Concepts in Parallel Programming

Arrows for Parallel Computation

Martin Braun (1249080)

First supervisor: Dr. Oleg Lobachev

Second supervisor: Prof. Dr. Thomas Rauber

AI5 – Visual Computing
University of Bayreuth

16.05.2018

Arrows are a general interface for computation and an alternative to Monads for API design. In contrast to Monad-based parallelism, we explore the use of Arrows for specifying generalised parallelism. Specifically, we define an Arrow-based language and implement it using multiple parallel Haskells. As each parallel computation is an Arrow, such parallel Arrows (PArrows) can be readily composed and transformed as such. To allow for more sophisticated communication schemes between computation nodes in distributed systems, we utilise the concept of Futures to wrap direct communication. To show that PArrows have similar expressive power as existing parallel languages, we implement several algorithmic skeletons and four benchmarks. Benchmarks show that our framework does not induce any notable performance overhead. We conclude that Arrows have considerable potential for composing parallel programs and for producing programs that can execute on multiple parallel language implementations.

Contents

1 Introduction

Functional languages have a long history of being used for experimenting with novel parallel programming paradigms. Haskell, which we focus on in this paper, has several mature implementations. We regard here in-depth Glasgow parallel Haskell or short GpH (its Multicore SMP implementation, in particular), the Par Monad, and Eden, a distributed memory parallel Haskell. These languages represent orthogonal approaches. Some use a Monad, even if only for the internal representation. Some introduce additional language constructs. Chapter 3.2 gives a short overview over these languages.

A key novelty in this paper is to use Arrows to represent parallel computations. They seem a natural fit as they can be thought of as a more general function arrow ($\rightarrow$) and serve as general interface to computations while not being as restrictive as Monads (Hughes, 2000). Chapter 3.1.4 gives a short introduction to Arrows.

We provide an Arrows-based type class and implementations for the three above mentioned parallel Haskells. Instead of introducing a new low-level parallel backend to implement our Arrows-based interface, we define a shallow-embedded DSL for Arrows. This DSL is defined as a common interface with varying implementations in the existing parallel Haskells. Thus, we not only define a parallel programming interface in a novel manner – we tame the zoo of parallel Haskells. We provide a common, very low-penalty programming interface that allows to switch the parallel implementations at will. The induced penalty is in the single-digit percent range, with means typically under 2% overhead in measurements over the varying cores configuration (Chapter 7). Further implementations, based on HdpH or a Frege implementation (on the Java Virtual Machine), are viable, too.

1.1 Contributions

We propose an Arrow-based encoding for parallelism based on a new Arrow combinator $parEvalN :: [arr\ a\ b] \rightarrow arr\ [a]\ [b]$. A parallel Arrow is still an Arrow, hence the resulting parallel Arrow can still be used in the same way as a potential sequential version. In this paper we evaluate the expressive power of such a formalism in the context of parallel programming.

- We introduce a parallel evaluation formalism using Arrows. One big advantage of this specific approach is that we do not have to introduce any new types, facilitating composability (Chapter 4).
- We show that PArrow programs can readily exploit multiple parallel language implementations. We demonstrate the use of GpH, a Par Monad, and Eden. We do not re-implement all the parallel internals, as this functionality is hosted in the $ArrowParallel$ type class, which abstracts all parallel implementation logic. The implementations can easily be swapped, so we are not bound to any specific one.

This has many practical advantages. For example, during development we can run the program in a simple GHC-compiled variant using GpH and afterwards deploy it on a cluster by converting it into an Eden program, by just replacing the $ArrowParallel$ instance and compiling with Eden's GHC variant (Chapter 4).

- We extend the PArrows formalism with $Future$s to enable direct communication of data between nodes in a distributed memory setting similar to Eden's Remote Data (Dieterle et al., 2010b). Direct communication is useful in a distributed memory setting because it allows for inter-node communication without blocking the master-node. (Chapter 5.1)
- We demonstrate the expressiveness of PArrows by using them to define common algorithmic skeletons (Chapter 4.4), and by using these skeletons to implement four benchmarks (Chapter 7).
- We practically demonstrate that Arrow parallelism has a low performance overhead compared with existing approaches, e.g. the mean over all cores of relative mean overhead was less than 3.5% and less than 0.8% for all benchmarks with GpH and Eden, respectively. As for |Par| Monad, the mean of mean overheads was in favour of PArrows in all benchmarks (Chapter 7).

PArrows are open source and are available from `https://github.com/s4ke/Parrows`.

2 Related Work

2.1 Parallel Haskells

The non-strict semantics of Haskell, and the fact that reduction encapsulates computations as closures, makes it relatively easy to define alternate parallelisations. A range of approaches have been explored, including data parallelism (Chakravarty et al., 2007, Keller et al. (2010)), GPU-based approaches (Mainland and Morrisett, 2010,obsidian-phd), software transactional memory (Harris et al., 2005, Perfumo et al. (2008)). The Haskell–GPU bridge Accelerate (Chakravarty et al., 2011, Clifton-Everest et al. (2014), McDonell et al. (2015)) is completely orthogonal to our approach. A good survey of parallel Haskells can be found in Marlow (2013).

Our PArrow implementation uses three task parallel languages as backends: the GpH (Trinder et al., 1996, Trinder et al. (1998)) parallel Haskell dialect and its multicore version (Marlow et al., 2009), the Par Monad (Marlow et al., 2011, Foltzer et al. (2012)), and Eden (Loogen et al., 2005, Loogen (2012)). These languages are under active development, for example a combined shared and distributed memory implementation of GpH is available (Aljabri et al., 2014, Aljabri et al. (2015)). Research on Eden includes low-level implementation (Berthold, 2008, Berthold (2016)), skeleton composition (Dieterle et al., 2016), communication (Dieterle et al., 2010b), and generation of process networks (Horstmeyer and Loogen, 2013). The definitions of new Eden skeletons is a specific focus (Hammond et al., 2003, Berthold and Loogen (2006), Berthold et al. (2009b), Berthold et al. (2009c), Dieterle et al. (2010a), Encina et al. (2011), Dieterle et al. (2013), Janjic et al. (2013))

Other task parallel Haskells related to Eden, GpH, and the Par Monad include: HdpH (Maier et al., 2014, Stewart (2016)) is an extension of Par Monad to heterogeneous clusters. LVish (Kuper et al., 2014) is a communication-centred extension of the Par Monad.

2.2 Algorithmic skeletons

Algorithmic skeletons were introduced by Cole (1989). Early publications on this topic include Danelutto et al. (1992), Darlington et al. (1993), Botorog and Kuchen (1996), Lengauer et al. (1997), Gorlatch (1998). Rabhi and Gorlatch (2003) consolidated early reports on high-level programming approaches. Types of algorithmic skeletons include map-, $fold$-, and $scan$-based parallel programming patterns, special applications such as divide-and-conquer or topological skeletons.

The $farm$ skeleton (Hey, 1990, Peña and Rubio (2001), Poldner and Kuchen (2005)) is a statically task-balanced parallel map. When tasks' durations cannot be foreseen, a dynamic load balancing ($workpool$) brings a lot of improvement (Rudolph et al., 1991, Hammond et al. (2003), Hippold and Rünger (2006), Berthold et al. (2008),Marlow2009). For special tasks $workpool$ skeletons can be extended with dynamic task creation (Priebe, 2006, Dinan et al. (2009), Brown and Hammond (2010)). Efficient load-balancing schemes for $workpool$s are subject of research (Blumofe and Leiserson, 1999, Acar et al. (2000), Nieuwpoort et al. (2001), Chase and Lev (2005), Olivier and Prins (2008), Michael et al. (2009)).

The $fold$ (or $reduce$) skeleton was implemented in various skeleton libraries (Kuchen, 2002, Karasawa and Iwasaki (2009), Buono et al. (2010), Dastgeer et al. (2011)), as also its inverse, $scan$ (Bischof and Gorlatch, 2002, Harris et al. (2007)). Google map–$reduce$ (Dean and Ghemawat, 2008, Dean and Ghemawat (2010)) is more special than just a composition of the two skeletons (Lämmel, 2008, Berthold et al. (2009b)).

The effort is ongoing, including topological skeletons (Berthold and Loogen, 2006), special-purpose skeletons for computer algebra (Berthold et al., 2009c, Lobachev (2011), Lobachev (2012), Janjic et al. (2013)), iteration skeletons (Dieterle et al., 2013). The idea of Linton et al. (2010) is to use a parallel Haskell to orchestrate further software systems to run in parallel. Dieterle et al. (2016) compare the composition of skeletons to stable process networks.

2.3 Arrows

Arrows were introduced by Hughes (2000) as a less restrictive alternative to Monads, in essence they are a generalised function arrow $\rightarrow$. Hughes (2005) presents a tutorial on Arrows. Jacobs (2009), Lindley et al. (2011), Atkey (2011) develop theoretical background of Arrows. (Paterson, 2001) introduced a new notation for Arrows. Arrows have applications in information flow research (Li and Zdancewic, 2006, Li

and Zdancewic (2010), Russo et al. (2008)), invertible programming (Alimarine et al., 2005), and quantum computer simulation (Vizzotto, 2006). But probably most prominent application of Arrows is Arrow-based functional reactive programming, AFRP (Nilsson et al., 2002, Hudak et al. (2003), Czaplicki and Chong (2013)). (Liu et al., 2009) formally define a more special kind of Arrows that capsule the computation more than regular Arrows do and thus enable optimisations. Their approach would allow parallel composition, as their special Arrows would not interfere with each other in concurrent execution. In contrast, we capture a whole parallel computation as a single entity: our main instantiation function $parEvalN$ makes a single (parallel) Arrow out of list of Arrows. Huang et al. (2007) utilise Arrows for parallelism, but strikingly different from our approach. They use Arrows to orchestrate several tasks in robotics. We, however, propose a general interface for parallel programming, while remaining completely in Haskell.

Arrows in other languages

Although this work is centered on Haskell implementation of Arrows, it is applicable to any functional programming language where parallel evaluation and Arrows can be defined. Basic definitions of PArrows are possible in the Frege language[1] (which is basically Haskell on the JVM). However, they are beyond the scope of this work, as are similar experiments with the Eta language[2], a new approach to Haskell on the JVM.

(Achten et al., 2004, Achten et al. (2007)) use an Arrow implementation in Clean for better handling of typical GUI tasks. (Dagand et al., 2009) used Arrows in OCaml in the implementation of a distributed system.

[1] GitHub project page at `https://github.com/Frege/frege`
[2] Eta project page at `http://eta-lang.org`

3 Background

Before we delve into our novel approach for parallel programming using Arrows, we give a short overview of all our main concepts and technologies. We start by giving an introduction to functional programming (Chapter 3.1) including a short tutorial on Monads (Chapter 3.1.3) before explaining the concept of Arrows (Chapter 3.1.4). Finally we give a short introduction to the main parallel Haskells used as backends for our DSL in this thesis (Chapter 3.2) - GpH, the Par Monad, and Eden.

3.1 Functional Programming

This section covers the basics of functional programming. We start by citing Hughes (1990) why functional programming matters including a characterisation of the concept in general (Chapter 3.1.1). Then, we give a short introduction to functional programming with Haskell (Chapter 3.1.2) and also explain the concept of Monads (Chapter 3.1.3) which some parallel Haskells use. Finally, we introduce Arrows and explain their type class in Haskell (Chapter 3.1.4).

3.1.1 Why Functional Programming?

Hughes (1990) describes the fundamental idea of functional programming like this:

> Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

Functional programming is also often - wrongly - only defined by what it does not allow programmers to do. Hughes (1990) furthermore describes this aspect elegantly while naming the usual advantages of functional programs:

> The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant — since no side- effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa — that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.
>
> [...]
>
> Even a functional programmer should be dissatisfied with these so-called advantages, because they give no help in exploiting the power of functional languages. One cannot write a program that is particularly lacking in assignment statements, or particularly referentially transparent. There is no yardstick of program quality here, and therefore no ideal to aim at.

To argue that there is merit in functional programming besides having fewer error-prone features Hughes (1990) also goes into detail about one of the actual aspects why functional programming matters - composability. He does this by showing how higher order functions help in expressing programs in a modular way. The focus on composability can be seen in all the definitions of Haskell functions in the following sections of this thesis.

3.1.2 A Short introduction to Haskell

In the following section, we will give a short introduction to functional programming with Haskell. While this will give a good idea of how programming in Haskell works, this is not aimed to be a complete tutorial on Haskell, but merely a quick overview over the most relevant features of the language used in this thesis. The following is loosely based on the book „Learn you a haskell for great good!“ (Michaelson, 2013).

From Imperative Programming to Functional Programming

In order to ease the introduction to functional programming, we will give a short introduction to functional programming in Haskell in this section by comparing the general style of imperative C code to functional Haskell using the example of the Fibonacci sequence.

To start off, we take a look at the iterative implementation of the Fibonacci sequence in Fig. 3.1. It contains assignments and a loop, which in pure[1] functional programming we we can not use[2].

```
int fib( int n ) {
    int pre = 0;
    int cur = 1;
    int res = 0;
    for ( int i = 0; i < n; ++i ) {
        res = pre + cur;
        pre = cur;
        cur = res;
    }
    return cur;
}
```

Figure 3.1: Iterative Fiboncacci in C

If we translate this Fibonacci example into a recursive definition (Fig. 3.2), however, we get pure functional C code without any assignment statements, that resembles the Haskell variant in Fig. 3.3. Note the flow of the programming without requiring any modifiable state.

```
int fib( int n ) {
    if ( n <= 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```

Figure 3.2: Recursive Fiboncacci in C

In functional languages like Haskell we only express computations in this matter by composition of functions (recursion in essence is also just a composition of a function

[1] Pure code is code without side-effects. Assignments are side-effects.

[2] It is however possible to introduce monadic DSLs in Haskell that mimic C style behaviour, see `https://hackage.haskell.org/package/ImperativeHaskell-2.0.0.1`.

with itself). Because of this and since we can not change the state of any associated variables, we generally also do not have to worry about the order of execution in functional programs and let the compiler decide how to resolve the recursive formula. In general, we can say that in functional programming we primarily focus on what information is required and by which transformations to compute it instead of how we perform them and how we track the changes in state.[3]

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n \\
&\quad \mid n \leqslant 0 = 0 \\
&\quad \mid n \equiv 1 = 0 \\
&\quad \mid \mathit{otherwise} = \\
&\qquad (\mathit{fib}\ (n-2)) \\
&\qquad\quad + (\mathit{fib}\ (n-1))
\end{aligned}
$$

Figure 3.3: Standard Fibonacci in Haskell.

Haskell being a functional language does not mean, that we do not have the usual problem of a too small call-stack size encountered when programming with recursion. While Haskell programs can naturally handle much bigger call-stacks without overflowing, at some point the limit will be reached and the program will crash. But since the class of tail-recursive programs is equivalent to the class of all recursive programs (which is in turn equivalent to all imperative programs), this is no big problem: We can just translate our *fib* definition into a tail-recursive variant (Fig. 3.4) which Haskell's compiler is capable of automatically translating into looping machine code.

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n \\
&\quad \mid n \leqslant 0 = 0 \\
&\quad \mid \mathit{otherwise} = \mathit{fib}'\ n\ 0\ 1 \\
&\quad \textbf{where} \\
&\qquad \mathit{fib}' :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\qquad \mathit{fib}'\ n\ \mathit{prev}\ \mathit{res} \\
&\qquad\quad \mid n \equiv 0 = \mathit{res} \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{fib}'\ (n-1)\ \mathit{res}\ (\mathit{res} + \mathit{prev})
\end{aligned}
$$

Figure 3.4: Tail Recursive Fibonacci in Haskell.

[3] from `https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/linq/functional-programming-vs-imperative-programming`

Functions

As already mentioned above, the basic building blocks of a Haskell program are functions. We define them like this:

$$f :: Int \to Int \to Int$$
$$f\ x\ y = multiply\ x\ y$$

Here, we declared a function f which takes two arguments of type Int and returns yet another Int. In the definition we say that f is the function $multiply$ applied to both its arguments x and y. We define $multiply$ as:

$$multiply :: Int \to Int \to Int$$
$$multiply\ x\ y = x * y$$

In Haskell, since f and $multiply$ seem to be the same, we can even write this relationship directly:

$$f :: Int \to Int \to Int$$
$$f = multiply$$

We can do so because in Haskell functions can be treated just like any other type. For example, if we wanted to have another function g which applied f on two lists of integers, we can write

$$g :: [Int] \to [Int] \to [Int]$$
$$g = zipWith\ f$$

where zipWith would be of type $(Int \to Int \to Int) \to [Int] \to [Int] \to [Int]$. In Haskell it is common to express calculations in such a way using higher-order functions. We will see more of this later in this Chapter.

Type inference

Taking the same example function g from above, it does not make sense to be so restrictive in terms of which type to allow in such a function since all it does is apply some function to zip two lists. Thankfully, in Haskell we can define functions in a completely generic way such that we can write the actual type of zipWith as $(a \to b \to c) \to [a] \to [b] \to [c]$ as in it can zip a list containing some as with a list

containing a list of bs with a function $a \to b \to c$ to get a list of cs. Only because we use this function in the context of our function g it is specialized into the Int form.

Furthermore we can even define g without writing down the type definition and let the compiler determine the actual type of g.

$$g = zipWith\ f$$

While this is possible, it is generally encouraged to always specify the type of top-level functions for better readability, but sometimes this is useful for some nested helper functions.

Function composition, higher-order functions, and function application

As we have seen, in Haskell, functions can be handled similar to data types. This way, we can for example define a function that computes a number to the power of four as

$$\begin{aligned} &toThePowerOfFour :: Int \to Int \\ &toThePowerOfFour = toThePowerOfTwo \circ toThePowerOfTwo \end{aligned}$$

with $\circ$ being the functional composition operator with type $(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$[4] and where $toThePowerOfTwo$ is defined simply as

$$\begin{aligned} &toThePowerOfTwo :: Int \to Int \\ &toThePowerOfTwo\ x = multiply\ x\ x \end{aligned}$$

Another aspect of functions being similar to data types is that, in functional programming, we frequently use higher order functions to express calculations. We have seen this earlier with the use of $zipWith$. Other often used higher-order functions include mapping ($map :: (a \to b) \to [a] \to [b]$, i.e. convert a list of as into a list of bs with the given function $a \to b$) and folding (e.g. $foldLeft :: (b \to a \to b) \to b \to [a] \to b$, i.e. reduce the list with the given function $b \to a \to b$ into a singular value given a starting value of type b). These are often used in some sort of composition like

$$\begin{aligned} &euclidDistance :: [Int] \to [Int] \to Int \\ &euclidDistance = sqrt \circ foldLeft\ (+)\ 0 \circ map\ (toThePowerOfTwo) \circ zipWith\ (-) \end{aligned}$$

[4] note the order of the arguments, $g \circ f$ means to first apply f and then g and not the other way around

Note that while this could have easily been written shorter as something along the lines of $sqrt\ (foldLeft\ (+)\ 0\ (zipWith\ (\lambda a\ b \rightarrow toThePowerOfTwo\ (a-b))))$ it is easy to see that the above declaration is easier to understand because of the simple steps the computation takes. We first zip the list of inputs with element-wise subtraction and then square this difference, sum these results up and finally take the square root. This is something we see a lot in Haskell code: Complex computations can be expressed with the help of higher-order functions instead of having to write it manually. This is not only much shorter, but also easier to understand for other programmers which have to read-up on the implementation for some reason.

Something which is also quite useful in Haskell is the function application operator $(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$ which allows for the application of a function $a \rightarrow b$ to a given argument a. It is simply defined as:

$$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f\ \$\ x = f\ x$$

While the use-case for such an operator might not be immediately clear, it will, if we take a look at the following function $listApp :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$ where we take a list of functions $[a \rightarrow b]$ and apply them one-by-one with their respective input values from the input list $[a]$ to generate a list of results b:

$$listApp :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$$
$$listApp = zipWith\ (\$)$$

Here, if we had not used the $(\$)$ operator, we would have to write $zipWith\ (\lambda f\ a \rightarrow f\ a)$ which obviously seems a bit redundant.

Something the $(\$)$ operator is also used quite often is to write shorter code. For example, code snippets like

$$someFunc = f1\ (f2\ param1\ (f3\ param2\ (f4\ param3)))$$

can also be written without the braces as

$$someFunc = f1\ \$\ f2\ param1\ \$\ f3\ param2\ \$\ f4\ param3$$

which is sometimes preferred to the brace-style, but is semantically identical.

Conditional Computation

Haskell has different styles of dealing with conditional evaluation. We will now show the most common variants to express conditional statements.

The most obvious one in terms of functionality is the **if** ... **then** ... **else** construct:

$$
\begin{aligned}
&myFunc :: Int \to Int \\
&myFunc\ x = \textbf{if}\ x < 10\ \textbf{then}\ x * 2\ \textbf{else}\ x * 4
\end{aligned}
$$

While having the same well-known semantics of any **if** ... **then** ... **else** like they could be found in imperative languages like e.g. C, in Haskell, being a functional language, the **else** is non-optional as expressions are required to be total.[5]

An alternative to this are guards, which make expressions easier to read if many alternatives are involved in a function:

$$
\begin{aligned}
&myFunc :: Int \to Int \\
&myFunc\ x \\
&\quad \mid x < 10 = x * 2 \\
&\quad \mid x < 12 = x * 3 \\
&\quad \mid x < 14 = x \\
&\quad \mid x > 18 \wedge x < 20 = 42 \\
&\quad \mid otherwise = x * 4
\end{aligned}
$$

Yet another technique for conditional computation is using pattern matching. For conditional statements we can use it by writing definitions of the function for specific values, like

$$
\begin{aligned}
&myFunc :: Int \to Int \\
&myFunc\ 5 = 10 \\
&myFunc\ x\ @\ 10 = x * 10 \\
&myFunc\ x = x * 2
\end{aligned}
$$

, where the first matching definition is chosen during computation. Alternatively, we can do pattern matching with the help of **case** expressions:

$$
\begin{aligned}
&myFunc :: Int \to Int \\
&myFunc\ x = \textbf{case}\ x\ \textbf{of}
\end{aligned}
$$

[5] total in terms of computation, unsuccessful calculations can still be expressed with constructs like $Maybe\ a$.

$$5 \rightarrow 10$$
$$x \ @ \ 10 = x * 10$$
$$x = x * 2$$

These can be used just like ordinary expressions.

We can not, however, express boolean statements in this way. This is because pattern matching is done on the structure of the value that is being pattern matched. Later in this section we will see what other powerful things we can do with this technique.

where, let

While Haskell does not have variables, it still allows the programmer to name sub-expressions so that either the code becomes more clear or that it can be reused more easily. Here, two different variants are available: **where** and **let**.

With the help of **where** we can write code like the following:

```
whereReuse :: Double → Double → String
whereReuse a b
  | divided > 10 = "a is more than 10 times b"
  | divided ≡ 10 = "a is 10 times b"
  | otherwise = "a is less than 10 times b"
  where divided = a / b
```

where is just syntactic sugar, though, and can not used in expressions like $f\ (a * 2$ **where** $a = 3)$. This is possible with **let** where we can write f (**let** $a = 3$ **in** $a * 2)$ or **let** $a = 3$ **in** $f\ (a * 2)$. **let** can in contrast, however, not be used in conjunction with guards.

Type safety

Haskell is a statically typed functional language. This means that during compilation all types are checked for compatibility and type declarations are not just treated as optional „hints“ to the type-checker. Pairing this with the pure aspect of the language means that Haskell programs seem to be correct more often if the program compiles than in imperative languages. The compiler essentially helps the programmer to write *semantically correct* instead of just syntactically correct code. It should be noted

that this does not mean that testing can be omitted. It is still extremely important, but becomes less cumbersome because state is mostly a non-issue.

Type classes

The example function $multiply$ from above seems a bit restrictive as it only allows for the usage of Ints. Ints are obviously not the only type which can be multiplied. Haskell has a way to express this fact: type classes. We can express a type class $Multiplicable\ a$ that encapsulates the contract of $multiply$ on some type a as

$$\begin{aligned}&\textbf{class}\ Multiplicable\ a\ \textbf{where}\\&\quad multiply :: a \to a \to a\end{aligned}$$

With this class in place, we can then introduce instances - implementations of the contract - for specific types. For example the instance for Int, $Multiplicable\ Int$ can be defined as

$$\begin{aligned}&\textbf{instance}\ Multiplicable\ Int\ \textbf{where}\\&\quad multiply\ x\ y = x * y\end{aligned}$$

Now if we want to use this new contract on a generic function f, we require a $Multiplicable$ instance for every type that we want to use $multiply$ on inside the function:

$$\begin{aligned}&f :: Multiplicable\ a \Rightarrow a \to a \to a\\&f\ x\ y = multiply\ (multiply\ x\ y)\ x\end{aligned}$$

Such a function f does work with the contract of $Multiplicable$ instead of requiring some specific type. This way we can reuse many definitions in Haskell even though it is a statically typed language.

In Haskell we can also write type classes with more than one type parameter. This allows for encapsulation of contracts of arbitrary complexity. Furthermore type classes can itself have constraints placed on what types are allowed. Both can be seen here:

$$\begin{aligned}&\textbf{class}\ (SomeClass\ a, SomeOtherClass\ b) \Rightarrow MyClass\ a\ b\ c\ \textbf{where}\\&\quad ...\end{aligned}$$

Lazy Evaluation

One thing that is not obvious when looking at the definitions from this chapter is that Haskell is a lazy language[6]. This means that values are only evaluated when required. This has one major benefit: We get a Producer/Consumer pattern behaviour for free. For example if we have the lazy function $producer :: Int \to [Int]$ producing some list of integers and some consumer consuming $consumer :: [Int] \to Int$ this list. Then, in a program $consumer \circ producer$, $producer$ generates the elements of the result-list as they are consumed. This also means that, if $consumer$ only requires the first few elements of the list to compute the result, $consumer$ does not produce unneeded results.

Laziness even allows us to express infinite streams, which can be helpful in some cases. As an example, an infinite list of ones is defined as

$$
\begin{aligned}
&ones :: [Int] \\
&ones = 1 : ones
\end{aligned}
$$

or, if we require a list of some value at least n times so that it can be consumed with some list of length n, we can just use an infinite list instead of computing the actual required amount (which would take n steps for a linked list). The helper function for this is called $repeat$ and can be written as

$$
\begin{aligned}
&repeat :: a \to [a] \\
&repeat\ a = a : (repeat\ a)
\end{aligned}
$$

Another good example where Laziness simplifies things is when branching is involved:

$$
\begin{aligned}
&calculateStuff :: [Int] \to Int \\
&calculateStuff = \textbf{if} < someCondition > \\
&\quad \textbf{then}\ doStuff\ list1\ list2 \\
&\quad \textbf{else}\ doSomeOtherStuff\ list1 \\
&\qquad \textbf{where} \\
&\qquad\quad list1 = ... \\
&\qquad\quad list2 = ...
\end{aligned}
$$

[6] Haskell is actually defined as a non-strict language, meaning that only as much as required is evaluated, not when it is done. Laziness is just a way to achieve non-strictness. The same could be achieved with an eager, but non-strict evaluation mechanism. But as Haskell's main compilers all implement non-strictness via lazy evaluation, it is okay to call Haskell a lazy language here. See `https://wiki.haskell.org/Lazy_vs._non-strict`

Here, *list2* is not required in both branches of the **if** statement. Thanks to laziness it is therefore only evaluated upon a successful if-check. While such a behaviour is obviously possible in non-lazy languages the elegance of the above definition is apparent. We can define as many variables in the same clear way without having unnecessary computations or code dealing with conditional computation like nested **wheres**.

Usually laziness is beneficial to programs and programmers as it allows for easy composition and better structure in code, but sometimes we require more control about when something is evaluated. Haskell has several ways to control when and how values are evaluated. The basic primitive to force values is $seq :: a \to b \to b$, which is by nature part of the compiler and can not be expressed in Haskell directly. It's semantics however, are as follows: We tell the compiler that the first argument (of type a) is to be evaluated before the second argument. For example, in an expression like

```
myFun :: Int -> (Int, Int)
myFun x = let y = f x in y `seq` g y
    where
        f = ...
        g = ...
```

we can then hint to the compiler that we want $y = f\ x$ evaluated before returning the (still non-evaluated) result of $g\ y$. This trick is usually used if during profiling a big chunk of non-evaluated values are noticed to aggregate before or in the process of evaluation of $f\ x$. As this is a common pattern seen in Haskell programs, there exists the strict function application operator $(\$!) :: (a \to b) \to a \to b$ to encapsulate it. It is straightforwardly defined as:

$$(\$!) :: (a \to b) \to a \to b$$
$$f\ \$!\ x = x\ `seq`\ f\ x$$

With it we can then write our example function as

```
myFun :: Int -> (Int, Int)
myFun x = g $! f x
    where
        f = ...
        g = ...
```

These two operations do not *completely* evaluate values, however as they only force to weak-head-normal-form (WHNF) meaning that evaluation is only forced until the outermost constructor in contrast to normal-form (NF) which stands for full evaluation. This means that if we were to evaluate some calculation $f\ (g\ (h\ (i\ x)))$ embedded in some lazy tuple (y, z) to WHNF, y and z would not be touched as the evaluation stops at the tuple constructor (for more about constructors see the next section, „Custom types“). All the computations to get to that constructor however, are forced to be evaluated. Therefore, if we want to make the insides of a tuple strict, we would have to write something along the lines of

```
let tup @ (y, z) = f (g (h (i x))) in y `seq` z `seq` tup
```

instead of just

```
let tup = f (g (h (i x))) in y `seq` y
```

But as seq and $! both only evaluate to WHNF, y and z might still not be completely evaluated, since they could be of some more complex type than just Int or any other primitive. This is the reason why in the Haskell eco system, there exists the library $deepseq$[7] which comes with the typeclass $NFData$ defined as

$$\begin{aligned}&\textbf{class}\ NFData\ a\ \textbf{where}\\&\quad rnf :: a \to ()\end{aligned}$$

Instances of this typeclass for some type a are required to provide an appropriate implementation of rnf for *full* evaluation to normal-form, where rnf stands for „reduce-to-normal-form“. With this we can then implement the NF equivalent to seq, $deepseq$, as

$$\begin{aligned}&deepseq :: NFData\ a \Rightarrow a \to b \to b\\&deepseq\ a = rnf\ a\ `seq`\ a\end{aligned}$$

A deep analogue to $!! is then easily definable as well as

$$\begin{aligned}&(\$!) :: NFData\ a \Rightarrow (a \to b) \to a \to b\\&f\ \$!\ x = x\ `deepseq`\ f\ x\end{aligned}$$

[7] see `haskell.org/package/deepseq-1.4.3.0/docs/Control-DeepSeq.html`.

When dealing with WHNF and NF, note that in all computations annotated with some forcing construct, be it seq or $deepseq$, laziness does go away entirely. All forced values, even the ones forced to NF, can still be considered somewhat lazy as they are only forced when they are requested. This is in practice, however, usually a desired property in Haskell programs.

Custom types

As in any mature programming language, in Haskell programmers obviously do not have to represent everything with only some base-set of types. Types are usually defined in three different ways. For starters, we can give types aliases with the **type** keyword like

$$
\begin{aligned}
&\text{-- Tuple of a and b}\\
&\mathbf{type}\ Tuple\ a\ b = (a, b)\\
&\text{-- Tuple of ints}\\
&\mathbf{type}\ IntTuple = (Int, Int)
\end{aligned}
$$

, which are treated just like original (a, b) or Int, Int would. This means, we can use such types loosely and pass e.b. a $Tuple\ Int\ Int$ into a function $f :: (Int, Int) \rightarrow ...$. The same also holds for typeclasses.

The second way to declare types, **data** however declares new-types as in actual new types in the type system like

$$
\begin{aligned}
&\mathbf{data}\ Direction =\\
&\quad North\\
&\quad \mid NorthEast\\
&\quad \mid East\\
&\quad \mid SouthEast\\
&\quad \mid South\\
&\quad \mid SouthWest\\
&\quad \mid West\\
&\quad \mid NorthWest
\end{aligned}
$$

, where $North$ - $NorthWest$ are called constructors.

data types are not limited to enum-style types though, they can also hold values, like the $Maybe\ a$ type from Haskell. This type - which *may* hold a value a internally

- is usually used as a return type for functions which not always return an actual result. We can define it as follows:

$$
\begin{aligned}
&\text{-- unnamed field}\\
&\mathbf{data}\ Maybe\ a = Just\ a \mid Nothing
\end{aligned}
$$

where values are created by calling the constructors with the appropriate parameters (if any), i.e. when passed into a function: $f\ (Just\ 1)$. Furthermore, **data** constructors can have named fields defined like

$$
\begin{aligned}
&\text{-- named field}\\
&\mathbf{data}\ Maybe\ a =\\
&\quad Just\ \{theThing :: a\}\\
&\quad \mid Nothing
\end{aligned}
$$

where values are created by calling the constructor and passing the appropriate parameters to the properties, i.e. $f\ (Maybe\ \{theThing = 1\})$ The final way to define custom types is via **newtype**:

$$
\begin{aligned}
&\text{-- unnamed field}\\
&\mathbf{newtype}\ MyNewType\ a = Constructor\ a\\
&\text{-- named field}\\
&\mathbf{newtype}\ MyOtherNewType\ a = Constructor\ \{myOnlyThing :: a\}
\end{aligned}
$$

Types declared this way are similar to **data** types, but can only contain a single constructor with just a single field. Also, unlike **data**, constructors declared with **newtype** are strict, meaning the compiler can optimize away the surrounding declaration. Everything else is handled exactly like with **data** types. **newtype** types are also a useful tool if we were to write a wrapper for a type while not wanting to inherit all instances of typeclasses, but are also often used when declaring more complicated types.

Pattern Matching

While we have seen pattern matching as an alternative to $\mathbf{if} \ldots \mathbf{then} \ldots \mathbf{else}$ and guard statements, it can do more things. For example, if we have a datatype $MyType\ a$ type defined as

$$\mathbf{data}\ MyType\ a = SomeConstructor\ a \mid SomeOtherConstructor\ a$$

and we want to write a function $unwrap :: MyType\ a \to a$ to unwrap the a value, we use pattern matching like this:

$$
\begin{aligned}
&unwrap :: MyType\ a \to a \\
&unwrap\ (SomeConstructor\ x) = x \\
&unwrap\ (SomeOtherConstructor\ x) = x
\end{aligned}
$$

This type of unwrapping can also be done with the help of **case** statements so that we do not require a new function definition:

$$
\begin{aligned}
&someFunc :: MyType\ a \to a \\
&someFunc\ t = \mathbf{case}\ t\ \mathbf{of} \\
&\quad (SomeConstructor\ a) \to ... \\
&\quad (SomeOtherConstructor\ a) \to ...
\end{aligned}
$$

In Haskell programs we can also write unwrapping code with the help of the **let** notation for single constructor types like $\mathbf{let}\ (x, y) = vec2d\ \mathbf{in}\ sqrt\ (x * x + y * y)$. Predictably, we can do this with **where** as well with constructs like $\mathbf{where}\ (x, y) = vec2d$.

Sometimes we only care about some part of the value. For example, in a definition of $maybeHead :: [a] \to Maybe\ a$, which should return the first element of the list or $Nothing$ if it is an empty list, we can write this with the help of wildcards (_) as:

$$
\begin{aligned}
&maybeHead :: [a] \to Maybe\ a \\
&maybeHead\ (x : \_) = Just\ x \\
&maybeHead\ [\,] = Nothing
\end{aligned}
$$

We could even write

$$
\begin{aligned}
&maybeHead :: [a] \to Maybe\ a \\
&maybeHead\ (x : \_) = Just\ x \\
&maybeHead\ \_ = Nothing
\end{aligned}
$$

as the second equation will only ever match when the list is empty.[8] Furthermore, in Haskell, functions $isJust :: Maybe\ a \to Bool$ when they only care about the structure of the type, can be written with only with wildcards:

$$
\begin{aligned}
&isJust :: Maybe\ a \to Bool \\
&isJust\ (Just\ \_) = True \\
&isJust\ \_ = False
\end{aligned}
$$

[8] A single element list has two forms in Haskell, $a : [\,]$ and $[a]$ of which the latter is just syntactic sugar of the former.

Additionally, if we want to preserve laziness and we are 100% sure that a match will work (e.g. if we have called $isJust$), we can use irrefutable patterns like $\sim(Just\ a) = someMaybe$.

Lambdas and Partial application

As Functions are just another type that can be passed into higher-order functions it makes sense to have a short-hand to write anonymous functions - lambdas. In Haskell they look like this:

$$\lambda(a, b) \to a + b$$

This can easily be passed into functions, like zipWith^ [While $(\lambda(a, b) \to a + b)$ is obviously the same as $(+)$, we just write it as a lambda here for demonstration purposes]:

$$someFunc :: [Int] \to [Int] \to [Int]$$
$$someFunc\ xs\ ys = zipWith\ (\lambda(a, b) \to a + b)\ xs\ ys$$

Here, we notice the reason for yet another feature in Haskell that is commonly used: Partial application. While the definition of $someFunc$ is definitely not wrong, we could have written it more elegantly as

$$someFunc :: [Int] \to [Int] \to [Int]$$
$$someFunc = zipWith\ (\lambda(a, b) \to a + b)$$

where we this means that $someFunc$ is defined as $zipWith :: (a \to b \to c) \to [a] \to [b] \to [c])$ partially applied with the passed lambda to get a function with type $[a] \to [b] \to [c]$ which the compiler then automatically binds to the type of $someFunc :: [Int] \to [Int] \to [Int]$.

3.1.3 Monads

Functional programmers try to avoid mutable state at all cost, but programs that do not only just compute some function usually involve some sort of it. So, doesn't this make Haskell useless being be a pure functional language without *any* mutable state? No. Functional Programs generally just avoid *unnecessary* mutable state at all cost. The fact of the matter is that in functional programming, we can represent mutable state as well, but we do so in a meaningful and controlled manner.

While in most computations, we could represent state by passing it into every function that can possibly change it and returning it alongside of the actual returned value like

$$
\begin{aligned}
&comp :: MyState \to Int \to (Int, MyState)\\
&comp\ curState\ x = (x + 3, nextState)\\
&\quad \textbf{where}\ nextState = changeState\ curState
\end{aligned}
$$

this can become unnecessarily complicated to handle by hand. A better alternative is the use of monads, which are the main concept generally used in computations involving some sort of mutable state. The typeclass for the $Monad$ typeclass can be defined as

$$
\begin{aligned}
&\textbf{class}\ Monad\ m\ \textbf{where}\\
&\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\\
&\quad (\gg) :: m\ a \to m\ b \to m\ b\\
&\quad m \gg k = m \gg\!= \backslash\_ \to k\\
&\quad return :: a \to m\ a
\end{aligned}
$$

Thinking of Monads as computations, we can come up with the following explanation: $return$ is used to create a computation $m\ a$ just returning some given value a. Next, $(\gg\!=)$ is used to compose some monadic computation $m\ a$ returning some a with a monadic function $a \to m\ b$ to return some computation $m\ b$ returning some b. Finally, $(\gg)$ is used to define the order of two monadic computations $m\ a$ and $m\ b$ so that $m\ a$ is computed before $m\ b$ while discarding the result of the first one as can also be seen in its default implementation above.

Given this definition of a Monad, we can now take a look at how we would implement a $State$ Monad. It is defined as (Michaelson, 2013)

$$
\textbf{newtype}\ State\ s\ a = State\ \{runState :: s \to (a, s)\}
$$

where a $State\ s\ a$ encapsulates a stateful computation on some state type s yielding some value of type a. For easier understanding it is often useful to think of $State\ s\ a$ just as a usability wrapper around a function $s \to (a, s)$ that returns some a and the final state s if we pass it some starting state s. The State monad therefore only contains the „blueprint“ of the computation that can only be run if we start it by passing a state. The instance for the Monad type class can then be defined as

$$
\begin{aligned}
&\textbf{instance}\ Monad\ (State\ s)\ \textbf{where}\\
&\quad (State\ h) \gg\!= f = State\ \$\ \lambda s \to \textbf{let}\ (a, newState) = h\ s\\
&\qquad\quad (State\ g) = f\ a
\end{aligned}
$$

$$
\begin{aligned}
&\quad \textbf{in } g\ newState \\
&return\ x = State\ \{ runState = \lambda s \rightarrow (x, s) \}
\end{aligned}
$$

where we declare the Monad deliberately on top of $State\ s$ meaning that $State$ itself is not a monad, but it is a monad together with some state representation s.[9] Note how the operations are defined here: $return$ encapsulates the given value $x :: a$ inside the internal function and therefore is equal to the identity $id :: a \rightarrow a$ function on tuples with one parameter already applied.[10] In the composition operator $\gg\!=$, the monadic computation $State\ h :: State\ s\ a$ is composed with the function $f :: a \rightarrow State\ s\ b$ into a new monadic computation of type $State\ s\ b$. The internal function of the state is essentially taken out of the first argument and composed with the second argument inside the returned Monad.

Additionally, we have helper operations to use this construct with. The first is $put :: s \rightarrow State\ s\ ()$ which overwrites the current state returning a unit $()$ as result:

$$
\begin{aligned}
&put :: s \rightarrow State\ s\ () \\
&put\ newState = State\ \{ runState = \lambda s \rightarrow ((), newState) \}
\end{aligned}
$$

The second one is $get :: State\ s\ s$ which returns the current state, but does not change it:

$$
\begin{aligned}
&get :: State\ s\ s \\
&get = State\ \{ runState = \lambda s \rightarrow (s, s) \}
\end{aligned}
$$

With these operations, we can easily write stateful programs like this one:[11]

$$
\begin{aligned}
&\textbf{type } Stack = [Int] \\
&empty :: Stack \\
&empty = [\,] \\
&pop :: State\ Stack\ Int \\
&pop = get \gg\!= (\lambda(x : xs) \rightarrow put\ xs \gg return\ x) \\
&push :: Int \rightarrow State\ Stack\ () \\
&push\ a = State\ \$\ \lambda xs \rightarrow ((), a : xs) \\
&peek :: State\ Stack\ Int \\
&peek = get \gg\!= \lambda(x : xs) \rightarrow return\ x \\
&computeStateful :: State\ Stack\ Int
\end{aligned}
$$

[9] We can't declare $State$ a monad anyways since the Monad is a type class with just one type parameter

[10] With the help of $curry :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$, we could have therefore also written $return\ x = State\ \{ runState = (curry\ id)\ x \}$

[11] inspired and adapted from `https://gist.github.com/sdiehl/8d991a718f7a9c80f54b`

```
computeStateful = push 10 >>
  push 20 >>
  pop >>= λa →
    (pop >>= λb → push (a + b)) >>
  peek
  -- main program inside the IO monad
main :: IO ()
main = print (evalState computeStateful empty)
```

Here, *computeStateful* first pushes some values on top of a stack represented by a list $[Int]$ (the actual state inside of the *State* monad) and then *pop*s these values and *push*es their sum back on the stack to finally *peek* the actual value which then is the result of the computation. To make writing such code easier, Haskell has syntactic sugar called **do** notation. With it we can write the above method *computeStateful* in a way that resembles imperative-style code (but with side-effects clearly encapsulated) as:

```
computeStateful :: State Stack Int
computeStateful = do
  push 10
  push 20
  a ← pop
  b ← pop
  push (a + b)
  peek
```

In this example, we can also see the direct relationship between $(\gg)$ and simple new lines and the $(\gg=)$ operator and the special $\leftarrow$ operator in do notation which facilitates the binding to a variable.[12]

Other often used Monads in the Haskell eco-system include the *Writer* monad, which is useful for e.g. logging or the *IO* monad, which is used to encapsulate I/O computations as well as low level internal operations such as modifiable variables *IORef* or *MVar* among others. Furthermore, as one of many other applications, Monads are used in some parallel Haskells as we will see later in this thesis.

3.1.4 Arrows

Arrows were introduced by Hughes (2000) as a general interface for computation and a less restrictive generalisation of Monads. Hughes (2000) motivates the broader

[12] $(\gg=)$ is also often called *bind* in languages which do not support operator overloading.

$$
\begin{aligned}
&\textbf{class } \mathit{Arrow}\ \mathit{arr}\ \textbf{where}\\
&\quad \mathit{arr} :: (a \to b) \to \mathit{arr}\ a\ b\\
&\quad (\ggg) :: \mathit{arr}\ a\ b \to \mathit{arr}\ b\ c \to \mathit{arr}\ a\ c\\
&\quad \mathit{first} :: \mathit{arr}\ a\ b \to \mathit{arr}\ (a, c)\ (b, c)\\
&\textbf{instance } \mathit{Arrow}\ (\to)\ \textbf{where}\\
&\quad \mathit{arr}\ f = f\\
&\quad f \ggg g = g \circ f\\
&\quad \mathit{first}\ f = \lambda(a, c) \to (f\ a, c)\\
&\textbf{data } \mathit{Kleisli}\ m\ a\ b = \mathit{Kleisli}\ \{\mathit{run} :: a \to m\ b\}\\
&\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{Arrow}\ (\mathit{Kleisli}\ m)\ \textbf{where}\\
&\quad \mathit{arr}\ f = \mathit{Kleisli}\ (\mathit{return} \circ f)\\
&\quad f \ggg g = \mathit{Kleisli}\ (\lambda a \to f\ a \gg\!= g)\\
&\quad \mathit{first}\ f = \mathit{Kleisli}\ (\lambda(a, c) \to f\ a \gg\!= \lambda b \to \mathit{return}\ (b, c))
\end{aligned}
$$

Figure 3.5: The *Arrow* type class and its two most typical instances.

interface of Arrows with the example of a parser with added static meta-information that can not satisfy the monadic bind operator $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$ (with m being a Monad).[13]

An Arrow $\mathit{arr}\ a\ b$ represents a computation that converts an input a to an output b. This is defined in the *Arrow* type class shown in Fig. 3.5. To lift an ordinary function to an Arrow, *arr* is used, analogous to the monadic *return*. Similarly, the composition operator $\ggg$ is analogous to the monadic composition $\gg\!=$ and combines two Arrows $\mathit{arr}\ a\ b$ and $\mathit{arr}\ b\ c$ by „wiring“ the outputs of the first to the inputs to the second to get a new Arrow $\mathit{arr}\ a\ c$. Lastly, the *first* operator takes the input Arrow $\mathit{arr}\ a\ b$ and converts it into an Arrow on pairs $\mathit{arr}\ (a, c)\ (b, c)$ that leaves the second argument untouched. It allows us to to save input across Arrows. Fig. 3.6 shows a graphical representation of these basic Arrow combinators. The most prominent instances of this interface (Fig. 3.5) are regular functions $(\to)$ and the Kleisli type, which wraps monadic functions, e.g. $a \to m\ b$.

Hughes also defined some syntactic sugar (Fig.3.7): *second*, $***$ and &&&. *second* is the mirrored version of *first* (Appendix 10.1). The $***$ function combines *first* and *second* to handle two inputs in one arrow, and is defined as follows:

$$
\begin{aligned}
&(***) :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ c\ d \to \mathit{arr}\ (a, c)\ (b, d)\\
&f *** g = \mathit{first}\ f \ggg \mathit{second}\ g
\end{aligned}
$$

[13] In the example a parser of the type $\mathit{Parser}\ s\ a$ with static meta information s and result a is shown to not be able to use the static s without applying the monadic function $a \to m\ b$. With Arrows this is possible.

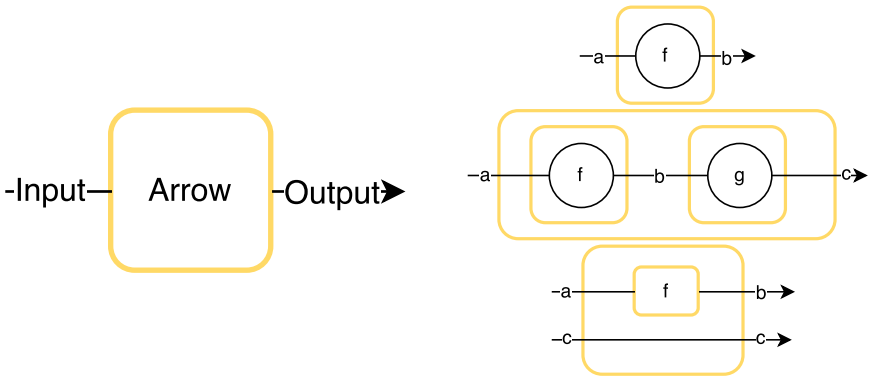

Figure 3.6: Schematic depiction of an Arrow (left) and its basic combinators arr, $\ggg$ and $first$ (right).

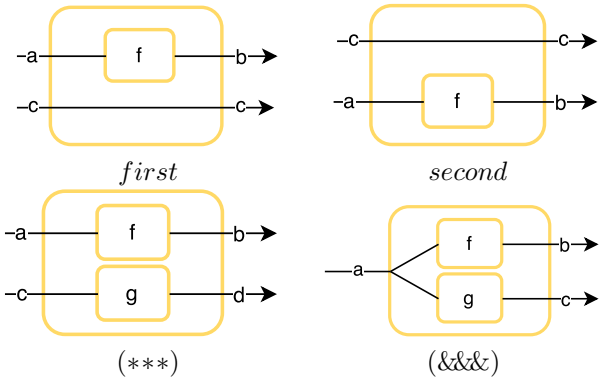

Figure 3.7: Visual depiction of syntactic sugar for Arrows.

The $\&\&\&$ combinator, which constructs an Arrow that outputs two different values like $***$, but takes only one input, is:

$$(\&\&\&) :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ a\ c \rightarrow arr\ a\ (b, c)$$
$$f\ \&\&\&\ g = arr\ (\lambda a \rightarrow (a, a)) \ggg f *** g$$

A first short example given by Hughes on how to use Arrows is addition with Arrows:

$$add :: Arrow\ arr \Rightarrow arr\ a\ Int \rightarrow arr\ a\ Int \rightarrow arr\ a\ Int$$
$$add\ f\ g = f\ \&\&\&\ g \ggg arr\ (\lambda(u, v) \rightarrow u + v)$$

As we can rewrite the monadic bind operation $(\gg=)$ with only the Kleisli type into $m\ a \rightarrow Kleisli\ m\ a\ b \rightarrow m\ b$, but not with a general Arrow $arr\ a\ b$, we can intuitively get an idea of why Arrows must be a generalisation of Monads. While this also means that a general Arrow can not express everything a Monad can, Hughes (2000) shows in his parser example that this trade-off is worth it in some cases.

In this thesis we will show that parallel computations can be expressed with this more general interface of Arrows without requiring Monads (we will see an example of monadic parallelism in Chapter 3.2). We also do not restrict the compatible Arrows to ones which have $ArrowApply$ instances but instead only require instances for $ArrowChoice$ (for if-then-else constructs) and $ArrowLoop$ (for looping). Because of this, we have a truly more general interface as compared to a monadic one.

While we could have based our DSL on Profunctors as well, we chose Arrows in this thesis since they they allow for a more direct way of thinking about parallelism than general Profunctors because of their composability. However, they are a promising candidate for future improvements of our DSL. Some Profunctors, especially ones supporting a composition operation, choice, and looping, can already be adapted to our interface as shown in Appendix 10.2.

3.2 Short introduction to parallel Haskells

In 3.1, we cited Hughes (1990) saying that in functional programming, the order of evaluation is irrelevant. In parallel programs this is not the case, as at least some kind of structure of evaluation is required to have actual speedup in programs. In the following we will take a look at how parallelism can be achieved in Haskell programs in general. Now, one might think that we would want side effects (parallel evaluation is a side-effect) and require the need to think about order of evaluation in

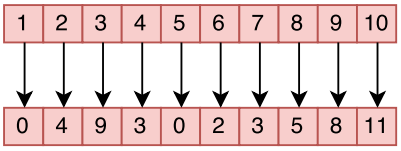

Figure 3.8: Schematic illustration of $parEvalN$. A list of inputs is transformed by different functions in parallel.

a pure functional program seems a bit odd. The fact of the matter is that functional programs only aim to avoid *unnecessary* side-effects and in the case of parallelism it is obvious that some amount of side-effects are required. Also, parallel Haskells generally aim to encapsulate all the necessary and complicated code in a way such that the room for code-breaking errors is almost impossible. If some parallel evaluation code is written in a sub-optimal way, only the performance is affected, but not the result, which will always be tractable no matter the order of evaluation.[14]

In its purest form, parallel computation (on functions) can be looked at as the execution of some functions $a \to b$ in parallel or $parEvalN :: [a \to b] \to [a] \to [b]$, as also Fig. 3.8 symbolically shows. In this section, we will implement this non-Arrow version which will later be adapted for usage in our Arrow-based parallel Haskell.

There exist several parallel Haskells already. Among the most important are probably GpH (based on par and $pseq$ „hints“, Trinder et al. (1996), Trinder et al. (1998)), the Par Monad (a Monad for deterministic parallelism, Marlow et al. (2011), Foltzer et al. (2012)), Eden (a parallel Haskell for distributed memory, Loogen et al. (2005), Loogen (2012)), HdpH (a Template Haskell-based parallel Haskell for distributed memory, Maier et al. (2014), Stewart (2016)) and LVish (a Par extension with focus on communication, Kuper et al. (2014)).

As the goal of this thesis is not to re-implement yet another parallel runtime, but to represent parallelism with Arrows, we base our efforts on existing work which we wrap as backends behind a common interface. For this thesis we chose GpH for its simplicity, the Par Monad to represent a monadic DSL, and Eden as a distributed parallel Haskell.

LVish and HdpH were not chosen as the former does not differ from the original Par Monad with regard to how we would have used it in this thesis, while the latter (at least in its current form) does not comply with our representation of parallelism due to its heavy reliance on Template Haskell.

[14] Some exceptions using unsafe and non-deterministic operations exist, though. These situations can however only be achieved if the programmer actively chooses to use these kinds of operations.

We will now go into some detail on GpH, the *Par* Monad and Eden, and also give their respective implementations of the non-Arrow version of *parEvalN*.

3.2.1 Glasgow parallel Haskell – GpH

GpH (Marlow et al., 2009, Trinder et al. (1998)) is one of the simplest ways to do parallel processing found in standard GHC.[15] Besides some basic primitives (*par* and *pseq*), it ships with parallel evaluation strategies for several types which can be applied with $using :: a \to Strategy\ a \to a$, which is exactly what is required for an implementation of *parEvalN*.

$$
\begin{aligned}
&parEvalN :: (NFData\ b) \Rightarrow [a \to b] \to [a] \to [b] \\
&parEvalN\ fs\ as = \mathbf{let}\ bs = zipWith\ (\$)\ fs\ as \\
&\quad \mathbf{in}\ bs\ `using`\ parList\ rdeepseq
\end{aligned}
$$

In the above definition of *parEvalN* we just apply the list of functions $[a \to b]$ to the list of inputs $[a]$ by zipping them with the application operator \$. We then evaluate this lazy list $[b]$ according to a $Strategy\ [b]$ with the $using :: a \to Strategy\ a \to a$ operator. We construct this strategy with $parList :: Strategy\ a \to Strategy\ [a]$ and $rdeepseq :: NFData\ a \Rightarrow Strategy\ a$ where the latter is a strategy which evaluates to normal form. Other strategies like e.g. evaluation to weak head normal form are available as well. It also allows for custom *Strategy* implementations to be used. Fig. 3.9 shows a visual representation of this code.

[15] The Multicore implementation of GpH is available on Hackage under `https://hackage.haskell.org/package/parallel-3.2.1.0`, compiler support is integrated in the stock GHC.

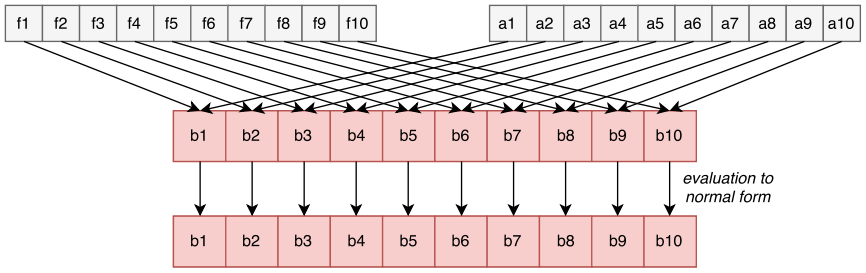

Figure 3.9: *parEvalN* (GpH).

3.2.2 Par Monad

The Par Monad[16] introduced by (Marlow et al., 2011), is a Monad designed for composition of parallel programs. Let:

$$
\begin{aligned}
&parEvalN :: (NFData\ b) \Rightarrow [a \to b] \to [a] \to [b] \\
&parEvalN\ fs\ as = runPar\ \$ \\
&\quad (sequenceA\ (map\ (return \circ spawn)\ (zipWith\ (\$)\ fs\ as))) \gg\!= mapM\ get
\end{aligned}
$$

The Par Monad version of our parallel evaluation function $parEvalN$ is defined by zipping the list of $[a \to b]$ with the list of inputs $[a]$ with the application operator $ just like with GpH. Then, we map over this not yet evaluated lazy list of results $[b]$ with $spawn :: NFData\ a \Rightarrow Par\ a \to Par\ (IVar\ a)$ to transform them to a list of not yet evaluated forked away computations $[Par\ (IVar\ b)]$, which we convert to $Par\ [IVar\ b]$ with $sequenceA$. We wait for the computations to finish by mapping over the $IVar\ b$ values inside the Par Monad with get. This results in $Par\ [b]$. We execute this process with $runPar$ to finally get the fully evaluated list of results$[b]$. While we used $spawn$ in the definition above, a head-strict variant can easily be defined by replacing $spawn$ with $spawn\_ :: Par\ a \to Par\ (IVar\ a)$. Fig. 3.10 shows a graphical representation.

[16]The Par Monad can be found in the $monad - par$ package on Hackage under `https://hackage.haskell.org/package/monad-par-0.3.4.8/`.

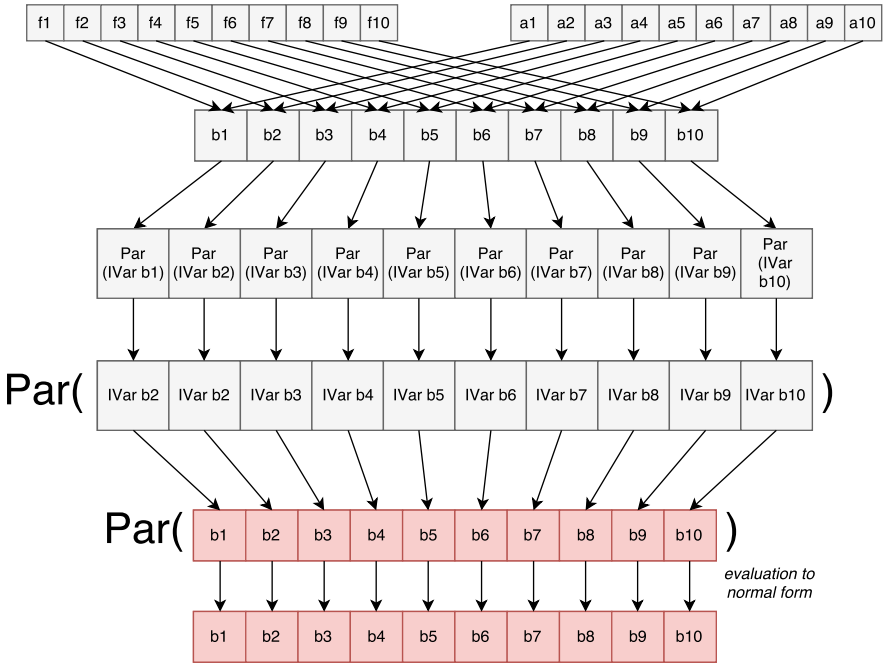

Figure 3.10: $parEvalN$ (Par Monad).

3.2.3 Eden

Eden (Loogen et al., 2005, Loogen (2012)) is a parallel Haskell for distributed memory and comes with MPI and PVM as distributed backends.[17] It is targeted towards clusters, but also functions well in a shared-memory setting with a further simple backend. However, in contrast to many other parallel Haskells, in Eden each process has its own heap. This seems to be a waste of memory, but with distributed programming paradigm and individual GC per process, Eden yields good performance results on multicores, as well (Berthold et al., 2009a, Aswad et al. (2009)).

While Eden comes with a Monad PA for parallel evaluation, it also ships with a completely functional interface that includes a $spawnF :: (Trans\ a, Trans\ b) \Rightarrow [a \to b] \to [a] \to [b]$ function that allows us to define $parEvalN$ directly:

$$parEvalN :: (Trans\ a, Trans\ b) \Rightarrow [a \to b] \to [a] \to [b]$$
$$parEvalN = spawnF$$

Eden TraceViewer

To comprehend the efficiency and the lack thereof in a parallel program, an inspection of its execution is extremely helpful. While some large-scale solutions

[17] The projects homepage can be found at `http://www.mathematik.uni-marburg.de/~eden/`. The Hackage page is at `https://hackage.haskell.org/package/edenmodules-1.2.0.0/`.

exist (Geimer et al., 2010), the parallel Haskell community mainly utilises the tools Threadscope (Wheeler and Thain, 2009) and Eden TraceViewer[18] (Berthold and Loogen, 2007). In the next sections we will present some *trace visualisations*, the post-mortem process diagrams of Eden processes and their activity.

The trace visualisations are colour-coded. In such a visualisation (Fig. 5.1), the x axis shows the time, the y axis enumerates the machines and processes. The visualisation shows a running process in green, a blocked process is red. If the process is „runnable“, i.e. it may run, but does not, it is yellow. The typical reason for this is GC. An inactive machine, where no processes are started yet, or all are already terminated, shows as a blue bar. A communication from one process to another is represented with a black arrow. A stream of communications, e.g. a transmitted list is shows as a dark shading between sender and receiver processes.

[18] See `http://hackage.haskell.org/package/edentv` on Hackage for the last available version of Eden TraceViewer.

4 Parallel Arrows

While Arrows are a general interface to computation, we introduce here specialised Arrows as a general interface to *parallel computations*. We present the $ArrowParallel$ type class and explain the reasoning behind it before discussing some parallel Haskell implementations and basic extensions.

4.1 The $ArrowParallel$ type class

A parallel computation (on functions) can be seen as execution of some functions $a \rightarrow b$ in parallel, as our $parEvalN$ prototype shows (Chapter 3.2). Translating this into Arrow terms gives us a new operator $parEvalN$ that lifts a list of Arrows $[arr\ a\ b]$ to a parallel Arrow $arr\ [a]\ [b]$. This combinator is similar to the evaluation combinator $evalN$ from Appendix 10.1, but does parallel instead of serial evaluation.

$$parEvalN :: (Arrow\ arr) \Rightarrow [arr\ a\ b] \rightarrow arr\ [a]\ [b]$$

With this definition of $parEvalN$, parallel execution is yet another Arrow combinator. But as the implementation may differ depending on the actual type of the Arrow arr - or even the input a and output b - and we want this to be an interface for different backends, we introduce a new type class $ArrowParallel\ arr\ a\ b$:

$$\begin{aligned}&\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ \textbf{where}\\&\quad parEvalN :: [arr\ a\ b] \rightarrow arr\ [a]\ [b]\end{aligned}$$

Sometimes parallel Haskells require or allow for additional configuration parameters, e.g. an information about the execution environment or the level of evaluation (weak head normal form vs. normal form). For this reason we introduce an additional $conf$ parameter as we do not want $conf$ to be a fixed type, as the configuration parameters can differ for different instances of $ArrowParallel$.

$$\begin{aligned}&\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ conf\ \textbf{where}\\&\quad parEvalN :: conf \rightarrow [arr\ a\ b] \rightarrow arr\ [a]\ [b]\end{aligned}$$

By restricting the implementations of our backends to a specific $conf$ type, we also get interoperability between backends for free. We can parallelize one part of a program using one backend, and parallelize the next with another one.

4.2 $ArrowParallel$ instances

With the type class defined, we will now give implementations of it with GpH, the Par Monad and Eden.

4.2.1 Glasgow parallel Haskell

The GpH implementation of $ArrowParallel$ is implemented in a straightforward manner in Fig. 4.1, but a bit different compared to the variant from Chapter 3.2.1. We use $evalN :: [arr\ a\ b] \to arr\ [a]\ [b]$ (definition in Appendix 10.1, think $zipWith\ (\$)$ on Arrows) combined with $withStrategy :: Strategy\ a \to a \to a$ from GpH, where $withStrategy$ is the same as $using :: a \to Strategy\ a \to a$, but with flipped parameters. Our $Conf\ a$ datatype simply wraps a $Strategy\ a$, but could be extended in future versions of our DSL.

$$
\begin{aligned}
&\mathbf{data}\ Conf\ a = Conf\ (Strategy\ a)\\
&\mathbf{instance}\ (ArrowChoice\ arr) \Rightarrow\\
&\quad ArrowParallel\ arr\ a\ b\ (Conf\ b)\ \mathbf{where}\\
&\qquad parEvalN\ (Conf\ strat)\ fs =\\
&\qquad\quad evalN\ fs \ggg\\
&\qquad\quad arr\ (withStrategy\ (parList\ strat))
\end{aligned}
$$

Figure 4.1: GpH $ArrowParallel$ instance.

4.2.2 Par Monad

As for GpH we can easily lift the definition of $parEvalN$ for the Par Monad to Arrows in Fig. 4.2. To start off, we define the $Strategy\ a$ and $Conf\ a$ type so we can have a configurable instance of ArrowParallel:

$$
\begin{aligned}
&\mathbf{type}\ Strategy\ a = a \to Par\ (IVar\ a)\\
&\mathbf{data}\ Conf\ a = Conf\ (Strategy\ a)
\end{aligned}
$$

Now we can once again define our $ArrowParallel$ instance as follows: First, we convert our Arrows $[arr\ a\ b]$ with $evalN\ (map\ (\ggg arr\ strat)\ fs)$ into an Arrow $arr\ [a]\ [(Par\ (IVar\ b))]$ that yields composable computations in the Par

monad. By combining the result of this Arrow with $arr\ sequenceA$, we get an Arrow $arr\ [a]\ (Par\ [IVar\ b])$. Then, in order to fetch the results of the different threads, we map over the $IVar$s inside the Par Monad with $arr\ (\gg\!= mapM\ get)$ – our intermediary Arrow is of type $arr\ [a]\ (Par\ [b])$. Finally, we execute the computation $Par\ [b]$ by composing with $arr\ runPar$ and get the final Arrow $arr\ [a]\ [b]$.

```
instance (ArrowChoice arr) ⇒ ArrowParallel arr a b (Conf b) where
  parEvalN (Conf strat) fs =
    evalN (map (>>> arr strat) fs) >>>
    arr sequenceA >>>
    arr (>>= mapM Control.Monad.Par.get) >>>
    arr runPar
```

Figure 4.2: Par Monad $ArrowParallel$ instance.

4.2.3 Eden

For both the GpH Haskell and Par Monad implementations we could use general instances of $ArrowParallel$ that just require the $ArrowChoice$ type class. With Eden this is not the case as we can only spawn a list of functions, which we cannot extract from general Arrows. While we could still manage to have only one instance in the module by introducing a type class

```
class (Arrow arr) ⇒ ArrowUnwrap arr where
  unwrap :: arr a b → (a → b)
```

we avoid doing so for aesthetic reasons. For now, we just implement $ArrowParallel$ for normal functions and the Kleisli type in Fig. 4.3, where $Conf$ is simply defined as **data** $Conf = Nil$ since Eden does not have a configurable $spawnF$ variant.

```
instance (Trans a, Trans b) ⇒ ArrowParallel (→) a b Conf where
    parEvalN _ = spawnF
instance (ArrowParallel (→) a (m b) Conf,
  Monad m, Trans a, Trans b, Trans (m b)) ⇒
  ArrowParallel (Kleisli m) a b conf where
    parEvalN conf fs =
      arr (parEvalN conf (map (λ(Kleisli f) → f) fs)) >>>
      Kleisli sequence
```

Figure 4.3: Eden $ArrowParallel$ instance.

4.2.4 Default configuration instances

While the configurability in the instances of the *ArrowParallel* instances above is nice, users probably would like to have proper default configurations for many parallel programs as well. These can also easily be defined as we can see by the example of the default implementation of *ArrowParallel* for GpH:

```
instance (NFData b, ArrowChoice arr, ArrowParallel arr a b (Conf b)) =>
  ArrowParallel arr a b () where
    parEvalN _ fs = parEvalN (defaultConf fs) fs
defaultConf :: (NFData b) => [arr a b] -> Conf b
defaultConf fs = stratToConf fs rdeepseq
stratToConf :: [arr a b] -> Strategy b -> Conf b
stratToConf _ strat = Conf strat
```

The other backends have similarly structured implementations which we do not discuss here for the sake of brevity. We can, however, only have one instance of *ArrowParallel arr a b ()* present at a time, which should not be a problem, though.

Up until now we discussed Arrow operations more in detail, but in the following sections we focus more on the data-flow between the Arrows, now that we have seen that Arrows are capable of expressing parallelism. We do explain new concepts in greater detail if required for better understanding, though.

4.3 Extending the interface

With the *ArrowParallel* type class in place and implemented, we can now define other parallel interface functions. These are basic algorithmic skeletons that are used to define more sophisticated skeletons.

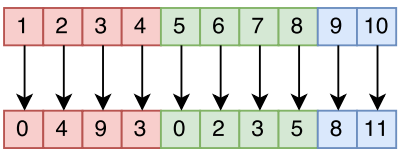

Figure 4.4: *parEvalNLazy* depiction.

```
parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) ⇒
  conf → ChunkSize → [arr a b] → (arr [a] [b])
parEvalNLazy conf chunkSize fs =
  arr (chunksOf chunkSize) >>>
  evalN fchunks >>>
  arr concat
  where
    fchunks = map (parEvalN conf) (chunksOf chunkSize fs)
```

Figure 4.5: Definition of *parEvalNLazy*.

4.3.1 Lazy *parEvalN*

The function *parEvalN* fully traverses the list of passed Arrows as well as their inputs. Sometimes this might not be feasible, as it will not work on infinite lists of functions like e.g. $map\ (arr \circ (+))\ [1..]$ or just because we need the Arrows evaluated in chunks. *parEvalNLazy* (Figs. 4.4, 4.5) fixes this. It works by first chunking the input from $[a]$ to $[[a]]$ with the given *chunkSize* in $arr\ (chunksOf\ chunkSize)$. These chunks are then fed into a list $[arr\ [a]\ [b]]$ of chunk-wise parallel Arrows with the help of our lazy and sequential *evalN*. The resulting $[[b]]$ is lastly converted into $[b]$ with $arr\ concat$.

4.3.2 Heterogeneous tasks

We have only talked about the parallelization of Arrows of the same set of input and output types until now. But sometimes we want to parallelize heterogeneous types as well. We can implement such a *parEval2* combinator (Figs. 4.6, 4.7) which combines two Arrows $arr\ a\ b$ and $arr\ c\ d$ into a new parallel Arrow $arr\ (a, c)\ (b, d)$ quite easily with the help of the *ArrowChoice* type class. Here, the general idea is to use the $+++$ combinator which combines two Arrows $arr\ a\ b$ and $arr\ c\ d$ and transforms them into $arr\ (Either\ a\ c)\ (Either\ b\ d)$ to get a common Arrow type that we can then feed into *parEvalN*.

We can implement this idea as follows: Starting off, we transform the (a, c) input into a two-element list $[Either\ a\ c]$ by first tagging the two inputs with *Left* and

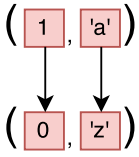

Figure 4.6: *parEval2* depiction.

```
parEval2 :: (ArrowChoice arr,
  ArrowParallel arr (Either a c) (Either b d) conf) ⇒
  conf → arr a b → arr c d → arr (a, c) (b, d)
parEval2 conf f g =
  arr Left *** (arr Right >>> arr return) >>>
  arr (uncurry (:)) >>>
  parEvalN conf (replicate 2 (f +++ g)) >>>
  arr partitionEithers >>>
  arr head *** arr head
```

Figure 4.7: *parEval2* definition.

Right and wrapping the right element in a singleton list with *return* so that we can combine them with *arr* (*uncurry* (:)). Next, we feed this list into a parallel Arrow running on two instances of f +++ g as described in the paper. After the calculation is finished, we convert the resulting [*Either b d*] into ([b], [d]) with *arr partitionEithers*. The two lists in this tuple contain only one element each by construction, so we can finally just convert the tuple to (b, d) in the last step.

4.4 Basic *map*-based skeletons

Now we have developed Parallel Arrows far enough to define some useful algorithmic skeletons that abstract typical parallel computations. We start here with some basic *map*-based skeletons. The essential differences between these skeletons presented here are in terms of order of evaluation and work distribution but still provide the same semantics as a sequential *map*.

4.4.1 Parallel *map* and laziness

The *parMap* skeleton (Figs. 4.8, 4.10) is probably the most common skeleton for parallel programs. We can implement it with *ArrowParallel* by repeating an Arrow *arr a b* and then passing it into *parEvalN* to obtain an Arrow *arr* [a] [b].

```
parMap :: (ArrowParallel arr a b conf) ⇒ conf → (arr a b) → (arr [a] [b])
parMap conf f = parEvalN conf (repeat f)
```

Figure 4.8: *parMap* definition.

Just like *parEvalN*, *parMap* traverses all input Arrows as well as the inputs. Because of this, it has the same restrictions as *parEvalN* as compared to *parEvalNLazy*. So

it makes sense to also have a *parMapStream* (Figs. 4.9, 4.11) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*.

$$
\begin{aligned}
&parMapStream :: (ArrowParallel\ arr\ a\ b\ conf, ArrowChoice\ arr, ArrowApply\ arr) \Rightarrow \\
&\quad conf \to ChunkSize \to arr\ a\ b \to arr\ [a]\ [b] \\
&parMapStream\ conf\ chunkSize\ f = parEvalNLazy\ conf\ chunkSize\ (repeat\ f)
\end{aligned}
$$

Figure 4.9: *parMapStream* definition.

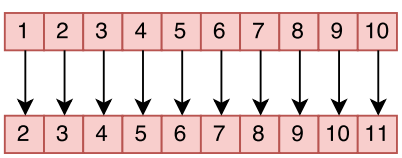

Figure 4.10: *parMap* depiction.

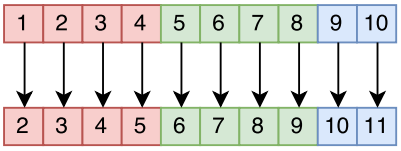

Figure 4.11: *parMapStream* depiction.

4.4.2 Statically load-balancing parallel *map*

Our *parMap* spawns every single computation in a new thread (at least for the instances of *ArrowParallel* we presented in this thesis). This can be quite wasteful and a statically load-balancing *farm* (Figs. 4.12, 4.14) that equally distributes the workload over *numCores* workers seems useful. The definitions of the helper functions *unshuffle*, *takeEach*, *shuffle* (Fig. 10.5) originate from an Eden skeleton[1].

$$
\begin{aligned}
&farm :: (ArrowParallel\ arr\ a\ b\ conf,\\
&\quad ArrowParallel\ arr\ [a]\ [b]\ conf, ArrowChoice\ arr) \Rightarrow\\
&\quad conf \to NumCores \to arr\ a\ b \to arr\ [a]\ [b]\\
&farm\ conf\ numCores\ f =\\
&\quad unshuffle\ numCores \ggg\\
&\quad parEvalN\ conf\ (repeat\ (mapArr\ f)) \ggg\\
&\quad shuffle
\end{aligned}
$$

Figure 4.12: *farm* definition.

Since a *farm* is basically just *parMap* with a different work distribution, it has the same restrictions as *parEvalN* and *parMap*. We can, however, define *farmChunk* (Figs. 4.13, 4.15) which uses *parEvalNLazy* instead of *parEvalN*. It is basically the same definition as for *farm*, but with *parEvalNLazy* instead of *parEvalN*.

$$
\begin{aligned}
&farmChunk :: (ArrowParallel\ arr\ a\ b\ conf, ArrowParallel\ arr\ [a]\ [b]\ conf,\\
&\quad ArrowChoice\ arr, ArrowApply\ arr) \Rightarrow\\
&\quad conf \to ChunkSize \to NumCores \to arr\ a\ b \to arr\ [a]\ [b]\\
&farmChunk\ conf\ chunkSize\ numCores\ f =\\
&\quad unshuffle\ numCores \ggg\\
&\quad parEvalNLazy\ conf\ chunkSize\ (repeat\ (mapArr\ f)) \ggg\\
&\quad shuffle
\end{aligned}
$$

Figure 4.13: *farmChunk* definition.

[1] Available on Hackage under `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/src/Control-Parallel-Eden-Map.html`.

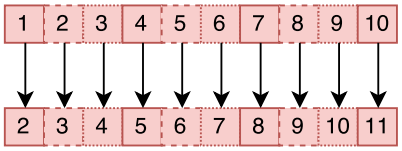

Figure 4.14: *farm* depiction.

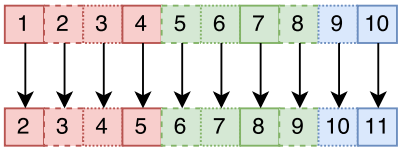

Figure 4.15: *farmChunk* depiction.

5 Further development of Parallel Arrows

5.1 Futures

Consider the following outline parallel Arrow combinator:

```
someCombinator :: (ArrowChoice arr,
  ArrowParallel arr a b (),
  ArrowParallel arr b c ()) ⇒
  [arr a b] → [arr b c] → arr [a] [c]
someCombinator fs1 fs2 =
  parEvalN () fs1 >>>
  rightRotate >>>
  parEvalN () fs2
```

In a distributed environment this first evaluates all $[arr\ a\ b]$ in parallel, sends the results back to the master node, rotates the input once (in the example we require $ArrowChoice$ for this) and then evaluates the $[arr\ b\ c]$ in parallel to then gather the input once again on the master node. Such situations arise, e.g. in scientific computations when data distributed across the nodes needs to be transposed. A concrete example is 2D FFT computation (Gorlatch and Bischof, 1998, Berthold et al. (2009c)).

While the example could be rewritten into a single $parEvalN$ call by directly wiring the Arrows together before spawning, it illustrates an important problem. When using a $ArrowParallel$ backend that resides on multiple computers, all communication between the nodes is done via the master node, as shown in the Eden trace in Figure 5.1. This can become a serious bottleneck for a larger amount of data and number of processes as e.g. Berthold et al. (2009c) showcases.

This is only a problem in distributed memory (in the scope of this thesis) and we should allow nodes to communicate directly with each other. Eden already provides „remote data“ that enable this (Alt and Gorlatch, 2003,Dieterle et al. (2010b)). But as we want code using our DSL to be implementation agnostic, we have to wrap this concept. We do this with the $Future$ type class:

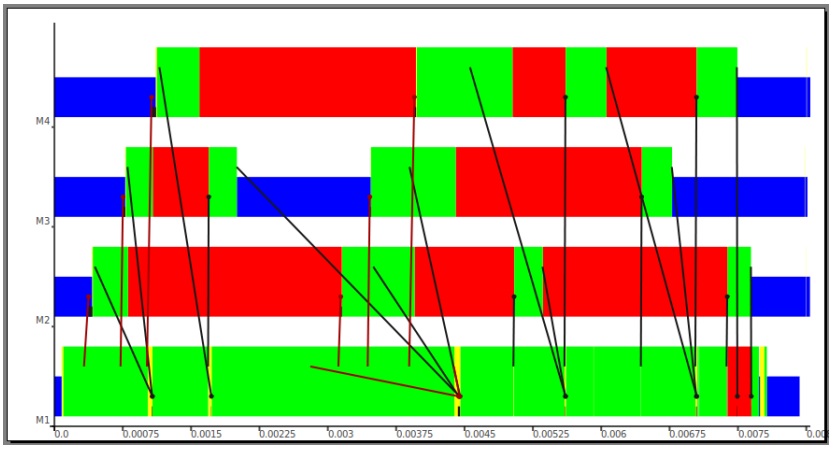

Figure 5.1: Communication between 4 Eden processes without Futures. All communication goes through the master node. Each bar represents one process. Black lines represent communication. Colours: blue ≙ idle, green ≙ running, red ≙ blocked, yellow ≙ suspended.

$$\begin{aligned}
&\textbf{class } \mathit{Future}\ \mathit{fut}\ a\ \mathit{conf} \mid a\ \mathit{conf} \to \mathit{fut} \textbf{ where}\\
&\quad \mathit{put} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{conf} \to \mathit{arr}\ a\ (\mathit{fut}\ a)\\
&\quad \mathit{get} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{conf} \to \mathit{arr}\ (\mathit{fut}\ a)\ a
\end{aligned}$$

A *conf* parameter is required here as well, but only so that Haskells type system allows us to have multiple Future implementations imported at once without breaking any dependencies similar to what we did with the *ArrowParallel* type class earlier. Note that we can also define default utility instances $\mathit{Future}\ \mathit{fut}\ a\ ()$ for each backend similar to how $\mathit{ArrowParallel}\ \mathit{arr}\ a\ b\ ()$ was defined in Chapter 4 as we will shortly see in the implementations for the backends.

Since *RD* is only a type synonym for a communication type that Eden uses internally, we have to use some wrapper classes to fit that definition, though, as the following code showcases:

$$\begin{aligned}
&\textbf{data } \mathit{RemoteData}\ a = \mathit{RD}\ \{\mathit{rd} :: \mathit{RD}\ a\}\\
&\mathit{put}' :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{arr}\ a\ (\mathit{BasicFuture}\ a)\\
&\mathit{put}' = \mathit{arr}\ \mathit{BF}\\
&\mathit{get}' :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{arr}\ (\mathit{BasicFuture}\ a)\ a\\
&\mathit{get}' = \mathit{arr}\ (\lambda(\sim(\mathit{BF}\ a)) \to a)\\
&\textbf{instance } \mathit{NFData}\ (\mathit{RemoteData}\ a) \textbf{ where}\\
&\quad \mathit{rnf} = \mathit{rnf} \circ \mathit{rd}\\
&\textbf{instance } \mathit{Trans}\ (\mathit{RemoteData}\ a)\\
&\textbf{instance } (\mathit{Trans}\ a) \Rightarrow \mathit{Future}\ \mathit{RemoteData}\ a\ \mathit{Conf} \textbf{ where}\\
&\quad \mathit{put}\ _ = \mathit{put}'\\
&\quad \mathit{get}\ _ = \mathit{get}'
\end{aligned}$$

```
instance (Trans a) ⇒ Future RemoteData a () where
  put _ = put'
  get _ = get'
```

For GpH and *Par* Monad, we can simply use *BasicFuture*s, which are just simple wrappers around the actual data with boiler-plate logic so that the type class is satisfied. This is because the concept of a *Future* does not change anything for shared-memory execution as there are no communication problems to fix. Nevertheless, we require a common interface so the parallel Arrows are portable across backends. Here, the implementation is:

```
data BasicFuture a = BF a

put' :: (Arrow arr) ⇒ arr a (BasicFuture a)
put' = arr BF

get' :: (Arrow arr) ⇒ arr (BasicFuture a) a
get' = arr (λ(∼(BF a)) → a)

instance NFData a ⇒ NFData (BasicFuture a) where
  rnf (BF a) = rnf a

instance Future BasicFuture a (Conf a) where
  put _ = put'
  get _ = get'

instance Future BasicFuture a () where
  put _ = put'
  get _ = get'
```

Now, we can use this *Future* concept in our communication example for direct communication between nodes:

```
someCombinator :: (ArrowChoice arr,
  ArrowParallel arr a (fut b) (),
  ArrowParallel arr (fut b) c (),
  Future fut b ()) ⇒
  [arr a b] → [arr b c] → arr [a] [c]
someCombinator fs1 fs2 =
  parEvalN () (map (⋙put ()) fs1) ⋙
  rightRotate ⋙
  parEvalN () (map (get ()⋙) fs2)
```

In a distributed environment, this gives us a communication scheme with messages going through the master node only if it is needed – similar to what is shown in

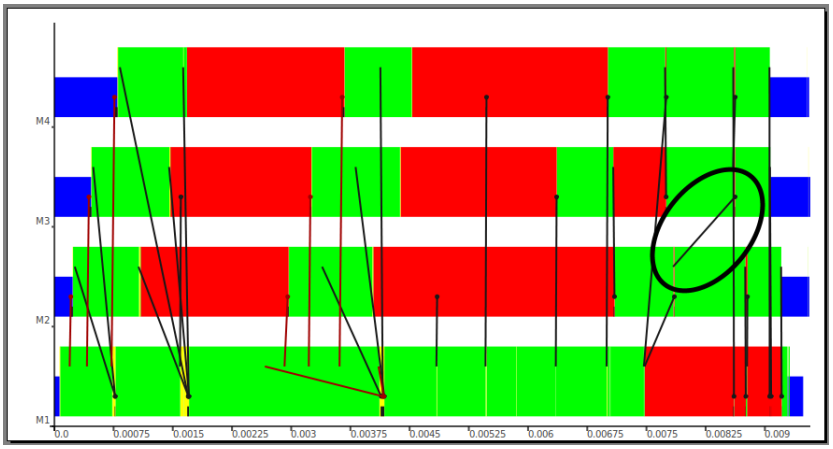

Figure 5.2: Communication between 4 Eden processes with Futures. Other than in Fig. 5.1, processes communicate directly (one example message is highlighted) instead of always going through the master node (bottom bar).

the trace visualisation in Fig. 5.2. One especially elegant aspect of the definition of our Future type class is that we can specify the type of *Future* to be used per backend with full interoperability between code using different backends, without even requiring to know about the actual type used for communication. We only specify that there has to be a compatible Future and do not care about any specifics as can be seen in the future version of *someCombinator*.

5.2 Advanced topological skeletons

Even though many algorithms can be expressed by *parMap*s, some problems require more sophisticated skeletons. The Eden library leverages this problem and already comes with more predefined skeletons[1], among them a *pipe*, a *ring*, and a *torus* implementation (Loogen et al., 2003). These seem like reasonable candidates to be ported to our Arrow-based parallel Haskell. While doing so, we aim to showcase that we can express more sophisticated skeletons with parallel Arrows as well.

If we were to use the original definition of *parEvalN*, however, these skeletons would produce an infinite loop with the GpH and *Par* Monad which during runtime would result in the program crashing. This materialises with the usage of *loop* of the *ArrowLoop* type class and we think that this is due to difference of how evaluation is done in these backends when compared to Eden. An investigation of why this difference exists is beyond the scope of this work, we only provide a workaround for these types of skeletons as such they probably are not of much importance outside

[1] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/Control-Parallel-Eden-Topology.html`.

of a distributed memory environment. However our workaround enables users of the DSL to test their code within a shared memory setting.

The idea of the fix is to provide a *ArrowLoopParallel* type class that has two functions – *loopParEvalN* and *postLoopParEvalN*, where the first is to be used inside an *loop* construct while the latter will be used right outside of the *loop*. This way we can delegate to the actual *parEvalN* in the spot where the backend supports it.

```
class ArrowParallel arr a b conf ⇒
  ArrowLoopParallel arr a b conf where
  loopParEvalN :: conf → [arr a b] → arr [a] [b]
  postLoopParEvalN :: conf → [arr a b] → arr [a] [b]
```

As Eden has no problems with the looping skeletons, we use this instance:

```
instance (ArrowChoice arr, ArrowParallel arr a b Conf) ⇒
  ArrowLoopParallel arr a b Conf where
  loopParEvalN = parEvalN
  postLoopParEvalN _ = evalN
```

As the *Par* Monad and GpH have problems with *parEvalN* inside of *loop* their respective instances for *ArrowLoopParallel* look like this:

```
instance (ArrowChoice arr, ArrowParallel arr a b (Conf b)) ⇒
  ArrowLoopParallel arr a b (Conf b) where
  loopParEvalN _ = evalN
  postLoopParEvalN = parEvalN
```

5.2.1 Parallel pipe

We start with the parallel *pipe* skeleton, which is semantically equivalent to folding over a list $[arr\ a\ a]$ of Arrows with $\ggg$, but in parallel, meaning that the Arrows do not have to reside on the same thread/machine. We implement this skeleton using the *ArrowLoop* type class which provides us with the $loop :: arr\ (a, b)\ (c, b) \to arr\ a\ c$ combinator allowing us to express recursive fix-point computations in which output values are fed back as input. For example

$$loop\ (arr\ (\lambda(a, b) \to (b, a : b)))$$

which is the same as

```
pipeSimple :: (ArrowLoop arr, ArrowLoopParallel arr a a conf) ⇒
  conf → [arr a a] → arr a a
pipeSimple conf fs =
  loop (arr snd &&&
    (arr (uncurry (:) >>> lazy) >>> loopParEvalN conf fs)) >>>
  arr last
```

Figure 5.3: Simple *pipe* skeleton. The use of *lazy* (Fig. 10.6) is essential as without it programs using this definition would never halt. We need to ensure that the evaluation of the input $[a]$ is not forced fully before passing it into *loopParEvalN*.

```
pipe :: (ArrowLoop arr,
  ArrowLoopParallel arr (fut a) (fut a) conf,
  Future fut a conf) ⇒
  conf → [arr a a] → arr a a
pipe conf fs = unliftFut conf (pipeSimple conf (map (liftFut conf) fs))
liftFut :: (Arrow arr, Future fut a conf, Future fut b conf) ⇒
  conf → arr a b → arr (fut a) (fut b)
liftFut conf f = get conf >>> f >>> put conf
unliftFut :: (Arrow arr, Future fut a conf, Future fut b conf) ⇒
  conf → arr (fut a) (fut b) → arr a b
unliftFut conf f = put conf >>> f >>> get conf
```

Figure 5.4: *pipe* skeleton definition with Futures.

```
loop (arr snd &&& arr (uncurry (:)))
```

defines an Arrow that takes its input a and converts it into an infinite stream $[a]$ of it. Using *loop* to our advantage gives us a first draft of a pipe implementation (Fig. 5.3) by plugging in the parallel evaluation call *loopParEvalN conf fs* inside the second argument of &&& and then only picking the first element of the resulting list with *arr last* outside of the *loop*.

However, using this definition directly will make the master node a potential bottleneck in distributed environments as described in Chapter 5.1. Therefore, we introduce a more sophisticated version that internally uses Futures and obtain the final definition of *pipe* in Fig. 5.4.

Sometimes, this *pipe* definition can be a bit inconvenient, especially if we want to pipe Arrows of mixed types together, i.e. *arr a b* and *arr b c*. By wrapping these two Arrows inside a bigger Arrow *arr* $(([a],[b]),[c])$ $(([a],[b]),[c])$ suitable for *pipe*, we can define *pipe2* as in Fig. 5.5.

```
pipe2 :: (ArrowLoop arr, ArrowChoice arr,
  ArrowLoopParallel arr (fut (([a],[b]),[c])) (fut (([a],[b]),[c])) conf,
  Future fut (([a],[b]),[c]) conf) ⇒
  conf → arr a b → arr b c → arr a c
pipe2 conf f g =
  (arr return &&& arr (const [])) &&& arr (const []) >>>
  pipe conf (replicate 2 (unify f g)) >>>
  arr snd >>>
  arr head
  where
    unify :: (ArrowChoice arr) ⇒
      arr a b → arr b c → arr (([a],[b]),[c]) (([a],[b]),[c])
    unify f' g' =
      (mapArr f' *** mapArr g') *** arr (const []) >>>
      arr (λ((b, c), a) → ((a, b), c))
(|>>>|) :: (ArrowLoop arr, ArrowChoice arr,
  ArrowLoopParallel arr (fut (([a],[b]),[c])) (fut (([a],[b]),[c])) (),
  Future fut (([a],[b]),[c]) ()) ⇒
  arr a b → arr b c → arr a c
(|>>>|) = pipe2 ()
```

Figure 5.5: Definition of *pipe2* and (|>>>|), a parallel >>>.

Extensive use of *pipe2* over *pipe* with a hand-written combination data type will probably result in worse performance because of more communication overhead from the many calls to *parEvalN* inside of *evalN*. Nonetheless, we can define a parallel piping operator |>>>|, which is semantically equivalent to >>> similarly to other parallel syntactic sugar from Appendix 10.4.

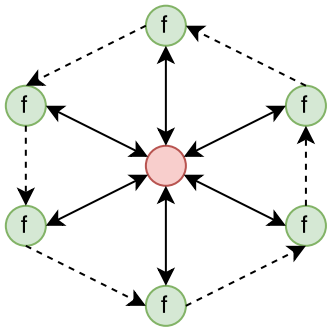

Figure 5.6: *ring* skeleton depiction.

5.2.2 Ring skeleton

Eden comes with a ring skeleton[2] (Fig. 5.6) implementation that allows the computation of a function $[i] \to [o]$ with a ring of nodes that communicate with each other. Its input is a node function $i \to r \to (o, r)$ in which r serves as the intermediary output that is sent to the neighbour of each node. It uses the direct „remote data“ communication channels that were already mentioned in Chapter 5.1. We depict it in the Appendix, Fig. 10.8.

We can rewrite this functionality easily with the use of *loop* as the definition of the node function, $arr\ (i, r)\ (o, r)$, after being transformed into an Arrow, already fits quite neatly into *loop*'s signature: $arr\ (a, b)\ (c, b) \to arr\ a\ c$. In each iteration we start by rotating the intermediary input from the nodes $[fut\ r]$ with $second\ (rightRotate \ggg lazy)$ (Fig. 10.6). Similarly to the *pipe* from Chapter 5.2.1 (Fig. 5.3), we have to feed the intermediary input into our *lazy* (Fig. 10.6) Arrow here, or the evaluation would fail to terminate. The reasoning is explained by Loogen (2012) as a demand problem.

Next, we zip the resulting $([i], [fut\ r])$ to $[(i, fut\ r)]$ with $arr\ (uncurry\ zip)$. We then feed this into our parallel Arrow $arr\ [(i, fut\ r)]\ [(o, fut\ r)]$ obtained by transforming our input Arrow $f :: arr\ (i, r)\ (o, r)$ into $arr\ (i, fut\ r)\ (o, fut\ r)$ before *repeat*ing and lifting it with *loopParEvalN*. Finally we unzip the output list $[(o, fut\ r)]$ list into $([o], [fut\ r])$.

Plugging this Arrow $arr\ ([i], [fut\ r])\ ([o], fut\ r)$ into the definition of *loop* from earlier gives us $arr\ [i]\ [o]$, our ring Arrow (Fig. 5.7). To make sure this algorithm has speedup on shared-memory machines as well, we pass the result of this Arrow to $postLoopParEvalN\ conf\ (repeat\ (arr\ id))$. This combinator can, for example, be used to calculate the shortest paths in a graph using Warshall's algorithm.

[2] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/Control-Parallel-Eden-Topology.html`.

$$
\begin{aligned}
&ring :: (Future\ fut\ r\ conf,\\
&\quad ArrowLoop\ arr,\\
&\quad ArrowLoopParallel\ arr\ (i, fut\ r)\ (o, fut\ r)\ conf,\\
&\quad ArrowLoopParallel\ arr\ o\ o\ conf) \Rightarrow\\
&\quad conf \to arr\ (i, r)\ (o, r) \to arr\ [i]\ [o]\\
&ring\ conf\ f =\\
&\quad loop\ (second\ (rightRotate \ggg lazy) \ggg\\
&\qquad arr\ (uncurry\ zip) \ggg\\
&\qquad loopParEvalN\ conf\\
&\qquad\quad (repeat\ (second\ (get\ conf) \ggg f \ggg second\ (put\ conf))) \ggg\\
&\qquad arr\ unzip) \ggg\\
&\quad postLoopParEvalN\ conf\ (repeat\ (arr\ id))
\end{aligned}
$$

Figure 5.7: *ring* skeleton definition.

5.2.3 Torus skeleton

If we take the concept of a *ring* from Chapter 5.2.2 one dimension further, we obtain a *torus* skeleton (Fig. 5.8, 5.9). Every node sends and receives data from horizontal and vertical neighbours in each communication round. With our Parallel Arrows we re-implement the *torus* combinator[3] from Eden – yet again with the help of the *ArrowLoop* type class.

Similar to the *ring*, we start by rotating the input (Fig. 10.6), but this time not only in one direction, but in two. This means that the intermediary input from the neighbour nodes has to be stored in a tuple $([[fut\ a]], [[fut\ b]])$ in the second argument (*loop* only allows for two arguments) of our looped Arrow of type

$$arr\ ([[c]], ([[fut\ a]], [[fut\ b]]))\ ([[d]], ([[fut\ a]], [[fut\ b]]))$$

and our rotation Arrow becomes

[3] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/Control-Parallel-Eden-Topology.html`.

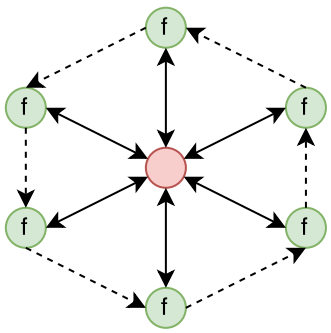

Figure 5.8: *torus* skeleton depiction.

$$second\ ((mapArr\ rightRotate \ggg lazy) *** (arr\ rightRotate \ggg lazy))$$

instead of the singular rotation in the ring as we rotate $[[fut\ a]]$ horizontally and $[[fut\ b]]$ vertically. Then, we zip the inputs for the input Arrow with

$$arr\ (uncurry3\ zipWith3\ lazyzip3)$$

from $([[c]], ([[fut\ a]], [[fut\ b]]))$ to $[[(c, fut\ a, fut\ b)]]$, which we then evaluate in parallel.

This, however, is more complicated than in the *ring* case as we have one more dimension of inputs that needs to be transformed. We first have to *shuffle* all the inputs to then pass them into $loopParEvalN\ conf\ (repeat\ (ptorus\ conf\ f))$ to get an output of $[(d, fut\ a, fut\ b)]$. We then unshuffle this list back to its original ordering by feeding it into $arr\ (uncurry\ unshuffle)$ which takes the input length we saved one step earlier as additional input to get a result matrix $[[[(d, fut\ a, fut\ b)]]]$. Finally, we unpack this matrix with $arr\ (map\ unzip3) \ggg arr\ unzip3 \ggg threetotwo$ to get $([[d]], ([[fut\ a]], [[fut\ b]]))$.

This internal looping computation is once again fed into *loop* and we also compose a final $postLoopParEvalN\ conf\ (repeat\ (arr\ id))$ for the same reasons as explained for the *ring* skeleton.

As an example of using this skeleton, Loogen et al. (2003) showed the matrix multiplication using the Gentleman algorithm (Gentleman, 1978). An adapted version can be found in Fig. 5.10.

If we compare the trace from a call using our Arrow definition of the *torus* (Fig. 5.11) with the Eden version (Fig. 5.12) we can see that the behaviour of the Arrow version and execution times are comparable. We discuss further benchmarks on larger clusters in more detail in Chapter 7.

```
torus :: (Future fut a conf, Future fut b conf,
  ArrowLoop arr, ArrowChoice arr,
  ArrowLoopParallel arr (c, fut a, fut b) (d, fut a, fut b) conf,
  ArrowLoopParallel arr [d] [d] conf) ⇒
  conf → arr (c, a, b) (d, a, b) → arr [[c]] [[d]]
torus conf f =
  loop (second ((mapArr rightRotate >>> lazy)
    *** (arr rightRotate >>> lazy)) >>>
    arr (uncurry3 (zipWith3 lazyzip3)) >>>
    arr length &&& (shuffle >>>
      loopParEvalN conf (repeat (ptorus conf f))) >>>
    arr (uncurry unshuffle) >>>
    arr (map unzip3) >>> arr unzip3 >>> threetotwo) >>>
  postLoopParEvalN conf (repeat (arr id))
ptorus :: (Arrow arr, Future fut a conf, Future fut b conf) ⇒
  conf →
  arr (c, a, b) (d, a, b) →
  arr (c, fut a, fut b) (d, fut a, fut b)
ptorus conf f =
  arr (λ~(c, a, b) → (c, get conf a, get conf b)) >>>
  f >>>
  arr (λ~(d, a, b) → (d, put conf a, put conf b))
```

Figure 5.9: *torus* skeleton definition. *lazyzip3*, *uncurry3* and *threetotwo* definitions are in Fig. 10.7.

```
type Matrix = [[Int]]
prMM_torus :: Int → Int → Matrix → Matrix → Matrix
prMM_torus numCores problemSizeVal m1 m2 =
  combine $ torus () (mult torusSize) $ zipWith zip (split1 m1) (split2 m2)
  where torusSize = (floor ∘ sqrt) $ fromIntegral $ numCoreCalc numCores
        combine x = concat (map ((map (concat)) ∘ transpose) x)
        split1 x = staggerHorizontally
          (splitMatrix (problemSizeVal `div` torusSize) x)
        split2 x = staggerVertically
          (splitMatrix (problemSizeVal `div` torusSize) x)
  -- Function performed by each worker
mult :: Int →
  ((Matrix, Matrix), [Matrix], [Matrix]) →
  (Matrix, [Matrix], [Matrix])
mult size ((sm1, sm2), sm1s, sm2s) = (result, toRight, toBottom)
  where toRight = take (size − 1) (sm1 : sm1s)
        toBottom = take (size − 1) (sm2 : sm2s)
        sms = zipWith prMM (sm1 : sm1s) (sm2 : sm2s)
        result = foldl1' matAdd sms
```

Figure 5.10: Adapted matrix multiplication in Eden using a the *torus* skeleton. *prMM_torus* is the parallel matrix multiplication. *mult* is the function performed by each worker. *prMM* is the sequential matrix multiplication in the chunks. *splitMatrix* splits the Matrix into chunks. *staggerHorizontally* and *staggerVertically* pre-rotate the matrices. *matAdd* calculates $A + B$. Omitted definitions can be found in 10.9.

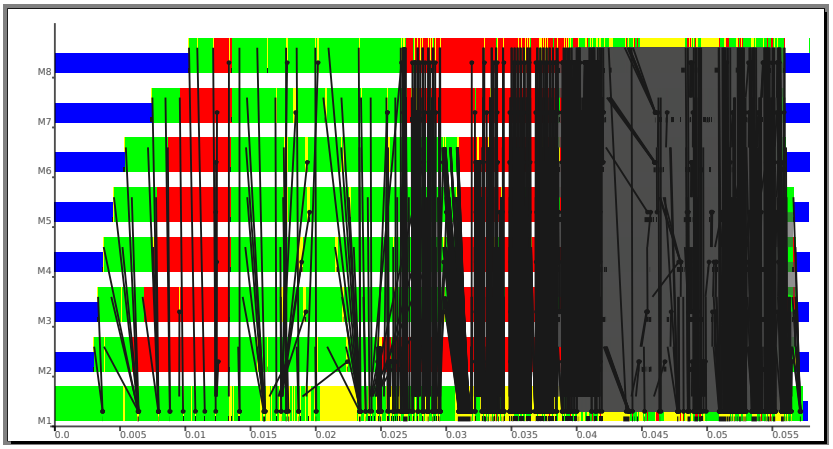

Figure 5.11: Matrix multiplication with *torus* (PArrows).

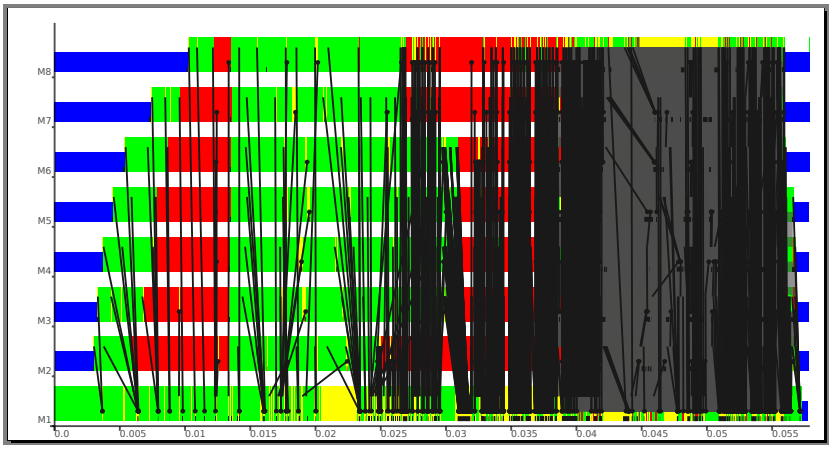

Figure 5.12: Matrix multiplication with *torus* (Eden).

6 Experiment: Cloud Haskell Backend

Cloud Computing has become more and more prevalent in recent years. Servers are replaced with virtual ones positioned all around the globe. These can easily be brought up when required and shut down when not in use. This trend in computing has also been embraced by the Haskell community and therefore, libraries such as Cloud Haskell were born. Cloud Haskell is described on the projects website[1] as:

> Cloud Haskell: Erlang-style concurrent and distributed programming in Haskell. The Cloud Haskell Platform consists of a generic network transport API, libraries for sending static closures to remote nodes, a rich API for distributed programming and a set of platform libraries modelled after Erlang's Open Telecom Platform.
>
> Generic network transport backends have been developed for TCP and in-memory messaging, and several other implementations are available including a transport for Windows Azure.[...]

It is basically a set of APIs and libraries for communication between networks of nodes and easy use in a cloud environment. With it programmers can write fully-featured Haskell based cloud solutions targeting a wide range of environments.

While users can already write concurrent applications with the help of Cloud Haskell using some of its libraries or even with the bare communication API, it seems like a good idea to write parallel programs requiring less involvement from the user. In the following section we will therefore explore the possibility of a Cloud Haskell based backend for the $ArrowParallel$ interface given in this thesis while explaining all the necessary parts of the API. For easier testing and as this is only meant as a proof of concept, we only work with a local-net Cloud Haskell backend in this thesis. The results of this experiment, however, are transferable to other architectures as well when building upon the results presented here.[2]

[1] see `http://haskell-distributed.github.io/`
[2] With the help of virtual private networks one could even use this local-net variant

The following is structured as follows. We start by explaining how to discover nodes with a master-slave structure while also defining a program startup harness that can be used with this scheme in Chapter 6.1. Then, we explain how parallel evaluation of arbitrary data is possible with Cloud Haskell in Chapter 6.2 and also discuss how we can implement the PArrows DSL with this knowledge in Chapter 6.3.

6.1 Node discovery and program harness

In cloud services it is more common that the architecture of the running network changes than in ordinary computing clusters where the participating nodes are usually known at startup. In the SimpleLocalNet[3] Cloud Haskell backend we are using for this experiment, this is reflected in the fact that there already exists a pre-implemented master-slave structure. The master node – the node that starts the computation is considered the master node here – has to keep track of all the available slave nodes. The slave nodes wait for tasks and handle them as required.

We will now first go into detail on the data-structure (Chapter 6.1.1) we use in order to handle this information to then explain how to start slave (Chapter 6.1.2) and master nodes (Chapter 6.1.3). We also explain how to create a startup harness (Chapter 6.1.4) to wrap all of this.

6.1.1 The $State$ data-structure

The data-structure containing all relevant information about the state of the computation network and the computation in general we will use, $State$, is defined as

$$
\begin{aligned}
&\mathbf{data}\ State = State\ \{\\
&\quad workers :: MVar\ [NodeId],\\
&\quad shutdown :: MVar\ Bool,\\
&\quad started :: MVar\ Bool,\\
&\quad localNode :: LocalNode,\\
&\quad serializeBufferSize :: Int\\
&\}
\end{aligned}
$$

Notice that $workers :: MVar\ [NodeId]$, $shutdown :: MVar\ Bool$ and $started :: MVar\ ()$ are all low level mutable locations instead of regular fields. This is because we pass this $State$ around between functions, but want to update it with new information on-the-fly. These modifiable variables can be created empty with $newEmptyMVar ::$

[3] see `http://hackage.haskell.org/package/distributed-process-simplelocalnet`

$IO\ (MVar\ a)$ or already with contents with $newMVar :: a \to IO\ (MVar\ a)$. They can be read with $readMVar :: MVar\ a \to IO\ a$ or emptied with $takeMVar :: MVar\ a \to IO\ a$. Values can be placed inside with $putMVar :: MVar\ a \to a \to IO\ ()$. $MVar$s are threadsafe and all reading operations block until some content is placed in them. We will see them used in other places of this backend as well.

In the $State$ type, $workers :: MVar\ [NodeId]$ holds information about all available slave nodes, $shutdown :: MVar\ Bool$ determines whether the backend is to be shut down, $started :: MVar\ ()$ returns a signalling $()$ if the backend has properly started when accessed with $readMVar$. $localNode :: LocalNode$ and $serializeBufferSize :: Int$ store information about all Cloud Haskell internals for the master node and the buffer size for serialization (we will discuss the system itself separately), respectively.

Note that as we will use the $State$ type as the $conf$ parameter in the $ArrowParallel$ instance, we use the type synonym $\mathbf{type}\ Conf = State$ in the following code sections. Furthermore, an initial config can be created with the function $initialConf :: Int \to LocalNode \to IO\ Conf$ where the resulting config contains a $serializeBufferSize$ as specified by the first parameter and the $LocalNode$ specified by the second parameter. Additionally, the list of workers $workers :: MVar\ [NodeId]$ is initialized with an empty list, $shutdown :: MVar\ Bool$ is set to $False$ and $started :: MVar\ ()$ is created as an empty $MVar$ so that it can be populated with the signalling $()$ when the startup is finished.

$$
\begin{aligned}
&initialConf :: Int \to LocalNode \to IO\ Conf \\
&initialConf\ serializeBufferSize\ localNode = \mathbf{do} \\
&\quad workersMVar \leftarrow newMVar\ [\,] \\
&\quad shutdownMVar \leftarrow newMVar\ False \\
&\quad startedMVar \leftarrow newEmptyMVar \\
&\quad return\ State\ \{ \\
&\qquad workers = workersMVar, \\
&\qquad shutdown = shutdownMVar, \\
&\qquad started = startedMVar, \\
&\qquad localNode = localNode, \\
&\qquad serializeBufferSize = serializeBufferSize \\
&\quad \}
\end{aligned}
$$

A utility function $defaultInitConf$ using a default serialization buffer size of 10MB is also defined as:

$$
\begin{aligned}
&defaultBufSize :: Int \\
&defaultBufSize = 10 * 2 \uparrow 20 \quad \text{-- 10 MB}
\end{aligned}
$$

```
defaultInitConf :: LocalNode → IO Conf
defaultInitConf = initialConf defaultBufSize
```

6.1.2 Starting Slave nodes

With the *State/Conf* data structure we can then implement our node-discovery scheme. Starting with the slave nodes, we can just use the basic utilities for a slave backend in the SimpleLocalNet library. The code to start a master node for the *Slave* backend is therefore:

```
type Host = String
type Port = String
initializeSlave :: RemoteTable → Host → Port → IO ()
initializeSlave remoteTable host port = do
  backend ← initializeBackend host port remoteTable
  startSlave backend
```

We start a slave node by initializing the Cloud Haskell backend with a given *host*, *port* and *remoteTable* via *initializeBackend* :: *String* → *String* → *RemoteTable* and then delegating the logic completely to the library function *startSlave* :: *Backend* → *IO* () which does not return unless the slave is shutdown manually from the master node. The *RemoteTable* contains all serialization information about static values required by Cloud Haskell. We will later see how we can automatically generate such a table.

6.1.3 Starting Master nodes

For master nodes, the implementation is a bit more involved. The actual *startMaster*:: *Backend* → *Process* → *IO* () supplied by SimpleLocalNet is meant to start a computation represented by a *Process* monad and then return. In our use-case we want to be able to spawn functions outside of the *Process* monad however. We use the *Process* passed into this startup function only for slave-node discovery and management:

```
master :: Conf → Backend → [NodeId] → Process ()
master conf backend slaves = do
  forever $ do
    shutdown ← liftIO $ readMVar $ shutdown conf
    if shutdown
      then do
```

```
    terminateAllSlaves backend
    die "terminated"
  else do
    slaveProcesses ← findSlaves backend
    redirectLogsHere backend slaveProcesses
    let slaveNodes = map processNodeId slaveProcesses
    liftIO $ do
      modifyMVar_ (workers conf) (\_ → return slaveNodes)
      isEmpty ← isEmptyMVar $ started conf
      if (isEmpty ∧ length slaveNodes > 0) then
        putMVar (started conf) ()
      else
        return ()
```

Basically, this continuously updates the list of slaves inside the configuration by first querying for all slave processes with *findSlaves backend* and redirecting the log output to the master node with *redirectLogsHere backend slaveProcesses* to then finally update *workers* :: *MVar* [*NodeId*] inside the configuration. Additionally, as soon as one slave is found, *started* :: *MVar* () is supplied with the signalling () so that any thread waiting for node discovery can start its actual computation.[4] All of this is embedded in a check whether a shutdown is requested with *liftIO* $ *readMVar* $ *shutdown conf*. If instructed to do so, the program does the necessary cleanup - terminating all slaves with *terminateAllSlaves backend* and shutting itself down with *die* `"terminated"` - otherwise continuing with the updating process.

With this *master* function, we can now define our initialization function *initializeMaster*:: *RemoteTable* → *Host* → *Port* → *IO Conf*:

```
initializeMaster :: RemoteTable → Host → Port → IO Conf
initializeMaster remoteTable host port = do
  backend ← initializeBackend host port remoteTable
  localNode ← newLocalNode backend
  conf ← defaultInitConf localNode
  forkIO $ startMaster backend (master conf backend)
  waitForStartup conf
  return conf
```

Similar to the slave code, we again initialize the Cloud Haskell backend via *initializeBackend*:: *String* → *String* → *RemoteTable*, but then also create a new local node that is used

[4] Notice that while we could add an additional sleep here to not generate too much network noise in this function, we leave it out for the sake of simplicity.

to start computations outside of the initialization logic. With this node we can create a default initial config via $defaultInitConf :: LocalNode \to Conf$ which is then passed into the discovery function with $startMaster\ backend\ (master\ conf\ backend)$. We have to fork this IO action away with $forkIO$, because the IO action will run forever as long as the program has not been manually shutdown via the corresponding variable in the $State$. Finally, we wait for the startup to finish via $waitForStartup :: Conf \to IO\ ()$ to end with returning a $IO\ Conf$ action containing the initial config/state. Here, $waitForStartup$ can simply be defined as

$$
\begin{aligned}
&waitForStartup :: Conf \to IO\ () \\
&waitForStartup\ conf = readMVar\ (started\ conf)
\end{aligned}
$$

because of the blocking behaviour of empty $MVar$s and the fact that we are signalling the startup with a simple dummy value $()$ as described earlier.

6.1.4 Startup harness

If we put all of the startup logic we have discussed until now together we can easily write a startup harness where we simply delegate to the proper initialization code depending on the command line arguments:

```
myRemoteTable :: RemoteTable
myRemoteTable = Main.__remoteTable initRemoteTable

main :: IO ()
main = do
  args <- getArgs
  case args of
    ["master", host, port] -> do
      conf <- initializeMaster myRemoteTable host port
        -- read and print the list of available workers
      readMVar (workers conf) >>= print
        -- TODO: parallel computation here
    ["slave", host, port] -> do
      initializeSlave myRemoteTable host port
      print "slave shutdown."
```

In order to launch a program using this harness, we have to start slave nodes for each cpu core with commands like „<executable> slave 127.0.0.1 8000“ where the last parameter determines the port the slave will listen to and wait for requests on.

Similarly a single master node can be started with „<executable> master 127.0.0.1 7999“ where, once again, the last parameter determines the communication port.

This example also shows how a $RemoteTable$ is obtained so that it can be used inside $main :: IO\ ()$. Note, that the definition of $Main.\_\_remoteTable :: RemoteTable \rightarrow RemoteTable$ used in $myRemoteTable :: RemoteTable$ is an automatically, Template-Haskell[5] generated function that builds a $RemoteTable$ from the $initRemoteTable$ that is supplied by Cloud Haskell by adding all relevant static declarations of the our program. In Cloud Haskell, we can for example generate such a declaration for some function $f :: Int \rightarrow Int$, with a call to $remotable$ inside a Template-Haskell splice as $\$(remotable\ ['f])$.

As can be seen from this, any function passed to $remotable$ must have a top-level declaration. Furthermore, we must also add any function manually. This is usually okay for basic applications where the user usually knows which functions/values need to be serialized statically at compile time, but not in our use case as we want to be able to evaluate arbitrary functions/$Arrow$s on remote nodes. In Chapter 6.2 we will see how we will resolve this problem.

6.2 Parallel Evaluation with Cloud Haskell

As already mentioned earlier, with Cloud Haskell we can not send arbitrary functions or Arrows to the slave nodes. Thankfully, there is an alternative: Eden's serialization mechanism has been made available separately in a package called „packman“.[6] This mechanism allows values to be serialized in the exact evaluation state they are currently in.

We can use this to our advantage. Instead of sending inputs and functions/Arrows to the slave nodes and sending the result back (which does not work with the current Cloud Haskell API), we can instead apply the function, serialize this unevaluated thunk, send it to the evaluating slave, and send the fully evaluated value back.

With this idea in mind we will now explain how to achieve parallel evaluation of Arrows with Cloud Haskell. We start by explaining the communication basics in Chapter 6.2.1. Next, we describe how to achieve evaluation of single values on slave nodes in Chapter 6.2.2. Finally we use these results to implement a parallel evaluation scheme in Chapter 6.2.3.

[5] Template-Haskell is a code generator for Haskell written in Haskell, and can be enabled with a language pragma {-# LANGUAGE TemplateHaskell #-} at the top of the source file.
[6] see `https://hackage.haskell.org/package/packman`

6.2.1 Communication basics

We will now go over the communication basics we require in the later parts of this Chapter. This includes a quick introduction on how we actually send the data in Cloud Haskell and also a quick definition of our serialized data wrapper we use to send unevaluated data between nodes and also

Sending and Receiving data

In order to send and receive data between nodes, Cloud Haskell uses typed channels. A typed channel consists of a $SendPort\ a$ and a $ReceivePort\ a$. We can create a new typed cannel with the help of $newChan{::}Serializable\ a \Rightarrow Process\ (SendPort\ a, ReceivePort\ a)$:

$$
\begin{aligned}
&myProc :: Process\ () \\
&myProc = \mathbf{do} \\
&\quad (sendPort, receivePort) \leftarrow newChan \\
&\qquad \text{-- do stuff}
\end{aligned}
$$

Data can be sent with the help of $sendChan :: Serializable\ a \Rightarrow SendPort\ a \rightarrow a \rightarrow Process\ ()$:

$$
\begin{aligned}
&sendTen :: SendPort\ Int \rightarrow Process\ () \\
&sendTen\ sendPort = sendChan\ sendPort\ 10
\end{aligned}
$$

Values are received in a blocking manner with $receiveChan :: Serializable\ a \Rightarrow ReceivePort\ a \rightarrow Process\ a$:

$$
\begin{aligned}
&receiveVal :: ReceivePort\ Int \rightarrow Process\ Int \\
&receiveVal\ receivePort = receivechan\ receivePort
\end{aligned}
$$

Note that only $SendPort\ a$ is serializable here. So in order to have a two way communication where process A sends some input to process B and awaits its result like we require in our use case, we have to first receive a $SendPort\ a$ in A via some $ReceivePort\ (SendPort\ a))$ of some channel $(SendPort\ (SendPort\ a), ReceivePort\ (SendPort\ a))$. This $SendPort\ a$ is sent by B and belongs to the channel $(SendPort\ a, ReceivePort\ a)$ where B expects its input to come through the $ReceivePort\ a$. Additionally, we also require a channel $(SendPort\ b, ReceivePort\ b)$ on which B sends its result through the $SendPort\ b$ and A awaits its result on the $ReceivePort\ b$. This idea is executed in the following code example. Process A looks like

```
procA :: ReceivePort (SendPort a) → ReceivePort b → Process ()
procA aSenderReceiver bReceiver = do
  aSender ← receiveChan aSenderReceiver
  let someA = ...
  sendChan aSender someA
  someB ← receiveChan bReceiver
  ...
  return ()
```

while process B is schematically defined as

```
procB :: SendPort (SendPort a) → SendPort b → Process ()
procB aSenderSender bSender = do
  (aSender, aReceiver) ← newChan
  sendChan aSenderSender aSender
  someA ← receiveChan aReceiver
  let someB = useAToMakeB someA
  sendChan bSender someB
```

Serialized data type

The packman package comes with a serialization function $trySerializeWith :: a \rightarrow Int \rightarrow IO\ (Serialized\ a)$ (the second parameter is the buffer size) and a deserialization function $deserialize :: Serialized\ a \rightarrow IO\ a$. Here, $Serialized\ a$ is the type containing the serialized value of a.

In order to have a clean slate in terms of type class instances, we define a wrapper type $Thunk\ a$ around $Serialized\ a$ as

```
  -- Wrapper for the packman type Serialized
newtype Thunk a = Thunk {fromThunk :: Serialized a} deriving (Typeable)
toThunk a = Thunk {fromThunk = a}
```

Additionally, we require a $Binary$ for our wrapper in order to be able to send it with Cloud Haskell. This only delegates to the implementation of the actual $Serialized$ we wrap:

```
instance (Typeable a) ⇒ Binary (Thunk a) where
  put = Data.Binary.put ∘ fromThunk
```

$$
\begin{aligned}
&\quad get = \mathbf{do} \\
&\qquad (ser :: Serialized\ a) \leftarrow Data.Binary.get \\
&return\ \$\ Thunk\ \{fromThunk = ser\}
\end{aligned}
$$

6.2.2 Evaluation of values on slave nodes

Having discussed the communication scheme and serialization mechanism we want to use, we can explain how the evaluation of values on slave nodes works with Cloud Haskell, next. We give the master node's code for evaluation of a single value on a slave node and also the slave nodes' code.

Master node

The following function $forceSingle :: NodeId \rightarrow MVar\ a \rightarrow a \rightarrow Process\ ()$ is used to evaluate a single value a. It returns a monadic action $Process\ ()$ that evaluates a value of type a on the node with the given $NodeId$ and stores the evaluated result in the given $MVar\ a$. It starts by creating the necessary communication channels on the master side (the A side from Chapter 6.2.1). It then spawns the actual evaluation task (process B from Chapter 6.2.1)

$$evalTask :: (SendPort\ (SendPort\ (Thunk\ a)), SendPort\ a) \rightarrow Process\ ()$$

with the necessary $SendPort$s for input communication ($SendPort\ (SendPort\ (Thunk\ a))$) and result communication ($SendPort\ a$[7]) on the given node via

$$spawn\ node\ (evalTask\ (inputSenderSender, outputSender))$$

where $spawn$ is of type

$$spawn :: NodeId \rightarrow Closure\ (Process\ ()) \rightarrow Process\ ProcessId$$

Then, like process A in Chapter 6.2.1, $forceSingle$ waits for the input $SendPort\ a$ of the evaluation task with $receiveChan\ inputSenderReceiver$. It then sends the serialized version of a to be evaluated, $serialized \leftarrow liftIO\ \$\ trySerialize\ a$ over that $SendPort$ with $sendChan\ inputSender\ \$\ toThunk\ serialized$ to the evaluating slave node. Then, it awaits the result of the evaluation with $forcedA \leftarrow receiveChan\ outputReceiver$ to finally put it inside the passed $MVar\ a$ with $liftIO\ \$\ putMVar\ out\ forcedA$.

[7] here the type is a instead of some potentially other b because we only evaluate some a

```
forceSingle :: (Evaluatable a) ⇒ NodeId → MVar a → a → Process ()
forceSingle node out a = do
    -- create the Channel that we use to send the
    -- Sender of the input from the slave node from
  (inputSenderSender, inputSenderReceiver) ← newChan
    -- create the channel to receive the output from
  (outputSender, outputReceiver) ← newChan
    -- spawn the actual evaluation task on the given node
    -- and pass the two sender objects we created above
  spawn node (evalTask (inputSenderSender, outputSender))
    -- wait for the slave to send the input sender
  inputSender ← receiveChan inputSenderReceiver
  serialized ← liftIO $ trySerialize a
    -- send the input to the slave
  sendChan inputSender $ toThunk serialized
    -- wait for the result from the slave
  forcedA ← receiveChan outputReceiver
    -- put the output back into the passed MVar
  liftIO $ putMVar out forcedA
```

Slave node

In the definition of *forceSingle* we use a function

```
evalTask :: (SendPort (SendPort (Thunk a)), SendPort a) → Closure (Process ())
```

As indicated by the *Evaluatable a* in the type signature, this function is hosted on a *Evaluatable a* type class:

```
class (Binary a, Typeable a, NFData a) ⇒ Evaluatable a where
  evalTask :: (SendPort (SendPort (Thunk a)), SendPort a) →
    Closure (Process ())
```

This abstraction is required because of the way Cloud Haskell does serialization. We can not write a single definition *evalTask* and expect it to work even though it would be a valid definition. This is because for Cloud Haskell to be able to create the required serialization code, at least in our tests, we require a fixed type like for example for *Int*s: *evalTaskInt* :: (*SendPort* (*SendPort* (*Thunk Int*)), *SendPort Int*) →

$Closure\ (Process\ ())$ If we then make this function remotable with $\$(remotable\ ['evalTaskInt])$, we can write a valid Cloud Haskell compatible instance $Evaluatable\ Int$ simply as

```
instance Evaluatable Int where
  evalTask = evalTaskInt
```

These instances and evaluation tasks can however easily be generated with the Template Haskell code generator in Fig. 10.10 from the Appendix via calls to the following three Template Haskell functions:

```
$ (mkEvalTasks ["Int])
$ (mkRemotables ["Int])
$ (mkEvaluatables ["Int])
```

This is possible because $evalTaskInt$ just like any other function on types that have instances for $Binary\ a$, $Typeable\ a$, and $NFData\ a$ can be just delegated to $evalTaskBase$, which behaves as follows: Starting off, it creates the channel that it wants to receive its input from with $(sendMaster, rec) \leftarrow newChan$. Then it sends the $SendPort\ (Thunk\ a)$ of this channel back to the master process via $sendChan\ inputPipe\ sendMaster$ to then receive its actual input on the $ReceivePort\ (Thunk\ a)$ end with $thunkA \leftarrow receiveChan\ rec$. It then deserializes this thunk with $a \leftarrow liftIO\ \$\ deserialize\ \$\ fromThunk\ thunkA$ and sends the fully evaluated result back with $sendChan\ output\ (seq\ (rnf\ a)\ a)$. Its complete definition is

```
evalTaskBase :: (Binary a, Typeable a, NFData a) =>
  (SendPort (SendPort (Thunk a)), SendPort a) -> Process ()
evalTaskBase (inputPipe, output) = do
  (sendMaster, rec) <- newChan
    -- send the master the SendPort, that we
    -- want to listen the other end on for the input
  sendChan inputPipe sendMaster
    -- receive the actual input
  thunkA <- receiveChan rec
    -- and deserialize
  a <- liftIO $ deserialize $ fromThunk thunkA
    -- force the input and send it back to master
  sendChan output (seq (rnf a) a)
```

6.2.3 Parallel Evaluation Scheme

Since we now know how to evaluate a value on slave nodes via $forceSingle :: (Evaluatable\ a) \Rightarrow NodeId \rightarrow MVar\ a \rightarrow a \rightarrow Process\ ()$, we can use this to build up an internal parallel evaluation scheme we require in order to fit the $ArrowParallel$ type class. For this we start by defining an abstraction of a computation as

$$
\begin{aligned}
&\textbf{data}\ Computation\ a = Comp\ \{ \\
&\quad computation :: IO\ (), \\
&\quad result :: IO\ a \\
&\}
\end{aligned}
$$

where $computation :: IO\ ()$ is the $IO\ ()$ action that is required to be evaluated so that we can get a result from $result :: IO\ a$.

Next is the definition of $evalSingle :: Evaluatable \Rightarrow Conf \rightarrow NodeId \rightarrow a \rightarrow IO\ (Computation\ a)$. Its resulting IO action starts by creating an empty $MVar\ a$ with $mvar \leftarrow newEmptyMVar$. Then it creates an IO action that forks away the evaluation process of $forceSingle$ on the single passed value a by means of $forkProcess :: LocalNode \rightarrow Process\ () \rightarrow IO\ ProcessId$ on the the master node with

$$forkProcess\ (localNode\ conf)\ \$\ forceSingle\ node\ mvar\ a$$

The action concludes by returning a $Computation\ a$ encapsulating the evaluation $IO\ ()$ action and the result communication action $takeMVar\ mvar :: IO\ a$:

$$
\begin{aligned}
&evalSingle :: Evaluatable\ a \Rightarrow Conf \rightarrow NodeId \rightarrow a \rightarrow IO\ (Computation\ a) \\
&evalSingle\ conf\ node\ a = \textbf{do} \\
&\quad mvar \leftarrow newEmptyMVar \\
&\quad \textbf{let}\ comp = forkProcess\ (localNode\ conf)\ \$\ forceSingle\ node\ mvar\ a \\
&\quad return\ \$\ Comp\ \{ \\
&\qquad computation = comp \gg return\ () \\
&\qquad result = takeMVar\ mvar \\
&\qquad \}
\end{aligned}
$$

With this we can then easily define a function $evalParallel :: Evaluatable\ a \Rightarrow Conf \rightarrow [a] \rightarrow IO\ (Computation\ [a])$ that builds an IO action containing a parallel $Computation\ [a]$ from an input list $[a]$. This IO action starts by retrieving the current list of workers with $workers \leftarrow readMVar\ \$\ workers\ conf$. It then continues by shuffling this list of workers with $shuffledWorkers \leftarrow randomShuffle\ workers$[8]

[8] $randomShuffle :: [a] \rightarrow IO\ [a]$ from `https://wiki.haskell.org/Random_shuffle`

to ensure at least some level of equal work distribution between multiple calls to *evalParallel*. It then assigns the input values a to their corresponding workers to finally build the list of parallel computations $[Computation\ a]$ with $comps \leftarrow sequence\ \$\ map\ (uncurry\ \$\ evalSingle\ conf)\ workAssignment$. It concludes by turning this list $[Computation\ a]$ into a computation of a list $Computation\ [a]$ with $return\ \$\ sequenceComp\ comps$.

```
evalParallel :: Evaluatable a ⇒ Conf → [a] → IO (Computation [a])
evalParallel conf as = do
  workers ← readMVar $ workers conf
    -- shuffle the list of workers, so we don't end up spawning
    -- all tasks in the same order everytime
  shuffledWorkers ← randomShuffle workers
    -- complete the work assignment node to task (NodeId, a)
  let workAssignment = zipWith (,) (cycle shuffledWorkers) as
    -- build the parallel computation with sequence
  comps ← sequence $ map (uncurry $ evalSingle conf) workAssignment
  return $ sequenceComp comps
```

Here, the definition of $sequenceComp :: [Computation\ a] \rightarrow Computation\ [a]$ is

```
sequenceComp :: [Computation a] → Computation [a]
sequenceComp comps = Comp {computation = newComp, result = newRes}
  where newComp = sequence_ $ map computation comps
        newRes = sequence $ map result comps
```

Now, in order to start the actual computation from a blueprint in $Computation\ a$ and get the result back as a pure value a, we have to use the function $runComputation :: IO\ (Computation\ a) \rightarrow a$ defined as follows: Internally it uses an $IO\ a$ action that starts by unwrapping $Computation\ a$ from the input $IO\ (Computation\ a)$ with $comp \leftarrow x$ to then launch the actual evaluation with $computation\ comp$. It then finally returns the result with $result\ comp$. Finally, in order to turn the $IO\ a$ action into a, we have to use $unsafePerformIO :: IO\ a \rightarrow a$ which is a useful function to turn IO actions into their pure values and is generally avoided because it can introduce severe bugs if not handled with utmost care. Here its use is necessary and absolutely fine, though, since we only do evaluation inside the IO monad evaluation and if this were to fail, the computation would be wrong anyways. Also in order to force the compiler to not inline the result - which is generally okay in pure functions but not in this case as we do not want to spawn the computation multiple times - we protect the definition of $runComputation$ with a $NOINLINE$ pragma:

```
{-# NOINLINE runComputation #-}
runComputation :: IO (Computation a) → a
runComputation x = unsafePerformIO $ do
  comp ← x
  computation comp
  result comp
```

6.3 Implementing the PArrows API

Finally, we describe in this Chapter how to implement the PArrows API with the Cloud Haskell code provided in this experiment and evaluate our results.

We start by explaining how to implement *ArrowParallel* in Chapter 6.3.1. Then, we discuss the limits of the current code: Why we can not yet give a proper instance for *ArrowLoopParallel* or a proper *Future* implementation in Chapter 6.3.2. We finally lay out a possible solution to this which could be implemented in the future in Chapter 6.3.3.

6.3.1 *ArrowParallel* instance

We will now give an experimental implementation of the *ArrowParallel* type class with Cloud Haskell. Obviously, as already mentioned earlier, here the additional conf parameter is the *State / Conf* type we have discussed in detail earlier.

We implement *parEvalN* of our *ArrowParallel arr a b Conf* instance as follows: We start off by forcing the input $[a]$ into normal form with *arr force*. During testing this was found necessary because a not fully evaluated value a can still have attached things like a file handle which may be not serializable. Then, the parallel Arrow goes on to feed the now fully forced input list $[a]$ into the evaluation Arrow obtained by applying $evalN :: [arr\ a\ b] \rightarrow arr\ [a]\ [b]$ to the list of arrows to be parallelized $[arr\ a\ b]$ with *evalN fs*. This results in a not yet evaluated list of results $[b]$ which is then forked away with $arr\ (evalParallel\ conf) :: arr\ [a]\ (Computation\ [b])$. The resulting computation blueprint is then executed with $arr\ runComputation :: arr\ (Computation\ [b])\ [b]$.

```
instance (NFData a, Evaluatable b, ArrowChoice arr) ⇒
  ArrowParallel arr a b Conf where
    parEvalN conf fs =
      arr force >>>
      evalN fs >>>
```

$arr\ (evalParallel\ conf) \ggg$
$arr\ runComputation$

6.3.2 Limits of the current implementation

Similar to the GpH and Par Monad backends, the current code as explained earlier in this Chapter, the experimental Cloud Haskell backend suffers from the problem that it does not work in conjunction with the looping skeletons $pipe/ring/torus$ described in this thesis. All testing programs would refuse to compute anything and hang indefinitely. While this is no big problem for the shared-memory backends where we could just implement a workaround with the help of an $ArrowLoopParallel$ instance

$\textbf{instance}\ (ArrowChoice\ arr, ArrowParallel\ arr\ a\ b\ Conf) \Rightarrow$
$\quad ArrowLoopParallel\ arr\ a\ b\ Conf\ \textbf{where}$
$\quad loopParEvalN\ \_ = evalN$
$\quad postLoopParEvalN = parEvalN$

a similar solution would not be feasible here because we are in a distributed-memory setting with Cloud Haskell. The skeletons would become meaningless as all benefits of using a sophisticated distributed skeleton would be lost.

Since it wouldn't make sense to have a $Future$ instance without proper support for skeletons that could make use of it, we also do not give an implementation for a $CloudFuture$ in this thesis.

6.3.3 Possible mitigation of the limits

While investigating the problem with the looping skeletons, we noticed a difference in behaviour between Eden and all other backends including our experimental Cloud Haskell backend: Eden streams lists of data $[a]$ instead of sending the complete list as one big serialized chunk. Another difference is that tuples of data $(a, b, ...)$ are also sent in parallel on n threads for a tuple of n entries. When investigating the $torus$ or $ring$ skeletons we ported from Eden, we notice how these two specialities are important. For example, in the $ring$ skeleton we build up the resulting Arrow so that it calculates the result in multiple rounds:

$ring\ conf\ f =$
$\quad loop\ (second\ (rightRotate \ggg lazy) \ggg$
$\qquad$ -- convert the current input into a form we can process in this round

$$
\begin{aligned}
&\quad arr\ (uncurry\ zip) \ggg \\
&\quad\quad \text{-- here, we evaluate the current round} \\
&\quad loopParEvalN\ conf\ (repeat\ (second\ (get\ conf) \ggg f \ggg second\ (put\ conf))) \ggg \\
&\quad\quad \text{-- put the current result back into the original input form} \\
&\quad arr\ unzip) \ggg \\
&postLoopParEvalN\ conf\ (repeat\ (arr\ id))
\end{aligned}
$$

Here, anything other than the exact same behaviour as Eden will result in a dead-lock when using $loopParEvalN = parEvalN$. We therefore believe that it is crucial for a proper Cloud Haskell backend to have the same streaming behaviour as Eden does. While we are confident that this is definitely possible to achieve with Cloud Haskell as early experiments on this suggest, we have to date not been able to achieve proper streaming behaviour. We stopped developing this further as this would have bursted the scope of this thesis.

The most promising idea to implement this is to use a more sophisticated mechanism to stream data back to the master node by using pipes along the lines of

$$
\begin{aligned}
&\textbf{type}\ PipeIn\ a = SendPort\ (SendPort\ (Maybe\ (SendPort\ (Maybe\ a)))) \\
&\textbf{type}\ PipeOut\ a = ReceivePort\ (SendPort\ (Maybe\ (SendPort\ (Maybe\ a))))
\end{aligned}
$$

where $PipeIn\ a$ would be the port where the evaluating process on the slave node would send its result through to the corresponding $PipeOut\ a$ on the master node. Note that we here encode a „stream“ of some a with $SendPort\ (Maybe\ a)$: For types with singular values, we just request one value. And on types like e.g. a list $[a]$ we expect multiple singleton lists $Just\ [a]$, on the $SendPort$ and the end of the input with $Nothing$. For other multi-valued types, this would work similar even if some hacks would be required.

Then in order to communicate the result from the slave node, we would first send $SendPort\ (Maybe\ (SendPort\ (Maybe\ a)))$ on which the slave-node would want to receive the stream of $SendPort\ (Maybe\ a)$. This stream of $SendPort$s is required instead of a singular $SendPort$ because of types that have to be sent by multiple threads like e.g. tuples. Via these $SendPort\ (Maybe\ a)$s the slave can then finally communicate the stream of evaluated results back to the master node. A corresponding communication scheme doing the necessary opposite tasks would obviously be required on the master node.

During testing, as already mentioned, we were not successful in making this idea work with the looping skeletons.[9] We still believe that this path is worth exploring

[9] It did however still work with non-looping skeletons.

further in the future, though. For the sake of this thesis we however leave the experiment as it is presented in the earlier parts of this Chapter.

7 Performance results and discussion

The preceding sections have shown what PArrows are and how expressive they are. In this Chapter we will now evaluate the performance overhead of our compositional abstraction in comparison to GpH and the Par Monad on shared memory architectures and Eden on a distributed memory cluster.[1] We describe our measurement platform (Chapter 7.1), the benchmark results (Chapter 7.1.2) – the shared-memory variants (GpH, Par Monad and Eden CP) followed by Eden in a distributed-memory setting, and conclude that PArrows hold up in terms of performance when compared to the original parallel Haskells (Chapter 7.3).

7.1 Measurement platform

We start by explaining the hardware and software stack and outline the benchmark programs and motivation for choosing them. We also shortly address hyper-threading and why we do not use it in our benchmarks.

7.1.1 Hardware and software

The benchmarks are executed both in a shared and in a distributed memory setting using the Glasgow GPG Beowulf cluster, consisting of 16 machines with 2 Intel® Xeon® E5-2640 v2 and 64 GB of DDR3 RAM each. Each processor has 8 cores and 16 (hyper-threaded) threads with a base frequency of 2 GHz and a turbo frequency of 2.50 GHz. This results in a total of 256 cores and 512 threads for the whole cluster. The operating system was Ubuntu 14.04 LTS with Kernel 3.19.0-33. Non-surprisingly, we found that hyper-threaded 32 cores do not behave in the same manner as real 16 cores (numbers here for a single machine). We disregarded the hyper-threading ability in most of the cases.

Apart from Eden, all benchmarks and libraries were compiled with Stack's[2] lts-7.1 GHC compiler which is equivalent to a standard GHC 8.0.1 with the base package

[1] We do not include the Cloud Haskell backend here, as it is still a work-in-progress.
[2] see `https://www.haskellstack.org/`

in version 4.9.0.0. Stack itself was used in version 1.3.2. For GpH in its Multicore variant we used the parallel package in version 3.2.1.0[3], while for the Par Monad we used monad-par in version 0.3.4.8[4]. For all Eden tests, we used its GHC-Eden compiler in version 7.8.2[5] together with OpenMPI 1.6.5[6].

Furthermore, all benchmarks were done with help of the bench[7] tool in version 1.0.2 which uses criterion (>=1.1.1.0 && < 1.2)[8] internally. All runtime data (mean runtime, max stddev, etc.) was collected with this tool.

We used a single node with 16 real cores as a shared memory test-bed and the whole grid with 256 real cores as a device to test our distributed memory software.

7.1.2 Benchmarks

We measure four benchmarks from different sources. Most of them are parallel mathematical computations, initially implemented in Eden. Table 7.1 summarises.

Table 7.1: The benchmarks we use in this paper.

Name	Area	Type	Origin	Source
Rabin–Miller test	Mathematics	$parMap$ + $reduce$	Eden	Lobachev (2012)
Jacobi sum test	Mathematics	$workpool$ + $reduce$	Eden	Lobachev (2012)
Gentleman	Mathematics	$torus$	Eden	Loogen et al. (2003)
Sudoku	Puzzle	$parMap$	Par Monad	Marlow et al. (2011)[9]

Rabin–Miller test is a probabilistic primality test that iterates multiple (here: 32–256) „subtests“. Should a subtest fail, the input is definitely not a prime. If all n subtest pass, the input is composite with the probability of $1/4^n$.

Jacobi sum test or APRCL is also a primality test, that however, guarantees the correctness of the result. It is probabilistic in the sense that its run time is not certain. Unlike Rabin–Miller test, the subtests of Jacobi sum test have very different durations. Lobachev (2011) discusses some optimisations of parallel APRCL.

[3] see `https://hackage.haskell.org/package/parallel-3.2.1.0`
[4] see `https://hackage.haskell.org/package/monad-par-0.3.4.8`
[5] see `http://www.mathematik.uni-marburg.de/~eden/?content=build_eden_7_&navi=build`
[6] see `https://www.open-mpi.org/software/ompi/v1.6/`
[7] see `https://hackage.haskell.org/package/bench`
[8] see `https://hackage.haskell.org/package/criterion-1.1.1.0`
[9] actual code from: `http://community.haskell.org/~simonmar/par-tutorial.pdf` and `https://github.com/simonmar/parconc-examples}`

Generic parallel implementations of Rabin–Miller test and APRCL were presented in Lobachev (2012).

„Gentleman“ is a standard Eden test program, developed for their $torus$ skeleton. It implements a Gentleman's algorithm for parallel matrix multiplication (Gentleman, 1978). We ported an Eden-based version (Loogen et al., 2003) to PArrows.

A parallel Sudoku solver was used by Marlow et al. (2011) to compare Par Monad to GpH, we ported it to PArrows.

7.1.3 What parallel Haskells run where

The Par monad and GpH – in its multicore version (Marlow et al., 2009) – can be executed on shared memory machines only. Although GpH is available on distributed memory clusters, and newer distributed memory Haskells such as HdpH exist, current support of distributed memory in PArrows is limited to Eden. We used the MPI backend of Eden in a distributed memory setting. However, for shared memory Eden features a „CP“ backend that merely copies the memory blocks between disjoint heaps. In this mode, Eden still operates in the „nothing shared“ setting, but is adapted better to multicore machines. We call this version of Eden „Eden CP“.

7.2 Benchmark results

We compare the PArrow performance with direct implementations of the benchmarks in Eden, GpH and the Par Monad. We start with the definition of mean overhead to compare both PArrows-enabled and standard benchmark implementations. We continue comparing speedups and overheads for the shared memory implementations and then study OpenMPI variants of the Eden-enabled PArrows as a representative of a distributed memory backend. We plot all speedup curves and all overhead values in the Appendix in 10.6 and and 10.7 for the shared memory and distributed memory benchmarks, respectively.

7.2.1 Defining overhead

We compare the mean overhead, i.e. the mean of relative wall-clock run time differences between the PArrow and direct benchmark implementations executed multiple times with the same settings. The error margins of the time measurements, supplied by criterion package[10], yield the error margin of the mean overhead.

[10] `https://hackage.haskell.org/package/criterion-1.1.1.0`

Quite often the zero value lies in the error margin of the mean overhead. This means that even though we have measured some difference (against or even in favour of PArrows), it could be merely the error margin of the measurement and the difference might not be existent. We are mostly interested in the cases where above issue does not persist, we call them *significant*. We often denote the error margin with $\pm$ after the mean overhead value.

7.2.2 Shared memory

Speedup

The Rabin–Miller benchmark showed almost linear speedup for both 32 and 64 tasks, the performance is slightly better in the latter case: 13.7 at 16 cores for input $2^{11213} - 1$ and 64 tasks in the best case scenario with Eden CP. The performance of the Sudoku benchmark merely reaches a speedup of 9.19 (GpH), 8.78 (Par Monad), 8.14 (Eden CP) for 16 cores and 1000 Sudokus. In contrast to Rabin–Miller, here the GpH seems to be the best of all, while Rabin–Miller profited most from Eden CP (i.e. Eden with direct memory copy) implementation of PArrows. Gentleman on shared memory has a plummeting speedup curve with GpH and Par Monad and logarithmically increasing speedup for the Eden-based version. The latter reached a speedup of 6.56 at 16 cores.

Overhead

For the shared memory Rabin–Miller benchmark, implemented with PArrows using Eden CP, GpH, and Par Monad, the overhead values are within single percents range, but also negative overhead (i.e. PArrows are better) and larger error margins happen. To give a few examples, the overhead for Eden CP with input value $2^{11213}-1$, 32 tasks, and 16 cores is 1.5%, but the error margin is around 5.2%! Same implementation in the same setting with 64 tasks reaches -0.2% overhead, PArrows apparently fare better than Eden – but the error margin of 1.9% disallows this interpretation. We focus now on significant overhead values. To name a few: $0.41\% \pm 7 \cdot 10^{-2}\%$ for Eden∼CP and 64 tasks at 4 cores; $4.7\% \pm 0.72\%$ for GpH, 32 tasks, 8 cores; $0.34\% \pm 0.31\%$ for Par Monad at 4 cores with 64 tasks. The worst significant overhead was in case of GpH with $8\% \pm 6.9\%$ at 16 cores with 32 tasks and input value $2^{11213} - 1$. In other words, we notice no major slow-down through PArrows here.

For Sudoku the situation is slightly different. There is a minimal significant ($-1.4\% \pm 1.2\%$ at 8 cores) speed *improvement* with PArrows Eden CP version when compared with the base Eden CP benchmark. However, with increasing number of cores the error margin reaches zero again: $-1.6\% \pm 5.0\%$ at 16 cores. The Par Monad shows a similar development, e.g. with $-1.95\% \pm 0.64\%$ at 8 cores. The GpH version shows both a significant speed improvement of $-4.2\% \pm 0.26\%$ (for 16 cores) with PArrows and a minor overhead of $0.87\% \pm 0.70\%$ (4 cores).

The Gentleman multiplication with Eden CP shows a minor significant overhead of $2.6\% \pm 1.0\%$ at 8 cores and an insignificant improvement at 16 cores. Summarising, we observe a low (if significant at all) overhead, induced by PArrows in the shared memory setting.

7.2.3 Distributed memory

Speedup

The speedup of distributed memory Rabin–Miller benchmark with PArrows and Eden showed an almost linear speedup excepting around 192 cores where an unfortunate task distribution reduces performance. As seen in Fig. 7.1, we reached a speedup of 213.4 with PArrrows at 256 cores (vs. 207.7 with pure Eden). Because of memory limitations, the speedup of Jacobi sum test for large inputs (such as $2^{4253} - 1$) could be measured only in a massively distributed setting. PArrows improved there from $9193s$ (at 128 cores) to $1649s$ (at 256 cores). A scaled-down version with input $2^{3217} - 1$ stagnates the speedup at about 11 for both PArrows and Eden for more than 64 cores. There is apparently not enough work for that many cores. The Gentleman test with input 4096 had an almost linear speedup first, then plummeted between 128 and 224 cores, and recovered at 256 cores with speedup of 129.

Overhead

We use our mean overhead quality measure and the notion of significance also for distributed memory benchmarks. The mean overhead of Rabin–Miller test in the distributed memory setting ranges from 0.29% to -2.8% (last value in favour of PArrows), but these values are not significant with error margins $\pm 0.8\%$ and $\pm 2.9\%$ correspondingly. A sole significant (by a very low margin) overhead is $0.35\% \pm 0.33\%$ at 64 cores. We measured the mean overhead for Jacobi benchmark for an input of $2^{3217} - 1$ for up to 256 cores. We reach the flattering value $-3.8\% \pm 0.93\%$ at 16 cores in favour of PArrows, it was the sole significant overhead value. The

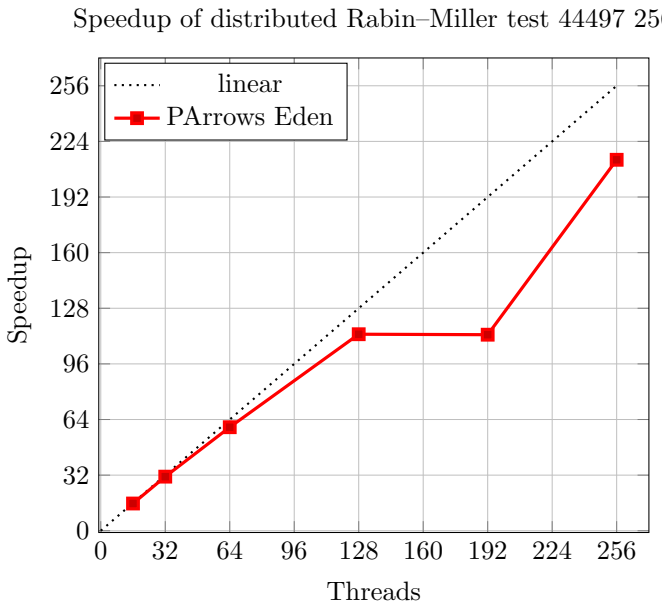

Figure 7.1: Speedup of the distributed Rabin–Miller benchmark using PArrows with Eden.

value for 256 cores was $0.31\% \pm 0.39\%$. Mean overhead for distributed Gentleman multiplication was also low. Significant values include $1.23\% \pm 1.20\%$ at 64 cores and $2.4\% \pm 0.97\%$ at 256 cores. It took PArrows 64.2 seconds at 256 cores to complete the benchmark.

Similar to the shared memory setting, PArrows only imply a very low penalty with distributed memory that lies in lower single-percent digits at most.

7.3 Evaluation of results

Table 7.2: Overhead in the shared memory benchmarks. Bold marks values in favour of PArrows.

Benchmark	Base	Mean of mean overheads	Maximum normalised stdDev	Runtime for 16 cores (s)
Sudoku 1000	Eden CP	**-2.1%**	5.1%	1.17
	GpH	**-0.82%**	0.7%	1.11
	Par Monad	**-1.3%**	2.1%	1.14
Gentleman 512	Eden CP	*0.81%*	6.8%	1.66
Rabin–Miller 11213 32	Eden CP	*0.79%*	5.2%	5.16
	GpH	*3.5%*	6.9%	5.28
	Par Monad	**-2.5%**	19.0%	5.84
Rabin–Miller 11213 64	Eden CP	*0.21%*	1.9%	10.3
	GpH	*1.6%*	1.3%	10.6
	Par Monad	**-4.0%**	17.0%	11.4

Table 7.3: Overhead in the distributed memory benchmarks. Bold marks values in favour of PArrows.

Benchmark	Base	Mean of mean overheads	Maximum normalised stdDev	Runtime for 256 cores (s)
Gentleman 4096	Eden	*0.67%*	1.5%	110.0
Rabin–Miller 44497 256	Eden	**-0.5%**	2.9%	165.0
Jacobi Test 3217	Eden	**-0.74%**	1.6%	635.0

PArrows performed in our benchmarks with little to no overhead. Tables 7.2 and 7.3 clarify this once more: The PArrows-enabled versions trade blows with their vanilla counterparts when comparing the means over all cores of the mean overheads. If we combine these findings with the usability of our DSL, the minor overhead induced by PArrows is outweighed by their convenience and usefulness to the user.

8 Discussion

In this thesis we have seen that...

PArrows are an extendable formalism, they can be easily ported to further parallel Haskells while still maintaining interchangeability. It is straightforward to provide further implementations of algorithmic skeletons in PArrows.

9 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are first to represent *parallel* computation with Arrows, and hence to show their usefulness for composing parallel programs. We have shown that for a generic and extensible parallel Haskell, we do not have to restrict ourselves to a monadic interface. We argue that Arrows are better suited to parallelise pure functions than Monads, as the functions are already Arrows and can be used directly in our DSL. Arrows are a better fit to parallelise pure code than a monadic solution as regular functions are already Arrows and can be used with our DSL in a more natural way. We use a non-monadic interface (similar to Eden or GpH) and retain composability. The DSL allows for a direct parallelisation of monadic code via the Kleisli type and additionally allows to parallelise any Arrow type that has an instance for $ArrowChoice$. (Some skeletons require an additional $ArrowLoop$ instance.)

We have demonstrated the generality of the approach by exhibiting PArrow implementations for Eden, GpH, and the Par Monad. Hence, parallel programs can be ported between task parallel Haskell implementations with little or no effort. We are confident that it will be straightforward to add other task-parallel Haskells. Our measurements of four benchmarks on both shared and distributed memory platforms shows that the generality and portability of PArrows has very low performance overheads, i.e. never more than $8\% \pm 6.9\%$ and typically under 2%.

9.1 Future work

Our PArrows DSL can be expanded to other task parallel Haskells, and a specific target is HdpH (Maier et al., 2014). Further Future-aware versions of Arrow combinators can be defined. Existing combinators could also be improved, for example more specialised versions of $\ggg$ and $***$ combinators are viable.

In ongoing work we are expanding both our skeleton library and the number of skeleton-based parallel programs that use our DSL. It would also be interesting to see a hybrid of PArrows and Accelerate (McDonell et al., 2015). Ports of our approach to other languages such as Frege, Eta, or Java directly are at an early development stage.

10 Appendix

10.1 Utility Arrows

Following are definitions of some utility Arrows used in this paper that have been left out for brevity. We start with the $second$ combinator from Hughes (2000), which is a mirrored version of *first* used for example in the definition of $***$:

$$
\begin{aligned}
&second :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ (c, a)\ (c, b) \\
&second\ f = arr\ swap \ggg first\ f \ggg arr\ swap \\
&\quad \mathbf{where}\ swap\ (x, y) = (y, x)
\end{aligned}
$$

Next, we give the definition of $evalN$ which also helps us to define map, and $zipWith$ on Arrows. The $evalN$ combinator in Fig. 10.1 converts a list of Arrows $[arr\ a\ b]$ into an Arrow $arr\ [a]\ [b]$.

$$
\begin{aligned}
&evalN :: (ArrowChoice\ arr) \Rightarrow [arr\ a\ b] \rightarrow arr\ [a]\ [b] \\
&evalN\ (f : fs) = arr\ listcase \ggg \\
&\quad arr\ (const\ [\,]) \Vert\!\Vert (f *** evalN\ fs \ggg arr\ (uncurry\ (:))) \\
&\quad \mathbf{where}\ listcase\ [\,] = Left\ () \\
&\qquad listcase\ (x : xs) = Right\ (x, xs) \\
&evalN\ [\,] = arr\ (const\ [\,])
\end{aligned}
$$

Figure 10.1: The definition of $evalN$.

The $mapArr$ combinator (Fig. 10.2) lifts any Arrow $arr\ a\ b$ to an Arrow $arr\ [a]\ [b]$. The original inspiration was from Hughes (2005), but the definition was then unified with $evalN$.

$$
\begin{aligned}
&mapArr :: ArrowChoice\ arr \Rightarrow arr\ a\ b \rightarrow arr\ [a]\ [b] \\
&mapArr = evalN \circ repeat
\end{aligned}
$$

Figure 10.2: The definition of map over Arrows.

Finally, with the help of $mapArr$ (Fig. 10.2), we can define $zipWithArr$ (Fig. 10.3) that lifts any Arrow $arr\ (a, b)\ c$ to an Arrow $arr\ ([a], [b])\ [c]$.

$$zipWithArr :: ArrowChoice\ arr \Rightarrow arr\ (a, b)\ c \rightarrow arr\ ([a], [b])\ [c]$$
$$zipWithArr\ f = (arr\ (\lambda(as, bs) \rightarrow zipWith\ (,)\ as\ bs)) \ggg mapArr\ f$$

Figure 10.3: $zipWith$ over Arrows.

These combinators make use of the $ArrowChoice$ type class which provides the $\|\|\|$ combinator. It takes two Arrows $arr\ a\ c$ and $arr\ b\ c$ and combines them into a new Arrow $arr\ (Either\ a\ b)\ c$ which pipes all $Left\ a$'s to the first Arrow and all $Right\ b$'s to the second Arrow:

$$(|||) :: ArrowChoice\ arr\ a\ c \rightarrow arr\ b\ c \rightarrow arr\ (Either\ a\ b)\ c$$

10.2 Profunctor Arrows

In Fig. 10.4 we show how specific Profunctors can be turned into Arrows. This works because Arrows are strong Monads in the bicategory $Prof$ of Profunctors as shown by Asada (2010). In Standard GHC $(\ggg)$ has the type $(\ggg) :: Category\ cat \Rightarrow cat\ a\ b \rightarrow cat\ b\ c \rightarrow cat\ a\ c$ and is therefore not part of the $Arrow$ type class like presented in this thesis.[1]

$$\begin{aligned}
&\textbf{instance}\ (Category\ p, Strong\ p) \Rightarrow Arrow\ p\ \textbf{where}\\
&\quad arr\ f = dimap\ id\ f\ id\\
&\quad first = first'\\
&\textbf{instance}\ (Category\ p, Strong\ p, Costrong\ p) \Rightarrow ArrowLoop\ p\ \textbf{where}\\
&\quad loop = loop'\\
&\textbf{instance}\ (Category\ p, Strong\ p, Choice\ p) \Rightarrow ArrowChoice\ p\ \textbf{where}\\
&\quad left = left'
\end{aligned}$$

Figure 10.4: Profunctors as Arrows.

10.3 Additional function definitions

We have omitted some function definitions in the main text for brevity, and redeem this here.

We begin with Arrow versions of Eden's *shuffle*, *unshuffle* and the definition of *takeEach* are in Fig. 10.5. Similarly, Fig. 10.6 contains the definition of Arrow ver-

[1] For additional information on the type classes used, see: `https://hackage.haskell.org/package/profunctors-5.2.1/docs/Data-Profunctor.html` and `https://hackage.haskell.org/package/base-4.9.1.0/docs/Control-Category.html`.

```
shuffle :: (Arrow arr) ⇒ arr [[a]] [a]
shuffle = arr (concat ∘ transpose)
unshuffle :: (Arrow arr) ⇒ Int → arr [a] [[a]]
unshuffle n = arr (λxs → [takeEach n (drop i xs) | i ← [0 .. n − 1]])
takeEach :: Int → [a] → [a]
takeEach n [] = []
takeEach n (x : xs) = x : takeEach n (drop (n − 1) xs)
```

Figure 10.5: *shuffle*, *unshuffle*, *takeEach* definition.

```
lazy :: (Arrow arr) ⇒ arr [a] [a]
lazy = arr (λ∼(x : xs) → x : lazy xs)
rightRotate :: (Arrow arr) ⇒ arr [a] [a]
rightRotate = arr $ λlist → case list of
  [] → []
  xs → last xs : init xs
```

Figure 10.6: *lazy* and *rightRotate* definitions.

sions of Eden's *lazy* and *rightRotate* utility functions. Fig. 10.7 contains Eden's definition of *lazyzip3* together with the utility functions *uncurry3* and *threetotwo*. The full definition of *farmChunk* is in Fig. 4.13. Eden definition of *ring* skeleton is in Fig. 10.8. It follows Loogen (2012).

Furthermore, Fig. 10.9 contains the omitted definitions of *prMM* (sequential matrix multiplication), *splitMatrix* (which splits the a matrix into chunks), *staggerHorizontally* and *staggerVertically* (to pre-rotate the matrices), and lastly *matAdd*, that calculates $A + B$ for two matrices A and B.

```
lazyzip3 :: [a] → [b] → [c] → [(a, b, c)]
lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)
uncurry3 :: (a → b → c → d) → (a, (b, c)) → d
uncurry3 f (a, (b, c)) = f a b c
threetotwo :: (Arrow arr) ⇒ arr (a, b, c) (a, (b, c))
threetotwo = arr $ λ∼(a, b, c) → (a, (b, c))
```

Figure 10.7: *lazyzip3*, *uncurry3* and *threetotwo* definitions.

```
ringSimple :: (Trans i, Trans o, Trans r) ⇒ (i → r → (o, r)) → [i] → [o]
ringSimple f is = os
  where (os, ringOuts) = unzip (parMap (toRD $ uncurry f) (zip is $ lazy ringIns))
    ringIns = rightRotate ringOuts
toRD :: (Trans i, Trans o, Trans r) ⇒ ((i, r) → (o, r)) → ((i, RD r) → (o, RD r))
toRD f (i, ringIn) = (o, release ringOut)
  where (o, ringOut) = f (i, fetch ringIn)
rightRotate :: [a] → [a]
rightRotate [] = []
rightRotate xs = last xs : init xs
lazy :: [a] → [a]
lazy∼(x : xs) = x : lazy xs
```

Figure 10.8: Eden's definition of the *ring* skeleton.

```
prMM :: Matrix → Matrix → Matrix
prMM m1 m2 = prMMTr m1 (transpose m2)
  where
    prMMTr m1′ m2′ = [[sum (zipWith (∗) row col) | col ← m2′] | row ← m1′]
splitMatrix :: Int → Matrix → [[Matrix]]
splitMatrix size matrix = map (transpose ∘ map (chunksOf size)) $ chunksOf size $ matrix
staggerHorizontally :: [[a]] → [[a]]
staggerHorizontally matrix = zipWith leftRotate [0..] matrix
staggerVertically :: [[a]] → [[a]]
staggerVertically matrix = transpose $ zipWith leftRotate [0..] (transpose matrix)
leftRotate :: Int → [a] → [a]
leftRotate i xs = xs2 ++ xs1 where
    (xs1, xs2) = splitAt i xs
matAdd = chunksOf (dimX x) $ zipWith (+) (concat x) (concat y)
```

Figure 10.9: *prMMTr*, *splitMatrix*, *staggerHorizontally*, *staggerVertically* and *matAdd* definition.

10.4 Syntactic sugar

Next, we also give the definitions for some syntactic sugar for PArrows, namely $|***|$ and $|\&\&\&|$. For basic Arrows, we have the $***$ combinator (Fig. 3.7) which allows us to combine two Arrows $arr\ a\ b$ and $arr\ c\ d$ into an Arrow $arr\ (a, c)\ (b, d)$ which does both computations at once. This can easily be translated into a parallel version $|***|$ with the use of $parEval2$, but for this we require a backend which has an implementation that does not require any configuration (hence the $()$ as the $conf$ parameter:

$$(|***|) :: (ArrowChoice\ arr, ArrowParallel\ arr\ (Either\ a\ c)\ (Either\ b\ d)\ ()) \Rightarrow arr\ a\ b \rightarrow arr\ c\ d \rightarrow arr\ (a, c)\ (b, d)$$
$$(|***|) = parEval2\ ()$$

We define the parallel $|\&\&\&|$ in a similar manner to its sequential pendant $\&\&\&$ (Fig. 3.7):

$$(|\&\&\&|) :: (ArrowChoice\ arr, ArrowParallel\ arr\ (Either\ a\ a)\ (Either\ b\ c)\ ()) \Rightarrow arr\ a\ b \rightarrow arr\ a\ c \rightarrow arr\ a\ (b, c)$$
$$(|\&\&\&|)\ f\ g = (arr\ \$\ \lambda a \rightarrow (a, a)) \ggg f\ |***|\ g$$

10.5 Experimental Cloud Haskell backend code

Finally, we include the Template Haskell based code generator to make the experimental Cloud Haskell backend easier to use and a version of the main Sudoku benchmark program as an example.

The code generator can be found in Fig. 10.10. Here, if we enclose this in a Haskell module, the functions $mkEvalTasks$ (to generate the $evalTask$s for the specific types), $mkRemotables$ (to mark the evaluation tasks as remotable in Cloud Haskell) and $mkEvaluatables$ (to create the $Evaluatable$ instance) are the ones exposed to the user.

The Template Haskell version of the main Sudoku benchmark program can be found in Fig. 10.11[2]. We have to write type aliases for $Maybe\ Grid$ ($MaybeGrid$) and $[Maybe\ Grid]$ ($MaybeGridList$). We can then use these to generate the code required to to evaluate these types in the Cloud Haskell backend with. In the $main$ program we have two cases: a) the program is started in master mode and starts

[2]For the full code, see the GitHub repository at `https://github.com/s4ke/Parrows/blob/e1ab76018448d9d4ca3ed48ef1f0c5be26ae34ab/CloudHaskell/testing/Test.hs`

the computation, b) the program is started in slave mode and waits for computation requests. In order to launch this program and have speedup as well, we have to start slave nodes for each cpu core with commands like „<executable> slave 127.0.0.1 8000“ where the last parameter determines the port the slave will listen to and wait for requests on. Similarly a single master node can be started with „<executable> master 127.0.0.1 7999“ where, once again, the last parameter determines the communication port.

```
nested :: Type → Type → Type
nested a b = a `AppT` (ParensT b)

tuple2 :: Type → Type → Type
tuple2 a b = (TupleT 2 `AppT` a) `AppT` b

fn :: Type → Type → Type
fn a b = (ArrowT `AppT` a) `AppT` b

nameToFnName :: Name → Name
nameToFnName (Name (OccName str) _) = mkName $ ("__" ++ str ++ "_evalTaskImpl")

evalTaskFn :: Name → Name → Q [Dec]
evalTaskFn typeName fnName = do
  let sendPort = ConT $ mkName "SendPort"
      thunk = ConT $ mkName "Thunk"
      process = ConT $ mkName "Process"
      firstTup = (sendPort `nested` (sendPort `nested` (thunk `nested` (ConT typeName))))
      secondTup = sendPort `nested` (ConT typeName)
      procNil = process `AppT` (TupleT 0)
  return [
    SigD fnName ((firstTup `tuple2` secondTup) `fn` procNil),
    FunD fnName [Clause [] (NormalB (VarE $ mkName "evalTaskBase")) []]
    ]

evaluatableInstance :: Name → Name → Q [Dec]
evaluatableInstance typeName fnName = do
  let evaluatable = ConT $ mkName "Evaluatable"
  closure ← mkClosure fnName
  return [
    InstanceD (Nothing) [] (evaluatable `nested` ConT typeName) [
      FunD (mkName "evalTask") [Clause [] (NormalB closure) []]
    ]
    ]

mkEvalTasks :: [Name] → Q [Dec]
mkEvalTasks names = do
  let fnNames = map nameToFnName names
  (mapM (uncurry evalTaskFn) (zipWith (,) names fnNames)) >>= (return ∘ concat)

mkRemotables :: [Name] → Q [Dec]
mkRemotables names = do
  let fnNames = map nameToFnName names
  remotable fnNames

mkEvaluatables :: [Name] → Q [Dec]
mkEvaluatables names = do
  let fnNames = map nameToFnName names
  (mapM (uncurry evaluatableInstance) (zipWith (,) names fnNames)) >>= (return ∘ concat)
```

Figure 10.10: The Template Haskell code generator for the Cloud Haskell backend.

```
type MaybeGrid = Maybe Grid
type MaybeGridList = [Maybe Grid]
  -- remotable declaration for all eval tasks
$(mkEvalTasks ["MaybeGrid, "MaybeGridList])
$(mkRemotables ["MaybeGrid, "MaybeGridList])
$(mkEvaluatables ["MaybeGrid, "MaybeGridList])
myRemoteTable :: RemoteTable
myRemoteTable = Main.__remoteTable initRemoteTable
main :: IO ()
main = do
  args ← getArgs
  case args of
    ["master", host, port] → do
      conf ← startBackend myRemoteTable Master host port
      readMVar (workers conf) ≫= print
      grids ← fmap lines $ readFile "sudoku.txt"
      print (length (filter isJust (farm conf 4 solve grids)))
    ["slave", host, port] → do
      startBackend myRemoteTable Slave host port
      print "slave shutdown."
```

Figure 10.11: The Template Haskell version of the Sudoku benchmark program.

10.6 Plots for the shared memory benchmarks

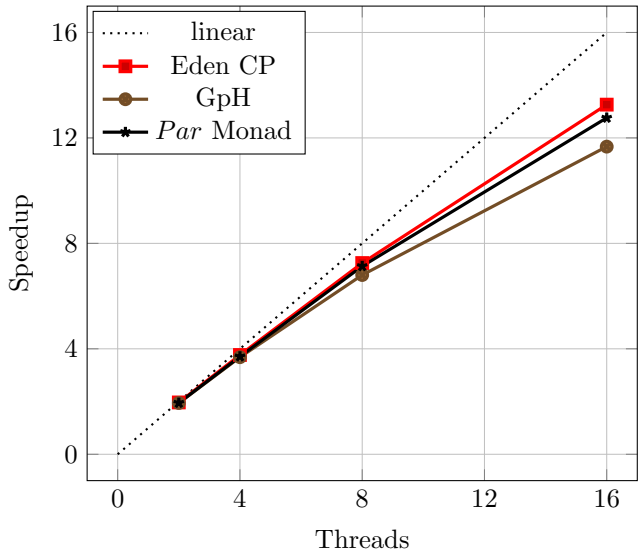

Figure 10.12: Parallel speedup of shared-memory Rabin–Miller test „11213 32“

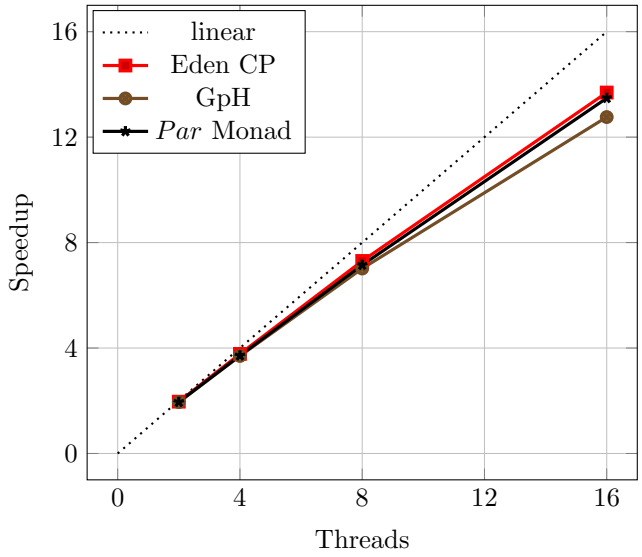

Figure 10.13: Parallel speedup of shared-memory Rabin–Miller test „11213 64“

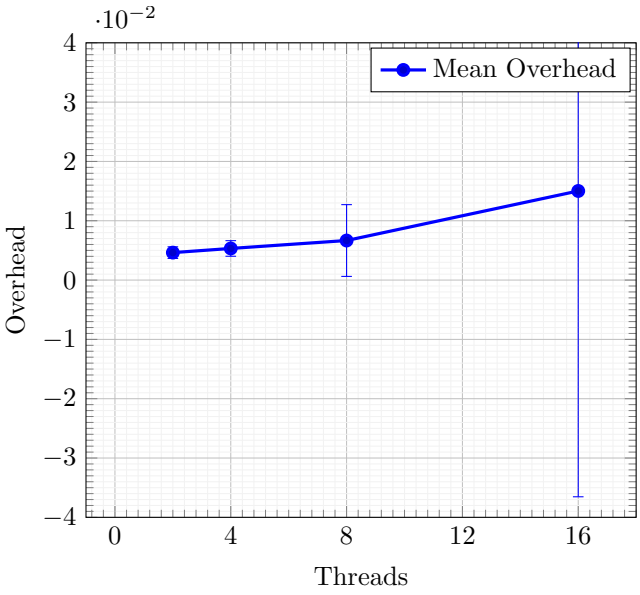

Figure 10.14: Mean overhead for shared-memory Rabin—Miller test „11213 32“ vs Eden CP

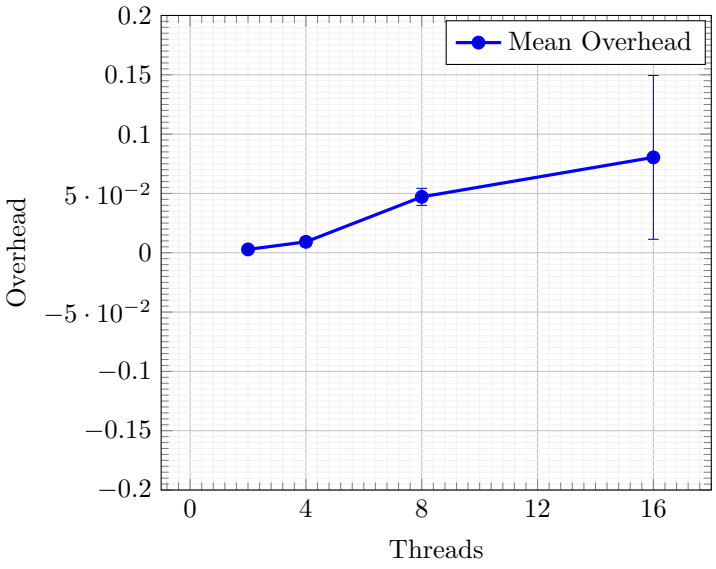

Figure 10.15: Mean overhead for shared-memory Rabin—Miller test „11213 32“ vs GpH

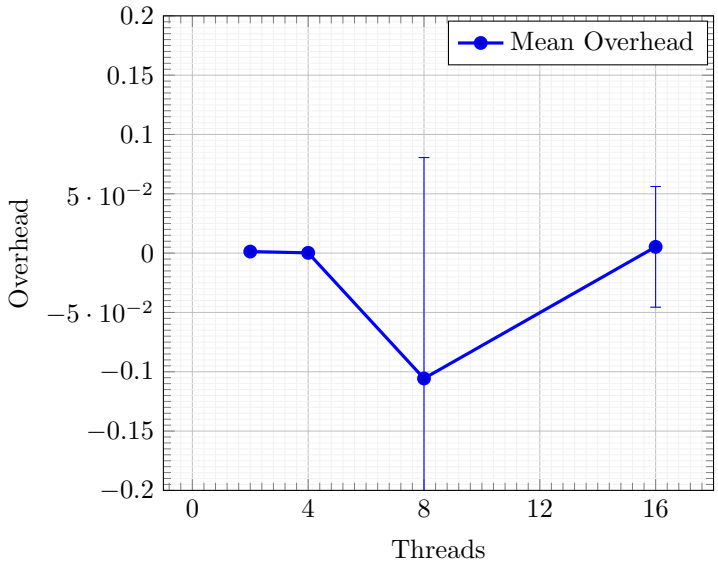

Figure 10.16: Mean overhead for shared-memory Rabin—Miller test „11213 32“ vs *Par* monad

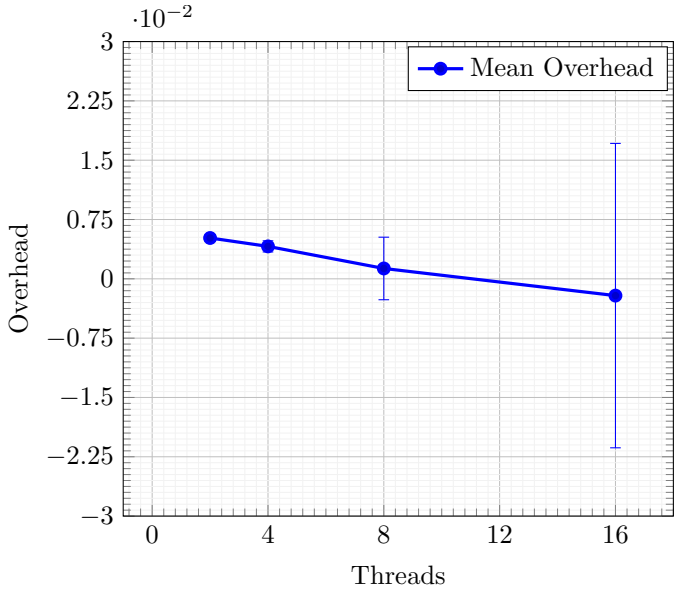

Figure 10.17: Mean overhead for shared-memory Rabin—Miller test „11213 64“ vs Eden CP

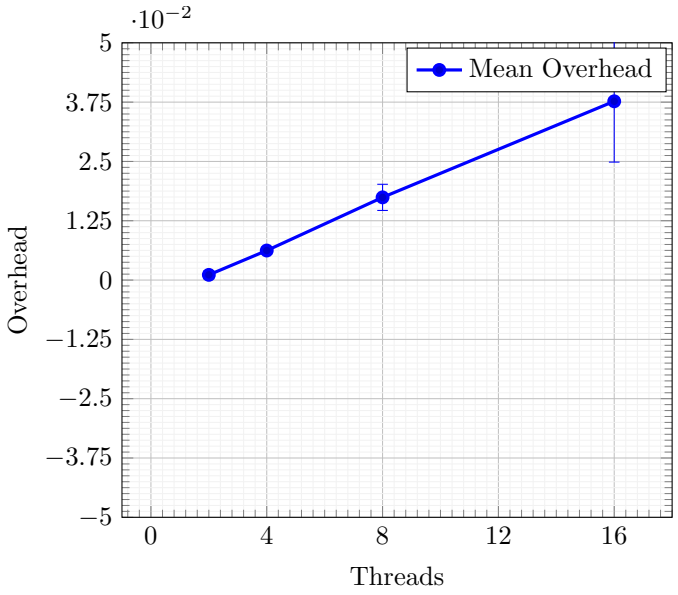

Figure 10.18: Mean overhead for shared-memory Rabin—Miller test „11213 64“ vs GpH

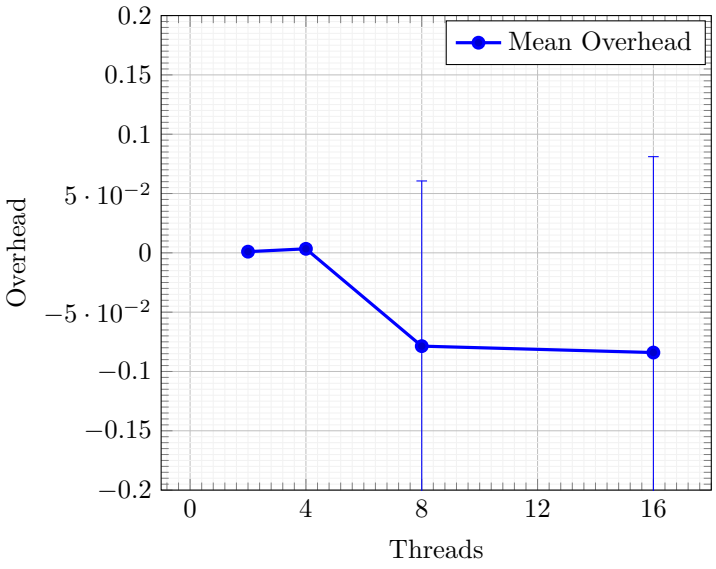

Figure 10.19: Mean overhead for shared-memory Rabin—Miller test „11213 64“ vs *Par* monad

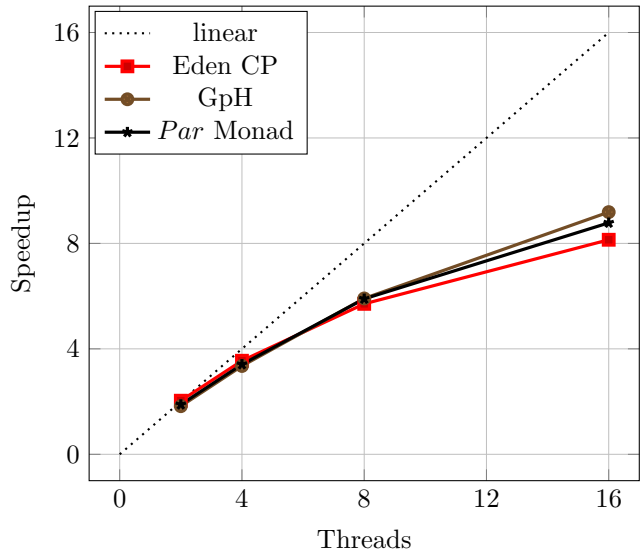

Figure 10.20: Parallel speedup of shared-memory Sudoku „1000“

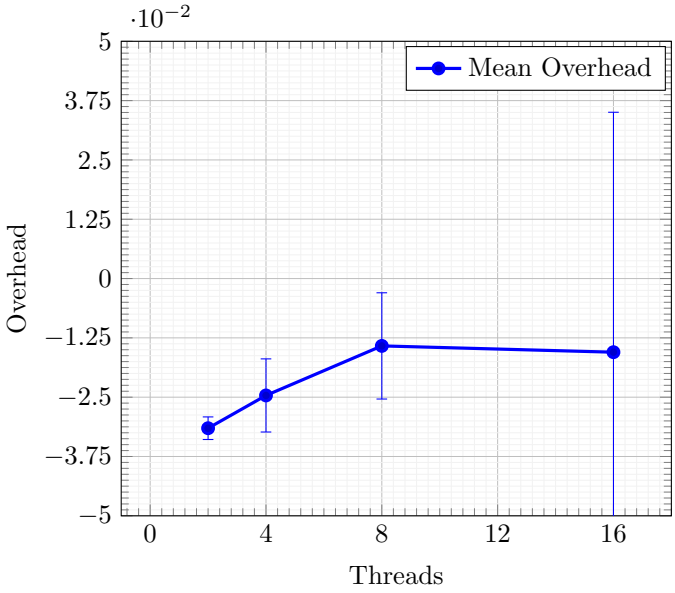

Figure 10.21: Mean overhead for shared-memory Sudoku „1000“ vs Eden CP

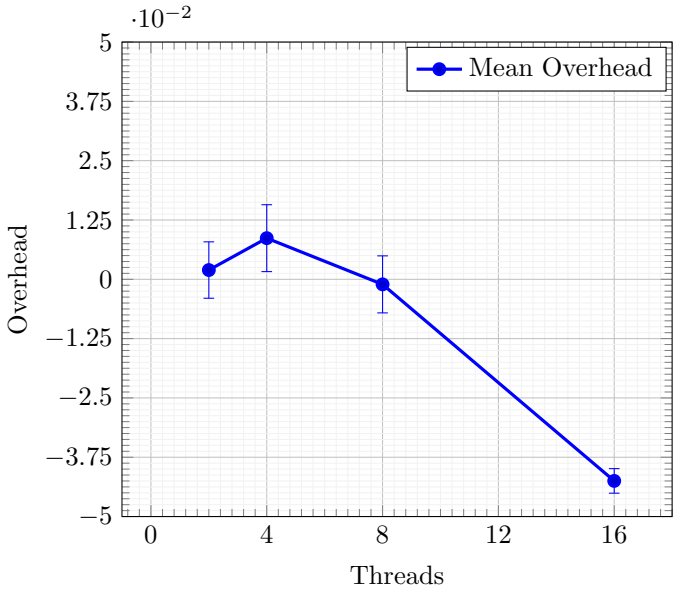

Figure 10.22: Mean overhead for shared-memory Sudoku „1000“ vs GpH

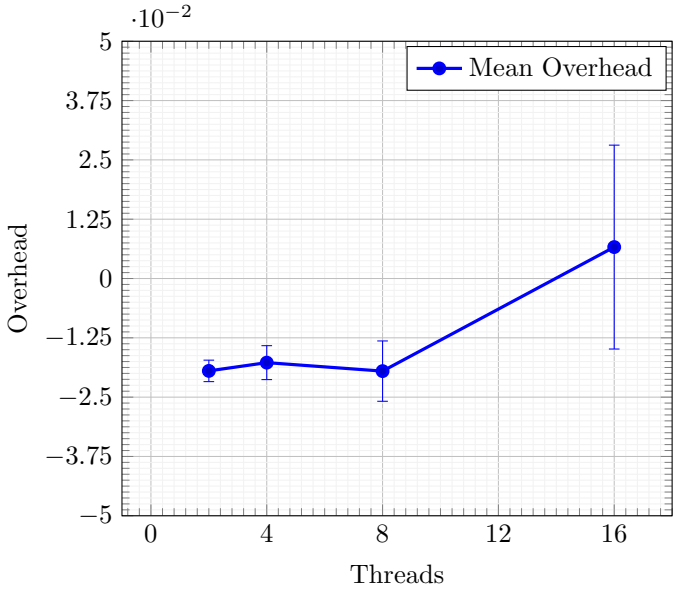

Figure 10.23: Mean overhead for shared-memory Sudoku „1000“ vs *Par* monad

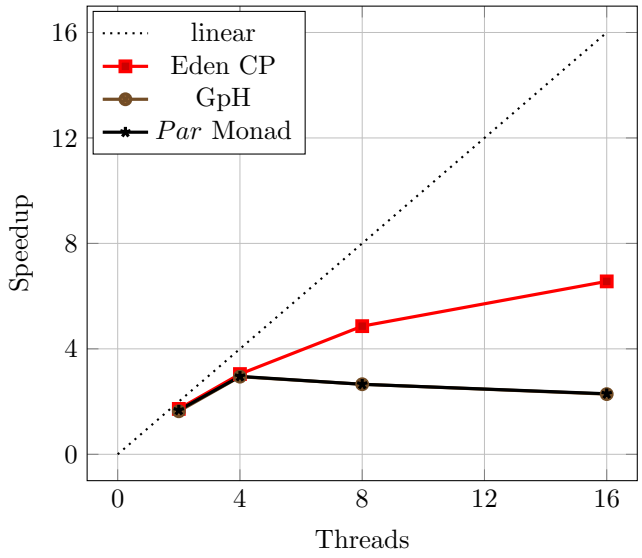

Figure 10.24: Parallel speedup of shared-memory Gentleman „512“

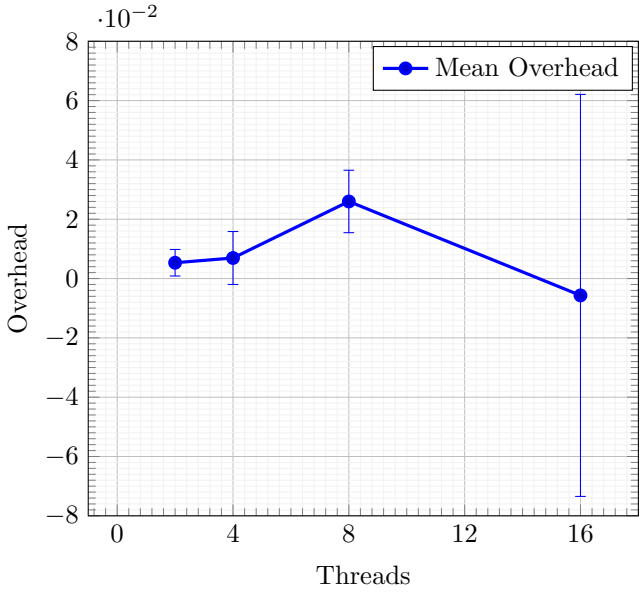

Figure 10.25: Mean overhead for shared-memory speedup of Gentleman „512“ vs Eden CP

10.7 Plots for the distributed memory benchmarks

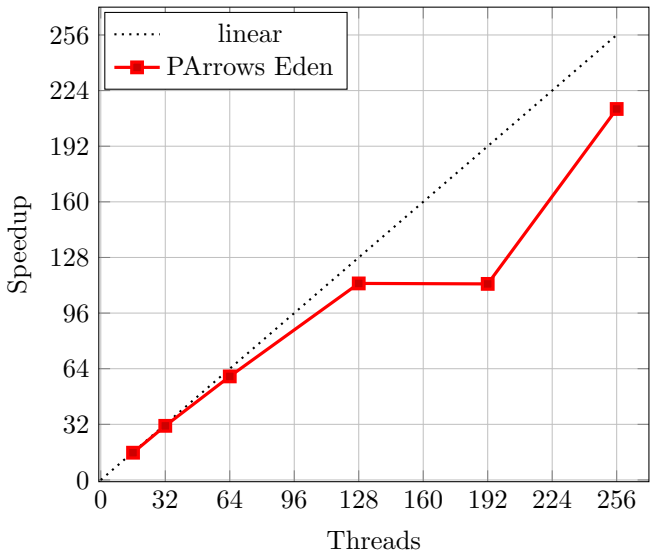

Figure 10.26: Parallel speedup of distributed-memory Rabin—Miller test „44497 256“

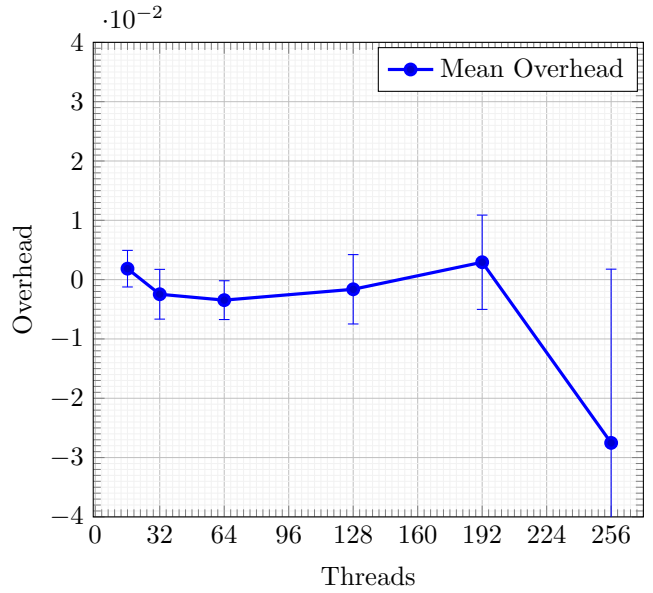

Figure 10.27: Mean overhead for distributed-memory Rabin—Miller test „44497 256“ vs Eden

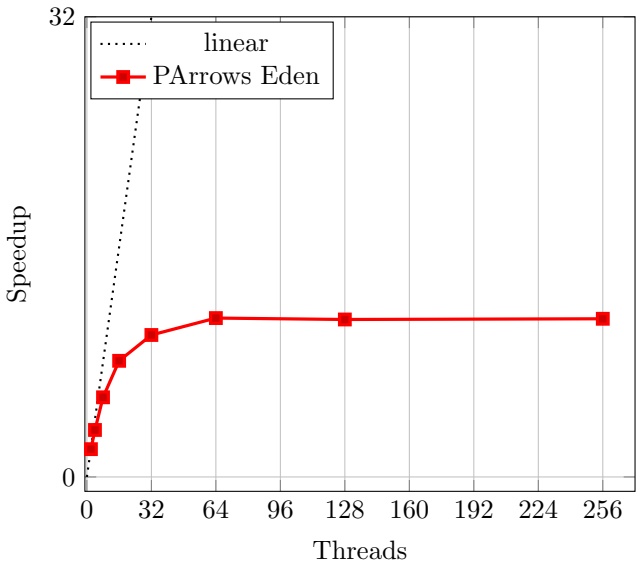

Figure 10.28: Parallel speedup of distributed-memory Jacobi sum test „3217“

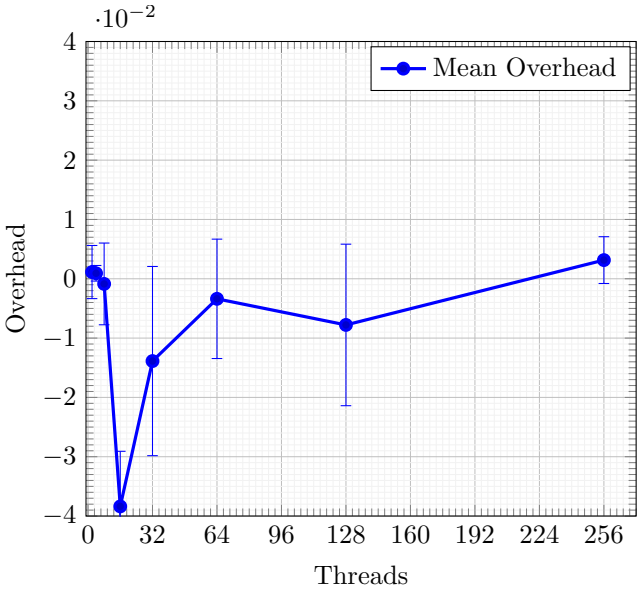

Figure 10.29: Mean overhead for distributed-memory Jacobi sum test „3217“ vs Eden

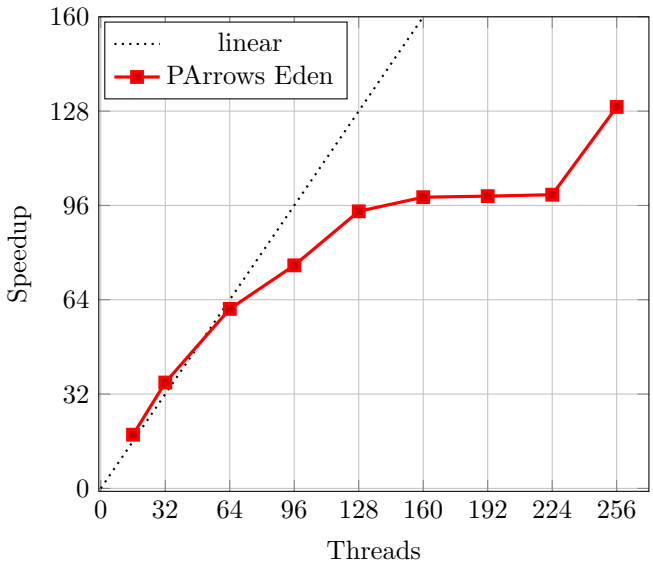

Figure 10.30: Parallel speedup of distributed-memory Gentleman „4096“

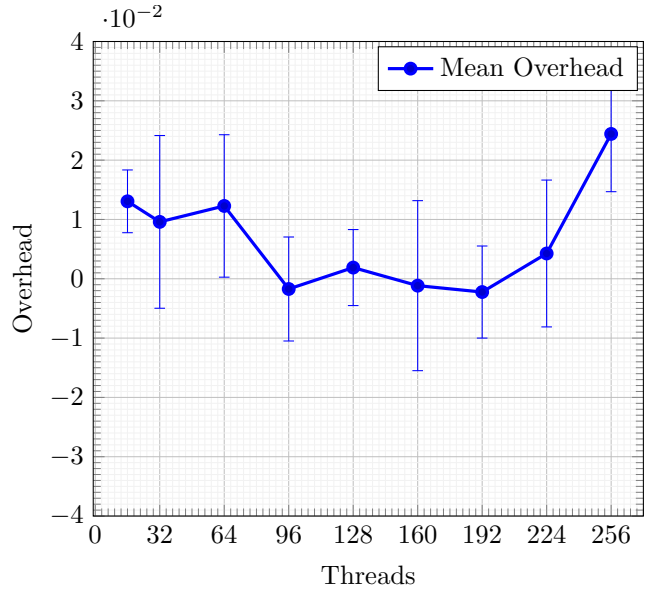

Figure 10.31: Mean overhead for distributed-memory Gentleman „4096“ vs Eden

References

Acar, U. A., Blelloch, G. E. and Blumofe, R. D.: The data locality of work stealing, in Proceedings of the 12Annual acm symposium on parallel algorithms and architectures, pp. 1–12, ACM., 2000.

Achten, P., Eekelen, M. C. van, Plasmeijer, M. and Weelden, A. van: Arrows for generic graphical editor components, Nijmegen Institute for Computing; Information Sciences, Faculty of Science, University of Nijmegen, The Netherlands. [online] Available from: `ftp://ftp.cs.ru.nl/pub/Clean/papers/2004/achp2004-ArrowGECs.pdf`, 2004.

Achten, P., Eekelen, M. van, Mol, M. de and Plasmeijer, R.: An arrow based semantics for interactive applications, in Draft proceedings of the symposium on trends in functional programming., 2007.

Alimarine, A., Smetsers, S., Weelden, A. van, Eekelen, M. van and Plasmeijer, R.: There and back again: Arrows for invertible programming, in Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, pp. 86–97, ACM., 2005.

Aljabri, M., Loidl, H.-W. and Trinder, P. W.: The design and implementation of GUMSMP: A multilevel parallel haskell implementation, in Proceedings of the 25Symposium on implementation and application of functional languages, pp. 37:37–37:48, ACM., 2014.

Aljabri, M., Loidl, H.-W. and Trinder, P.: Balancing shared and distributed heaps on NUMA architectures, in 15International symposium on trends in functional programming, revised selected papers, edited by J. Hage and J. McCarthy, pp. 1–17, Springer., 2015.

Alt, M. and Gorlatch, S.: Future-Based RMI: Optimizing compositions of remote method calls on the Grid, in Euro-par 2003, edited by H. Kosch, L. Böszörményi, and H. Hellwagner, pp. 682–693, Springer-Verlag., 2003.

Asada, K.: Arrows are strong monads, in Proceedings of the third ACM SIGPLAN workshop on mathematically structured functional programming, pp. 33–42, ACM, New York, NY, USA., 2010.

Aswad, M., Trinder, P., Al Zain, A., Michaelson, G. and Berthold, J.: Low pain vs no pain multi-core Haskells, in Trends in functional programming, pp. 49–64., 2009.

Atkey, R.: What is a categorical model of arrows?, Electronic Notes in Theoretical Computer Science, 229(5), 19–37, doi:10.1016/j.entcs.2011.02.014, 2011.

Berthold, H.-W. A. H., Jost And Loidl: PAEAN: Portable and scalable runtime support for parallel Haskell dialects, Journal of Functional Programming, 26, doi:10.1017/S0956796816000010, 2016.

Berthold, J.: Explicit and implicit parallel functional programming — concepts and implementation, PhD thesis, Philipps-Universität Marburg., 2008.

Berthold, J. and Loogen, R.: Skeletons for recursively unfolding process topologies, in Parallel computing: Current & future issues of high-end computing, parco 2005, malaga, spain, edited by G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, Central Institute for Applied Mathematics, Jülich, Germany., 2006.

Berthold, J. and Loogen, R.: Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer, in ParCo '07. parallel computing: Architectures, algorithms and applications, IOS Press., 2007.

Berthold, J., Dieterle, M., Loogen, R. and Priebe, S.: Hierarchical master-worker skeletons, in Practical aspects of declarative languages (padl'08), edited by D. S. Warren and P. Hudak, Springer-Verlag, San Francisco (CA), USA., 2008.

Berthold, J., Dieterle, M., Lobachev, O. and Loogen, R.: Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons, in Workshop on many-cores at arcs '09 – 22 international conference on architecture of computing systems 2009, edited by K.-E. Großpitsch, A. Henkersdorf, S. Uhrig, T. Ungerer, and J. Hähner, pp. 47–55, VDE-Verlag., 2009a.

Berthold, J., Dieterle, M. and Loogen, R.: Implementing parallel Google map-reduce in Eden, in Euro-par 2009 parallel processing, edited by H. Sips, D. Epema, and H.-X. Lin, pp. 990–1002, Springer Berlin Heidelberg., 2009b.

Berthold, J., Dieterle, M., Lobachev, O. and Loogen, R.: Parallel FFT with Eden skeletons, in 10International conference on parallel computing technologies, edited by V. Malyshkin, pp. 73–83, Springer., 2009c.

Bischof, H. and Gorlatch, S.: Double-scan: Introducing and implementing a new data-parallel skeleton, in Parallel processing, edited by B. Monien and R. Feldmann, pp. 640–647, Springer., 2002.

Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, J. ACM, 46(5), 720–748, doi:10.1145/324133.324234, 1999.

Botorog, G. H. and Kuchen, H.: Euro-Par'96 Parallel Processing, pp. 718–731, Springer-Verlag., 1996.

Brown, C. and Hammond, K.: Ever-decreasing circles: A skeleton for parallel orbit calculations in Eden, 2010.

Buono, D., Danelutto, M. and Lametti, S.: Map, reduce and mapreduce, the skeleton way, Procedia Computer Science, 1(1), 2095–2103, doi:https://doi.org/10.1016/j.procs.2010.04.234, 2010.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L. and Grover, V.: Accelerating Haskell array codes with multicore GPUs, in Proceedings of the 6Workshop on declarative aspects of multicore programming, pp. 3–14, ACM., 2011.

Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. L. Peyton, Keller, G. and Marlow, S.: Data Parallel Haskell: A status report, in DAMP '07, pp. 10–18, ACM Press., 2007.

Chase, D. and Lev, Y.: Dynamic circular work-stealing deque, in Proceedings of the 17Annual acm symposium on parallelism in algorithms and architectures, pp. 21–28, ACM., 2005.

Clifton-Everest, R., McDonell, T. L., Chakravarty, M. M. T. and Keller, G.: Embedding Foreign Code, in PADL '14: The 16th international symposium on practical aspects of declarative languages, Springer-Verlag., 2014.

Cole, M. I.: Algorithmic skeletons: Structured management of parallel computation, in Research monographs in parallel and distributed computing, Pitman., 1989.

Czaplicki, E. and Chong, S.: Asynchronous functional reactive programming for guis, SIGPLAN Not., 48(6), 411–422, doi:10.1145/2499370.2462161, 2013.

Dagand, P.-É., Kostić, D. and Kuncak, V.: Opis: Reliable distributed systems in OCaml, in Proceedings of the 4International workshop on types in language design and implementation, pp. 65–78, ACM., 2009.

Danelutto, M., Meglio, R. D., Orlando, S., Pelagatti, S. and Vanneschi, M.: A methodology for the development and the support of massively parallel programs, Future Generation Computer Systems, 8(1), 205–220, doi:10.1016/0167-739X(92)90040-I, 1992.

Darlington, J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q. and While, R.: Parallel programming using skeleton functions, 146–160, 1993.

Dastgeer, U., Enmyren, J. and Kessler, C. W.: Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems, in Proceedings of the 4International workshop on multicore software engineering, pp. 25–32, ACM., 2011.

Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, Communications of the ACM, 51(1), 107–113, doi:http://doi.acm.org/10.1145/1327452.1327492, 2008.

Dean, J. and Ghemawat, S.: MapReduce: A flexible data processing tool, Communications of the ACM, 53(1), 72–77, doi:http://doi.acm.org/10.1145/1629175.1629198, 2010.

Dieterle, M., Berthold, J. and Loogen, R.: A skeleton for distributed work pools in Eden, in 10International symposium on functional and logic programming, edited by M. Blume, N. Kobayashi, and G. Vidal, pp. 337–353, Springer., 2010a.

Dieterle, M., Horstmeyer, T. and Loogen, R.: Skeleton composition using remote data, in 12International symposium on practical aspects of declarative languages, vol. 5937, edited by M. Carro and R. Peña, pp. 73–87, Springer-Verlag., 2010b.

Dieterle, M., Horstmeyer, T., Berthold, J. and Loogen, R.: Iterating skeletons, in 24International symposium on implementation and application of functional languages, revised selected papers, edited by R. Hinze, pp. 18–36, Springer., 2013.

Dieterle, M., Horstmeyer, T., Loogen, R. and Berthold, J.: Skeleton composition versus stable process systems in Eden, Journal of Functional Programming, 26, doi:10.1017/S0956796816000083, 2016.

Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S. and Nieplocha, J.: Scalable work stealing, in Proceedings of the conference on high performance computing networking, storage and analysis, pp. 53:1–53:11, ACM., 2009.

Encina, A. de la, Hidalgo-Herrero, M., Rabanal, P. and Rubio, F.: A parallel skeleton for genetic algorithms, in Advances in computational intelligence: 11International work-conference on artificial neural networks, edited by J. Cabestany, I. Rojas, and G. Joya, pp. 388–395, Springer., 2011.

Foltzer, A., Kulkarni, A., Swords, R., Sasidharan, S., Jiang, E. and Newton, R.: A metascheduler for the Par-monad: Composable scheduling for the heterogeneous cloud, SIGPLAN Not., 47(9), 235–246, doi:10.1145/2398856.2364562, 2012.

Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D. and Mohr, B.: The Scalasca performance toolset architecture, Concurrency and Computation: Practice and Experience, 22(6), 2010.

Gentleman, W. M.: Some complexity results for matrix computations on parallel processors, Journal of the ACM, 25(1), 112–115, doi:10.1145/322047.322057, 1978.

Gorlatch, S.: Programming with divide-and-conquer skeletons: A case study of FFT, Journal of Supercomputing, 12(1-2), 85–97, 1998.

Gorlatch, S. and Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT, Parallel Processing Letters, 8(4), 1998.

Hammond, K., Berthold, J. and Loogen, R.: Automatic skeletons in Template Haskell, Parallel Processing Letters, 13(03), 413–424, doi:10.1142/S0129626403001380, 2003.

Harris, M., Sengupta, S. and Owens, J. D.: Parallel prefix sum (scan) with CUDA, GPU gems, 3(39), 851–876, 2007.

Harris, T., Marlow, S., Peyton Jones, S. and Herlihy, M.: Composable memory transactions, in Proceedings of the 10ACM SIGPLAN symposium on principles and practice of parallel programming, pp. 48–60, ACM., 2005.

Hey, A. J. G.: Experiments in MIMD parallelism, Future Generation Computer Systems, 6(3), 185–196, 1990.

Hippold, J. and Rünger, G.: Task pool teams: A hybrid programming environment for irregular algorithms on SMP clusters, Concurrency and Computation: Practice and Experience, 18, 1575–1594, 2006.

Horstmeyer, T. and Loogen, R.: Graph-based communication in Eden, Higher-Order and Symbolic Computation, 26(1), 3–28, doi:10.1007/s10990-014-9101-y, 2013.

Huang, L., Hudak, P. and Peterson, J.: HPorter: Using arrows to compose parallel processes, in Practical aspects of declarative languages: 9th international symposium, padl 2007, nice, france, january 14-15, 2007. proceedings, edited by M. Hanus, pp. 275–289, Springer Berlin Heidelberg, Berlin, Heidelberg., 2007.

Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, robots, and functional reactive programming, in 4International school on advanced functional programming, edited by J. Jeuring and S. L. Peyton Jones, pp. 159–187, Springer., 2003.

Hughes, J.: Research topics in functional programming, edited by D. A. Turner, pp. 17–42, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. [online] Available from: `http://dl.acm.org/citation.cfm?id=119830.119832`, 1990.

Hughes, J.: Generalising monads to arrows, Science of Computer Programming, 37(1–3), 67–111, doi:10.1016/S0167-6423(99)00023-4, 2000.

Hughes, J.: Programming with arrows, in 5International school on advanced functional programming, edited by V. Vene and T. Uustalu, pp. 73–129, Springer., 2005.

Jacobs, C. A. H., Bart And Heunen: Categorical semantics for arrows, Journal of Functional Programming, 19(3-4), 403–438, doi:10.1017/S0956796809007308, 2009.

Janjic, V., Brown, C. M., Neunhoeffer, M., Hammond, K., Linton, S. A. and Loidl, H.-W.: Space exploration using parallel orbits: A study in parallel symbolic computing, Parallel Computing, 2013.

Karasawa, Y. and Iwasaki, H.: A parallel skeleton library for multi-core clusters, in International conference on parallel processing 2009, pp. 84–91., 2009.

Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S. and Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell, SIGPLAN Not., 45(9), 261–272, doi:10.1145/1932681.1863582, 2010.

Kuchen, H.: A skeleton library, in Parallel processing, edited by B. Monien and R. Feldmann, pp. 620–629, Springer., 2002.

Kuper, L., Todd, A., Tobin-Hochstadt, S. and Newton, R. R.: Taming the parallel effect zoo: Extensible deterministic parallelism with LVish, SIGPLAN Not., 49(6), 2–14, doi:10.1145/2666356.2594312, 2014.

Lämmel, R.: Google's mapreduce programming model — revisited, Science of Computer Programming, 70(1), 1–30, doi:10.1016/j.scico.2007.07.001, 2008.

Lengauer, C., Gorlatch, S. and Herrmann, C.: The static parallelization of loops and recursions, The Journal of Supercomputing, 11(4), 333–353, doi:10.1023/A:1007904422322, 1997.

Li, P. and Zdancewic, S.: Encoding information flow in Haskell, in 19IEEE computer security foundations workshop, pp. 12–16., 2006.

Li, P. and Zdancewic, S.: Arrows for secure information flow, Theoretical Computer Science, 411(19), 1974–1994, doi:10.1016/j.tcs.2010.01.025, 2010.

Lindley, S., Wadler, P. and Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous, Electronic Notes in Theoretical Computer Science, 229(5), 97–117, doi:10.1016/j.entcs.2011.02.018, 2011.

Linton, S., Hammond, K., Konovalov, A., Al Zain, A. D., Trinder, P., Horn, P. and Roozemond, D.: Easy composition of symbolic computation software: A new lingua franca for symbolic computation, in Proceedings of the 2010 international symposium on symbolic and algebraic computation, pp. 339–346, ACM Press., 2010.

Liu, H., Cheng, E. and Hudak, P.: Causal commutative arrows and their optimization, SIGPLAN Not., 44(9), 35–46, doi:10.1145/1631687.1596559, 2009.

Lobachev, O.: Implementation and evaluation of algorithmic skeletons: Parallelisation of computer algebra algorithms, PhD thesis, Philipps-Universität Marburg., 2011.

Lobachev, O.: Parallel computation skeletons with premature termination property, in 11International symposium on functional and logic programming, edited by T. Schrijvers and P. Thiemann, pp. 197–212, Springer., 2012.

Loogen, R.: Eden – parallel functional programming with Haskell, in Central european functional programming school: 4th summer school, cefp 2011, budapest, hungary, june 14-24, 2011, revised selected papers, edited by V. Zsók, Z. Horváth, and R. Plasmeijer, pp. 142–206, Springer., 2012.

Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S. and Rubio, F.: Parallelism Abstractions in Eden, in Patterns and Skeletons for Parallel and Distributed Computing, edited by F. A. Rabhi and S. Gorlatch, pp. 71–88, Springer-Verlag., 2003.

Loogen, R., Ortega-Mallén, Y. and Peña-Marí, R.: Parallel Functional Programming in Eden, Journal of Functional Programming, 15(3), 431–475, 2005.

Maier, P., Stewart, R. and Trinder, P.: The HdpH DSLs for scalable reliable computation, SIGPLAN Not., 49(12), 65–76, doi:10.1145/2775050.2633363, 2014.

Mainland, G. and Morrisett, G.: Nikola: Embedding compiled GPU functions in Haskell, SIGPLAN Not., 45(11), 67–78, doi:10.1145/2088456.1863533, 2010.

Marlow, S.: Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming, "O'Reilly Media, Inc.", 2013.

Marlow, S., Peyton Jones, S. and Singh, S.: Runtime support for multicore Haskell, SIGPLAN Not., 44(9), 65–78, 2009.

Marlow, S., Newton, R. and Peyton Jones, S.: A monad for deterministic parallelism, SIGPLAN Not., 46(12), 71–82, doi:10.1145/2096148.2034685, 2011.

McDonell, T. L., Chakravarty, M. M. T., Grover, V. and Newton, R. R.: Type-safe runtime code generation: Accelerate to LLVM, SIGPLAN Not., 50(12), 201–212, doi:10.1145/2887747.2804313, 2015.

Michael, M. M., Vechev, M. T. and Saraswat, V. A.: Idempotent work stealing, SIGPLAN Not., 44(4), 45–54, doi:10.1145/1594835.1504186, 2009.

Michaelson, G.: Learn you a haskell for great good! A beginner's guide, by miran lipovaca, no starch press, april 2011, isbn-10: 1593272839; isbn-13: 978-1593272838, 376 pp., 23, 351–352, 2013.

Nieuwpoort, R. V. van, Kielmann, T. and Bal, H. E.: Efficient load balancing for wide-area divide-and-conquer applications, SIGPLAN Not., 36(7), 34–43, doi:10.1145/568014.379563, 2001.

Nilsson, H., Courtney, A. and Peterson, J.: Functional reactive programming, continued, in Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, pp. 51–64, ACM, New York, NY, USA., 2002.

Olivier, S. and Prins, J.: Scalable dynamic load balancing using UPC, in 37International conference on parallel processing, pp. 123–131., 2008.

Paterson, R.: A new notation for arrows, SIGPLAN Not., 36(10), 229–240, doi:10.1145/507546.507664, 2001.

Peña, R. and Rubio, F.: Parallel Functional Programming at Two Levels of Abstraction, in PPDP'01 — intl. conf. on principles and practice of declarative programming, pp. 187–198, Firenze, Italy, September 5–7., 2001.

Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T. and Valero, M.: The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment, in Proceedings of the 5Conference on computing frontiers, pp. 67–78, ACM, Ischia, Italy., 2008.

Poldner, M. and Kuchen, H.: Scalable farms, in PARCO, vol. 33, edited by G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, pp. 795–802, Central Institute for Applied Mathematics, Jülich, Germany., 2005.

Priebe, S.: Dynamic task generation and transformation within a nestable workpool skeleton, in Euro-par., 2006.

Rabhi, F. A. and Gorlatch, S., Eds.: Patterns and Skeletons for Parallel and Distributed Computing, Springer-Verlag., 2003.

Rudolph, L., Slivkin-Allalouf, M. and Upfal, E.: A simple load balancing scheme for task allocation in parallel machines, in Proceedings of the 3Annual acm symposium on parallel algorithms and architectures, pp. 237–245, ACM., 1991.

Russo, A., Claessen, K. and Hughes, J.: A library for light-weight information-flow security in Haskell, in Proceedings of the 1ACM SIGPLAN symposium on Haskell, pp. 13–24, ACM., 2008.

Stewart, P. A. T., Robert And Maier: Transparent fault tolerance for scalable functional computation, Journal of Functional Programming, 26, doi:10.1017/S095679681600006X, 2016.

Trinder, P., Hammond, K., Loidl, H.-W. and Peyton Jones, S.: Algorithm + Strategy = Parallelism, J. Funct. Program., 8(1), 23–60 [online] Available from: `http://www.macs.hw.ac.uk//~dsg/gph/papers/ps/strategies.ps.gz`, 1998.

Trinder, P. W., Hammond, K., Mattson Jr., J. S., Partridge, A. S. and Peyton Jones, S. L.: GUM: a Portable Parallel Implementation of Haskell, in PLDI'96, ACM Press., 1996.

Vizzotto, T. A. S., Juliana And Altenkirch: Structuring quantum effects: Superoperators as arrows, Mathematical Structures in Computer Science, 16(3), 453–468, doi:10.1017/S0960129506005287, 2006.

Wheeler, K. B. and Thain, D.: Visualizing massively multithreaded applications with ThreadScope, Concurrency and Computation: Practice and Experience, 22(1), 45–67, 2009.

List of Figures

List of Tables