1

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 utilise the concept of Futures to wrap existing direct communication concepts in backends.

To show that PArrows have similar expressive power as 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 across 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, very low-penalty programming interface that allows to switch the parallel backends at will. The induced penalty was max. 8.3 % in our measurements **MB: , and typically under X %**. Further backends based on HdpH or a Frege implementation (on the Java Virtual Machine) are viable, too.

Contributions We propose an Arrow-based encoding for parallelism based on a new Arrow combinator *parEvalN* :: [*arr a b*] $\rightarrow$ *arr* [*a*] [*b*] in Section 4, which converts a list of Arrows into a new parallel Arrow. A parallel Arrow is still an Arrow, hence the resulting parallel Arrow can still be used in the same way as a potential sequential version.

In this paper evaluate the expressive power of our Arrow formalism in the context of parallel programming.

- We introduce a parallel evaluation formalism using Arrows. One big advantage is that we do not have to introduce any new types (Sec. 4) in contrast to Monad solutions such as the *Par* Monad. This behaviour encourages better composability.

- We utilitze multiple backends – currently a GpH, a *Par* Monad, and Eden. We do not reimplement all the parallel internals, as we host this functionality in the *ArrowParallel* type class, which abstracts all parallel implementation logic. The backends can easily be swapped, so we are not bound to any specific one.
  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. (Sec. 4)
- We extend our PArrows formalism with *Future*s. Our goal here is 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 it allows for inter-node communication without blocking the master-node. (Sec. 5)
- It is possible to define algorithmic skeletons with PArrows (Sec. 6). All our benchmarks were skeleton-based.
- Finally, we practically demonstrate that Arrow parallelism is a viable alternative to existing approaches. It introduces only low performance overhead (Sec. 7).

PArrows are open source and available from `https://github.com/s4ke/Parrows`.
MB: Überleitung auf Related Work hier machen
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) as a less restrictive alternative to Monads, 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.

3 Background

3.1 Arrows

Arrows were introduced by Hughes (2000) as a general interface for computation and a less restrictive generalisation of Monads. Hughes motivates the broader interface of Arrows with the example of a parser with added static meta-information that can not satisfy the monadic bind operator $(>>=) :: m\ a → (a → m\ b) → m\ b$ (with m being a Monad). In the example a parser of the type *Parser s a* with static meta information s and result a is shown to not be able to use the static information s without applying the monadic function $a → m\ b$. With Arrows this is possible.

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 $(→)$ and the Kleisli type (Fig. 1), which wraps monadic functions, e.g. $a → m\ b$.

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 ⇒ arr a b → arr c d → 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:
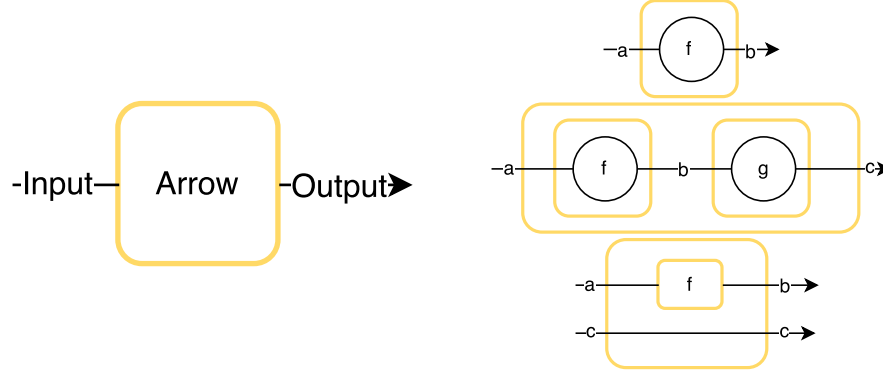
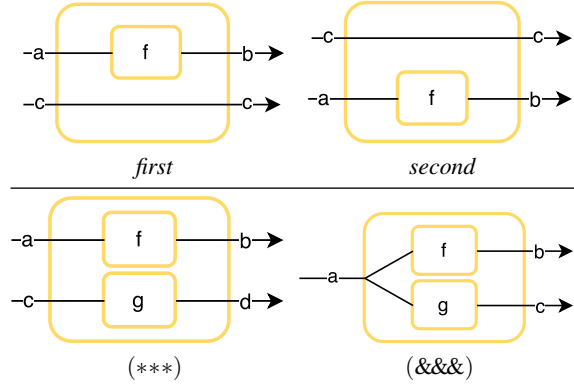Figure 2: Schematic depiction of an Arrow (left) and its basic combinators *arr*, $\ggg$ and *first* (right).

Figure 3: Visual depiction of syntactic sugar for Arrows.

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

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

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

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

In this paper we will show that we can express parallel computations with Arrows and do not require Monads to do so. We also do not restrict the compatible Arrows to ones which have *ArrowApply* instances but instead only require instances for *ArrowChoice* and

1	2	3	4	5	6	7	8	9	10

0	4	9	3	0	2	3	5	8	11

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

ArrowLoop. Because of this, we have a truly more general interface as compared to a monadic one.

Also note that, while we could have based our DSL on Profunctors as well, we chose Arrows **MB: cite missing?**. for this paper since they they allow for a more direct way of thinking about parallelism than general Profunctors because of their composability. However, they are a promising candidate for future improvements of our DSL. Note that some Profunctors, especially ones supporting a composition operation among other things, can already be adapted to our interface as shown in Appendix B.

3.2 Short introduction to parallel Haskells

In its purest form, parallel computation (on functions) can be looked at as the execution of some functions $a \to b$ in parallel or $parEvalN :: [a \to b] \to [a] \to [b]$, as also Figure 4 symbolically shows.

In this section, we will implement this non-Arrow version which will later be adapted for usage in our Arrow-based parallel Haskell.

There exist several parallel Haskells already. Among the most important are probably GpH (based on *par* & *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*.
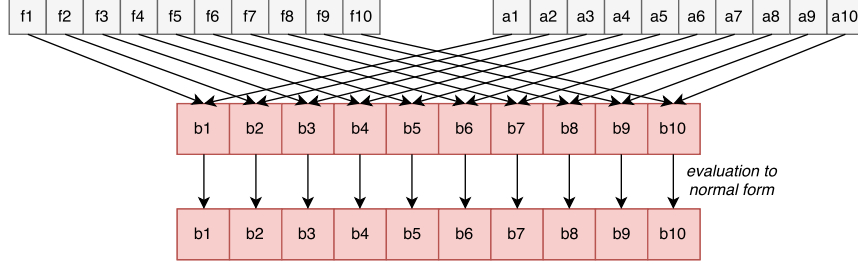
Figure 5: Data flow of the GpH *parEvalN* version.

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* :: $a \rightarrow Strategy\ a \rightarrow a$, which is exactly what is required for an implementation of *parEvalN*.

$$parEvalN :: (NFData\ b) \Rightarrow [a \rightarrow b] \rightarrow [a] \rightarrow [b]$$
$$parEvalN\ fs\ as = \textbf{let}\ bs = zipWith\ (\$)\ fs\ as$$
$$\textbf{in}\ bs\ `using`\ parList\ rdeepseq$$

In the above definition of *parEvalN* we just apply the list of functions $[a \rightarrow b]$ 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* :: $a \rightarrow Strategy\ a \rightarrow a$ operator. We construct this strategy with *parList* :: $Strategy\ a \rightarrow Strategy\ [a]$ 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.

3.2.2 Par Monad

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

$$parEvalN :: (NFData\ b) \Rightarrow [a \rightarrow b] \rightarrow [a] \rightarrow [b]$$
$$parEvalN\ fs\ as = runPar\ \$$$
$$(sequenceA\ (map\ (return \circ spawn)\ (zipWith\ (\$)\ fs\ as))) \ggg mapM\ get$$

The *Par* Monad version of our parallel evaluation function *parEvalN* is defined by zipping the list of $[a \rightarrow b]$ with the list of inputs $[a]$ with the application operator $ just like with GpH. Then, we map over this not yet evaluated lazy list of results $[b]$ with *spawn* :: $NFData\ a \Rightarrow Par\ a \rightarrow Par\ (IVar\ a)$ to transform them to a list of not yet evaluated

---

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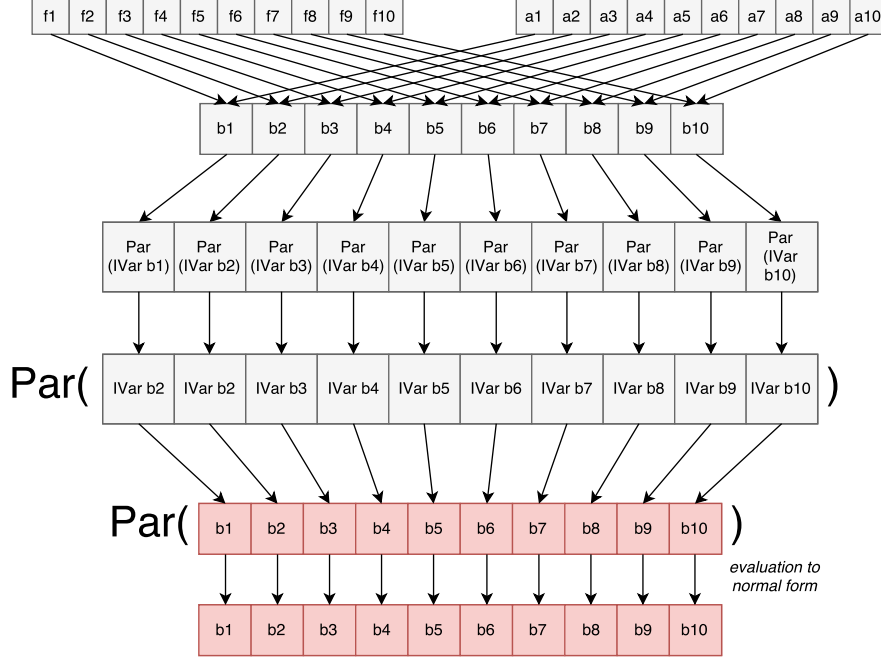
Figure 6: Data flow of the *Par* Monad *parEvalN* version.

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*]. While we used *spawn* in the definition above, a head-strict variant can easily be defined by replacing *spawn* with *spawn_* :: *Par a → Par* (*IVar a*). 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*) ⇒ [*a → b*] → [*a*] → [*b*] function that allows us to define *parEvalN* directly:

---

[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/`.

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

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. 12), 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.

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 $a \rightarrow b$ 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 *evalN* from Appendix A, but does parallel instead of serial evaluation.

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

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

class *Arrow arr* ⇒ *ArrowParallel arr a b* **where**
   $parEvalN :: [arr\ a\ b] \rightarrow arr\ [a]\ [b]$

---

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

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

Figure 7: The *ArrowParallel* instance for the GpH backend.

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*.

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

4.2 ArrowParallel instances

4.2.1 Glasgow parallel Haskell

The GpH implementation of *ArrowParallel* is implemented in a straightforward manner in Fig. 7, but a bit different compared to the variant from Section 3.2.1, by using *evalN* :: $[arr\ a\ b] → arr\ [a]\ [b]$ (definition in Appendix A, think *zipWith* ($) on Arrows) combined with *withStrategy* :: $Strategy\ a → a → a$ from GpH, where *withStrategy* is the same as *using* :: $a → Strategy\ a → a$, but with flipped parameters. Our *Conf a* datatype simply wraps a *Strategy a*, but could be extended in future versions of our DSL.

4.2.2 Par Monad

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

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

Now we can once again define our *ArrowParallel* instance as follows: First, we convert our Arrows $[arr\ a\ b]$ with *evalN* (*map* (>>>*arr strat*) *fs*) into an Arrow $arr\ [a]\ [(Par\ (IVar\ b))]$ that yields composable computations in the *Par* monad, where *evalN* is once again of the type *evalN* :: $[arr\ a\ b] → arr\ [a]\ [b]$. By combining the result of this Arrow with *arr sequenceA*, we get an Arrow $arr\ [a]\ (Par\ [IVar\ b])$. Then, in order to fetch the results of the different threads, we map over the *IVar*s inside the *Par* Monad with *arr* (>>=*mapM get*)

- our intermediary Arrow is of type *arr* [*a*] (*Par* [*b*]). Finally, we execute the comptuation *Par* [*b*] by composing with *arr runPar* and get the final Arrow *arr* [*a*] [*b*].

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

Figure 8: ArrowParallel instance for the Par Monad backend.

4.2.3 Eden

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

> **class** (*Arrow arr*) ⇒ *ArrowUnwrap arr* **where**
>   *arr a b* → (*a* → *b*)

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

> **instance** (*Trans a*, *Trans b*) ⇒ *ArrowParallel* (→) *a b Conf* **where**
>   *parEvalN* _ = *spawnF*

and the Kleisli type:

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

where *Conf* is simply defined as **data** *Conf* = *Nil* since Eden does not have a configurable *spawnF* variant.

4.2.4 Default configuration instances

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

> **instance** (*NFData b*, *ArrowChoice arr*, *ArrowParallel arr a b* (*Conf b*)) ⇒
>   *ArrowParallel arr a b* () **where**
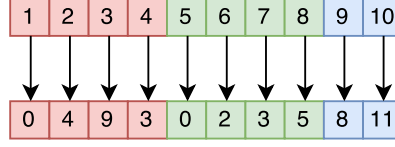
Figure 9: Schematic depiction of *parEvalNLazy*.

$$parEvalN \_ fs = parEvalN \, (defaultConf \, fs) \, fs$$

$$defaultConf :: (NFData \, b) \Rightarrow [arr \, a \, b] \to Conf \, b$$
$$defaultConf \, fs = stratToConf \, fs \, rdeepseq$$

$$stratToConf :: [arr \, a \, b] \to Strategy \, b \to Conf \, b$$
$$stratToConf \_ strat = Conf \, strat$$

The other backends have similarly structured implementations which we do not discuss here for brevity reasons. Note that by restricting the original implementations of our backends to their specific *Conf* type, we also get interoperability between backends for free. We can parallelize one part of a program using one backend, and parallelize the next with another one. We can however only have one instance of *ArrowParallel arr a b* () present at a time.

Up until now we discussed Arrow operations more in detail, but in the following sections we focus more on the dataflow between the arrows, now that we have seen that Arrows are capable of expressing parallelism. We do however explain new concepts with more details if required for easier understanding.

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. 9, 10) 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*.

4.3.2 Heterogeneous tasks

We have only talked about the parallelisation of arrows of the same type until now. But sometimes we want to parallelize heterogeneous types as well. We can implement such a

$$parEvalNLazy :: (ArrowParallel\ arr\ a\ b\ conf, ArrowChoice\ arr, ArrowApply\ arr) \Rightarrow$$
$$conf \rightarrow ChunkSize \rightarrow [arr\ a\ b] \rightarrow (arr\ [a]\ [b])$$
$$parEvalNLazy\ conf\ chunkSize\ fs =$$
$$arr\ (chunksOf\ chunkSize) >\!\!>\!\!>$$
$$evalN\ fchunks >\!\!>\!\!>$$
$$arr\ concat$$
where
$$fchunks = map\ (parEvalN\ conf)\ (chunksOf\ chunkSize\ fs)$$

Figure 10: Definition of *parEvalNLazy*.
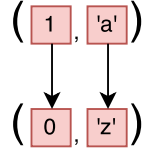
Figure 11: Schematic depiction of *parEval2*.

parEval2 combinator (Figs. 11, C 12) which combines two arrows *arr a b* and *arr c d* into a new parallel arrow *arr* (a, c) (b, d) quite easily with the help of the *ArrowChoice* type class. 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:

$$someCombinator :: (ArrowChoice\ arr,$$
$$ArrowParallel\ arr\ a\ b\ (),$$
$$ArrowParallel\ arr\ b\ c\ ()) \Rightarrow$$
$$[arr\ a\ b] \rightarrow [arr\ b\ c] \rightarrow arr\ [a]\ [c]$$
$$someCombinator\ fs1\ fs2 =$$
$$parEvalN\ ()\ fs1 >\!\!>\!\!>$$
$$rightRotate >\!\!>\!\!>$$
$$parEvalN\ ()\ fs2$$

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

While the 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 12. This can become a serious bottleneck for larger amount of data and number of processes (as e.g. Berthold *et al.*, 2009c, showcases).
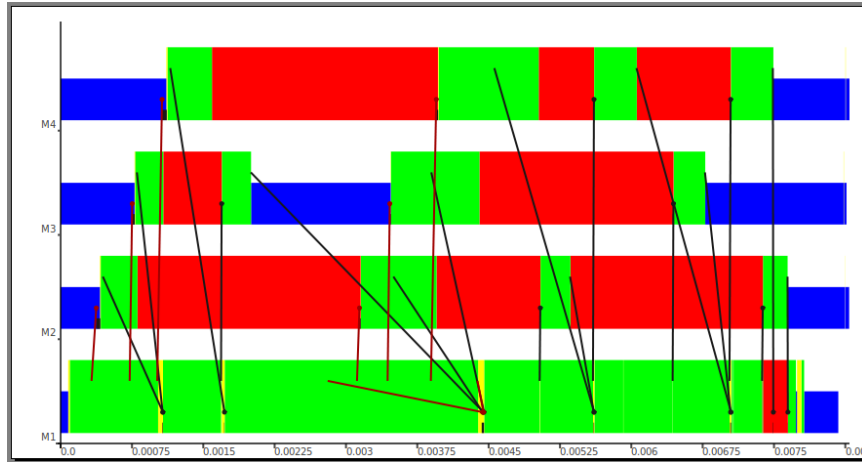

Figure 12: Communication between 4 Eden processes without Futures. All communication goes through the master node. Each bar represents one process. Black lines between processes represent communication. Colors: blue $\hat{=}$ idle, green $\hat{=}$ running, red $\hat{=}$ blocked, yellow $\hat{=}$ suspended.

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 concept. We do this with the *Future* type class (Fig. 13). Note that we require a conf parameter here as well, but only so that Haskells type system allows us to have multiple Future implementations imported at once without breaking any dependencies similar to what we did with the *ArrowParallel* typeclass earlier. Since *RD* is only a type synonym for a communication type

$$\textbf{class } \textit{Future fut a conf} \mid \textit{a conf} \rightarrow \textit{fut } \textbf{where}$$
$$\textit{put} :: (\textit{Arrow arr}) \Rightarrow \textit{conf} \rightarrow \textit{arr a } (\textit{fut a})$$
$$\textit{get} :: (\textit{Arrow arr}) \Rightarrow \textit{conf} \rightarrow \textit{arr } (\textit{fut a}) \textit{ a}$$

Figure 13: Definition of the *Future* type class.

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

For the *Par* Monad and GpH Haskell backends, we can simply use *BasicFuture*s (Fig. C 2), which are just simple wrappers around the actual data with boiler-plate logic so that the type class is satisfied. This is because the concept of a *Future* does not change anything for shared-memory execution as there are no communication problems to fix. Nevertheless, we require a common interface so the parallel Arrows are portable across backends. The implementation can also be found in Section C.

In our communication example we can use this *Future* concept for direct communication between nodes as shown in Fig. 14. Note that with our PArrows DSL we can also define default instances *Future fut a* () for each backend similar to how *ArrowParallel arr a b* () was defined in Section 4. Details can be found in Section C. In a distributed environment,

$$
\begin{aligned}
&someCombinator :: (ArrowChoice\ arr, NFData\ b, NFData\ c, \\
&\quad ArrowParallel\ arr\ a\ (fut\ b)\ (), \\
&\quad ArrowParallel\ arr\ (fut\ b)\ c\ (), \\
&\quad Future\ fut\ b\ ()) \Rightarrow \\
&\quad [arr\ a\ b] \to [arr\ b\ c] \to arr\ [a]\ [c] \\
&someCombinator\ fs1\ fs2 = \\
&\quad parEvalN\ ()\ (map\ (>>>put\ ())\ fs1) >>> \\
&\quad rightRotate >>> \\
&\quad parEvalN\ ()\ (map\ (get\ ()>>>)\ fs2)
\end{aligned}
$$

Figure 14: The mock-up combinator in parallel.

this gives us a communication scheme with messages going through the master node only if it is needed – similar to what is shown in the trace visualization in Fig. 15. One especially elegant aspect of this solution is that we can specify the type of *Future* to be used per backend with full interoperability between code using different backends, without even requring to know about the actual type used for communication. We only specify that there has to be a compatible Future and do not care about any specifics. **OL: Fig. is not really clear. Do Figs with a lot of load? — fft?**

6 Skeletons

Now we have developed Parallel Arrows far enough to define some useful algorithmic skeletons that abstract typical parallel computations. While there are many possible skeletons to implement, we regard in detail here only some *map*-based and topological skeletons to demonstrate the power of PArrows.

6.1 *map*-based Skeletons

We start with *map*-based skeletons. The essential differences between the skeletons presented here are in terms of order of evaluation and work distribution but still provide the same output of a standard *map*.

Parallel *map* and laziness. The *parMap* skeleton (Figs. C 3, C 4) is probably the most common skeleton for parallel programs. We can implement it with *ArrowParallel* by
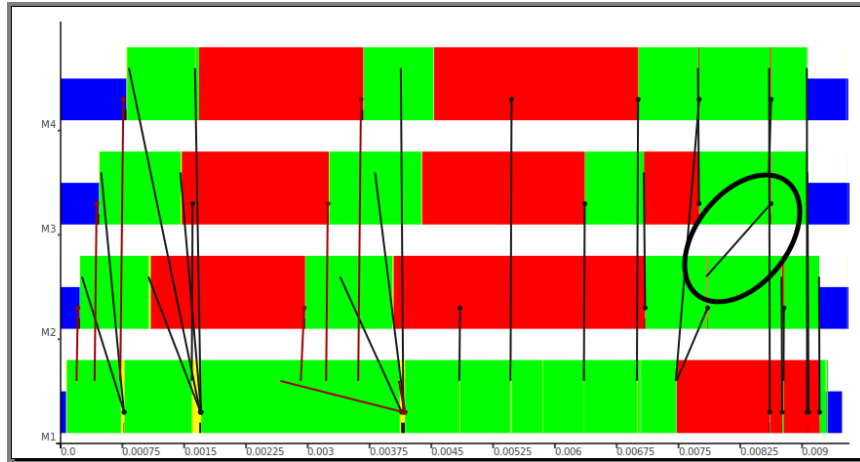
Figure 15: Communication between 4 Eden processes with Futures. Other than in Fig. 12, processes communicate directly (black lines between the bars, one example message is marked in the Figure) instead of always going through the master node (bottom bar).

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. Because of this, it has the same restrictions as *parEvalN* as compared to *parEvalNLazy*. So it makes sense to also have a *parMapStream* (Figs. C 5, C 6) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*. The code of these two skeletons is quite straightforward, we show it in Appendix.

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. 16, 17) that equally distributes the workload over *numCores* workers seems useful. The definitions of the helper functions *unshuffle*, *takeEach*, *shuffle* (Fig. C 7) originate from an Eden skeleton[7].
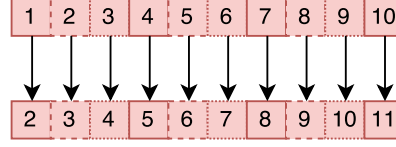
Since a *farm* is basically just *parMap* with a different work distribution, it is, again, 100% strict. So we can define *farmChunk* (Figs. 18, C 10) which uses *parEvalNLazy* instead of *parEvalN*. It is basically the same definition as for *farm*, with *parEvalN* replaced with *parEvalNLazy*.

6.2 Topological Skeletons

Even though many algorithms can be expressed by parallel maps, some problems require more sophisticated skeletons. The Eden library leverages this problem and already comes with more predefined skeletons[8], among them a *pipe*, a *ring*, and a *torus* implementations

---

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

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

M. Braun, O. Lobachev and P. Trinder

Figure 16: Schematic depiction of a *farm*, a statically load-balanced *parMap*.

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

Figure 17: The definition of *farm*.
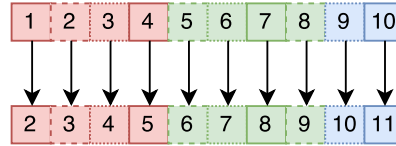
Figure 18: Schematic depiction of *farmChunk*.

(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 we think that this is due to difference of how evaluation is done in these backends when compared to Eden. An investigation of where this difference comes from is beyond the scope of this work, we only provide a workaround for these types of skeletons as such skeletons probably are not of much importance outside of a distributed memory environment. However our workaround enables users of the DSL to test their code within a shared memory setting even though other skeletons would be better suited to be run with them. The idea of the fix is to provide a *ArrowLoopParallel* typeclass that has two functions - *loopParEvalN* and *postLoopParEvalN* where the first is to be used inside an *loop* construct while the latter will be used right outside of the *loop*. This way we can delegate to the actual *parEvalN* in the spot where the backend supports it.

class *ArrowParallel arr a b conf* $\Rightarrow$ *ArrowLoopParallel arr a b conf* **where**
    loopParEvalN :: *conf* $\rightarrow$ [*arr a b*] $\rightarrow$ *arr* [*a*] [*b*]
    postLoopParEvalN :: *conf* $\rightarrow$ [*arr a b*] $\rightarrow$ *arr* [*a*] [*b*]

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

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

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

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

6.2.1 Parallel pipe

The parallel *pipe* skeleton is semantically equivalent to folding over a list [*arr a a*] of arrows with ⋙, but does this in parallel, meaning that the arrows do not have to reside on the same thread/machine. We implement this skeleton using the *ArrowLoop* type class which gives us the *loop* :: *arr* (*a*, *b*) (*c*, *b*) → *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* (λ(*a*, *b*) → (*b*, *a* : *b*)))

which is the same as

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

defines an arrow that takes its input *a* and converts it into an infinite stream [*a*] of it. Using this to our advantage gives us a first draft of a pipe implementation (Fig. 19) 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*.

Sometimes, this *pipe* definition can be a bit inconvenient, especially if we want to pipe arrows of mixed types together, i.e. *arr a b* and *arr b c*. By wrapping these two arrows inside a bigger arrow *arr* (([*a*], [*b*]), [*c*]) (([*a*], [*b*]), [*c*]) suitable for *pipe*, we can define *pipe2* as in Fig. 21.

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 D.

6.2.2 Ring skeleton

Eden comes with a ring skeleton[9] (Fig. 22) implementation that allows the computation of a function [*i*] → [*o*] with a ring of nodes that communicate with each other. Its input is a node function *i* → *r* → (*o*, *r*) in which *r* serves as the intermediary output that gets send to

---

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

$$
\begin{aligned}
&pipeSimple :: (ArrowLoop\ arr, ArrowLoopParallel\ arr\ a\ a\ conf) \Rightarrow \\
&\quad conf \rightarrow [arr\ a\ a] \rightarrow arr\ a\ a \\
&pipeSimple\ conf\ fs = \\
&\quad loop\ (arr\ snd\ \&\&\& \\
&\qquad (arr\ (uncurry\ (:) \ggg lazy) \ggg loopParEvalN\ conf\ fs)) \ggg \\
&\quad arr\ last
\end{aligned}
$$

Figure 19: A first implementation of the *pipe* skeleton expressed with Parallel Arrows. Note that the use of *lazy* (Fig. C 8) is essential as without it programs using this definition would never halt. We need to enforce that the evaluation of the input $[a]$ terminates before passing it into *evalN*.

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. 20.

$$
\begin{aligned}
&pipe :: (ArrowLoop\ arr, ArrowLoopParallel\ arr\ (fut\ a)\ (fut\ a)\ conf, \\
&\quad Future\ fut\ a\ conf) \Rightarrow \\
&\quad conf \rightarrow [arr\ a\ a] \rightarrow arr\ a\ a \\
&pipe\ conf\ fs = unliftFut\ conf\ (pipeSimple\ conf\ (map\ (liftFut\ conf)\ fs)) \\[4pt]
&liftFut :: (Arrow\ arr, Future\ fut\ a\ conf, Future\ fut\ b\ conf) \Rightarrow \\
&\quad conf \rightarrow arr\ a\ b \rightarrow arr\ (fut\ a)\ (fut\ b) \\
&liftFut\ conf\ f = get\ conf \ggg f \ggg put\ conf \\[4pt]
&unliftFut :: (Arrow\ arr, Future\ fut\ a\ conf, Future\ fut\ b\ conf) \Rightarrow \\
&\quad conf \rightarrow arr\ (fut\ a)\ (fut\ b) \rightarrow arr\ a\ b \\
&unliftFut\ conf\ f = put\ conf \ggg f \ggg get\ conf
\end{aligned}
$$

Figure 20: Final definition of the *pipe* skeleton with Futures.

the neighbour of each node. This data is sent over direct communication channels, the so called 'remote data'. We depict it in Appendix, Fig. C 11.

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) $(c,b) \rightarrow arr\ a\ c$. In each iteration we start by rotating the intermediary input from the nodes $[fut\ r]$ with *second* $(rightRotate \ggg lazy)$ (Fig. C 8). Similarly to the *pipe* from Section 6.2.1 (Fig. 19), we have to feed the intermediary input into our *lazy* (Fig. C 8) arrow here, or the evaluation would fail to terminate. The reasoning is explained by Loogen (2012) as a demand problem.

Next, we zip the resulting $([i], [fut\ r])$ to $[(i, fut\ r)]$ with *arr* $(uncurry\ zip)$ so we can feed that into our input arrow *arr* (i,r) (o,r), which we transform into *arr* $(i, fut\ r)$ $(o, fut\ r)$ with *second get* $\ggg f \ggg$ *second put* before converting it into a parallel arrow *arr* $[(i, fut\ r)]$ $[(o, fut\ r)]$ with the help of *repeat* and *loopParEvalN* to get an output 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. 23). To make sure this algorithm has speedup on shared-memory machines as well, we pass the result of this arrow to *postLoopParEvalN conf* (*repeat* (*arr id*)). This combinator can, for example, be used to calculate the shortest paths in a graph using Warshall's algorithm.

$pipe2 :: (ArrowLoop\ arr, ArrowChoice\ arr,$
$\quad ArrowLoopParallel\ arr\ (fut\ (([a],[b]),[c]))\ (fut\ (([a],[b]),[c]))\ conf,$
$\quad Future\ fut\ (([a],[b]),[c])\ conf) \Rightarrow$
$\qquad conf \rightarrow arr\ a\ b \rightarrow arr\ b\ c \rightarrow arr\ a\ c$
$pipe2\ conf\ f\ g =$
$\qquad (arr\ return\ \&\&\&\ arr\ (const\ [\,]))\ \&\&\&\ arr\ (const\ [\,]) >\!\!>\!\!>$
$\quad pipe\ conf\ (replicate\ 2\ (unify\ f\ g)) >\!\!>\!\!>$
$\quad arr\ snd >\!\!>\!\!>$
$\quad arr\ head$
$\quad \textbf{where}$
$\qquad unify :: (ArrowChoice\ arr) \Rightarrow$
$\qquad\quad arr\ a\ b \rightarrow arr\ b\ c \rightarrow arr\ (([a],[b]),[c])\ (([a],[b]),[c])$
$\qquad unify\ f'\ g' =$
$\qquad\quad (mapArr\ f' *\!*\!* mapArr\ g') *\!*\!* arr\ (const\ [\,]) >\!\!>\!\!>$
$\qquad\quad arr\ (\lambda((b,c),a) \rightarrow ((a,b),c))$
$(|\!>\!\!>\!\!>\!|) :: (ArrowLoop\ arr, ArrowChoice\ arr,$
$\quad ArrowLoopParallel\ arr\ (fut\ (([a],[b]),[c]))\ (fut\ (([a],[b]),[c]))\ (),$
$\quad Future\ fut\ (([a],[b]),[c])\ ()) \Rightarrow$
$\quad arr\ a\ b \rightarrow arr\ b\ c \rightarrow arr\ a\ c$
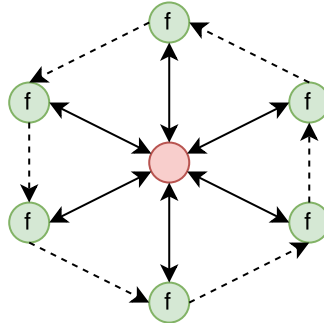$(|\!>\!\!>\!\!>\!|) = pipe2\ ()$

Figure 21: Definition of *pipe2* and $(|\!>\!\!>\!\!>\!|)$, a parallel $>\!\!>\!\!>$.

Figure 22: Schematic depiction of the ring skeleton.

6.2.3 Torus skeleton

If we take the concept of a *ring* from Section 6.2.2 one dimension further, we obtain a *torus* skeleton (Fig. 24, 25). 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 start by rotating the input (Fig. C 8), but this time not only in one direction, but in two. This means that the intermediary input from the neighbour nodes has to be stored in a tuple $([[fut\ a]],[[fut\ b]])$ in the second argument (loop only allows for two

---

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

Figure 23: Definition of the *ring* skeleton.
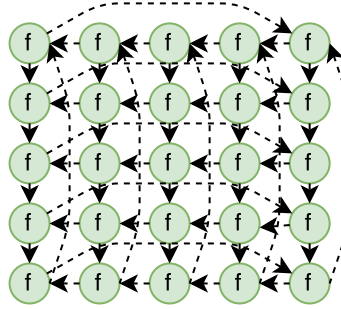
Figure 24: Schematic depiction of the *torus* skeleton.

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*))

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

   arr (*uncurry3 zipWith3 lazyzip3*)

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

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

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

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

ptorus :: (*Arrow arr*, *Future fut a*, *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 25: Definition of the *torus* skeleton. The definitions of *lazyzip3*, *uncurry3* and *threetotwo* have been omitted and can be found in Fig. C 9

.

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 26: Adapted matrix multiplication in Eden using a the *torus* skeleton. *prMM_torus* is the parallel matrix multiplication. *mult* is the function performed by each worker. *prMMTr* calculates AB^T and is used for the (sequential) calculation in the chunks. *splitMatrix* splits the Matrix into chunks. *matAdd* calculates $A + B$. Omitted definitions can be found in C 13.

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. 26. If we compare the trace from a call using our arrow definition of the torus (Fig. 27) with the Eden version (Fig. 28) we can see that the behaviour of the arrow version and execution times are comparable. We discuss further benchmarks on larger clusters and in a more detail in the next section.
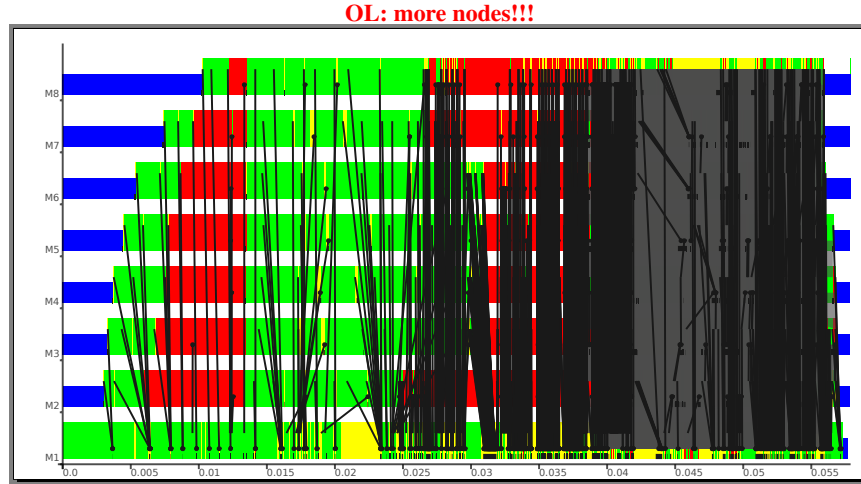
Figure 27: Matrix Multiplication with *torus* (PArrows).
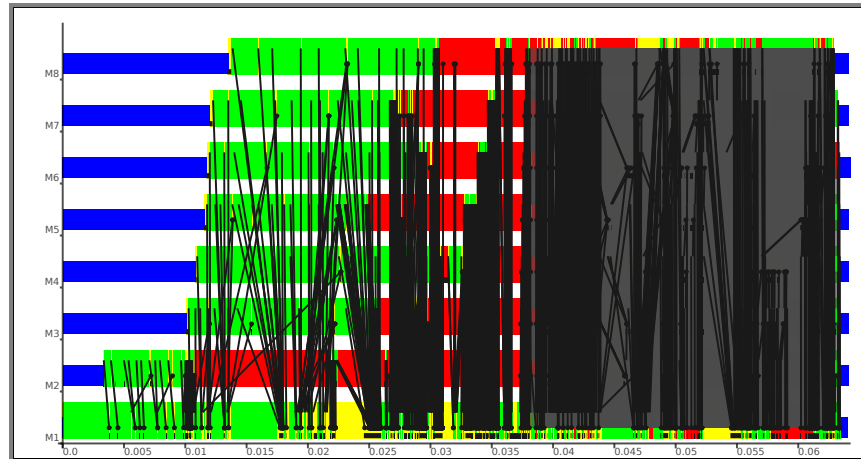
Figure 28: Matrix Multiplication with *torus* (Eden).

7 Performance results

In the following section, we describe the benchmarks of our parallel DSL and algorithmic
skeletons conducted. We start by explaining the hardware and software stack and also
elaborate on which benchmarks programs were used and also which parallel Haskell were
used in which setting. Before we go into detail on the benchmarks we also address the
issue of poor performance gains when using hyper-threading and our reasoning behind not
including hyper-threaded cores in our benchmarks. Finally, we show that PArrows hold up
in terms of performance when compared to the original parallel Haskells used as backends
in this paper, starting with the shared-memory variants (GpH, *Par* Monad and Eden CP)
and concluding with Eden as a distributed backend.

Table 1: The benchmarks we use in this paper.

Name	Area	Type	Origin	Source
Rabin–Miller test	Mathematics	*parMap + reduce*	Eden	Lobachev (2012)
Jacobi sum test	Mathematics	*workpool + reduce*	Eden	Lobachev (2012)
Gentleman	Mathematics	*torus*	Eden	Loogen *et al.* (2003)
Sudoku	Puzzle	*parMap*	*Par* Monad	Marlow *et al.* (2011)[18]

7.1 Preliminaries

7.1.1 Hardware and Software used

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

Apart from Eden, all benchmarks and libaries were compiled with Stack's[11] lts-7.1 GHC compiler which is equivalent to a standard GHC 8.0.1 with the base package in version 4.9.0.0. Stack itself was used in version 1.3.2. For GpH in its Multicore variant we used the the parallel package in version 3.2.1.0[12], while for the *Par* monad we used monad-par in version 0.3.4.8[13]. For all Eden tests, we used its GHC-Eden compiler in version 7.8.2[14] together with OpenMPI 1.6.5[15].

Furtermore, all benchmarks were done with help of the bench[16] tool in version 1.0.2 which uses criterion (>=1.1.1.0 && < 1.2)[17] internally. All runtime data (mean runtime, max stddev, etc.) was collected with this tool if not mentioned otherwise.

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

7.1.2 Benchmarks

We used multiple tests that originated from different sources. Most of them are parallel mathematical computations, initially implemented in Eden. Table 1 summarises.

---

[11] see https://www.haskellstack.org/
[12] see https://hackage.haskell.org/package/parallel-3.2.1.0
[13] see https://hackage.haskell.org/package/monad-par-0.3.4.8
[14] see http://www.mathematik.uni-marburg.de/~eden/?content=build_eden_7_&navi=build
[15] see https://www.open-mpi.org/software/ompi/v1.6/
[16] see https://hackage.haskell.org/package/bench
[17] see https://hackage.haskell.org/package/criterion-1.1.1.0
[18] actual code from: http://community.haskell.org/~simonmar/par-tutorial.pdf and https://github.com/simonmar/parconc-examples

Table 2: Overhead in the shared memory benchmarks

Benchmark	vs.	mean overhead	maximum normalized stdDev	runtime for 16 cores
Sudoku 1000	Eden CP	*0.05464*	0.03548	1.16433
	GpH	**-0.00762**	0.02074	1.11785
	Par Monad	**-0.04262**	0.01759	1.14049
Rabin–Miller test 11213 32	Eden CP	**-0.0036**	0.02502	5.18578
	GpH	*0.0252*	0.03199	5.30964
	Par Monad	**-0.00405**	0.02401	5.29689
Rabin–Miller test 11213 64	Eden CP	*0.00573*	0.015	10.32957
	GpH	*0.02427*	0.03006	10.56589
	Par Monad	**-0.04547**	0.26515	10.52869

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

Jacobi sum test or APRCL is also a primality test, that however, guarantees the correctness of the result. It is probabilistic in the sense that its run time is not certain. Unlike Rabin–Miller test, the subtests of Jacobi sum test have very different durations. Lobachev (2011) discusses some optimisations of parallel APRCL. Generic parallel implementations of Rabin–Miller test and APRCL were presented in Lobachev (2012).

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

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

7.1.3 What parallel Haskells run where

The *Par* monad and GpH – in its multicore version (Marlow *et al.*, 2009) – can be executed on 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".

Table 3: Overhead in the distributed memory benchmarks

Benchmark	vs.	mean overhead	maximum normalized stdDev	runtime for 256 cores
Gentleman 4096	Eden	*0.00972*	0.01484	110.07387
Rabin–Miller test 44497 256	Eden	**-0.00055**	0.01051	164.53853
Jacobi sum test 3217	Eden	**-0.00812**	0.02477	631.93115

7.1.4 Effect of hyper-threading

In preliminary tests, the PArrows version of Rabin–Miller test on a single node of the Glasgow cluster showed almost linear speedup on up to 16 shared-memory cores (Fig. 29). The speedup of 64-task PArrows/Eden at 16 real cores version was 13.65 giving a parallel efficiency of 85.3%. However, if we increased the number of requested cores to 32 – i.e. if we use hyper-threading on 16 real cores – the speedup did not increase that well. It was merely 15.99 for 32 tasks with PArrows/Eden. This was worse for other backends. As for 64 tasks, we obtained a speedup of 16.12 with PArrows/Eden at 32 hyper-threaded cores and only 13.55 with PArrows/GpH. The parallel eficiencies were 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 cluster fails to execute two parallel Haskell processes with full-fledged parallelism.

Hence, the performance measurements in Figs. 29–33 and Table **??** use only real cores as the purpose of this paper is to show the performance of PArrows and not to investigate parallel performance with hyper-threading in use.

7.2 Benchmark results

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. 30–32). 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

Speedup of Rabin–Miller test '32'
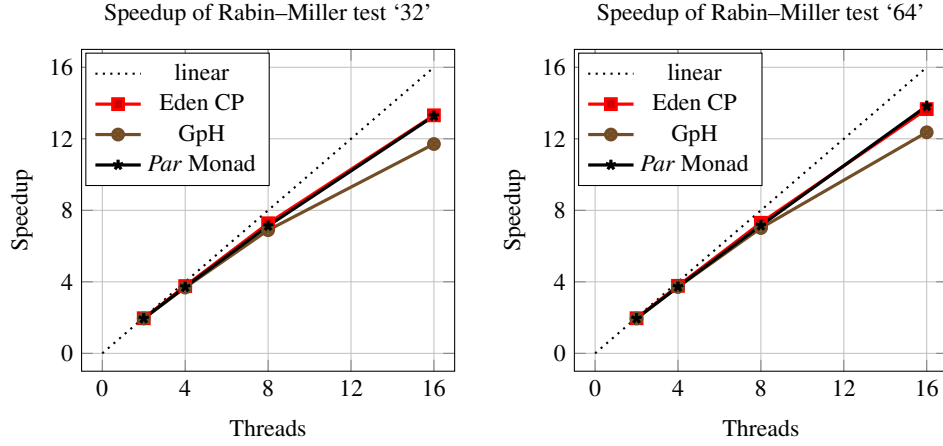
Speedup of Rabin–Miller test '64'

Figure 29: 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 cluster; 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.

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 benchmarking. 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.

7.2.1 Rabin–Miller test

The multicore version of our parallel Rabin–Miller test benchmark is depicted in Figure 29. 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.

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.

7.2.2 Jacobi sum test

Continuing, the results of the Jacobi sum test (MPI only) in Fig. 31 are as follows: The program does not seem to scale well beyond 64 threads with input $2^{3217} - 1$. We once again

Speedup of Rabin–Miller test '44497 256'
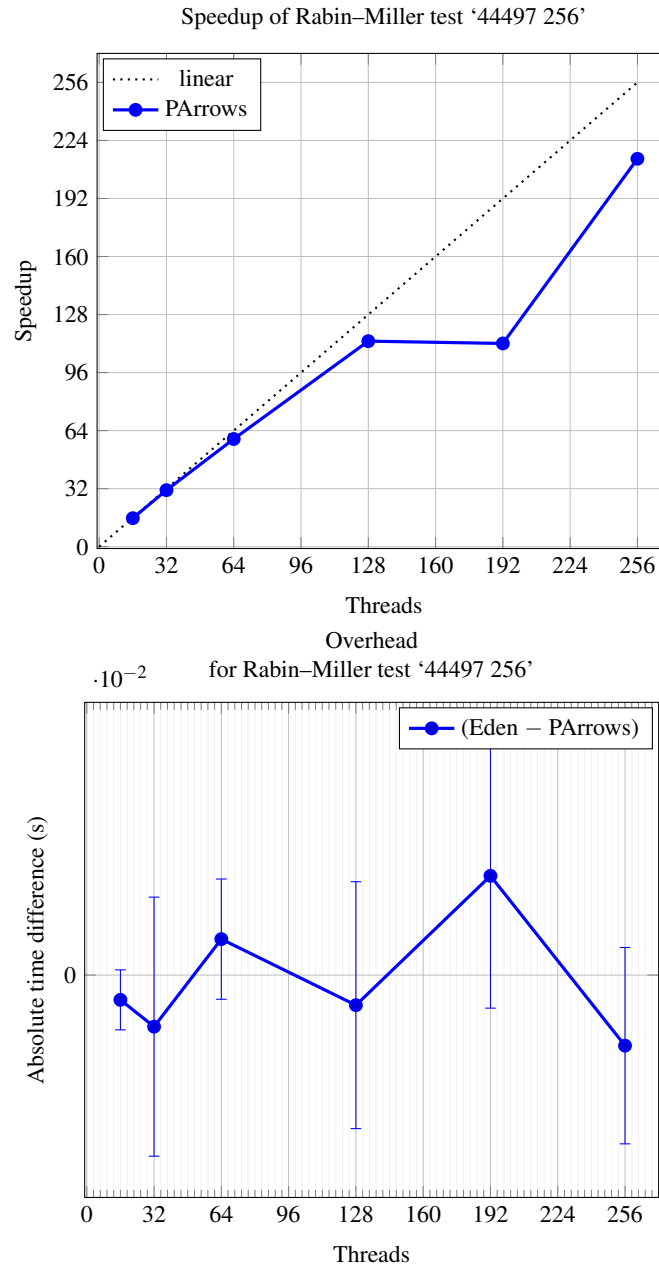

Overhead
for Rabin–Miller test '44497 256'

Figure 30: Parallel performance of the Rabin–Miller test on the Glasgow cluster 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**.

compare the Eden version with the PArrrows version in Fig. 31. 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. 31 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 core sare $-1005.18s$ 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.

7.2.3 Gentleman

Next is the Gentleman benchmark. The results of the comparison of vanilla Eden to our PArrows-based version can be found in Fig. 32. 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.

7.2.4 Sudoku

As the last benchmark in this paper we present the Sudoku in Fig. 33 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. 33 and 34. However, we notice that the GpH backend seems to choke on a bigger input (Fig. 34). 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.

7.3 Discussion

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.
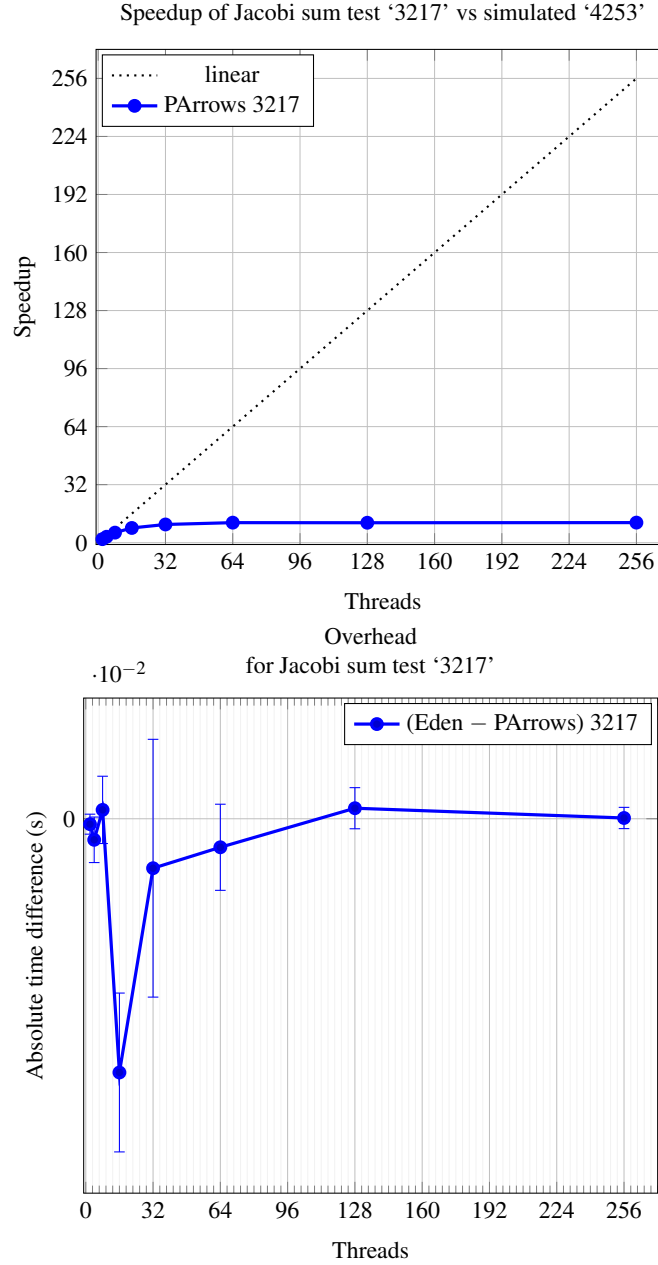
Speedup of Jacobi sum test '3217' vs simulated '4253'

Overhead
for Jacobi sum test '3217'

Figure 31: Parallel performance of the Jacobi sum test on the Glasgow cluster 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**.

M. Braun, O. Lobachev and P. Trinder

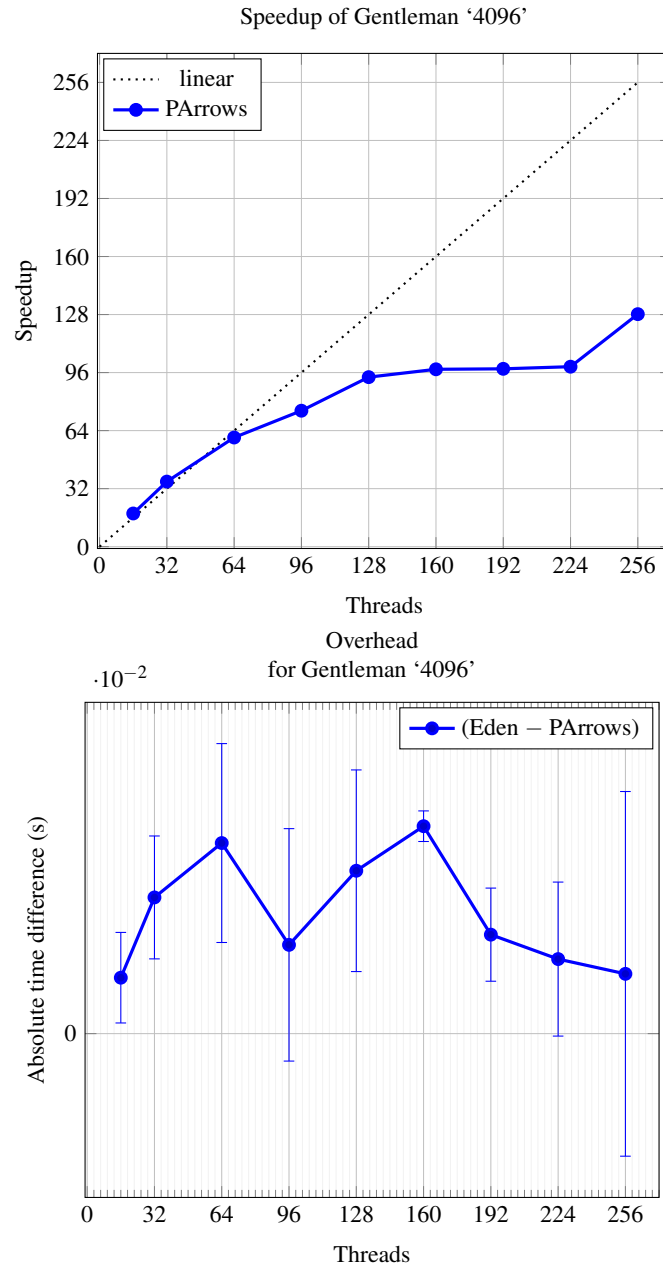Speedup of Gentleman '4096'

Overhead for Gentleman '4096'

Figure 32: Parallel performance of Gentleman on the Glasgow cluster 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**.

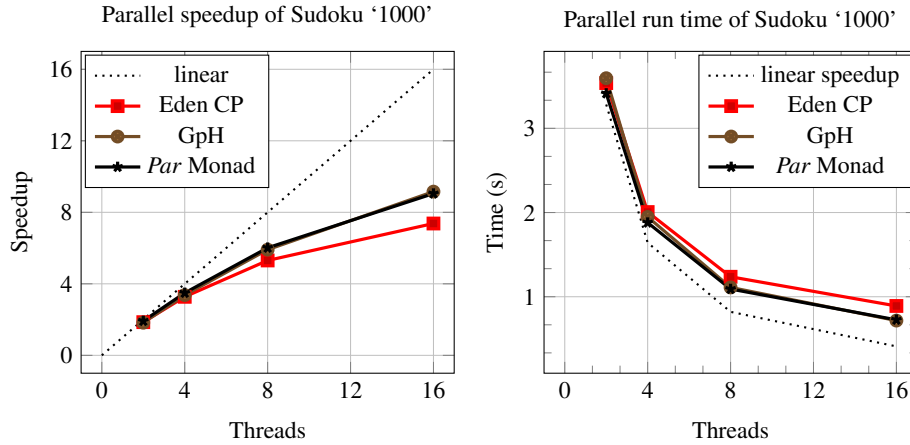Parallel speedup of Sudoku '1000'   Parallel run time of Sudoku '1000'

Figure 33: 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 cluster; it has 16 real cores and 32 threads. Input was a file of 1000 Sudokus. The closer to linear speedup the better.

8 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are first to represent *parallel* computation with Arrows. 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. Even though we make heavy use of the *ArrowApply* typeclass - which only Arrows that are equivalent to Monads can satisfy - in this paper, we believe that the general idea of an Arrow, the process of doing something - a composable *computation*, is the perfect fit to use in a library for parallel computation as it allows users to think about parallel programs more as a set of composed computations - like a data-flow graph. Furthermore, our approach allows for a more natural paralellization of any Arrow type (especially regular functions ($\rightarrow$)), when compared to Monads as we do not have to change anything in order to run them in parallel. 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 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.

8.1 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

Parallel speedup of Sudoku '16000'
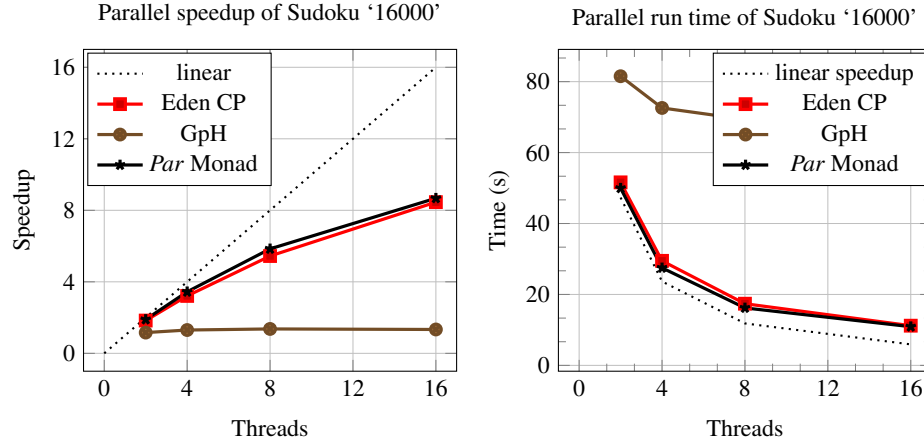
Parallel run time of Sudoku '16000'

Figure 34: 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 cluster; 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.

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 and Information Sciences, Faculty of Science, University of Nijmegen, The Netherlands. Technical Report NIII-R0416.

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

Aljabri, Malak, Loidl, Hans-Wolfgang, & Trinder, Phil W. (2014). The design and implementation of GUMSMP: A multilevel parallel haskell implementation. *Pages*

37:37–37:48 of: Proceedings of the 25th Symposium on Implementation and Application of Functional Languages. IFL '13. ACM.

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

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

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

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

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

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

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

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

Berthold, Jost, Dieterle, Mischa, 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^{3}L. *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 $***$:

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

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

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

Figure A 1: The definition of *evalN*

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

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

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

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

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

Figure A 3: *zipWith* over arrows.

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

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

B Profunctor Arrows

In Fig. B 1 we show how specific Profunctors can be turned into Arrows. This works because Arrows are strong Monads in the bicategory Prof of Profunctors as shown by Asada (2010). Note that in Standard GHC ($>\!\!>\!\!>$) has the type ($>\!\!>\!\!>$) :: *Category cat* $\Rightarrow$ *cat a b* $\rightarrow$ *cat b c* $\rightarrow$ *cat a c* and is therefore not part of the *Arrow* typeclass like presented in this paper.

```
instance (Category p, Strong p) ⇒ Arrow p where
   arr f = dimap id f id
   first = first′
instance (Category p, Strong p, Costrong p) ⇒ ArrowLoop p where
   loop = loop′
instance (Category p, Strong p, Choice p) ⇒ ArrowChoice p where
   left = left′
```

Figure B 1: Profunctors as Arrows[19]

C Omitted Function Definitions

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

We begin with warping Eden's build-in *RemoteData* to *Future* in Figure C 1

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

Figures C 3–C 6 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 C 7. Similarly, Figure C 8 contains the definition of arrow versions of Eden's *lazy* and *rightRotate* utility functions. Fig. C 9 contains Eden's definition of *lazyzip3* together with the utility functions *uncurry3* and *threetotwo*. The full definition of *farmChunk* is in Figure C 10. Eden definition of *ring* skeleton is in Figure C 11. It follows Loogen (2012).

The *parEval2* skeleton is defined in Figure C 12. We start by transforming the (a, c) input into a two-element list [*Either a c*] by first tagging the two inputs with *Left* and *Right* and

---

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

data *RemoteData a* = *RD* { *rd* :: *RD a* }

put′ :: (*Arrow arr*) ⇒ *arr a* (*BasicFuture a*)
put′ = *arr BF*

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

instance *NFData* (*RemoteData a*) **where**
  rnf = *rnf* ∘ *rd*

instance *Trans* (*RemoteData a*)

instance (*Trans a*) ⇒ *Future RemoteData a Conf* **where**
  put _ = *put′*
  get _ = *get′*

instance (*Trans a*) ⇒ *Future RemoteData a* () **where**
  put _ = *put′*
  get _ = *get′*

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

data *BasicFuture a* = *BF a*

put′ :: (*Arrow arr*) ⇒ *arr a* (*BasicFuture a*)
put′ = *arr BF*

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

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

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

instance *Future BasicFuture a* () **where**
  put _ = *put′*
  get _ = *get′*

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

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. C 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*.
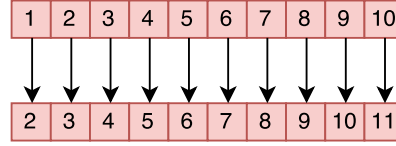
Figure C 3: Schematic depiction of *parMap*.

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

Figure C 4: Definition of parMap.
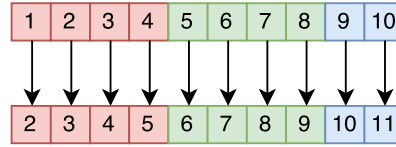
Figure C 5: Schematic depiction of *parMapStream*.

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 C 6: Definition of *parMapStream*.

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

Figure C 7: Definitions of *shuffle*, *unshuffle*, *takeEach*.

D Syntactic Sugar

Finally, we also give the definitions for some syntactic sugar for PArrows, namely ∗∗∗ and
&&&. For basic arrows, we have the ∗∗∗ combinator (Fig. 3) which allows us to combine
two arrows *arr a b* and *arr c d* into an arrow *arr* (*a*, *c*) (*b*, *d*) which does both computations
at once. This can easily be translated into a parallel version ∗∗∗ with the use of *parEval2*,
but for this we require a backend which has an implementation that does not require any
configuration (hence the () as the *conf* parameter):

(|∗∗∗|) :: (*ArrowChoice arr*, *ArrowParallel arr* (*Either a c*) (*Either b d*) ())) ⇒
    arr a b → *arr c d* → *arr* (*a*, *c*) (*b*, *d*)
(|∗∗∗|) = *parEval2* ()

$lazy :: (Arrow\ arr) \Rightarrow arr\ [a]\ [a]$
$lazy = arr\ (\lambda \sim (x:xs) \to x:lazy\ xs)$
$rightRotate :: (Arrow\ arr) \Rightarrow arr\ [a]\ [a]$
$rightRotate = arr\ \$\ \lambda\ list \to$ **case** $list$ **of**
$\quad [\,] \to [\,]$
$\quad xs \to last\ xs : init\ xs$

Figure C 8: Definitions of *lazy* and *rightRotate*.

$lazyzip3 :: [a] \to [b] \to [c] \to [(a,b,c)]$
$lazyzip3\ as\ bs\ cs = zip3\ as\ (lazy\ bs)\ (lazy\ cs)$
$uncurry3 :: (a \to b \to c \to d) \to (a,(b,c)) \to d$
$uncurry3\ f\ (a,(b,c)) = f\ a\ b\ c$
$threetotwo :: (Arrow\ arr) \Rightarrow arr\ (a,b,c)\ (a,(b,c))$
$threetotwo = arr\ \$\ \lambda \sim (a,b,c) \to (a,(b,c))$

Figure C 9: Definitions of *lazyzip3*, *uncurry3* and *threetotwo*.

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

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

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

Figure C 10: Definition of *farmChunk*.

```
ringSimple :: (Trans i, Trans o, Trans r) ⇒ (i → r → (o, r)) → [i] → [o]
ringSimple f is = os
    where (os, ringOuts) = unzip (parMap (toRD $ uncurry f) (zip is $ lazy ringIns))
        ringIns = rightRotate ringOuts

toRD :: (Trans i, Trans o, Trans r) ⇒ ((i, r) → (o, r)) → ((i, RD r) → (o, RD r))
toRD f (i, ringIn) = (o, release ringOut)
    where (o, ringOut) = f (i, fetch ringIn)

rightRotate :: [a] → [a]
rightRotate [] = []
rightRotate xs = last xs : init xs

lazy :: [a] → [a]
lazy ∼(x : xs) = x : lazy xs
```

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

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

Figure C 12: 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 C 13: Definition of *prMMTr*, *splitMatrix* and *matAdd*.