errors (see section 6.3).
- There is no indisputable overall winner.

Bibliography

[1] P. W. Trinder et al. "Algorithm + strategy = parallelism". In: *Journal of Functional Programming* (1998). ISSN: 09567968. DOI: 10.1017/S0956796897002967.

[2] Chris A. MacK. "Fifty years of Moore's law". In: *IEEE Transactions on Semiconductor Manufacturing*. 2011. DOI: 10.1109/TSM.2010.2096437.

[3] Oliver Haslam. "Apple's $799 2018 iPad Pro A12X Geekbench Benchmark Scores Are On Par With $2,799 2018 Core i7 15-Inch MacBook Pro". In: *Redmond Pie* (Nov. 2018). URL: https://www.redmondpie.com/apples-799-2018-ipad-pro-a12x-geekbench-benchmark-scores-are-on-par-with-2799-2018-core-i7-15-inch-macbook-pro.

[4] Mark D Hill and Michael R Marty. "Amdahl's law in the multicore era". In: *Computer* 41.7 (2008).

[5] H-W Loidl et al. "Comparing parallel functional languages: Programming and performance". In: *Higher-Order and Symbolic Computation* 16.3 (2003), pp. 203–251.

[6] Jost Berthold et al. "Comparing and optimising parallel Haskell implementations for multicore machines". In: *2009 International Conference on Parallel Processing Workshops*. IEEE. 2009, pp. 386–393.

[7] Simon Marlow et al. "Haskell 2010 language report". In: *Available online http://www.haskell.org/ (May 2011)* (2010).

[8] Joe Armstrong et al. "Concurrent programming in Erlang". In: (1993).

[9] Steve Vinoski. "Concurrency with erlang". In: *IEEE Internet Computing* 11.5 (2007), pp. 90–93.

[10] Martin Odersky et al. *The Scala language specification*. 2007.

[11] Matthew Fluet et al. "Manticore: A heterogeneous parallel language". In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. 2007, pp. 37–44.

[12] The University of Glasgow. *Control.Parallel.Strategies*. 2010. URL: https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html#t:Strategy.

[13] Simon Marlow and Ryan Newton. *Control.Monad.Par*. 2011. URL: http://hackage.haskell.org/package/monad-par-0.3.4.8/docs/Control-Monad-Par.html.

[14] Gabriele Keller et al. "Regular, shape-polymorphic, parallel arrays in Haskell". In: *ACM Sigplan Notices*. Vol. 45. 9. ACM. 2010, pp. 261–272.

[15] Manuel M T Chakravarty et al. "Data Parallel Haskell: a status report". In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM. 2007, pp. 10–18.

[16] *Erlang Reference Manual: Data Types*. URL: http://erlang.org/doc/reference_manual/data_types.html#atom.

[17] Weizhong Zhao, Huifang Ma, and Qing He. "Parallel K-means clustering based on MapReduce". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Jaatun Martin Gilje. Vol. 5931 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 674–679. ISBN: 3642106641. DOI: 10.1007/978-3-642-10665-1{_}71.

[18] Bryan O'Sullivan et. al. *Criterion*. 2016. DOI: criterion. URL: http://hackage.haskell.org/package/criterion.

[19] Simon Marlow. "Example: The K-Means Problem". In: *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013. Chap. 3. ISBN: 1449335926. URL: https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/ch03.html#sec_using-parlist.

[20] Simon Alling. *kmeans-par (forked from Hackage)*. 2019. URL: https://github.com/SimonAlling/kmeans-par.

[21] The University of Glasgow. *Compile-time options for SMP parallelism*. 2019. URL: https://downloads.haskell.org/~ghc/8.6.4/docs/html/users_guide/using-concurrent.html#compile-time-options-for-smp-parallelism.

[22] The University of Glasgow. *Running a compiled program*. 2019. URL: https://downloads.haskell.org/~ghc/8.6.4/docs/html/users_guide/runtime_control.html.

[23] Jesper L. Andersen. *Testing a Parallel map implementation*. 2015. URL: https://medium.com/@jlouis666/testing-a-parallel-map-implementation-2d9eab47094e.

[24] Josh Suereth and Matthew Farwell. *SBT in Action: The simple Scala build tool*. Manning Publications Co., 2015.

[25] Simon Alling. *CPS inliner issue with recursive function being mapped*. 2019. URL: https://github.com/ManticoreProject/manticore/issues/10.

[26] Simon Marlow. *Parallel and concurrent programming in Haskell*. Vol. 7241 LNCS. 2012, pp. 339–401. ISBN: 9783642320958. DOI: 10.1007/978-3-642-32096-5{_}7.

[27] Matthew Fluet et al. "Programming in Manticore, a heterogenous parallel functional language". In: *Central European Functional Programming School*. Springer. 2009, pp. 94–145.

[28] Evan Czaplicki. "Elm: Concurrent FRP for Functional GUIs". PhD thesis. 2012.

[29] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ICFP*. 2000. ISBN: 1581132026.

[30] Nicholas J Higham. "The accuracy of floating point summation". In: *SIAM Journal on Scientific Computing* 14.4 (1993), pp. 783–799.

[31] Stichting NixOS Foundation. *Installing NixOS*. URL: https://nixos.org/nixos/manual/index.html#ch-installation.

A

Appendix: Code

A.1 `kmeans-par` (our modified version)

Our version of `kmeans-par` can be found here:

`https://github.com/SimonAlling/kmeans-par`

In addition to the modifications detailed in section 4.1, we have also added the `-feager-blackholing` flag to the `ghc-options` field, which, according to the GHC documentation, is recommended for parallel code [21]. Eager blackholing prevents threads from starting to work on a *thunk* (unevaluated expression) which is already being evaluated on another thread.

We have notified the original author of `kmeans-par` about our improvements. He appreciates our contributions and has added us to the maintainer group, so that we can patch the package on Hackage as well.

A.2 Our benchmarks

All code related to this report can be found here:

`https://github.com/SimonAlling/comparafun`

A.3 Running the benchmarks

This section is intended as a practical guide on how to run the benchmarks.

A.3.1 Preparations

We have used the Nix language to describe dependencies, so we suggest using the Nix package manager, because then you can just run `nix-shell` to get a shell with everything available and ready.

There are two ways to get access to Nix: installing it on an existing UNIX-like system such as Ubuntu or macOS; or running NixOS, a complete Linux distribution built entirely around Nix.

Installing Nix on Linux/macOS

This procedure worked for us on a clean installation of Ubuntu 18.04.

The following command (run as a regular user — not `root`) should be enough to install Nix in single-user mode (meaning that the current user owns `/nix`). You should not run it with `sudo`, but you need `sudo` rights for it to work properly.

```
$ curl https://nixos.org/nix/install | sh
```

It is also possible to install Nix in multi-user mode:

```
$ sh <(curl https://nixos.org/nix/install) --daemon
```

However, in this case, the install script makes some assumptions that might give rise to unforeseen problems. For example, it tries to create a group with ID `30000` as well as a set of users for the Nix build system. We do not know of any easy solutions to such problems, should they arise.

When the installation is complete, you should have `nix` available on your command line. (You may have to log out and in again.)

NixOS

NixOS can be downloaded from the NixOS website. The installation procedure is *not self-explanatory*, and requires more manual work than most mainstream Linux distributions, so we highly recommend reading the *Installation* chapter [31] in the manual *carefully*.

Without Nix

Nix is not strictly necessary for running the benchmarks. If you can install the Haskell Stack, Erlang/OTP, Scala/`sbt` and Manticore manually, then that should work as well. From now on, we will assume that you either have Nix or have installed all dependencies manually.

A.3.2 Acquire the code

```
$ git clone https://github.com/SimonAlling/comparafun
$ cd comparafun
$ git checkout REPORT
```

You should now have the exact version of the codebase used to produce the results in this report.

A.3.3 Start a Nix shell

If you want to ensure you get the exact same dependencies as used in this report, you should first add `nixos-19.03` as the *channel* from which Nix should download packages:

```
$ nix-channel --add https://nixos.org/channels/nixos-19.03 nixpkgs
$ nix-channel --update
unpacking channels...
created 2 symlinks in user environment
```

Then, in the root of the repository:

```
$ nix-shell
```

Nix will now download a large number of precompiled binaries from `cache.nixos.org` and set them up properly. If it completes successfully, you will be dropped into a new shell, where you should be able to run these commands with similar output:

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.6.4
$ erl -version
Erlang (SMP,ASYNC_THREADS,HIPE) (BEAM) emulator version 10.2.2
$ scala -version
Scala code runner version 2.12.8 ...
$ pmlc -version
pmlc [x86_64-linux; 0.0.0 (); built 2019-02-22]
```

A.3.4 Run the benchmarks

Scripts that run the benchmarks are provided for all languages. Configuration is done in `config.sh`. You should change `MAX_THREADS` in that file to the maximum number of threads you want to use. In general, this should be the number of hardware threads your CPU supports.

All benchmarks

This script runs all benchmarks for all languages:

```
$ ./batch.sh
```

Haskell

```
$ cd haskell
$ ./batch-fib.sh
# Output saved to benchmark-fib-*.log
$ ./batch-kmeans.sh
# Output saved to benchmark-kmeans-*.log
$ ./batch-fac.sh
# Output saved to benchmark-fac-*.log
```

Erlang

```
$ cd erlang
$ ./batch-fib.sh
# Output saved to benchmark-fib-*.log
$ ./batch-kmeans.sh
# Output saved to benchmark-kmeans-*.log
$ ./batch-fac.sh
# Output saved to benchmark-fac-*.log
```

Scala

```
$ cd scala
$ ./batch.sh fib
# Output saved to fib-*.log
$ ./batch.sh kmeans
# Output saved to kmeans-*.log
$ ./batch.sh fac
# Output saved to fac-*.log
```

Manticore

```
$ cd manticore
$ ./batch-fib.sh
# Output saved to fib-*.log
$ ./batch-fac.sh
# Output saved to fac-*.log
```

A.3.5 Extract the results

There is a script named `extract.sh` in the `analysis` directory. It is intended to be run from a subdirectory of `analysis`, where it expects to find `.log` files. It is used like this:

```
$ cd haskell
$ ./batch-fib.sh
$ mkdir ../analysis/fib-haskell
$ mv *.log ../analysis/fib-haskell
$ cd ../analysis/fib-haskell
$ ../extract.sh haskell
```

The script scans the `.log` files in the current directory and prints only the execution times in milliseconds, so that they can be copied to a spreadsheet or similar. The files are scanned in the order given by `ls -1v`.

B

Appendix: Computers

Benchmarking machine

We run our benchmarks on a headless machine with these specifications:

- Processor: Intel Xeon Gold 6126 “Skylake-SP” @ 2.60 GHz (12C/24T)
- Memory: 192 GiB DDR4 @ 2666 MT/s
- OS: Red Hat Enterprise Linux Workstation 7.5 “Maipo”
  - Kernel: Linux 3.10.0-957.10.1.el7.x86_64

Laptop

Our laptop is mentioned in the report. It has these specifications:

- Processor: Intel Core i7-7500U “Kaby Lake” @ 2.7–3.5 GHz (2C/4T)
- Memory: 16 GiB LPDDR3
- OS: NixOS 18.09 “Jellyfish”
  - Kernel: Linux 4.14.105