Under consideration for publication in J. Functional Programming

Arrows for Parallel Computations

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 therefore form an alternative to Monads for API design. We express parallelism using this concept in a novel way: We define an Arrow-based language for parallelism and implement it using multiple parallel Haskells. In this manner we are able to bridge across various parallel Haskells.

Additionally, using these parallel Arrows (PArrows) has the benefit of being portable across multiple parallel Haskell implementations. Furthermore, as each parallel computation is an Arrow, PArrows can be readily composed and transformed as such. In order to allow for more sophisticated communication schemes between computation nodes in distributed systems we introduce the concept of Futures to wrap existing direct communication concepts in backends.

To show that PArrows have similar expressiveness to existing parallel languages, we also implement several parallel skeletons. Benchmarks show that our framework does not induce any notable overhead performance-wise. We finally conclude that Arrows turn out to be a useful tool for composing parallel programs and that our particular approach results in programs that are portable accross multiple backends. **OL: Summarize conclusions**

MB: Jedes Kapitel soll einmal ins Abstract. Conclusions sollen mit ins Abstract

Contents

1 Introduction

Parallel functional languages have a long history of being used for experimenting with novel parallel programming paradigms including the expression of parallelism. 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 are a generalization of the function arrow ($\rightarrow$) and serve as general interface to computations. 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 in order 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 **Phil: quantify, e.g. less than 10 % MB: we do this in the contributions section of the Introduction, enough?**, very low-penalty programming interface that allows to switch the parallel backends at will. Further backends based on HdpH or a Frege implementation (on the Java Virtual Machine) are viable, too.

Contributions We propose a new Arrow-based encoding for parallelism based on a new Arrow combinator $parEvalN :: [arr\ a\ b] \rightarrow arr\ [a]\ [b]$ in Section 4, which converts a list of Arrows into a new parallel Arrow. Because of this, we do not lose any benefits of using Arrows as the parallelism is encapsulated as another combinator instead of a different type. The resulting Arrow can be used in the same way as a potential serial version.

This is a big advantage as we do not introduce any new types as Monad solutions like the *Par* Monad do. We can just 'plug' in parallel parts into sequential Arrow-based programs and libraries without having to change integral parts of the code. **MB: mention Functions here? We have everything Eden has, but more.**

MB: Überleitung auf Related Work hier machen

We do not reimplement all the parallel internals, as we host this functionality in the *ArrowParallel* type class, which abstracts all parallel implementation logic. This way, we can have multiple backends – currently a GpH, a *Par* Monad and an Eden version are supported. The backends can easily be swapped, so we are not bound to any specific one. So as an 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.

We introduce a *Future* type class in Section 5 to enable direct communication of data between nodes in a distributed memory setting similar to Eden's Remote Data (*RD*). Direct communication is useful in a distributed memory setting because they allow for inter-node communication without blocking the master-node. As the *Par* Monad and GpH backends (in their current form) only support shared memory execution and do not have to fix any communication problems, we only use a wrapping dummy *BasicFuture*.

We show that it is possible to define algorithmic skeletons with Arrows (Sections 6, 7) and finally demonstrate that Arrow Parallelism is a viable alternative to existing approaches and prove that only low overhead is introduced in Section 8, i.e. less than 1.7% in cases outside of the error bar of the measurement. **MB: fix this after benchmarks are finally done**.

MB: Mention Future work here?

MB: Conclusion?

2 Related Work

Parallel Haskells. Of course, the three parallel Haskell flavours we use 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) are related to this work. We use these languages as backends: our DSL can switch from one to other at user's command.

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. Further parallel Haskell approaches include pH (Nikhil & Arvind, 2001), research work done on distributed variants of GpH (Trinder *et al.*, 1996; Aljabri *et al.*, 2014, 2015), and low-level Eden 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) are recent in-focus research topics in Eden. This also includes the definitions of new skeletons (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).

More different approaches include 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) deserves a special mention. Accelerate is completely orthogonal to our approach. Marlow authored a recent book in 2013 on parallel Haskells.

Algorithmic skeletons. Algorithmic skeletons were introduced by Cole (1989). Early publications on this topic include (Darlington *et al.*, 1993; Botorog & Kuchen, 1996; Danelutto *et al.*, 1997; Gorlatch, 1998; Lengauer *et al.*, 1997). Rabhi & Gorlatch (2003) consolidated early reports on high-level programming approaches. 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), basically they are a generalised function arrow $\rightarrow$. Hughes (2005a) presents a tutorial on Arrows. Some theoretical details on Arrows (Jacobs *et al.*, 2009; Lindley *et al.*, 2011; Atkey, 2011) are viable. 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 optimizations. 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.

Other languages. Although this work is centered on Haskell implementation of Arrows, it is applicable to any functional programming language where parallel evaluation and Arrows can be defined. Experiments with our approach in Frege language[1] (which is basically Haskell on the JVM) were quite successful. A new approach to Haskell on the JVM is the Eta language[2]. However, both are beyond the scope of this work.

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.

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

```
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 definition of the *Arrow* type class and its two most typical instances.

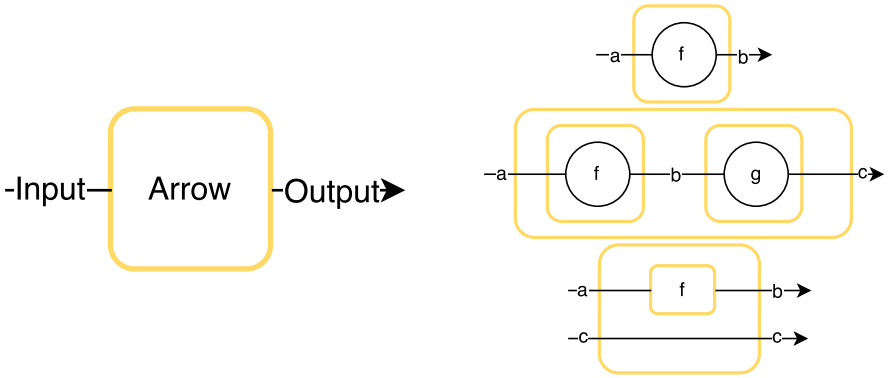

Figure 2: Schematic depiction of an Arrow (left) and its basic combinators *arr*, >>> and *first* (right).

3 Background

3.1 Arrows

Arrows were introduced by Hughes (2000) as a general interface for computation. 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 >>= 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 $(\rightarrow)$ and the Kleisli type (Fig. 1), which wraps monadic functions, e.g. $a \rightarrow m\ b$ (with m being a Monad).

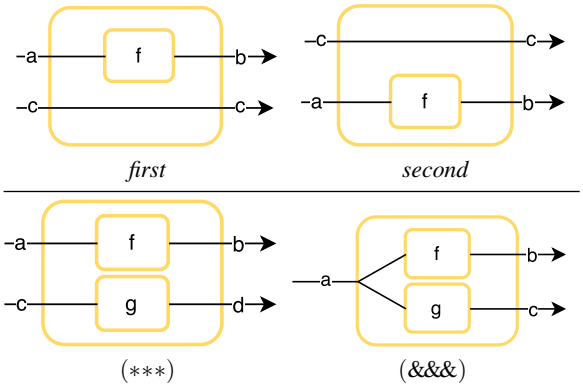

Figure 3: Visual depiction of syntactic sugar for Arrows.

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

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

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

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

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

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

The more restrictive interface of Arrows allows for more elaborate composition and transformation combinators—a Monad can be *anything*, an Arrow is a process of doing something, a *computation*. This is exactly one of the key challenges in parallel computing.

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 $a \rightarrow b$ in parallel or $parEvalN :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$, 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.

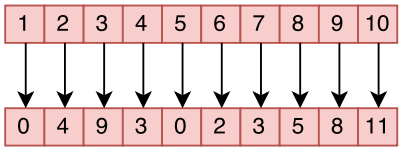

Figure 4: Schematic illustration of *parEvalN*. A list of inputs is transformed by different functions in parallel.

There exist several parallel Haskells already. Among the most important are probably GpH (based on *par* & *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 reimplement yet another parallel runtime, but to represent parallelism with Arrows, we base our efforts on existing work which we wrap as backends behind a common interface. For this 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 TemplateHaskell.

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.[3] Besides some basic primitives (*par* & *pseq* hints), 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. Fig. 5 shows a visual representation of this code.

[3] 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.

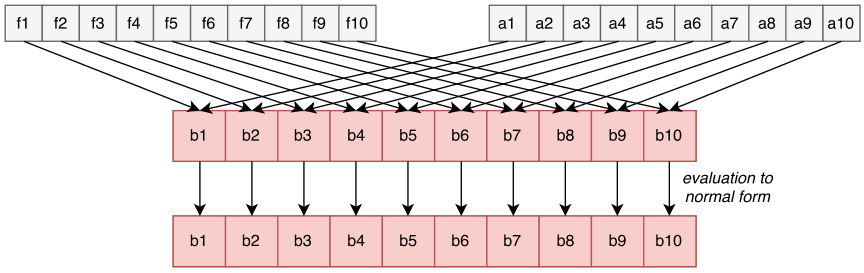

Figure 5: Data flow of the GpH *parEvalN* version.

3.2.2 Par Monad

The *Par* Monad[4] introduced by Marlow *et al.* (2011), is a Monad designed for composition of parallel programs. Let:

$$parEvalN :: (NFData\ b) \Rightarrow [a \to b] \to [a] \to [b]$$
$$parEvalN\ fs\ as = runPar\ \$$$
$$(sequenceA\ \$\ map\ (spawnP)\ \$\ zipWith\ (\$)\ fs\ as) >\!\!>\!\!= mapM\ get$$

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 $spawnP :: NFData\ a \Rightarrow a \to Par\ (IVar\ a)$ to transform them to a list of not yet evaluated forked away computations $[Par\ (IVar\ b)]$, which we convert to $Par\ [IVar\ b]$ with *sequenceA*. We wait for the computations to finish by mapping over the *IVar b* values inside the *Par* Monad with *get*. This results in $Par\ [b]$. We execute this process with *runPar* to finally get $[b]$. Fig. 6 shows a graphical representation.

3.2.3 Eden

Eden (Loogen *et al.*, 2005; Loogen, 2012) is a parallel Haskell for distributed memory and comes with a MPI and a PVM backends.[5] 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:

[4] 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/`.

[5] 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/`.

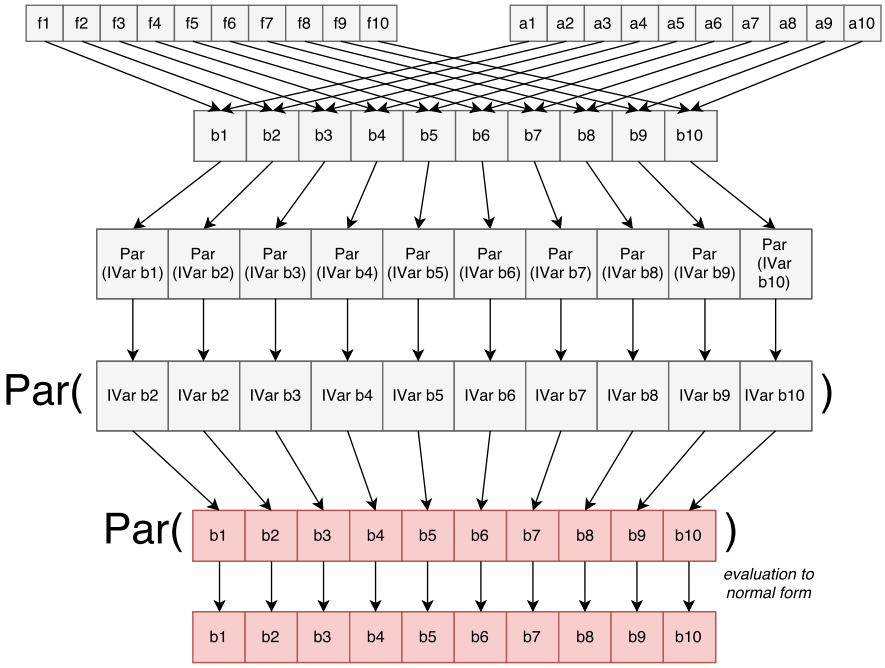

Figure 6: Data flow of the *Par* Monad *parEvalN* version.

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

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[6] (Berthold & Loogen, 2007). In the next sections we will present some *trace visualizations*, the post-mortem process diagrams of Eden processes and their activity.

The trace visualizations are color-coded. In such a visualization (Fig. 13), the x axis shows the time, the y axis enumerates the machines and processes. The visualization 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.

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

4 Parallel Arrows

Arrows are a general interface to computation. Here we introduce special Arrows as general interface to *parallel computations*. First, we present the interface and explain the reasonings behind it. Then, we discuss some implementations using existing parallel Haskells. Finally, we explain why using Arrows for expressing parallelism is beneficial.

4.1 The ArrowParallel type class

A parallel computation (on functions) in its purest form can be seen as execution of some functions $a \to b$ in parallel, as our *parEvalN* prototype shows (Sec. 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 our utility function *listApp* 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* and we want this to be an interface for different backends, we introduce a new type class *ArrowParallel arr a b*:

$$\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ \textbf{where}$$
$$\quad parEvalN :: [arr\ a\ b] \to arr\ [a]\ [b]$$

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 also introduce an additional *conf* parameter to the function. We also do not want *conf* to be a fixed type, as the configuration parameters can differ for different instances of *ArrowParallel*.

$$\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ conf\ \textbf{where}$$
$$\quad parEvalN :: conf \to [arr\ a\ b] \to arr\ [a]\ [b]$$

We do not require the *conf* parameter in every implementation. If it is not needed, we usually just default the *conf* type parameter to $()$ and even blank it out in the parameter list of the implemented *parEvalN*.

4.2 ArrowParallel instances

4.2.1 Glasgow parallel Haskell

The GpH implementation of *ArrowParallel* is implemented in a straightforward manner by using *listApp* (Appendix A) combined with the $withStrategy :: Strategy\ a \to a \to a$ combinators from GpH, where *withStrategy* is the same as $using :: a \to Strategy\ a \to a$, but with flipped parameters. For most cases a fully evaluating version like in Fig. 7 would probably suffice, but as the GpH interface allows the user to specify the level of evaluation to be done via the *Strategy* interface, our DSL should allow for this. We therefore introduce the *Conf a* data-type that simply wraps a *Strategy a*. With this definition in place, we can provide a delegating version of the non-configurable instance. We show this in Fig. B 1.

```
data Conf a = Conf (Strategy a)
instance (NFData b, ArrowApply arr, ArrowChoice arr) ⇒ ArrowParallel arr a b () where
  parEvalN _ fs =
    listApp fs >>>
    arr (withStrategy (parList rdeepseq))
```

Figure 7: Fully evaluating *ArrowParallel* instance for the GpH Haskell backend.

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) ⇒
  ArrowParallel arr a b (Conf b) where
  parEvalN (Conf strat) fs =
    listApp fs >>>
    arr (withStrategy (parList strat)) &&& arr id >>>
    arr (uncurry pseq)
```

Figure 8: Configurable *ArrowParallel* instance for the GpH Haskell backend.

4.2.2 Par Monad

OL: introduce a newcommand for par-monad, "arrows", "parrows" and replace all mentions to them to ensure uniform typesetting done!, we write Arrows. also "Monad"? done! The *Par* Monad implementation (Fig. 9) makes use of Haskells laziness and *Par* Monad's $spawnP :: NFData\ a \Rightarrow a \rightarrow Par\ (IVar\ a)$ function. The latter forks away the computation of a value and returns an *IVar* containing the result in the *Par* Monad.

We therefore apply each function to its corresponding input value with and then fork the computation away with *arr spawnP* inside a *zipWithArr* (Fig. A 3) call. This yields a list $[Par\ (IVar\ b)]$, which we then convert into $Par\ [IVar\ b]$ with *arr sequenceA*. In order to wait for the computation to finish, we map over the *IVar*s inside the *Par* Monad with *arr* $(\gg\!= mapM\ get)$. The result of this operation is a $Par\ [b]$ from which we can finally remove the Monad again by running *arr runPar* to get our output of $[b]$.

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) ⇒
  ArrowParallel arr a b conf where
    parEvalN _ fs =
      (arr $ λas → (fs, as)) >>>
      zipWithArr (app >>> arr spawnP) >>>
      arr sequenceA >>>
      arr (≫= mapM get) >>>
      arr runPar
```

Figure 9: ArrowParallel instance for the Par Monad backend.

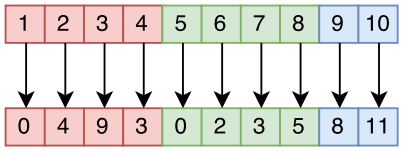

Figure 10: Schematic depiction of *parEvalNLazy*.

4.2.3 Eden

For both the GpH Haskell and *Par* Monad implementations we could use general instances of *ArrowParallel* that just require the *ArrowApply* and *ArrowChoice* type classes. With Eden this is not the case as we can only spawn a list of functions and we cannot extract simple functions out of arrows. While we could still manage to have only one class in the module by introducing a type class:

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

However, we avoid doing so for aesthetic reasons. For now, we just implement *ArrowParallel* for normal functions:

```
instance (Trans a, Trans b) ⇒ ArrowParallel (→) a b conf where
parEvalN _ fs as = spawnF fs as
```

and the Kleisli type:

```
instance (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)
```

4.3 Extending the Interface

With the *ArrowParallel* type class in place and implemented, we can now implement some further basic parallel interface functions. These are algorithmic skeletons that, however, mostly serve as a foundation to further, more specific algorithmic skeletons.

4.3.1 Lazy parEvalN

The function *parEvalN* is 100% strict, which means that it fully evaluates all passed arrows. 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 parallel arrows created by feeding chunks of the passed *ChunkSize* into the regular parEvalN by using *listApp*. The resulting $[[b]]$ is lastly converted into $[b]$ with *arr concat*.

```
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) >>>
  listApp fchunks >>>
  arr concat
  where fchunks = map (parEvalN conf) $ chunksOf chunkSize fs
```

Figure 11: Definition of *parEvalNLazy*.

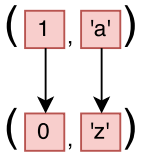

Figure 12: Schematic depiction of *parEval2*.

4.3.2 Heterogeneous tasks

We have only talked about the parallelization arrows of the same type until now. But sometimes we want to parallelize heterogeneous types as well. However, we can implement such a *parEval2* combinator (Figs. 12, B 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. The 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 a mock-up parallel arrow combinator:

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

In a distributed environment, a resulting arrow of this combinator first evaluates all $[arr\ a\ b]$ in parallel, sends the results back to the master node, rotates the input once 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 above example could be rewritten into only one *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 larger amount of data and number of processes (as e.g. Berthold *et al.*, 2009c, showcases).

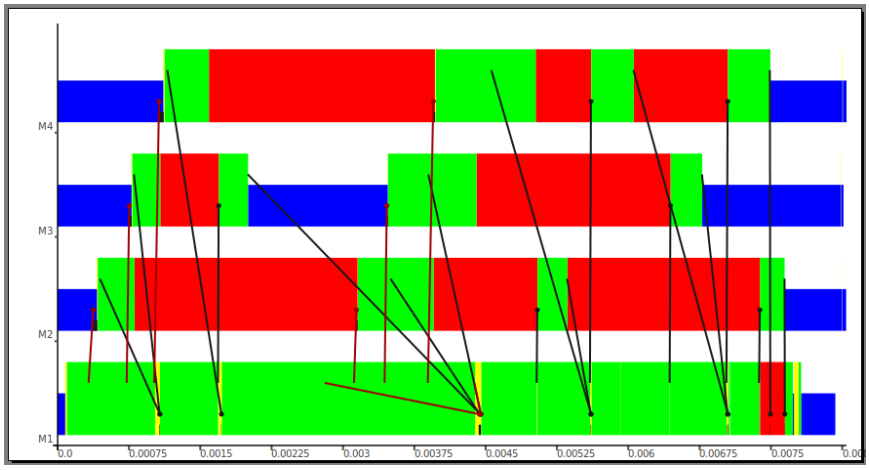

Figure 13: Communication between 4 Eden processes without Futures. All communication goes through the master node. Each bar represents one process. Black lines between processes represent communication. Colors: blue $\hat{=}$ idle, green $\hat{=}$ running, red $\hat{=}$ blocked, yellow $\hat{=}$ suspended.

OL: more practical and heavy-weight example! fft (I have the code)?
MB: Depends... Are the communications easy to read in such an example?
MB: Keep the description for the different colours, or link to the EdenTV description in 3.2.3
OL: ok as is OL: use the fft example (when it works)?

This is in fact 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 ships with '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 context. We do this with the *Future* type class (Fig. 14). Since *RD* is only a type synonym for a communication

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

Figure 14: Definition of the *Future* type class.

type that Eden uses internally, we have to use some wrapper classes to fit that definition, though, as Fig. B 6 shows. Technical details are in Appendix, in Section B.

For the *Par* Monad and GpH Haskell backends, we can simply use *BasicFuture*s (Fig. 15), 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.

In our communication example we can use this *Future* concept for direct communication between nodes as shown in Fig. 16. In a distributed environment, this gives us a communication scheme with messages going through the master node only if it is needed – similar to

```
data BasicFuture a = BF a
instance (NFData a) ⇒ NFData (BasicFuture a) where
  rnf (BF a) = rnf a
instance (NFData a) ⇒ Future BasicFuture a where
  put = arr BF
  get = arr (λ(BF a) → a)
```

Figure 15: The *BasicFuture* type and its *Future* instance for the *Par* Monad and GpH Haskell backends.

```
someCombinator :: (Arrow arr) ⇒ [arr a b] → [arr b c] → arr [a] [c]
someCombinator fs1 fs2 =
  parEvalN () (map (>>>put) fs1) >>>
  rightRotate >>>
  parEvalN () (map (get>>>) fs2)
```

Figure 16: The mock-up combinator in parallel.

what is shown in the trace visualization in Fig. 17. **OL: Fig. is not really clear. Do Figs with a lot of load? — fft?**

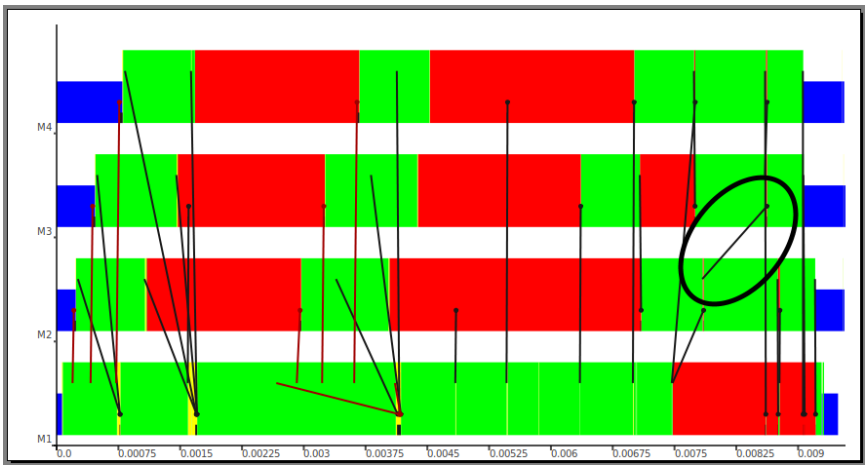

Figure 17: Communication between 4 Eden processes with Futures. Other than in Fig. 13, processes communicate directly (black lines between the bars, one example message is marked in the Figure) instead of always going through the master node (bottom bar).

6 Map-based Skeletons

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

Parallel *map* and laziness. The *parMap* skeleton (Figs. B 2, B 3) 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* is 100% strict. As *parMap* is 100% strict it has the same restrictions as *parEvalN* compared to *parEvalNLazy*. So it makes sense to also have a *parMapStream* (Figs. B 4, B 5) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*. The code is quite straightforward, we show it in Appendix.

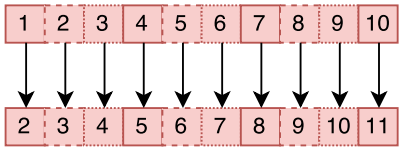

Figure 18: Schematic depiction of a *farm*, a statically load-balanced *parMap*.

```
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 19: The definition of *farm*.

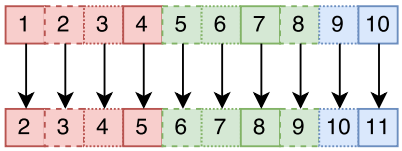

Figure 20: Schematic depiction of *farmChunk*.

Statically load-balancing parallel *map*. Our *parMap* spawns every single computation in a new thread (at least for the instances of *ArrowParallel* we gave in this paper). This can be quite wasteful and a statically load-balancing *farm* (Figs. 18, 19) that equally distributes the workload over *numCores* workers seems useful. The definitions of the helper functions *unshuffle*, *takeEach*, *shuffle* (Fig. B 7) originate from an Eden skeleton[7].

Since a *farm* is basically just *parMap* with a different work distribution, it is, again, 100% strict. So we can define *farmChunk* (Figs. 20, B 10) which uses *parEvalNLazy* instead of *parEvalN*. It is basically the same definition as for *farm*, with *parEvalN* replaced with *parEvalNLazy*.

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

7 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[8], 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 materializes with the usage of *loop* of the *ArrowLoop* type class and is probably due to the way their respective parallelism engines work internally. **MB: okay so?** As these skeletons probably do not make any practical sense besides for testing with these backends anyways (because of the shared memory between the threads), we create an extra abstraction layer for the original *parEvalN* in these skeletons called *evalN* in the *FutureEval* type class. This allows us for selective enabling and disabling of parallelism.

```
class ArrowParallel arr a b conf ⇒ FutureEval arr a b conf where
  evalN :: (ArrowParallel arr a b conf) ⇒ conf → [arr a b] → arr [a] [b]
```

As Eden has no problems with the looping skeletons, we declare a delegating instance:

```
instance ArrowParallel arr a b conf ⇒ FutureEval arr a b conf where
  evalN = parEvalN
```

The Par Monad and GpH backends have parallelism disabled in their instance of *FutureEval*. This way the skeletons can still run without errors on shared-memory machines and still be used to test programs locally.

```
instance (Arrow arr, ArrowChoice arr, ArrowApply arr,
  ArrowParallel arr a b conf) ⇒ FutureEval arr a b conf where
  evalN _ = listApp
```

7.1 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 $loop :: arr\ (a,b)\ (c,b) \to arr\ a\ c$ combinator which allows us to express recursive fix-point computations in which output values are fed back as input. For example

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

which is the same as

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

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

Figure 21: A first implementation of the *pipe* skeleton expressed with Parallel Arrows. Note that the use of *lazy* (Fig. B 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, FutureEval arr (fut a) (fut a) conf, Future fut a) ⇒
  conf → [arr a a] → arr a a
pipe conf fs = unliftFut (pipeSimple conf (map liftFut fs))
```

Figure 22: Final definition of the *pipe* skeleton with Futures.

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

defines an arrow that takes its input a and converts it into an infinite stream $[a]$ of it. Using this to our advantage gives us a first draft of a pipe implementation (Fig. 21) 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 **??**. Therefore, we introduce a more sophisticated version that internally uses Futures and obtain the final definition of *pipe* in Fig. 22.

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 common type we obtain *pipe2* (Fig. 23).

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 version of parallel piping operator $|>>>|$, which is semantically equivalent to $>>>$ similarly to other parallel syntactic sugar from Appendix C.

Another version of $>>>$ is:

$$f\ |>>>|\ g = (f \circ put) >>> (get \circ g)$$

It does not launch both arrows f and g in parallel, but allows for more smooth data communication between them. Basically, it is a *Future*-lifted *sequential* $>>>$, a way to compose parallel Arrows efficiently.

```
pipe2 :: (ArrowLoop arr, ArrowChoice arr, Future fut (([a],[b]),[c]),
  FutureEval arr (fut (([a],[b]),[c])) (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 (\_ → []) >>>
      arr (λ((a,b),c) → ((c,a),b))
(|>>>|) :: (ArrowLoop arr, ArrowChoice arr, Future fut (([a],[b]),[c]),
  FutureEval arr (fut (([a],[b]),[c])) (fut (([a],[b]),[c])) ()) ⇒
  arr a b → arr b c → arr a c
(|>>>|) = pipe2 ()
```

Figure 23: Definition of *pipe2* and a parallel >>>.

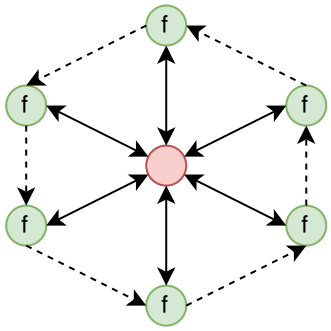

Figure 24: Schematic depiction of the ring skeleton.

7.2 Ring skeleton

Eden comes with a ring skeleton[9] (Fig. 24) implementation that allows the computation of a function $[i] \to [o]$ with a ring of nodes that communicate in a ring topology 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. B 11.

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 the *loop*'s *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. B 8). Similarly to the *pipe* from Section 7.1 (Fig. 21), we have to feed the intermediary input into our *lazy*

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

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

Figure 25: Final definition of the *ring* skeleton.

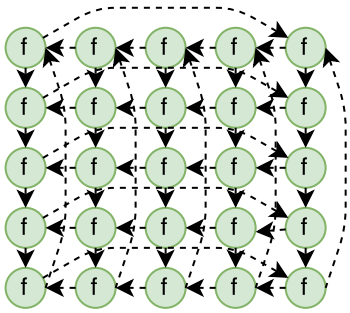

Figure 26: Schematic depiction of the *torus* skeleton.

(Fig. B 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 $arr\ (uncurry\ zip)$ so we can feed that into a our input arrow $arr\ (i,r)\ (o,r)$, which we transform into $arr\ (i,fut\ r)\ (o,fut\ r)$ before lifting it to $arr\ [(i,fut\ r)]\ [(o,fut\ r)]$ to get a list $[(o,fut\ r)]$. Finally we unzip this list into $([o],[fut\ r])$. Plugging this arrow $arr\ ([i],[fut\ r])\ ([o],fut\ r)$ into the definition of *loop* from earlier gives us $arr\ [i]\ [o]$, our ring arrow (Fig. 25). This combinator can, for example, be used to calculate the shortest paths in a graph using Warshall's algorithm.

7.3 Torus skeleton

If we take the concept of a *ring* from Section 7.2 one dimension further, we obtain a *torus* skeleton (Fig. 26, 27). 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[10] from Eden—yet again with the help of the *ArrowLoop* type class.

Similar to the *ring*, we once again start by rotating the input (Fig. B 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

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

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

```
torus :: (ArrowLoop arr, ArrowChoice arr, ArrowApply arr, Future fut a, Future fut b,
  FutureEval arr (c,fut a,fut b) (d,fut a,fut b) 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 >>> arr unshuffle) &&& (shuffle >>> evalN conf (repeat (ptorus f)) >>> app >>>
  arr (map unzip3) >>> arr unzip3 >>> threetotwo)
ptorus :: (Arrow arr, Future fut a, Future fut b) ⇒
  arr (c,a,b) (d,a,b) → arr (c,fut a,fut b) (d,fut a,fut b)
ptorus f = arr (λ∼(c,a,b) → (c,get a,get b)) >>> f >>> arr (λ∼(d,a,b) → (d,put a,put b))
```

Figure 27: Definition of the *torus* skeleton. The definitions of *lazyzip3*, *uncurry3* and *threetotwo* have been omitted and can be found in Fig. B 9

.

instead of the singular rotation in the ring as we rotate $[[fut\ a]]$ horizontally and $[[fut\ b]]$ vertically. Then, we once again 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 feed into our parallel execution.

This action is, however, more complicated than in the ring case as we have one more dimension of inputs to be transformed. We first have to *shuffle* all the inputs to then pass it into *evalN conf* (*repeat* (*ptorus f*)) which yields $[(d,fut\ a,fut\ b)]$. We can then unpack this shuffled list back to its original ordering by feeding it into the specific unshuffle arrow we created one step earlier with *arr length >>> arr unshuffle* with the use of *app* :: *arr* (*arr a b*, *a*) *c* from the *ArrowApply* type class. Finally, we unpack this matrix $[[[(d,fut\ a,fut\ b)]]]$ with *arr* (*map unzip3*) *>>> arr unzip3 >>> threetotwo* to get $([[d]],([[fut\ a]],[[fut\ b]]))$.

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. 28. If we compare the trace from a call using our arrow definition of the torus (Fig. 29) with the Eden version (Fig. 30) we can see that the behaviour of the arrow version and execution times are comparable. We discuss further examples on larger clusters and in a more detail in the next section.

8 Performance results

8.1 Hardware

In the following we describe the benchmarks of our parallel DSL and algorithmic skeletons we conducted. They were run both in a shared and in a distributed memory setting. All benchmarks were done on the 'Glasgow grid', 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

```
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 28: 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 B 13.

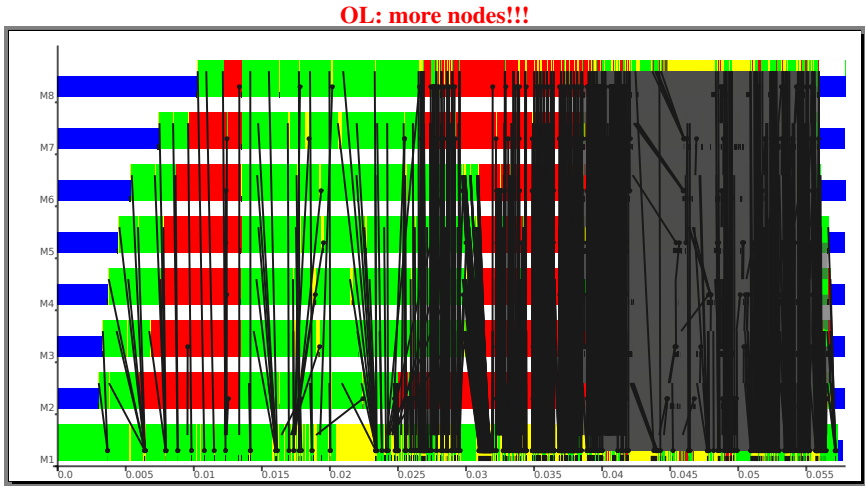

Figure 29: Matrix Multiplication with *torus* (PArrows).

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.

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

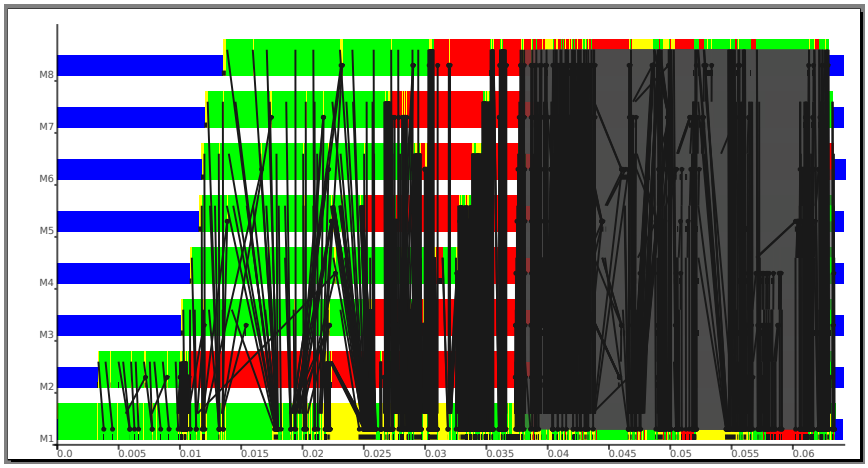

Figure 30: Matrix Multiplication with *torus* (Eden).

Table 1: The benchmarks we use in this paper.

Name	Area	Type	Origin	Citation
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) [11]

8.2 Test programs

We used multiple tests that originated 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 (32–256 here) '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 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 Haskell. We ported it to PArrows.

[11] actual code from: http://community.haskell.org/ simonmar/par-tutorial.pdf and https://github.com/simonmar/parconc-examples

vs.	best cores	best factor	worst cores	worst factor
Eden CP	16	1.05216	8	1.06262
GpH	16	0.97844	2	1.00347
Par Monad	4	0.95352	16	0.96667

(a) Sudoku 1000

vs.	best cores	best factor	worst cores	worst factor
Eden CP	16	1.04264	4	1.06058
GpH	2	0.99001	4	0.99999
Par Monad	4	0.95746	16	0.97572

(b) Sudoku 16000

vs.	best cores	best factor	worst cores	worst factor
Eden CP	16	0.97492	4	1.0061
GpH	32	0.9953	16	1.06316
Par Monad	16	0.98503	32	1.0032

(c) Rabin–Miller test 11213 32

vs.	best cores	best factor	worst cores	worst factor
Eden CP	8	1.00056	16	1.01204
GpH	2	1.00177	32	1.11614
Par Monad	16	0.8437	8	1.00235

(d) Rabin–Miller test 11213 64

vs.	best cores	best factor	worst cores	worst factor
Eden	16	1.00457	160	1.01717

(e) Gentleman (dist) 4096

vs.	best cores	best factor	worst cores	worst factor
Eden	256	0.99442	192	1.00795

(f) Rabin–Miller test (dist) 44497 256

vs.	best cores	best factor	worst cores	worst factor
Eden	256	1.06067	128	1.12277

(g) Jacobi sum test (dist) 4253

Table 2: Best & worst Benchmark results

8.3 What parallel Haskells run where

The *Par* monad and GpH Haskell – in its multicore version (Marlow *et al.*, 2009) – can be executed on a 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 operates in the "nothing shared" setting, but is adapted better to multicore machines. We label this version of Eden in the plots as "Eden CP".

8.4 Effect of hyper-threading

The PArrows version of Rabin–Miller test on a single node of the Glasgow grid showed almost linear speedup (Fig. 31). The speedup of 64-task PArrows/Eden at 16 real cores version was 13.65, the efficiency was 85.3%. However, if we increase the number of requested cores to be 32 – i.e. if we use hyper-threading on 16 real cores – the speedup does not increase that well. It is merely 15.99 for 32 tasks with PArrows/Eden. It is worse for other backends. As for 64 tasks, we obtain the speedup of 16.12 with PArrows/Eden at 32 hyper-threaded cores and only 13.55 with PArrows/GpH. Efficiency is 50.4% and 42.3%, respectively. The Eden version used here was Eden CP, the 'share nothing' SMP build.

In the distributed memory setting the same effect ensues. We obtain plummeting speedup of 124.31 at 512 hyper-threaded cores, whereas it was 213.172 for 256 real cores. Apparently, hyper-threading in the Glasgow grid fails to execute two parallel Haskell processes with full-fledged parallelism. For this reason, we did not regard hyper-threaded cores in our speedup plots in Figs. 31–35.

8.5 Benchmark results

The difference between, say, PArrows with *Par* Monad backend and a genuine *Par* Monad benchmark is very small. To give an example, it is $0.4s$ in favour of PArrows for 16 cores ($10.8s$ vs. $11.2s$) and $-0.8s$ in favour of the *Par* monad for 8 cores ($16.1s$ vs. $16.9s$) for the Sudoku benchmark in the shared memory setting. It is almost invisible in speedup and (non shown) run time plots. We thus show only the results for the PArrows-enabled versions in the backend-comparison.

To show that PArrows induce very small overhead even in a distributed context (and in general as well), we compare the original Eden versions of the benchmark to its PArrows-enabled counterpart in the Rabin–Miller test, Gentleman and Jacobi sum test benchmarks. We plot execution time differences between measurements for PArrows and the corresponding backend in a separate plot (Figs. 32–34). As an example, the differences range in about $0.5s$ for the execution time of $46s$ on 256 cores for distributed Rabin–Miller test with PArrows and Eden. For these comparisons, the plots show absolute time differences that are not relative w.r.t. the total execution time. Furthermore, the error bars ends were computed from point-wise maximum of both standard deviations from both measurements for PArrows and non-PArrows versions. These are the values provided by the *bench* package that we used for bench-marking. We call a difference between two versions significant when the border of the error bar of absolute time difference is above or below zero. In other words: the time difference is significant if it is outside of the measurement error.

8.5.1 Rabin–Miller test

The multicore version of our parallel Rabin–Miller test benchmark is depicted in Figure 31. We executed the test with 32 and 64 tasks. The plot shows the PArrows-enabled versions with corresponding backends. The performance of PArrows/Eden CP in shared memory is slightly better than for SMP variants such as PArrows/GpH and PArrows/*Par* Monad but most of the time the performance is still comparable with the GpH backend performing slightly worse than the other two in terms of speedup.

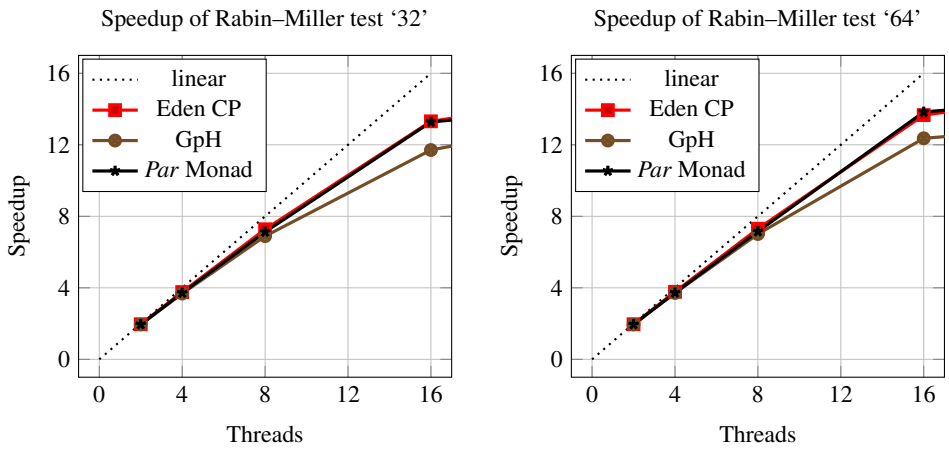

Figure 31: Relative speedup of Rabin–Miller test on a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores. Input was $2^{11213} - 1$, we used 32 (left) or 64 (right) tasks. The closer to linear speedup the better.

Comparing the PArrows version of the Rabin–Miller test with the original from Eden with the MPI backend in a distributed memory setting, we see an almost linear speedup of Rabin–Miller test with 256 tasks and input $2^{44497} - 1$ in both versions. The sequential run time was computed as the mean of three consecutive executions on a single core—the single run took two hours 43 minutes. The difference between PArrows/Eden and Eden almost always lies inside the error bar of the measurement.

8.5.2 Jacobi sum test

Continuing, the results of the Jacobi sum test (MPI only) in Fig. 33 are as follows: The program does not seem to scale well beyond 64 threads with input $2^{3217} - 1$. We once again compare the Eden version with the PArrrows version in Fig. 33. We see similar behaviour to the MPI version of the Rabin–Miller test: The difference once again almost always remains within the bounds of the error bar of the measurement.

Because of the bad scaling behaviour for $2^{3217} - 1$ beyond 64 threads, we also ran tests with input $2^{4253} - 1$. Because of the long running time, we could do this only for the 128 and 256 threads. Therefore we do not show these results in Fig. 33 as we cannot properly compute a speedup without results for 1 thread. Nonetheless, for 128 threads, the benchmark took a mean of 9192.9s, while with 256 threads, it only took a mean of 1649.1s. This means that the 128 thread version ran more than 5.5 times slower than the 256 one, which suggests an IO limitation for this big input that is somewhat mitigated by adding more cores. However it still proves that PArrows continue to scale, even if not perfectly for this test program. Comparing PArrows with Eden, for the larger input of $2^{4253} - 1$, we see slightly bigger differences between Eden and PArrows: The differences for 128 and 256

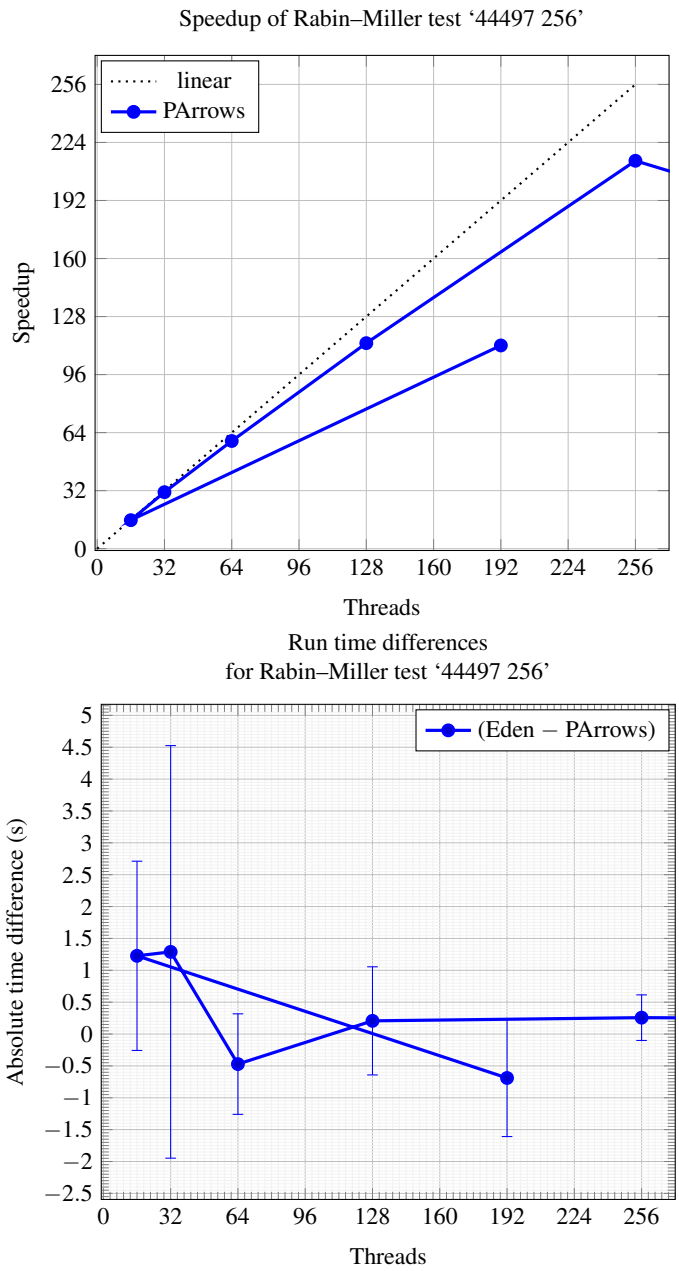

Figure 32: Parallel performance of the Rabin–Miller test on the Glasgow grid consisting of 256 cores. Input was $2^{44497} - 1$, we used 256 tasks. The top plot shows absolute speedup in a distributed memory setting. The closer to linear speedup the better. Time (and hence speedup) measurements for PArrows with Eden backend and Eden almost coincide. Hence, bottom plot shows absolute time differences for this benchmark. The higher the value, the better for PArrows**OL: CHECKME**.

coresare $-1005.18s$ and $-94.33s$ in favour of Eden, respectively. This maximum of 12.27% slower PArrows runtime, however, was still in the (quite big) error bar of our measurements.

8.5.3 Gentleman

Next is the Gentleman benchmark. The results of the comparison of vanilla Eden to our PArrows-based version can be found in Fig. 34. We see that the benchmark scales quite well with more cores until 64 cores. For $>= 96$ cores, we still have considerable speedup, but with less slope. We also prove that the difference between the Eden and PArrows version are only marginal with PArrows only being a maximum of 1.7% slower – for 160 cores – when outside of the error bar.

8.5.4 Sudoku

As the last benchmark in this paper we present the Sudoku in Fig. 35 running in a shared memory setting. Here we see all three SM backends performing similarly again like in the Rabin–Miller test SM benchmarks in Figs. 35 and 36. However, we notice that the GpH backend seems to choke on a bigger input (Fig. 36). This is due to the benchmark only using *parMap* instead of a chunking variant – however we did not change that for simplicity's sake. This issue is reflected by debug output which shows that of 16000 sparks being created (one for each Sudoku) only 8365 were converted (executed) with the rest (7635) overflowing the runtime spark pool. Another remarkable finding is that the Eden backend seems to lag behind for $\leqslant$16 threads, but manages to pull ahead noticeably with all 32 threads of the system in use.

8.6 Byline

In the shared memory setting we naturally find that the backends perform differently. Furthermore, in our distributed memory tests on the full grid the difference between PArrows and Eden was almost always below the error margin if PArrows performed worse. Even the biggest difference of 12.27% in the Jacobi sum test for input $2^{4253} - 1$ and 128 cores remained in line with these findings. The only exception to this was the Gentleman with only a 1.7% difference in favour of Eden which is only a marginal slowdown in our opinion.

9 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are first to represent *parallel* computation with Arrows. Arrows turn out to be a useful tool for composing parallel programs. We do not have to introduce new monadic types that wrap the computation. Instead, we use Arrows in the same manner one uses sequential pure functions. This work features multiple parallel backends: the already available parallel Haskell flavours. Parallel Arrows, as presented here, feature an implementation of the *ArrowParallel* type class for GpH Haskell, *Par* Monad, and Eden. With our approach parallel programs can be ported across these flavours with little to no effort. It is quite straightforward to add further backends. Performance-wise, Parallel Arrows

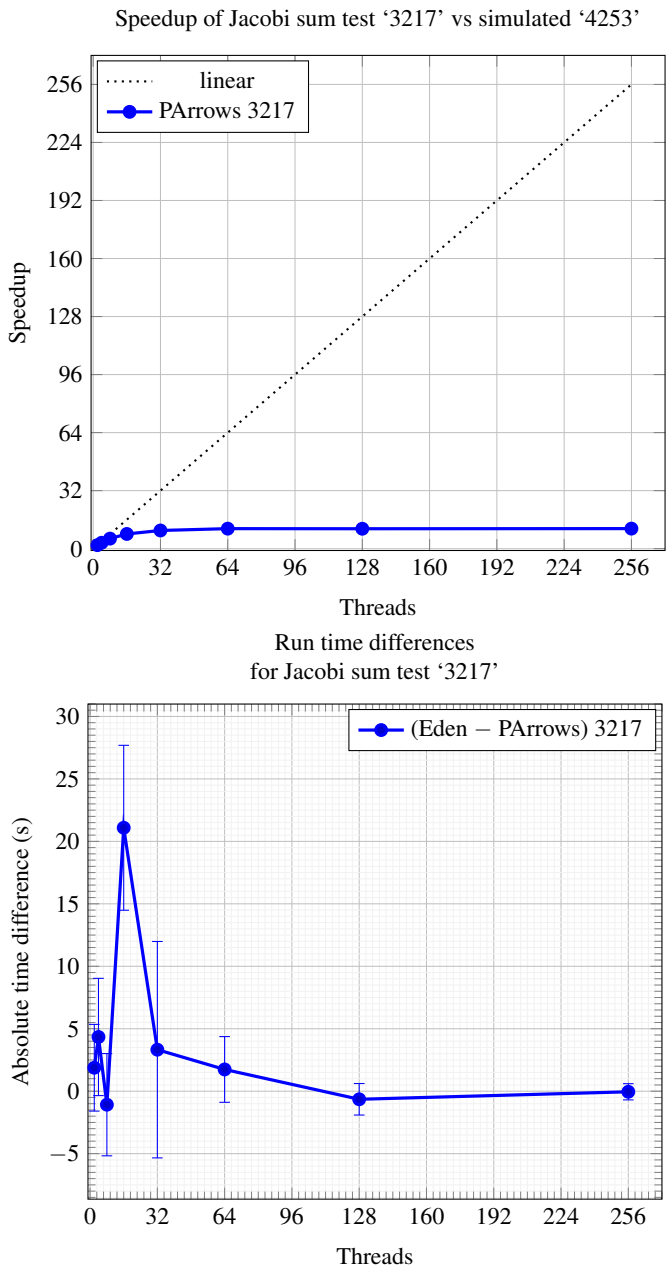

Figure 33: Parallel performance of the Jacobi sum test on the Glasgow grid consisting of 256 cores. Input was $2^{3217} - 1$, we used 256 tasks. The top plot shows relative speedup in a distributed memory setting compared to a simulated speedup for input $2^{4253} - 1$. The closer to linear speedup the better. Time (and hence speedup) measurements for PArrows with Eden backend and Eden almost coincide. Hence, bottom plot shows absolute time differences for this benchmark. The higher the value, the better for PArrows**OL: CHECKME**.

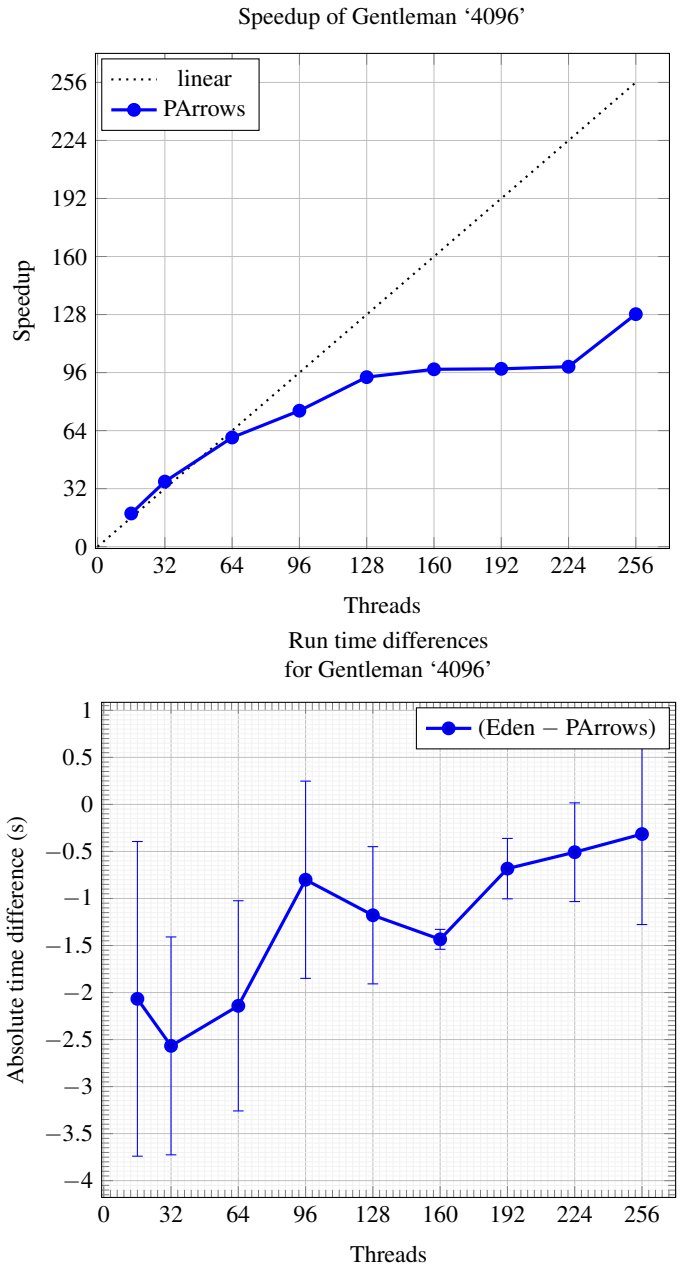

Figure 34: Parallel performance of Gentleman on the Glasgow grid consisting of 256 cores. Input was a matrix size of 4096. The top plot shows absolute speedup in a distributed memory setting. The closer to linear speedup the better. Time (and hence speedup) measurements for PArrows with Eden backend and Eden almost coincide. Hence, bottom plot shows absolute time differences for this benchmark. The higher the value, the better for PArrows**OL: CHECKME**.

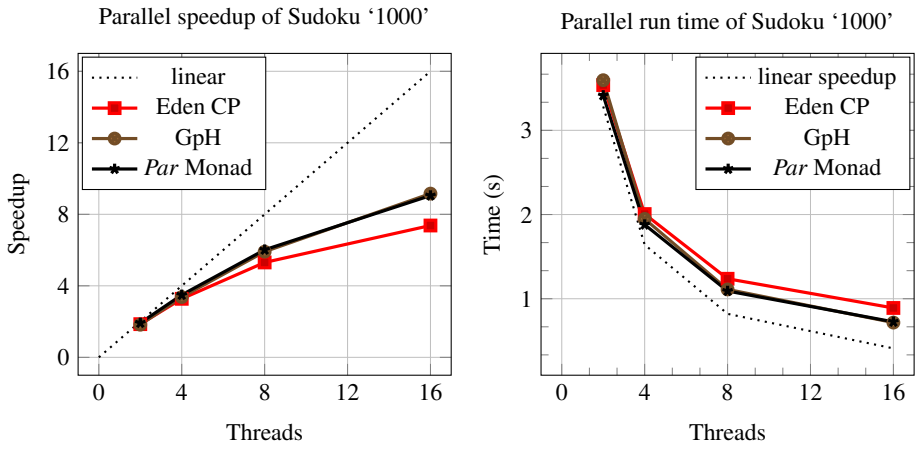

Figure 35: Absolute speedup of Sudoku on a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware and the *parMap* version from the *Par* Monad examples. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores and 32 threads. Input was a file of 1000 Sudokus. The closer to linear speedup the better.

are on par with existing parallel Haskells, as they only introduce minor overhead in some of our benchmarks. The benefit is, however, the greatly increased portability of parallel programs.

9.1 Future Work

Our PArrows DSL can be expanded to further parallel Haskells. More specifically we target HdpH (Maier *et al.*, 2014) for this future extension. HdpH is a modern distributed Haskell that would benefit from our Arrows notation. Further Future-aware versions of Arrow combinators can be defined. Existing combinators could also be improved. Arrow-based notation might enable further compiler optimizations.

More experiences with seamless porting of parallel PArrows-based programs across the backends are welcome. Of course, we are working ourselves on expanding both our skeleton library and the number of skeleton-based parallel programs that use our DSL to be portable across flavours of parallel Haskells. It would also be interesting to see a hybrid of PArrows and Accelerate (McDonell *et al.*, 2015). Ports of our approach to other languages like Frege or Java directly are in an early development stage.

References

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

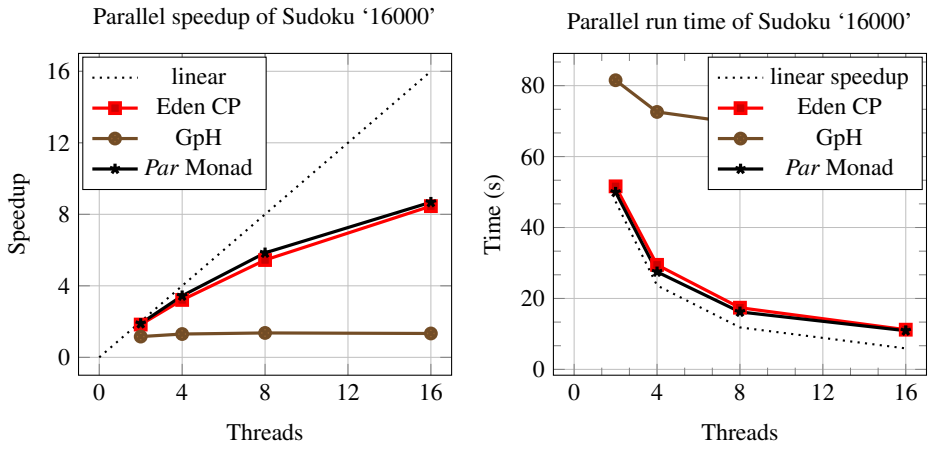

Figure 36: Absolute speedup of Sudoku on a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware and the *parMap* version from the *Par* Monad examples. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores and 32 threads. Input was a file of 16000 Sudokus. The closer to linear speedup the better. The GpH version shows signs of choking with too many sparks being created.

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.

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, 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**.

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.

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.

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, M., Pasqualetti, F., & Pelagatti, S. (1997). Skeletons for Data Parallelism in P^3L. *Pages 619–628 of:* Lengauer, C., Griebl, M., & Gorlatch, S. (eds), *Euro-Par'97*. LNCS 1300. Springer-Verlag.

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.

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

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.

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, 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.

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

Hughes, John. (2005b). *Programming with arrows*. Berlin, Heidelberg: Springer Berlin Heidelberg. 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*.

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.

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.

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.

Nikhil, R., & Arvind, L. A. (2001). *Implicit parallel programming in pH*. Morgan Kaufmann.

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.

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

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.

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

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.

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 $***$:

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

Next, we also define *map*, *foldl* and *zipWith* on Arrows. The *mapArr* combinator (Fig. A 1) lifts any arrow *arr a b* to an arrow *arr* $[a]\ [b]$ (Hughes, 2005b). Similarly, we can also define *foldlArr* (Fig. A 2) that lifts any arrow *arr* $(b,a)\ b$ with a neutral element *b* to *arr* $[a]\ b$.

Finally, with the help of *mapArr* (Fig. A 1), we can define *zipWithArr* (Fig. A 3) that lifts any arrow *arr* $(a,b)\ c$ to an arrow *arr* $([a],[b])\ [c]$.

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

```
mapArr :: ArrowChoice arr ⇒ arr a b → arr [a] [b]
mapArr f =
  arr listcase >>>
  arr (const []) ||| (f *** mapArr f >>> arr (uncurry (:)))
listcase [] = Left ()
listcase (x:xs) = Right (x,xs)
```

Figure A 1: The definition of *map* over Arrows and the *listcase* helper function.

```
foldlArr :: (ArrowChoice arr, ArrowApply arr) ⇒ arr (b,a) b → b → arr [a] b
foldlArr f b =
  arr listcase >>>
  arr (const b) |||
    (first (arr (λa → (b,a)) >>> f >>> arr (foldlArr f)) >>> app)
```

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

arr (*Either a b*) *c* which pipes all *Left a*'s to the first arrow and all *Right b*'s to the second arrow:

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

With the zipWithArr combinator we can also write a combinator *listApp*, that lifts a list of arrows [*arr a b*] to an arrow *arr* [*a*] [*b*].

$$listApp :: (ArrowChoice\ arr, ArrowApply\ arr) \Rightarrow [arr\ a\ b] \to arr\ [a]\ [b]$$
$$listApp\ fs = (arr\ \$\ \lambda as \to (fs, as)) >>> zipWithArr\ app$$

Note that this additionally makes use of the *ArrowApply* type class that allows us to evaluate arrows with $app :: arr\ (arr\ a\ b, a)\ c$.

B Omitted Function Definitions

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

To begin with, we give the actual implementation of the non-configurable instance of the *ArrowParallel* instance:

Next, we warp Eden's build-in Futures in PArrows as in Figure B 6, where *rd* is the accessor function for the *RD* wrapped inside *RemoteData*. Furthermore, in order for these *Future* types to fit with the *ArrowParallel* instances we gave earlier, we have to give the necessary *NFData* and *Trans* instances, the latter are only needed in Eden. The *Trans* instance does not have any functions declared as the default implementation suffices here. Furthermore, because *MVar* already ships with a *NFData* instance, we only have to supply a simple delegating *NFData* instance for our *RemoteData* type, where *rd* simply unwraps *RD*. The *Trans* instance does not have any functions declared as the default implementation suffices:

```
zipWithArr :: ArrowChoice arr ⇒ arr (a,b) c → arr ([a],[b]) [c]
zipWithArr f = (arr $ λ(as,bs) → zipWith (,) as bs) >>> mapArr f
```

Figure A 3: *zipWith* over arrows.

```
instance (NFData b,ArrowApply arr,ArrowChoice arr) ⇒ ArrowParallel arr a b () where
  parEvalN _fs = parEvalN (hack fs) fs
    where
      hack :: (NFData b) ⇒ [arr a b] → Conf b
      hack _ = Conf rdeepseq
```

Figure B 1: The actual default implementation of GpH's *ArrowParallel*.

```
instance NFData (RemoteData a) where
  rnf = rnf ∘ rd
instance Trans (RemoteData a)
```

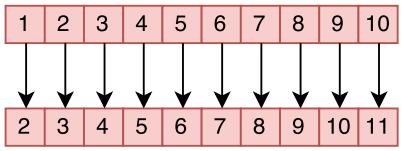

Figure B 2: Schematic depiction of *parMap*.

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

Figure B 3: Definition of parMap.

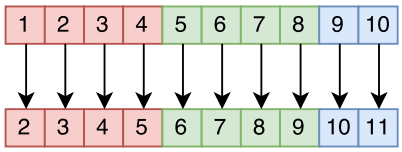

Figure B 4: Schematic depiction of *parMapStream*.

```
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 B 5: Definition of *parMapStream*.

Figures B 2–B 5 show the definitions and a visualizations 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 B 7. Similarly, Figure B 8 contains the definition of arrow versions of Eden's *lazy*

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

```
data RemoteData a = RD {rd :: RD a}
instance (Trans a) ⇒ Future RemoteData a where
  put = arr (λa → RD {rd = release a})
  get = arr rd >>> arr fetch
```

Figure B 6: *RD*-based *RemoteData* version of *Future* for the Eden backend.

```
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 B 7: Definitions of *shuffle*, *unshuffle*, *takeEach*.

```
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 B 8: Definitions of *lazy* and *rightRotate*.

The *parEval2* skeleton is defined in Figure B 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 above. 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. B 13 contains the omitted definitions of *prMMTr* (which calculates 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.

```
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 B 9: Definitions of *lazyzip3*, *uncurry3* and *threetotwo*.

```
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 B 10: Definition of *farmChunk*.

C 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
```

```
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 B 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 B 12: Definition of *parEval2*.

```
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 B 13: Definition of *prMMTr*, *splitMatrix* and *matAdd*.