Under consideration for publication in J. Functional Programming

Arrows for Parallel Computation

MARTIN BRAUN
University Bayreuth, 95440 Bayreuth, Germany
OLEG LOBACHEV
University Bayreuth, 95440 Bayreuth, Germany
and PHIL TRINDER
Glasgow University, Glasgow, G12 8QQ, Scotland

Abstract

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 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

Parallel 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. Section 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). Section 3.1 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 backends at will. The induced penalty was in the single-digit percent range, with means over the varying cores configuration typically under 2% overhead in our measurements (Section 7). Further backends based on HdpH or a Frege implementation (on the Java Virtual Machine) are viable, too.

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 our specific approach is that we do not have to introduce any new types, facilitating composability (Section 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 we host this functionality in the *ArrowParallel* type class, which abstracts all parallel implementation logic. The backends 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 a GpH backend 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. (Section 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 (*RD*, Dieterle *et al.*, 2010a). Direct communication is useful in a distributed memory setting because it allows for inter-node communication without blocking the master-node. (Section 5)
- We demonstrate the expressiveness of PArrows by using them to define common algorithmic skeletons ((Section 6), and by using these skeletons to implement four benchmarks (Section 7).
- We practically demonstrate that Arrow parallelism has a low performance overhead compared with existing approaches, e. g. with mean of mean overhead of less than 3.5% and less than 0.8% for all benchmarks with GpH and Eden, respectively. Compared to *Par* Monad we noticed no overhead: the mean of mean overheads was in our favour in all benchmarks (Section 7).

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

2 Related Work

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 & Morrisett, 2010; Svensson, 2011), 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, 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, 2015). Research on Eden includes low-level implementation (Berthold, 2008; Berthold *et al.*, 2016), skeleton composition (Dieterle *et al.*, 2016), communication (Dieterle *et al.*, 2010a), and generation of process networks (Horstmeyer & Loogen, 2013). The definitions of new Eden skeletons is a specific focus (Hammond *et al.*, 2003; Berthold & Loogen, 2006; Berthold *et al.*, 2009b,c; Dieterle *et al.*, 2010b; de la 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 the following. HdpH (Maier *et al.*, 2014; Stewart *et al.*, 2016) is an extension of *Par* Monad to heterogeneous clusters. LVish (Kuper *et al.*, 2014) is a communication-centred extension of *Par* Monad.

Algorithmic skeletons. Algorithmic skeletons were introduced by Cole (1989). Early publications on this topic include (Danelutto *et al.*, 1992; Darlington *et al.*, 1993; Botorog & Kuchen, 1996; Lengauer *et al.*, 1997; Gorlatch, 1998). Rabhi & Gorlatch (2003) consolidated early reports on high-level programming approaches. Typical 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 & Rubio, 2001; Poldner & 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 & Rünger, 2006; Berthold *et al.*, 2008; Marlow *et al.*, 2009). For special tasks *workpool* skeletons can be extended with dynamic task creation (Priebe, 2006; Dinan *et al.*, 2009; Brown & Hammond, 2010). Efficient load-balancing schemes for *workpool*s are subject of research (Blumofe & Leiserson, 1999; Acar *et al.*, 2000; van Nieuwpoort *et al.*, 2001; Chase & Lev, 2005; Olivier & Prins, 2008; Michael *et al.*, 2009). The *fold* (or *reduce*) skeleton was implemented in various skeleton libraries (Kuchen, 2002; Karasawa & Iwasaki, 2009; Buono *et al.*, 2010; Dastgeer *et al.*, 2011), as also its inverse, *scan* (Bischof & Gorlatch, 2002; Harris *et al.*, 2007). Google *map–reduce* (Dean & Ghemawat, 2008, 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 & Loogen, 2006), special-purpose skeletons for computer algebra (Berthold *et al.*, 2009c; Lobachev, 2011, 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.

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 *et al.* (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 & Zdancewic, 2006, 2010; Russo *et al.*, 2008), invertible programming (Alimarine *et al.*, 2005), and quantum computer simulation (Vizzotto *et al.*, 2006). But probably most prominent application of Arrows is Arrow-based functional reactive programming, AFRP (Nilsson *et al.*, 2002; Hudak *et al.*, 2003; Czaplicki & 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 centred 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, 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.

3 Background

As a introduction, we here give a short overview over Arrows and GpH, the *Par* Monad, and Eden, the three parallel Haskells which we base our DSL on.

3.1 Arrows

Arrows were introduced by Hughes (2000) as a general interface for computation and a less restrictive generalisation of Monads. Hughes motivates the broader 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)[3].

An Arrow *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. 1. To lift an ordinary function to an Arrow, *arr* is used, analogous to the monadic *return*. Similarly, the composition operator $>\!\!>\!\!>$ is analogous to the monadic composition $\gg\!=$ and combines two Arrows *arr a b* and *arr b c* by 'wiring' the outputs of the first to the inputs to the second to get a new Arrow *arr a c*. Lastly, the *first* operator takes the input Arrow *arr a b* and converts it into an Arrow on pairs *arr* $(a,c)\ (b,c)$ that leaves the second argument untouched. It allows us to to save input across Arrows. Figure 2 shows a graphical representation of these basic Arrow combinators. The most prominent instances of this interface are regular functions $(\to)$ and the Kleisli type (Fig. 1), which wraps monadic functions, e.g. $a \to m\ b$.

Hughes also defined some syntactic sugar (Fig. 3): *second*, $***$ and &&&. *second* is the mirrored version of *first* (Appendix A). The $***$ function combines *first* and *second* to handle two inputs in one arrow, and is defined as follows:

$$(***) :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ c\ d \to arr\ (a,c)\ (b,d)$$
$$f *** g = first\ f >\!\!>\!\!> second\ g$$

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

$$(\&\&\&) :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ a\ c \to arr\ a\ (b,c)$$
$$f \mathbin{\&\&\&} g = arr\ (\lambda a \to (a,a)) >\!\!>\!\!> (f *** g)$$

[1] GitHub project page at `https://github.com/Frege/frege`

[2] Eta project page at `http://eta-lang.org`

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

```
class Arrow arr where
  arr :: (a → b) → arr a b
  (>>>) :: arr a b → arr b c → arr a c
  first :: arr a b → arr (a,c) (b,c)
instance Arrow (→) where
  arr f = f
  f >>> g = g ∘ f
  first f = λ(a,c) → (f a,c)
data Kleisli m a b = Kleisli {run :: a → m b}
instance Monad m ⇒ Arrow (Kleisli m) where
  arr f = Kleisli (return ∘ f)
  f >>> g = Kleisli (λa → f a >>= g)
  first f = Kleisli (λ(a,c) → f a >>= λb → return (b,c))
```

Figure 1: The *Arrow* type class and its two most typical instances.

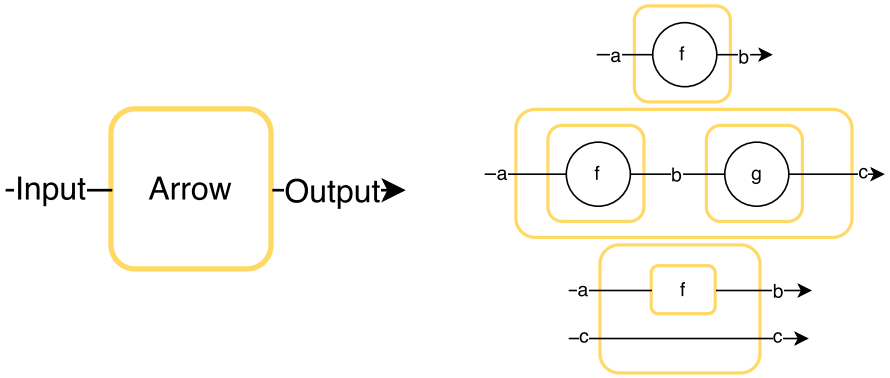

Figure 2: Schematic depiction of an Arrow (left) and its basic combinators *arr*, >>> and *first* (right).

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

```
add :: Arrow arr ⇒ arr a Int → arr a Int → arr a Int
add f g = (f &&& g) >>> arr (λ(u,v) → u + v)
```

As we can rewrite the monadic bind operation (>>=) with only the Kleisli type into *m a* → *Kleisli m a b* → *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 paper we will show that parallel computations can be expressed with this more general interface of Arrows without requiring Monads. 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.

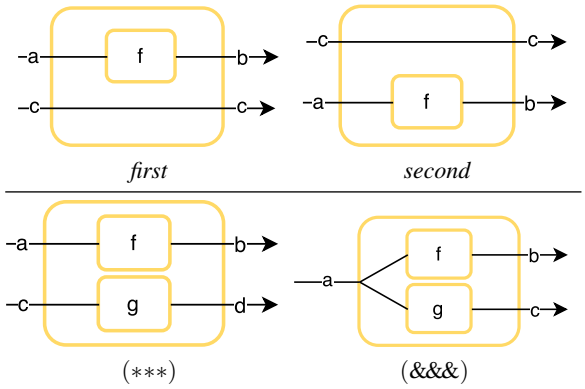

Figure 3: Visual depiction of syntactic sugar for Arrows.

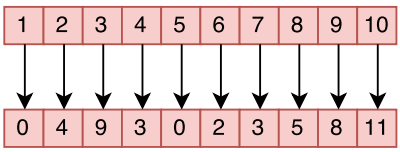

Figure 4: Schematic illustration of *parEvalN*. A list of inputs is transformed by different functions in parallel.

While we could have based our DSL on Profunctors as well, we chose Arrows for this paper 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 B.

3.2 Short introduction to parallel Haskells

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 Figure 4 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, 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 *et al.*, 2016) and LVish (a *Par* extension with focus on communication, Kuper *et al.*, 2014).

As the goal of this paper 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

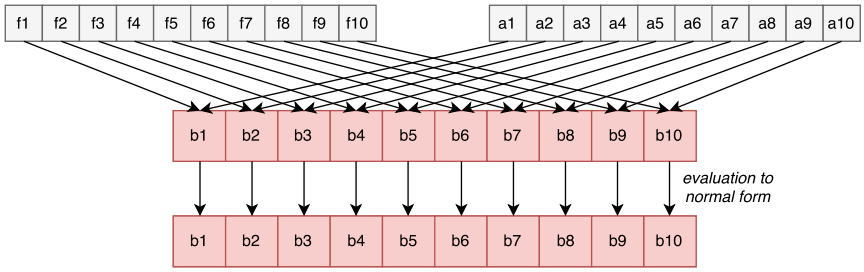

Figure 5: *parEvalN* (GpH).

backends behind a common interface. For this paper 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 paper, 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.

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.[4] 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. 5 shows a visual representation of this code.

[4] 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.

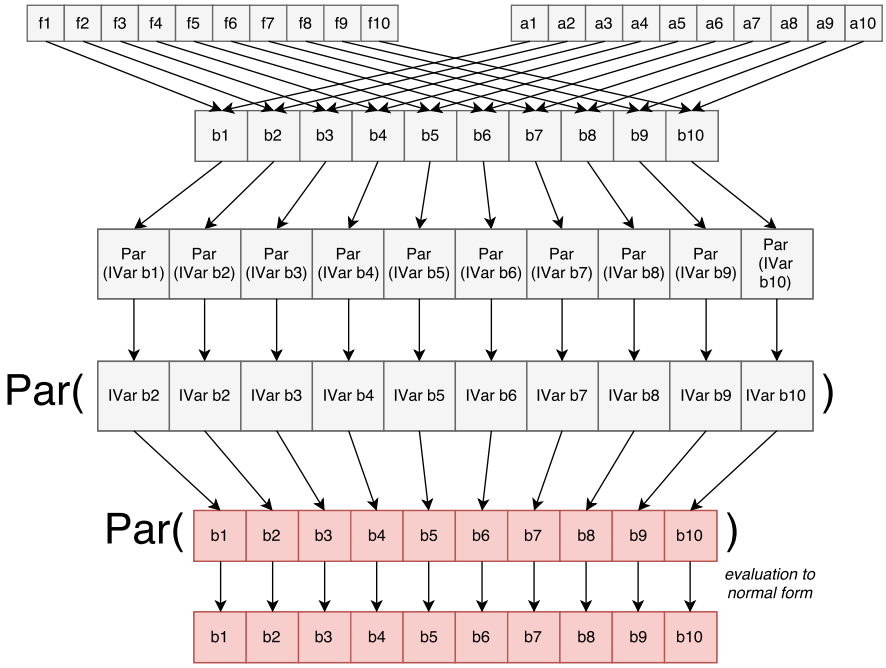

Figure 6: *parEvalN* (*Par* Monad).

3.2.2 *Par Monad*

The *Par* Monad[5] introduced by Marlow *et al.* (2011), is a Monad designed for composition of parallel programs. Let:

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

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 $\mathit{spawn} :: \mathit{NFData}\ a \Rightarrow \mathit{Par}\ a \to \mathit{Par}\ (\mathit{IVar}\ a)$ to transform them to a list of not yet evaluated forked away computations $[\mathit{Par}\ (\mathit{IVar}\ b)]$, which we convert to $\mathit{Par}\ [\mathit{IVar}\ b]$ with *sequenceA*. We wait for the computations to finish by mapping over the $\mathit{IVar}\ b$ values inside the *Par* Monad with *get*. This results in $\mathit{Par}\ [b]$. We execute this process with *runPar* to finally get $[b]$. While we used *spawn* in the definition above, a head-strict variant can easily be defined by replacing *spawn* with $\mathit{spawn\_} :: \mathit{Par}\ a \to \mathit{Par}\ (\mathit{IVar}\ a)$. Fig. 6 shows a graphical representation.

[5] 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/`.

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.[6] 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 exist (Geimer *et al.*, 2010), the parallel Haskell community mainly utilises the tools Threadscope (Wheeler & Thain, 2009) and Eden TraceViewer[7] (Berthold & 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. 13), 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.

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.

[6] 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/`.

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

4.1 The ArrowParallel type class

A parallel computation (on functions) can be seen as execution of some functions $a \to b$ in parallel, as our *parEvalN* prototype shows (Section 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 *evalN* from Appendix A, but does parallel instead of serial evaluation.

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

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*:

```
class Arrow arr ⇒ ArrowParallel arr a b where
  parEvalN :: [arr a b] → arr [a] [b]
```

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*.

```
class Arrow arr ⇒ ArrowParallel arr a b conf where
  parEvalN :: conf → [arr a b] → arr [a] [b]
```

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. 7, but a bit different compared to the variant from Section 3.2.1. We use $evalN :: [arr\ a\ b] \to arr\ [a]\ [b]$ (definition in Appendix A, 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.

4.2.2 Par Monad

As for GpH we can easily lift the definition of *parEvalN* for the *Par* Monad to Arrows in Fig. 8. To start off, we define the *Strategy a* and *Conf a* type so we can have a configurable instance of ArrowParallel:

```
data Conf a = Conf (Strategy a)
instance (NFData b,ArrowChoice arr) ⇒
  ArrowParallel arr a b (Conf b) where
    parEvalN (Conf strat) fs =
      evalN fs >>>
      arr (withStrategy (parList strat))
```

Figure 7: GpH *ArrowParallel* instance.

```
type Strategy a = a → Par (IVar a)
data Conf a = Conf (Strategy a)
```

Now we can once again define our *ArrowParallel* instance as follows: First, we convert our Arrows $[arr\ a\ b]$ with $evalN\ (map\ (>>>arr\ strat)\ fs)$ 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 $arr\ (>>=mapM\ get)$ - 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 (NFData b,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 8: *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
  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. 9, 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 9: 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 the GpH backend:

```
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 with more details 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.

4.3.1 Lazy parEvalN

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. 10, 11) 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

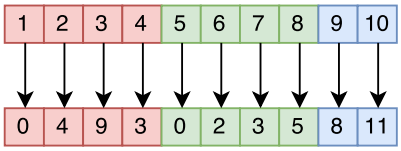

Figure 10: *parEvalNLazy* depiction.

```
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 11: Definition of *parEvalNLazy*.

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. 12, C 12) 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*.

5 Futures

Consider an outline parallel Arrow combinator:

```
someCombinator :: (ArrowChoice arr,
  ArrowParallel arr a b (),
```

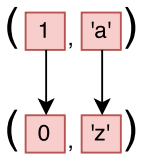

Figure 12: *parEval2* depiction.

```
    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 such a combinator 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 the data distributed across the nodes needs to be transposed. A concrete example is 2D FFT computation (Gorlatch & 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 13. This can become a serious bottleneck for a larger amount of data and number of processes (as e.g. Berthold *et al.*, 2009c, showcases).

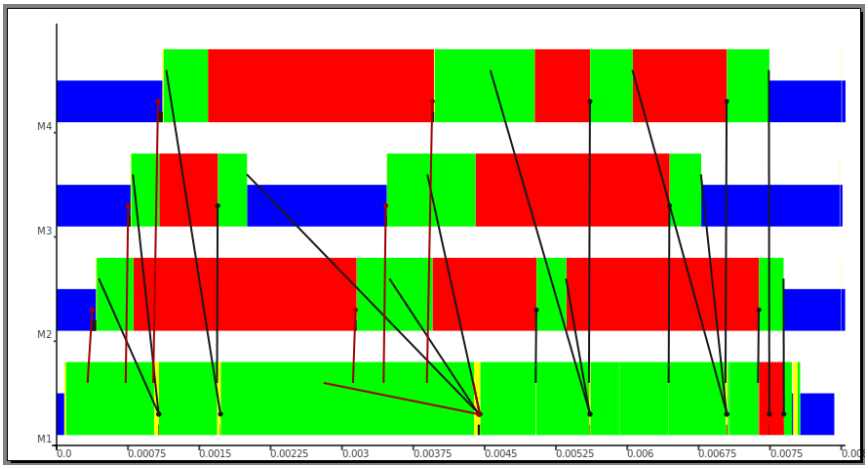

Figure 13: Communication between 4 Eden processes without Futures. All communication goes through the master node. Each bar represents one process. Black lines represent communication. Colours: blue ≙ idle, green ≙ running, red ≙ blocked, yellow ≙ suspended.

This is only a problem in distributed memory (in the scope of this paper) and we should allow nodes to communicate directly with each other. Eden already provides 'remote data' that enable this (Alt & Gorlatch, 2003; Dieterle *et al.*, 2010a). But as we want code with our DSL to be implementation agnostic, we have to wrap this concept. We do this with the *Future* type class (Fig. 14). We require a *conf* parameter 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. Since *RD* is only a type synonym for a communication type that Eden uses

```
class Future fut a conf | a conf → fut where
  put :: (Arrow arr) ⇒ conf → arr a (fut a)
  get :: (Arrow arr) ⇒ conf → arr (fut a) a
```

Figure 14: The *Future* type class.

internally, we have to use some wrapper classes to fit that definition, though, as Fig. C 1 shows. Technical details are in Appendix, in Section C.

For GpH and *Par* Monad, we can simply use *BasicFuture*s (Fig. C 2), 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. The implementation can also be found in Section C.

In our communication example we can use this *Future* concept for direct communication between nodes as shown in Fig. 15. In a distributed environment, this gives us a communic-

```
someCombinator :: (ArrowChoice arr, NFData b, NFData c,
  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)
```

Figure 15: The outline parallel combinator.

ation scheme with messages going through the master node only if it is needed – similar to what is shown in the trace visualisation in Fig. 16. One especially elegant aspect of the definition in Fig. 14 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 Fig. 15. With the PArrows DSL we can also define default instances *Future fut a* () for each backend similar to how *ArrowParallel arr a b* () was defined in Section 4. Details can be found in Section C.

6 Skeletons

Now we have developed Parallel Arrows far enough to define some useful algorithmic skeletons that abstract typical parallel computations. While there are many possible skeletons

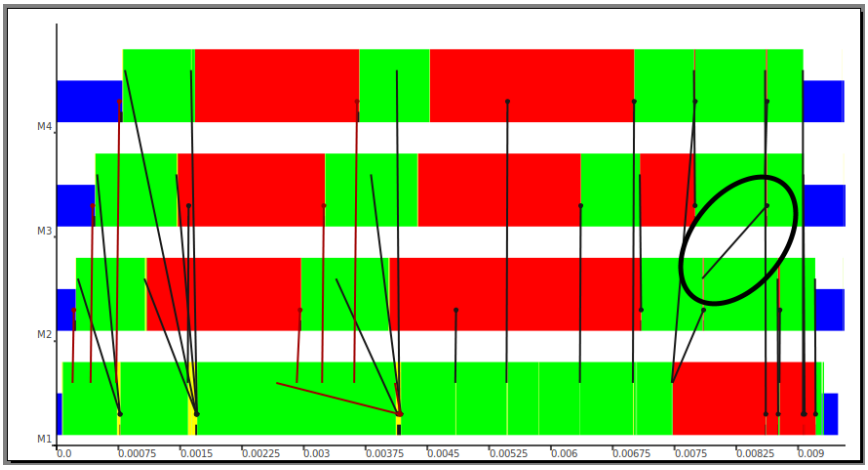

Figure 16: Communication between 4 Eden processes with Futures. Other than in Fig. 13, processes communicate directly (one example message is marked in the Figure) instead of always going through the master node (bottom bar).

to implement, we demonstrate the expressive power of PArrows here using four *map*-based and three toplogical skeletons.

6.1 map-based Skeletons

The essential differences between the mapping skeletons presented here are in terms of order of evaluation and work distribution but still provide the same semantics as a sequential *map*.

Parallel *map* and laziness. The *parMap* skeleton (Figs. C 3, C 4) 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* $[a]$ $[b]$. 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. C 5, C 6) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*. Implementing these skeletons is straightforward as in Appendix C in Figs.C 4 and C 6.

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 paper). This can be quite wasteful and a statically load-balancing *farm* (Figs. 17, 18) that equally distributes the workload over *numCores* workers seems useful. The definitions of the helper functions *unshuffle*, *takeEach*, *shuffle* (Fig. C 7) originate from an Eden skeleton[8].

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

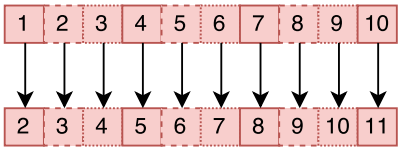

Figure 17: *farm* depiction.

```
farm :: (ArrowParallel arr a b conf,
  ArrowParallel arr [a] [b] conf, ArrowChoice arr) ⇒
  conf → NumCores → arr a b → arr [a] [b]
farm conf numCores f =
  unshuffle numCores >>>
  parEvalN conf (repeat (mapArr f)) >>>
  shuffle
```

Figure 18: *farm* definition.

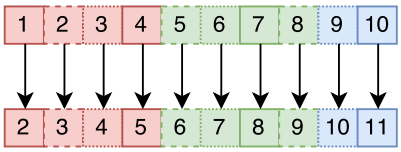

Figure 19: *farmChunk* depiction.

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. 19, C 10) which uses *parEvalNLazy* instead of *parEvalN*. It is basically the same definition as for *farm*, but with *parEvalNLazy* instead of *parEvalN*.

6.2 Topological Skeletons

Even though many algorithms can be expressed by parallel maps, some problems require more sophisticated skeletons. The Eden library leverages this problem and already comes with more predefined skeletons[9], among them a *pipe*, a *ring*, and a *torus* implementations (Loogen *et al.*, 2003). These seem like reasonable candidates to be ported to our Arrow-based parallel Haskell. We aim to showcase that we can express more sophisticated skeletons with parallel Arrows as well.

If we used the original definition of *parEvalN*, however, these skeletons would produce an infinite loop with the GpH and Par Monad backends 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 of a distributed memory environment. However our

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

workaround enables users of the DSL to test their code within a shared memory setting even though other skeletons would be better suited to be run with them. 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 backends for 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
```

6.2.1 Parallel pipe

The parallel *pipe* skeleton is semantically equivalent to folding over a list $[arr\ a\ a]$ of Arrows with $>>>$, but does this 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 gives us the $loop :: arr\ (a,b)\ (c,b) \rightarrow arr\ a\ c$ combinator which allows us to express recursive fix-point computations in which output values are fed back as input. For example

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

which is the same as

$$loop\ (arr\ snd\ \&\&\&\ arr\ (uncurry\ (:)))$$

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. 20) by plugging in the parallel evaluation call *evalN conf fs* inside the second argument of &&& and then only picking the first element of the resulting list with *arr last*.

However, using this definition directly will make the master node a potential bottleneck in distributed environments as described in Section 5. Therefore, we introduce a more sophisticated version that internally uses Futures and obtain the final definition of *pipe* in Fig. 21.

```
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 20: Simple *pipe* skeleton. The use of *lazy* (Fig. C 8) is essential as without it programs using this definition would never halt. We need to enforce that the evaluation of the input $[a]$ terminates before passing it into *evalN*.

```
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 21: *pipe* skeleton definition with Futures.

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. 22.

Note that 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 D.

6.2.2 Ring skeleton

Eden comes with a ring skeleton[10] (Fig. 23) 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 gets send to the neighbour of each node. This data is sent over direct communication channels, the so called 'remote data'. We depict it in Appendix, Fig. C 11.

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

```
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 22: Definition of *pipe2* and $(|>>>|)$, a parallel $>>>$.

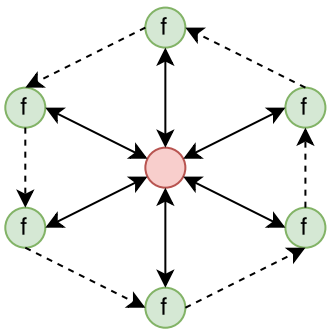

Figure 23: *ring* skeleton depiction.

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 $second\ (rightRotate >>> lazy)$ (Fig. C 8). Similarly to the *pipe* from Section 6.2.1 (Fig. 20), we have to feed the intermediary input into our *lazy* (Fig. C 8) 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 $([o],[fut\ r])$.

```
ring :: (Future fut r conf,
  ArrowLoop arr,
  ArrowLoopParallel arr (i,fut r) (o,fut r) conf,
  ArrowLoopParallel arr o o conf) ⇒
  conf → arr (i,r) (o,r) → arr [i] [o]
ring conf f =
  loop (second (rightRotate >>> lazy) >>>
    arr (uncurry zip) >>>
    loopParEvalN conf (repeat (second (get conf) >>> f >>> second (put conf))) >>>
    arr unzip) >>>
  postLoopParEvalN conf (repeat (arr id))
```

Figure 24: *ring* skeleton definition.

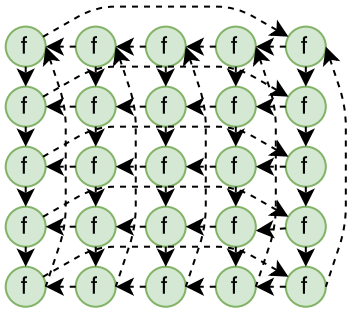

Figure 25: *torus* skeleton depiction.

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. 24). 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.

6.2.3 Torus skeleton

If we take the concept of a *ring* from Section 6.2.2 one dimension further, we obtain a *torus* skeleton (Fig. 25, 26). 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[11] from Eden—yet again with the help of the *ArrowLoop* type class.

Similar to the *ring*, we start by rotating the input (Fig. C 8), 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 *arr* $([[c]],([[fut\ a]],[[fut\ b]]))$ $([[d]],([[fut\ a]],[[fut\ b]]))$ and our rotation Arrow becomes

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

```
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 26: *torus* skeleton definition. *lazyzip3*, *uncurry3* and *threetotwo* definitions are in Fig. C 9

.

second ((*mapArr rightRotate* >>> *lazy*) *** (*arr rightRotate* >>> *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*) >>> *arr unzip3* >>> *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 (1978). An adapted version can be found in Fig. 27. If we compare the trace from a call using our Arrow definition of the torus (Fig. 28) with the Eden version (Fig. 29) we can see that the behaviour of the Arrow version and execution times are comparable. We discuss further benchmarks on larger clusters and in a more detail in the next section.

```
type Matrix = [[Int]]
prMM_torus :: Int → Int → Matrix → Matrix → Matrix
prMM_torus numCores problemSizeVal m1 m2 =
  combine $ torus () (mult torusSize) $ zipWith (zipWith (,)) (split m1) (split m2)
  where torusSize = (floor ∘ sqrt) $ fromIntegral numCores
        combine = concat ∘ (map (foldr (zipWith (++)) (repeat [])))
        split = splitMatrix (problemSizeVal `div` torusSize)
  -- 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)
    sm2′ = transpose sm2
    sms = zipWith prMMTr (sm1 : sm1s) (sm2′ : sm2s)
    result = foldl1′ matAdd sms
```

Figure 27: 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. *prMMTr* calculates AB^T and is used for the (sequential) calculation in the chunks. *splitMatrix* splits the Matrix into chunks. *matAdd* calculates $A+B$. Omitted definitions can be found in C 13.

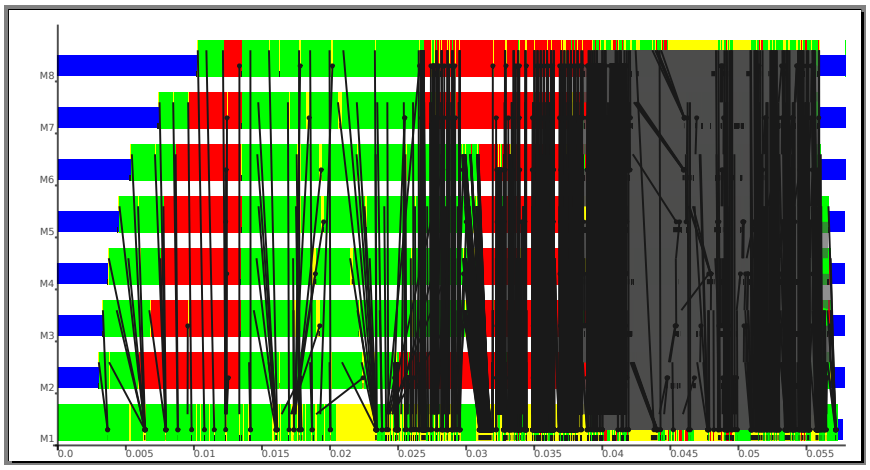

Figure 28: Matrix Multiplication with *torus* (PArrows).

7 Performance results and discussion

The preceding section has shown that PArrows are expressive. This section evaluates the performance overhead of this compositional abstraction in comparison to GpH and the *Par* Monad on shared memory architectures and Eden on a distributed memory cluster. We describe our measurement platform, the benchmark results – 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.

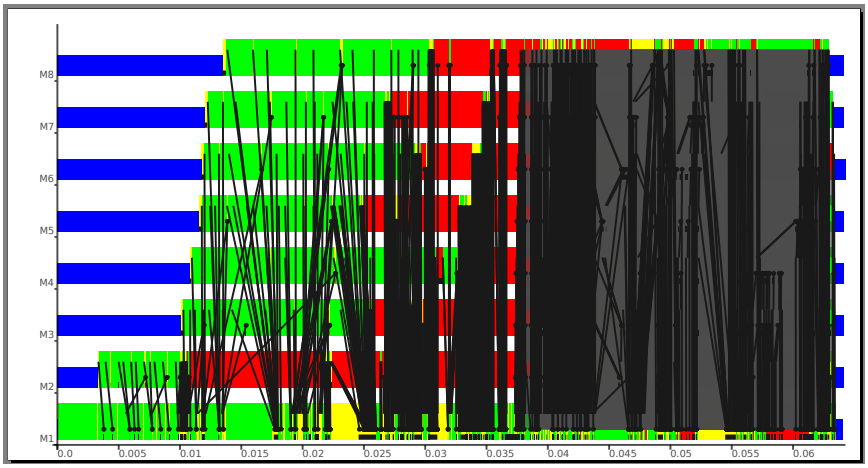

Figure 29: Matrix Multiplication with *torus* (Eden).

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[12] lts-7.1 GHC compiler which is equivalent to a standard GHC 8.0.1 with the base package in version 4.9.0.0. Stack itself was used in version 1.3.2. For GpH in its Multicore variant we used the the parallel package in version 3.2.1.0[13], while for the *Par* Monad we used monad-par in version 0.3.4.8[14]. For all Eden tests, we used its GHC-Eden compiler in version 7.8.2[15] together with OpenMPI 1.6.5[16].

[12] see `https://www.haskellstack.org/`
[13] see `https://hackage.haskell.org/package/parallel-3.2.1.0`
[14] see `https://hackage.haskell.org/package/monad-par-0.3.4.8`
[15] see `http://www.mathematik.uni-marburg.de/~eden/?content=build_eden_7_&navi=build`
[16] see `https://www.open-mpi.org/software/ompi/v1.6/`

Table 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)[19]

Furthermore, all benchmarks were done with help of the bench[17] tool in version 1.0.2 which uses criterion (>=1.1.1.0 && < 1.2)[18] internally. All runtime data (mean runtime, max stddev, etc.) was collected with this tool if not mentioned otherwise.

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 1 summarises.

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. 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, and 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 distributed heaps. In this mode, Eden still

[17] see `https://hackage.haskell.org/package/bench`
[18] see `https://hackage.haskell.org/package/criterion-1.1.1.0`
[19] actual code from: `http://community.haskell.org/~simonmar/par-tutorial.pdf` and `https://github.com/simonmar/parconc-examples`

operates in the 'nothing shared' setting, but is adapted better to multicore machines. We call this version of Eden 'Eden CP'.

7.1.4 Effect of hyper-threading

In preliminary tests, the PArrows version of Rabin–Miller test on a single node of the Glasgow cluster showed almost linear speedup on up to 16 shared-memory cores (as supplementary materials show). The speedup of 64-task PArrows/Eden at 16 real cores version was 13.65 giving a parallel efficiency of 85.3%. However, if we increased the number of requested cores to 32 – i.e. if we use hyper-threading on 16 real cores – the speedup did not increase that well. It was merely 15.99 for 32 tasks with PArrows/Eden. This was worse for other backends. As for 64 tasks, we obtained a speedup of 16.12 with PArrows/Eden at 32 hyper-threaded cores and only 13.55 with PArrows/GpH.

While this shows that hyper-threading can be of benefit in scenarios similar to the ones presented in the benchmarks, we only use real cores for the performance measurements in Section 7.2 as the purpose of this paper is to show the performance of PArrows and not to investigate parallel behaviour with hyper-threading.

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 backends and then study OpenMPI variants of the Eden-enabled backend as a representative of a distributed memory backend. We plot all speedup curves and all overhead values in the supplementary materials.

7.2.1 Defining overhead

We compare the mean overhead, i.e. the mean of relative 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[20], yield the error margin of the mean overhead.

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 benchmarks 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 test 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

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

$2^{11213}-1$ 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* backend 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 test benchmark, implemented with PArrows using Eden CP, GpH, and *Par* Monad backends, 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 backend 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 GpH backend 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* from PArrows of $-4.2\% \pm 0.26\%$ (for 16 cores) 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 distributed Eden backend showed an almost linear speedup excepting around 192 cores where an unfortunate task distribution reduces performance. As seen in Fig. 30, 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 9193 s (at 128 cores) to 1649 s (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.

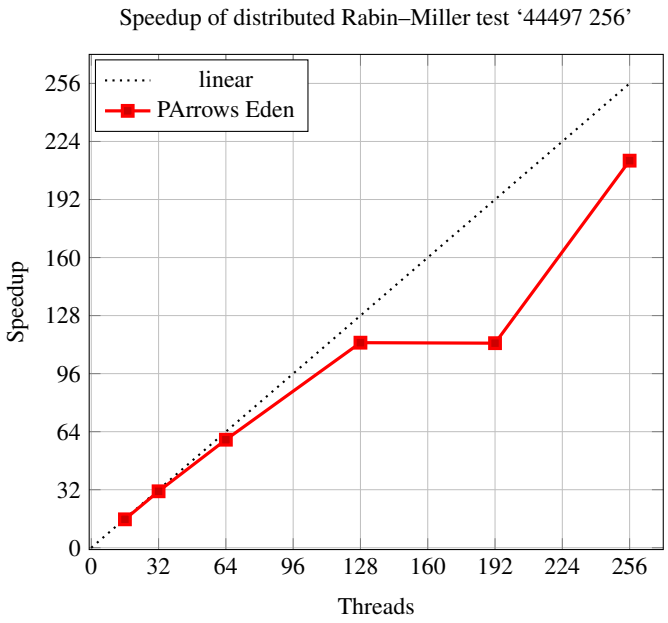

Figure 30: Speedup of the distributed Rabin–Miller test benchmark using PArrows with Eden backend.

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 ±0.8% and ±2.9% correspondingly. A sole significant (by a very low margin) overhead is 0.35% ±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% ±0.93% at 16 cores in favour of PArrows, it was the sole significant overhead value. The value for 256 cores was 0.31% ±0.39%. Mean overhead for distributed Gentleman multiplication was also low. Significant values include 1.23% ±1.20% at 64 cores and 2.4% ±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 Discussion

PArrows performed in our benchmarks with little to no overhead. Tables 2 and 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.

Table 2: Overhead in the shared memory benchmarks. Bold marks values in favour of PArrows. Overhead is unit-less. Runtime is in seconds.

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 test 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 test 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 3: Overhead in the distributed memory benchmarks. Bold marks values in favour of PArrows. Overhead is unit-less. Runtime is in seconds.

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 test 44497 256	Eden	**-0.5%**	2.9%	165.0
Jacobi sum test 3217	Eden	**-0.74%**	1.6%	635.0

PArrows is 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.

8 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 composibility. 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* (but 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, paralle 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 backends. In other words, PArrows greatly increase portability of parallel Haskell programs. 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%.

8.1 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 a more special versions of $>\!\!>\!\!>$ 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.

References

Acar, Umut A., Blelloch, Guy E., & Blumofe, Robert D. (2000). The data locality of work stealing. *Pages 1–12 of: Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '00. ACM.

Achten, Peter, van Eekelen, Marko, de Mol, Maarten, & Plasmeijer, Rinus. (2007). An arrow based semantics for interactive applications. *Draft Proceedings of the Symposium on Trends in Functional Programming*. TFP '07.

Achten, PM, van Eekelen, Marko CJD, Plasmeijer, MJ, & Weelden, A van. (2004). *Arrows for generic graphical editor components*. Tech. rept. Nijmegen Institute for Computing and Information Sciences, Faculty of Science, University of Nijmegen, The Netherlands. Technical Report NIII-R0416.

Alimarine, Artem, Smetsers, Sjaak, van Weelden, Arjen, van Eekelen, Marko, & Plasmeijer, Rinus. (2005). There and back again: Arrows for invertible programming. *Pages 86–97 of: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell '05. ACM.

Aljabri, Malak, Loidl, Hans-Wolfgang, & Trinder, Phil W. (2014). The design and implementation of GUMSMP: A multilevel parallel haskell implementation. *Pages 37:37–37:48 of: Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*. IFL '13. ACM.

Aljabri, Malak, Loidl, Hans-Wolfgang, & Trinder, Phil. (2015). *Balancing shared and distributed heaps on NUMA architectures*. Springer. Pages 1–17.

Alt, Martin, & Gorlatch, Sergei. (2003). Future-Based RMI: Optimizing compositions of remote method calls on the Grid. *Pages 682–693 of:* Kosch, Harald, Böszörményi, László, & Hellwagner, Hermann (eds), *Euro-Par 2003*. LNCS 2790. Springer-Verlag.

Asada, Kazuyuki. (2010). Arrows are strong monads. *Pages 33–42 of: Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*. MSFP '10. New York, NY, USA: ACM.

Aswad, Mustafa, Trinder, Phil, Al Zain, Abdallah, Michaelson, Greg, & Berthold, Jost. (2009). Low pain vs no pain multi-core Haskells. *Pages 49–64 of: Trends in Functional Programming*.

Atkey, Robert. (2011). What is a categorical model of arrows? *Electronic notes in theoretical computer science*, **229**(5), 19–37.

Berthold, Jost. (2008). *Explicit and implicit parallel functional programming — concepts and implementation*. Ph.D. thesis, Philipps-Universität Marburg.

Berthold, Jost, & Loogen, Rita. (2006). Skeletons for recursively unfolding process topologies. Joubert, Gerhard R., Nagel, Wolfgang E., Peters, Frans J., Plata, Oscar G., Tirado, P., & Zapata, Emilio L. (eds), *Parallel Computing: Current & Future Issues of High-End Computing, ParCo 2005, Malaga, Spain*. NIC Series 33. Central Institute for Applied Mathematics, Jülich, Germany.

Berthold, Jost, & Loogen, Rita. (2007). Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. *ParCo '07. Parallel Computing: Architectures, Algorithms and Applications*. IOS Press.

Berthold, Jost, Dieterle, Mischa, Loogen, Rita, & Priebe, Steffen. (2008). Hierarchical master-worker skeletons. Warren, David S., & Hudak, Paul (eds), *Practical Aspects of Declarative Languages (PADL'08)*. LNCS 4902. San Francisco (CA), USA: Springer-Verlag.

Berthold, Jost, Dieterle, Mischa, Lobachev, Oleg, & Loogen, Rita. (2009a). Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. *Pages 47–55 of:* Großpitsch, K.-E., Henkersdorf, A., Uhrig, S., Ungerer, T., & Hähner, J. (eds), *Workshop on Many-Cores at ARCS '09 – 22nd International Conference on Architecture of Computing Systems 2009*. VDE-Verlag.

Berthold, Jost, Dieterle, Mischa, & Loogen, Rita. (2009b). Implementing parallel Google map-reduce in Eden. *Pages 990–1002 of:* Sips, Henk, Epema, Dick, & Lin, Hai-Xiang (eds), *Euro-Par 2009 Parallel Processing*. LNCS 5704. Springer Berlin Heidelberg.

Berthold, Jost, Dieterle, Mischa, Lobachev, Oleg, & Loogen, Rita. (2009c). *Parallel FFT with Eden skeletons*. PaCT '09. Springer. Pages 73–83.

Berthold, Jost, Loidl, Hans-Wolfgang, & Hammond, Kevin. (2016). PAEAN: Portable and scalable runtime support for parallel Haskell dialects. *Journal of functional programming*, **26**.

Bischof, Holger, & Gorlatch, Sergei. (2002). *Double-scan: Introducing and implementing a new data-parallel skeleton*. Euro-Par '02. Springer. Pages 640–647.

Blumofe, Robert D., & Leiserson, Charles E. (1999). Scheduling multithreaded computations by work stealing. *J. acm*, **46**(5), 720–748.

Botorog, G. H., & Kuchen, H. (1996). *Euro-Par'96 Parallel Processing*. LNCS 1123. Springer-Verlag. Chap. Efficient parallel programming with algorithmic skeletons, pages 718–731.

Brown, C., & Hammond, K. (2010). Ever-decreasing circles: a skeleton for parallel orbit calculations in Eden. *Draft Proceedings of the Symposium on Trends in Functional Programming*. TFP '10.

Buono, D., Danelutto, M., & Lametti, S. (2010). Map, reduce and mapreduce, the skeleton way. *Procedia computer science*, **1**(1), 2095–2103. ICCS 2010.

Chakravarty, Manuel M. T., Leshchinskiy, Roman, Peyton Jones, Simon L., Keller, Gabriele, & Marlow, Simon. (2007). Data Parallel Haskell: a status report. *Pages 10–18 of: DAMP '07*. ACM Press.

Chakravarty, Manuel M.T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., & Grover, Vinod. (2011). Accelerating Haskell array codes with multicore GPUs. *Pages 3–14 of: Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. ACM.

Chase, David, & Lev, Yossi. (2005). Dynamic circular work-stealing deque. *Pages 21–28 of: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. ACM.

Clifton-Everest, Robert, McDonell, Trevor L, Chakravarty, Manuel M T, & Keller, Gabriele. (2014). Embedding Foreign Code. *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*. LNCS. Springer-Verlag.

Cole, M. I. (1989). Algorithmic skeletons: Structured management of parallel computation. *Research Monographs in Parallel and Distributed Computing*. Pitman.

Czaplicki, Evan, & Chong, Stephen. (2013). Asynchronous functional reactive programming for guis. *Sigplan not.*, **48**(6), 411–422.

Dagand, Pierre-Évariste, Kostić, Dejan, & Kuncak, Viktor. (2009). Opis: Reliable distributed systems in OCaml. *Pages 65–78 of: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. ACM.

Danelutto, Marco, Meglio, Roberto Di, Orlando, Salvatore, Pelagatti, Susanna, & Vanneschi, Marco. (1992). A methodology for the development and the support of massively parallel programs. *Future generation computer systems*, **8**(1), 205–220.

Darlington, J., Field, AJ, Harrison, PG, Kelly, PHJ, Sharp, DWN, Wu, Q., & While, RL. (1993). Parallel programming using skeleton functions. *Pages 146–160 of: Parallel architectures and languages Europe*. Springer-Verlag.

Dastgeer, Usman, Enmyren, Johan, & Kessler, Christoph W. (2011). Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. *Pages 25–32*

of: Proceedings of the 4th International Workshop on Multicore Software Engineering. IWMSE '11. ACM.

de la Encina, Alberto, Hidalgo-Herrero, Mercedes, Rabanal, Pablo, & Rubio, Fernando. (2011). *A parallel skeleton for genetic algorithms*. IWANN '11. Springer. Pages 388–395.

Dean, Jeffrey, & Ghemawat, Sanjay. (2008). MapReduce: simplified data processing on large clusters. *Communications of the acm*, **51**, 107–113.

Dean, Jeffrey, & Ghemawat, Sanjay. (2010). MapReduce: a flexible data processing tool. *Communications of the acm*, **53**, 72–77.

Dieterle, M., Horstmeyer, T., & Loogen, R. (2010a). Skeleton composition using remote data. *Pages 73–87 of:* Carro, M., & Peña, R. (eds), *12th International Symposium on Practical Aspects of Declarative Languages*. PADL '10, vol. 5937. Springer-Verlag.

Dieterle, M., Horstmeyer, T., Loogen, R., & Berthold, J. (2016). Skeleton composition versus stable process systems in Eden. *Journal of functional programming*, **26**.

Dieterle, Mischa, Berthold, Jost, & Loogen, Rita. (2010b). *A skeleton for distributed work pools in Eden*. FLOPS '10. Springer. Pages 337–353.

Dieterle, Mischa, Horstmeyer, Thomas, Berthold, Jost, & Loogen, Rita. (2013). *Iterating skeletons*. IFL '12. Springer. Pages 18–36.

Dinan, James, Larkins, D. Brian, Sadayappan, P., Krishnamoorthy, Sriram, & Nieplocha, Jarek. (2009). Scalable work stealing. *Pages 53:1–53:11 of: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. ACM.

Foltzer, Adam, Kulkarni, Abhishek, Swords, Rebecca, Sasidharan, Sajith, Jiang, Eric, & Newton, Ryan. (2012). A meta-scheduler for the Par-monad: Composable scheduling for the heterogeneous cloud. *Sigplan not.*, **47**(9), 235–246.

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

Gentleman, W. Morven. (1978). Some complexity results for matrix computations on parallel processors. *Journal of the acm*, **25**(1), 112–115.

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

Gorlatch, Sergei, & Bischof, Holger. (1998). A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel processing letters*, **8**(4).

Hammond, Kevin, Berthold, Jost, & Loogen, Rita. (2003). Automatic skeletons in Template Haskell. *Parallel processing letters*, **13**(03), 413–424.

Harris, Mark, Sengupta, Shubhabrata, & Owens, John D. (2007). Parallel prefix sum (scan) with CUDA. *GPU gems*, **3**(39), 851–876.

Harris, Tim, Marlow, Simon, Peyton Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. ACM.

Hey, Anthony J. G. (1990). Experiments in MIMD parallelism. *Future generation computer systems*, **6**(3), 185–196.

Hippold, J., & Rünger, G. (2006). Task pool teams: A hybrid programming environment for irregular algorithms on SMP clusters. *Concurrency and computation: Practice and experience*, **18**, 1575–1594.

Horstmeyer, Thomas, & Loogen, Rita. (2013). Graph-based communication in Eden. *Higher-order and symbolic computation*, **26**(1), 3–28.

Huang, Liwen, Hudak, Paul, & Peterson, John. (2007). *HPorter: Using arrows to compose parallel processes*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 275–289.

Hudak, Paul, Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003). *Arrows, robots, and functional reactive programming*. Springer. Pages 159–187.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1–3), 67–111.

Hughes, John. (2005). *Programming with arrows*. AFP '04. Springer. Pages 73–129.

Jacobs, Bart, Heunen, Chris, & Hasuo, Ichiro. (2009). Categorical semantics for arrows. *Journal of functional programming*, **19**(3-4), 403–438.

Janjic, Vladimir, Brown, Christopher Mark, Neunhoeffer, Max, Hammond, Kevin, Linton, Stephen Alexander, & Loidl, Hans-Wolfgang. (2013). Space exploration using parallel orbits: a study in parallel symbolic computing. *Parallel computing*.

Karasawa, Y., & Iwasaki, H. (2009). A parallel skeleton library for multi-core clusters. *Pages 84–91 of: International Conference on Parallel Processing 2009*.

Keller, Gabriele, Chakravarty, Manuel M.T., Leshchinskiy, Roman, Peyton Jones, Simon, & Lippmeier, Ben. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *Sigplan not.*, **45**(9), 261–272.

Kuchen, Herbert. (2002). A skeleton library. *Pages 620–629 of:* Monien, Burkhard, & Feldmann, Rainer (eds), *Parallel Processing*. Euro-Par '02. Springer.

Kuper, Lindsey, Todd, Aaron, Tobin-Hochstadt, Sam, & Newton, Ryan R. (2014). Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. *Sigplan not.*, **49**(6), 2–14.

Lengauer, Christian, Gorlatch, Sergei, & Herrmann, Christoph. (1997). The static parallelization of loops and recursions. *The journal of supercomputing*, **11**(4), 333–353.

Li, Peng, & Zdancewic, S. (2006). Encoding information flow in Haskell. *Pages 12–16 of: 19th IEEE Computer Security Foundations Workshop*. CSFW '06.

Li, Peng, & Zdancewic, Steve. (2010). Arrows for secure information flow. *Theoretical computer science*, **411**(19), 1974–1994.

Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic notes in theoretical computer science*, **229**(5), 97–117.

Linton, S., Hammond, K., Konovalov, A., Al Zain, A. D., Trinder, P., Horn, P., & Roozemond, D. (2010). Easy composition of symbolic computation software: a new lingua franca for symbolic computation. *Pages 339–346 of: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. ISSAC '10. ACM Press.

Liu, Hai, Cheng, Eric, & Hudak, Paul. (2009). Causal commutative arrows and their optimization. *Sigplan not.*, **44**(9), 35–46.

Lobachev, Oleg. (2011). *Implementation and evaluation of algorithmic skeletons: Parallelisation of computer algebra algorithms*. Ph.D. thesis, Philipps-Universität Marburg.

Lobachev, Oleg. (2012). *Parallel computation skeletons with premature termination property*. FLOPS 2012. Springer. Pages 197–212.

Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., & Rubio, F. (2003). Parallelism Abstractions in Eden. *Pages 71–88 of:* Rabhi, F. A., & Gorlatch, S. (eds), *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag.

Loogen, Rita. (2012). *Eden – parallel functional programming with Haskell*. Springer. Pages 142–206.

Loogen, Rita, Ortega-Mallén, Yolanda, & Peña-Marí, Ricardo. (2005). Parallel Functional Programming in Eden. *Journal of Functional Programming*, **15**(3), 431–475. Special Issue on Functional Approaches to High-Performance Parallel Programming.

Lämmel, Ralf. (2008). Google's mapreduce programming model — revisited. *Science of computer programming*, **70**(1), 1–30.

Maier, Patrick, Stewart, Robert, & Trinder, Phil. (2014). The HdpH DSLs for scalable reliable computation. *Sigplan not.*, **49**(12), 65–76.

Mainland, Geoffrey, & Morrisett, Greg. (2010). Nikola: Embedding compiled GPU functions in Haskell. *Sigplan not.*, **45**(11), 67–78.

Marlow, S., Peyton Jones, S., & Singh, S. (2009). Runtime support for multicore Haskell. *Acm sigplan notices*, **44**(9), 65–78.

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

Marlow, Simon, Newton, Ryan, & Peyton Jones, Simon. (2011). A monad for deterministic parallelism. *Sigplan not.*, **46**(12), 71–82.

McDonell, Trevor L., Chakravarty, Manuel M. T., Grover, Vinod, & Newton, Ryan R. (2015). Type-safe runtime code generation: Accelerate to LLVM. *Sigplan not.*, **50**(12), 201–212.

Michael, Maged M., Vechev, Martin T., & Saraswat, Vijay A. (2009). Idempotent work stealing. *Sigplan not.*, **44**(4), 45–54.

Nilsson, Henrik, Courtney, Antony, & Peterson, John. (2002). Functional reactive programming, continued. *Pages 51–64 of: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. New York, NY, USA: ACM.

Olivier, S., & Prins, J. (2008). Scalable dynamic load balancing using UPC. *Pages 123–131 of: 37th International Conference on Parallel Processing*.

Paterson, Ross. (2001). A new notation for arrows. *Sigplan not.*, **36**(10), 229–240.

Peña, R., & Rubio, F. (2001). Parallel Functional Programming at Two Levels of Abstraction. *Pages 187–198 of: PPDP'01 — Intl. Conf. on Principles and Practice of Declarative Programming*.

Perfumo, Cristian, Sönmez, Nehir, Stipic, Srdjan, Unsal, Osman, Cristal, Adrián, Harris, Tim, & Valero, Mateo. (2008). The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. *Pages 67–78 of: Proceedings of the 5th Conference on Computing Frontiers*. CF '08. ACM.

Poldner, Michael, & Kuchen, Herbert. (2005). Scalable farms. *Pages 795–802 of:* Joubert, Gerhard R., Nagel, Wolfgang E., Peters, Frans J., Plata, Oscar G., Tirado, P., & Zapata, Emilio L. (eds), *PARCO*. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany.

Priebe, Steffen. (2006). Dynamic task generation and transformation within a nestable workpool skeleton. *Euro-Par*. LNCS 4128.

Rabhi, F. A., & Gorlatch, S. (eds). (2003). *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag.

Rudolph, Larry, Slivkin-Allalouf, Miriam, & Upfal, Eli. (1991). A simple load balancing scheme for task allocation in parallel machines. *Pages 237–245 of: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '91. ACM.

Russo, Alejandro, Claessen, Koen, & Hughes, John. (2008). A library for light-weight information-flow security in Haskell. *Pages 13–24 of: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*. Haskell '08. ACM.

Stewart, Robert, Maier, Patrick, & Trinder, Phil. (2016). Transparent fault tolerance for scalable functional computation. *Journal of functional programming*, **26**.

Svensson, Joel. (2011). *Obsidian: Gpu kernel programming in haskell*. Ph.D. thesis, Chalmers University of Technology.

Trinder, Phil W., Hammond, Kevin, Mattson Jr., James S., Partridge, Andrew S., & Peyton Jones, Simon L. (1996). GUM: a Portable Parallel Implementation of Haskell. *PLDI'96*. ACM Press.

Trinder, P.W., Hammond, K., Loidl, H-W., & Peyton Jones, S. (1998). Algorithm + Strategy = Parallelism. *Journal of functional programming*, **8**(1), 23–60.

van Nieuwpoort, Rob V., Kielmann, Thilo, & Bal, Henri E. (2001). Efficient load balancing for wide-area divide-and-conquer applications. *Sigplan not.*, **36**(7), 34–43.

Vizzotto, Juliana, Altenkirch, Thorsten, & Sabry, Amr. (2006). Structuring quantum effects: superoperators as arrows. *Mathematical structures in computer science*, **16**(3), 453–468.

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

A 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*, which is for example used in the definition of $***$:

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

Next, we give the definition of *evalN* which also helps us to define *map*, and *zipWith* on Arrows. The *evalN* combinator in Fig. A 1 converts a list of Arrows $[arr\ a\ b]$ into an Arrow $arr\ [a]\ [b]$.

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

Figure A 1: The definition of *evalN*

The *mapArr* combinator (Fig. A 2) lifts any Arrow *arr a b* to an Arrow $arr\ [a]\ [b]$. The original inspiration was from Hughes (2005), but the definition as then unified with *evalN*.

$$mapArr :: ArrowChoice\ arr \Rightarrow arr\ a\ b \to arr\ [a]\ [b]$$
$$mapArr = evalN \circ repeat$$

Figure A 2: The definition of *map* over Arrows.

Finally, with the help of *mapArr* (Fig. A 2), we can define *zipWithArr* (Fig. A 3) that lifts any Arrow $arr\ (a,b)\ c$ to an Arrow $arr\ ([a],[b])\ [c]$.

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

Figure A 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:

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

B Profunctor Arrows

In Fig. B 1 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 $(>\!\!>\!\!>)$ has the type $(>\!\!>\!\!>) :: Category\ cat \Rightarrow cat\ a\ b \to cat\ b\ c \to cat\ a\ c$ and is therefore not part of the *Arrow* type class like presented in this paper.[21]

instance $(Category\ p, Strong\ p) \Rightarrow Arrow\ p$ **where**
  $arr\ f = dimap\ id\ f\ id$
  $first = first'$

instance $(Category\ p, Strong\ p, Costrong\ p) \Rightarrow ArrowLoop\ p$ **where**
  $loop = loop'$

instance $(Category\ p, Strong\ p, Choice\ p) \Rightarrow ArrowChoice\ p$ **where**
  $left = left'$

Figure B 1: Profunctors as Arrows.

[21] 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`.

C Omitted Function Definitions

We have omitted some function definitions in the main text for brevity, and redeem this here. We begin with warping Eden's build-in *RemoteData* to *Future* in Figure C 1

```
data RemoteData a = RD { rd :: RD 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 (RemoteData a) where
  rnf = rnf ∘ rd
instance Trans (RemoteData a)
instance (Trans a) ⇒ Future RemoteData a Conf where
  put _ = put'
  get _ = get'
instance (Trans a) ⇒ Future RemoteData a () where
  put _ = put'
  get _ = get'
```

Figure C 1: *RD*-based *RemoteData* version of *Future* for the Eden backend.

Next, we have the definition of *BasicFuture* in Fig. C 2 and the corresponding *Future* instances.

```
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'
```

Figure C 2: *BasicFuture* type and its *Future* instance for the *Par* Monad and GpH Haskell backends.

Figures C 3–C 6 show the definitions and a visualisations of two parallel *map* variants, defined using *parEvalN* and its lazy counterpart.

Arrow versions of Eden's *shuffle*, *unshuffle* and the definition of *takeEach* are in Figure C 7. Similarly, Figure C 8 contains the definition of Arrow versions of Eden's *lazy*

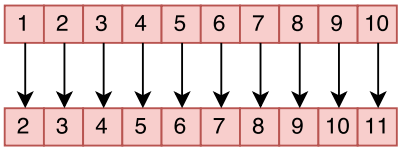

Figure C 3: *parMap* depiction.

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

Figure C 4: Definition of parMap.

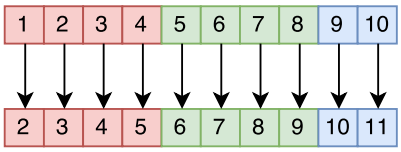

Figure C 5: *parMapStream* depiction.

```
parMapStream :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) ⇒
  conf → ChunkSize → arr a b → arr [a] [b]
parMapStream conf chunkSize f = parEvalNLazy conf chunkSize (repeat f)
```

Figure C 6: *parMapStream* definition.

and *rightRotate* utility functions. Fig. C 9 contains Eden's definition of *lazyzip3* together with the utility functions *uncurry3* and *threetotwo*. The full definition of *farmChunk* is in Figure C 10. Eden definition of *ring* skeleton is in Figure C 11. It follows Loogen (2012).

```
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 C 7: *shuffle*, *unshuffle*, *takeEach* definition.

The *parEval2* skeleton is defined in Figure C 12. We start by transforming the (a,c) input into a two-element list $[Either\ a\ c]$ by first tagging the two inputs with *Left* and *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. Furthermore, Fig. C 13 contains the omitted definitions of *prMMTr* (which calculates

```
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 C 8: *lazy* and *rightRotate* definitions.

```
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 C 9: *lazyzip3*, *uncurry3* and *threetotwo* definitions.

AB^T for two matrices A and B), *splitMatrix* (which splits the a matrix into chunks), and lastly *matAdd*, that calculates $A+B$ for two matrices A and B.

D Syntactic Sugar

Finally, we also give the definitions for some syntactic sugar for PArrows, namely $|***|$ and $|\&\&\&|$. For basic Arrows, we have the $***$ combinator (Fig. 3) 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) ()))) ⇒
  arr a b → arr c d → arr (a,c) (b,d)
(|***|) = parEval2 ()
```

We define the parallel $|\&\&\&|$ in a similar manner to its sequential pendant &&& (Fig. 3):

```
(|&&&|) :: (ArrowChoice arr, ArrowParallel arr (Either a a) (Either b c) ()) ⇒
  arr a b → arr a c → arr a (b,c)
(|&&&|) f g = (arr $ λa → (a,a)) >>> f |***| g
```

```
farmChunk :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
  ArrowChoice arr, ArrowApply arr) ⇒
  conf → ChunkSize → NumCores → arr a b → arr [a] [b]
farmChunk conf chunkSize numCores f =
  unshuffle numCores >>>
  parEvalNLazy conf chunkSize (repeat (mapArr f)) >>>
  shuffle
```

Figure C 10: *farmChunk* definition.

```
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 C 11: Eden's definition of the *ring* skeleton.

```
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 C 12: *parEval2* definition.

```
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
matAdd = chunksOf (dimX x) $ zipWith (+) (concat x) (concat y)
```

Figure C 13: *prMMTr*, *splitMatrix* and *matAdd* definition.