Under consideration for publication in J. Functional Programming

Arrows for Parallel Computations

Submission ID xxxxxx

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 arrows-based language for parallelism and implement it using multiple parallel Haskells. In this manner we are able to bridge across various parallel Haskells.

Additionally, our way of writing parallel programs has the benefit of being portable across multiple parallel Haskell implementations. Furthermore, as each parallel computation is an arrow, which means that they can be readily composed and transformed as such. We introduce some syntactic sugar to provide parallelism-aware arrow combinators.

To show that our arrow-based language has similar expressiveness to existing parallel languages, we also define several parallel skeletons with our framework. Benchmarks show that our framework does not induce too much overhead performance-wise. Summarize conclusions

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

Contents

Contents

1 Introduction

One particular reason for this is the ease of introducing new sophisticated language concepts. 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 the Multicore Haskell (a SMP implementation of GpH language), 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. A key novelty here is to use Arrows to represent parallel computations. They seem a natural fit as they are a generalization of the function $\rightarrow$ and serve as general interface to computations.

We provide an Arrows-based typeclass and implementations for three 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 and varying implementations in existing parallel Haskells: Multicore Haskell, *Par* Monad, and Eden. Thus, we not only define a parallel programming interface in a novel manner - we tame the zoo of parallel Haskells. We provide a common quantify, e.g. less than 10 %, very low-penalty programming interface that allows to switch the parallel Haskell backends at will. Further backends based on HdpH or a Frege implementation (on the Java Virtual Machine) are viable, too.

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

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

We host this functionality in the *ArrowParallel* typeclass, which abstracts all parallel implementation logic in a way that the backend easily be swapped. This means it is possible

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

Fig. 1. The definition of *Arrow* type class and its two most typical instances.

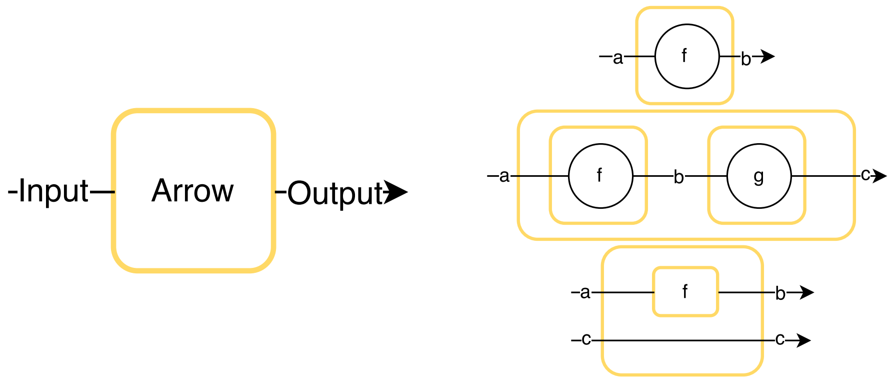


Fig. 2. Schematic depiction of Arrow (left) and its basic combinators *arr*, $\ggg$ and *first* (right).

to write code against the interface of a common typeclass without being bound to any specific backend. 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.

We also show that it is possible to define algorithmic skeletons with Arrows (Sections 6, 7) and finally demonstrate that Arrow Parallelism is a viable alternative to existing approaches and prove that only low overhead is introduced, i.e. less than x% fix this after benchmarks are finally done as shown in Section 8.

Mention Future work here?

2 Background

2.1 Arrows

Arrows were introduced by HughesArrows as a general interface for computation. An Arrow *arr a b* represents a computation that converts an input *a* to an output *b*. This is defined in the *Arrow* type class shown in Fig. 1. Its *arr* operation is used to lift an ordinary function to the specified arrow type, similarly to the monadic *return*. The $>>>$ operator is analogous to the monadic composition $\gg=$ and combines two arrows *arr a b* and *arr b c* by "wiring" the outputs of the first to the inputs to the second to get a new arrow *arr a c*. Lastly, the *first* operator takes the input arrow from b to c and converts it into an arrow on pairs with the second argument untouched. It allows us to to save input across arrows. Figure 2 shows a graphical representation of the basic Arrow combinators. The most prominent instances of this interface are regular functions $(\to)$ and the Kleisli type (Fig. 1).

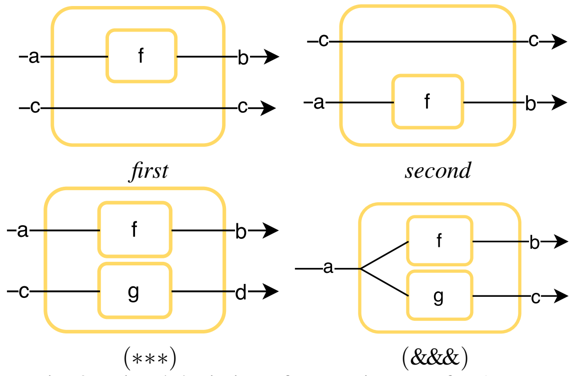


Fig. 3. Visual depiction of syntactic sugar for Arrows.

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

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

while the &&& combinator, that constructs an arrow which outputs two different values like $*\!*\!*$, but takes only one input, is:

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

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

$$\begin{aligned}&add :: Arrow\ arr \Rightarrow arr\ a\ Int \to arr\ a\ Int \to arr\ a\ Int\\ &add\ f\ g = (f\ \&\&\&\ g) >>> arr\ (\lambda(u,v) \to u+v)\end{aligned}$$

The more restrictive interface of Arrows allows for more elaborate composition and transformation combinators—a Monad can be *anything*, an Arrow is a process of doing something, a *computation*. One of the major problems in parallel computing is, however, composition of parallel processes.

2.2 Short introduction to parallel Haskells

There are already several ways to write parallel programs in Haskell:

Table 1. *Overview over some parallel Haskells*

Name	Memory setting	Concepts used
GpH	shared & distributed	*par*/*pseq* "hints"
Par Monad	shared	deterministic parallel Monad
Eden	shared & distributed	special compiler, pure & monadic interface
HdpH	shared? & distributed	parallel Monad, Template Haskell
LVish	shared	parallel Monad, parallel data-structures

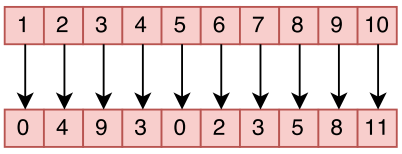


Fig. 4. Schematic illustration of *parEvalN*.

As we base our parallel Arrows on existing parallel Haskells, we will now give a short introduction to the ones we use as backends in this paper.

In its purest form, parallel computation (on functions) can be looked at as the execution of some functions $a \rightarrow b$ in parallel or $parEvalN :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$, as also Figure 4 symbolically shows. We implement this non-Arrow version here which will later in this paper be adapted for usage in our Arrow based parallel Haskell. Furthermore, the implementation of *parEvalN* serves as a short demonstration of how the used parallel Haskell backends work.

2.2.1 Multicore Haskell

Multicore Haskell (Marlow *et al.*, 2009; Trinder *et al.*, 1998) is one of the simplest ways to do parallel processing found in standard GHC.[1] Besides some basic primitives 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) ⇒ [a → b] → [a] → [b]
parEvalN fs as = let bs = zipWith ($) fs as
  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 evalutes to normal form. Fig. 5 shows a visual representation of this code.

[1] Multicore Haskell on Hackage is available under https://hackage.haskell.org/package/parallel-3.2.1.0, compiler support is integrated in the stock GHC.

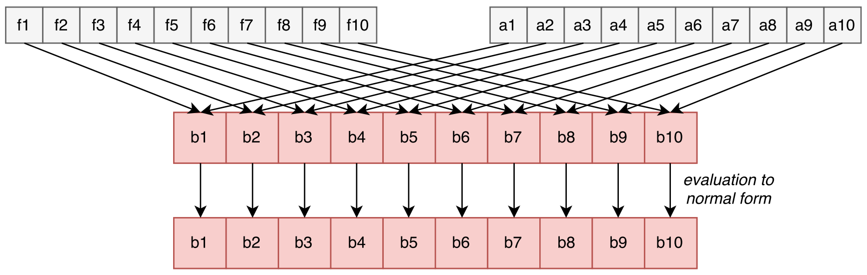

f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10
evaluation to normal form
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10

Fig. 5. Dataflow of the Multicore Haskell *parEvalN* version.

2.2.2 Par Monad

The *Par* Monad[2] introduced by par-monad, is a Monad designed for composition of parallel programs. Let:

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

The *Par* Monad version of our parallel evaluation function *parEvalN* is defined by zipping the list of $[a \to b]$ with the list of inputs $[a]$ with the application operator \$ just like with Multicore Haskell. Then, we map over this not yet evaluated lazy list of results $[b]$ with $\mathit{spawnP} :: \mathit{NFData}\ a \Rightarrow a \to \mathit{Par}\ (\mathit{IVar}\ a)$ to transform them to a list of not yet evaluated forked away computations $[\mathit{Par}\ (\mathit{IVar}\ b)]$, which we convert to $\mathit{Par}\ [\mathit{IVar}\ b]$ with *sequenceA*. We wait for the computations to finish by mapping over the *IVar b* values inside the *Par* Monad with *get*. This results in $\mathit{Par}\ [b]$. We execute this process with *runPar* to finally get $[b]$. Fig. 6 shows a graphical representation.

2.2.3 Eden

Eden (Loogen *et al.*, 2005; Loogen, 2012) is a parallel Haskell for distributed memory and comes with a MPI and a PVM backends.[3] 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 also on multicores (Berthold *et al.*, 2009a; Aswad *et al.*, 2009).

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

$$\begin{aligned}&\mathit{parEvalN} :: (\mathit{Trans}\ a, \mathit{Trans}\ b) \Rightarrow [a \to b] \to [a] \to [b]\\&\mathit{parEvalN} = \mathit{spawnF}\end{aligned}$$

[2] It can be found in the `monad-par` package on hackage under https://hackage.haskell.org/package/monad-par-0.3.4.8/.

[3] See also http://www.mathematik.uni-marburg.de/ eden/ and https://hackage.haskell.org/package/edenmodules-1.2.0.0/.

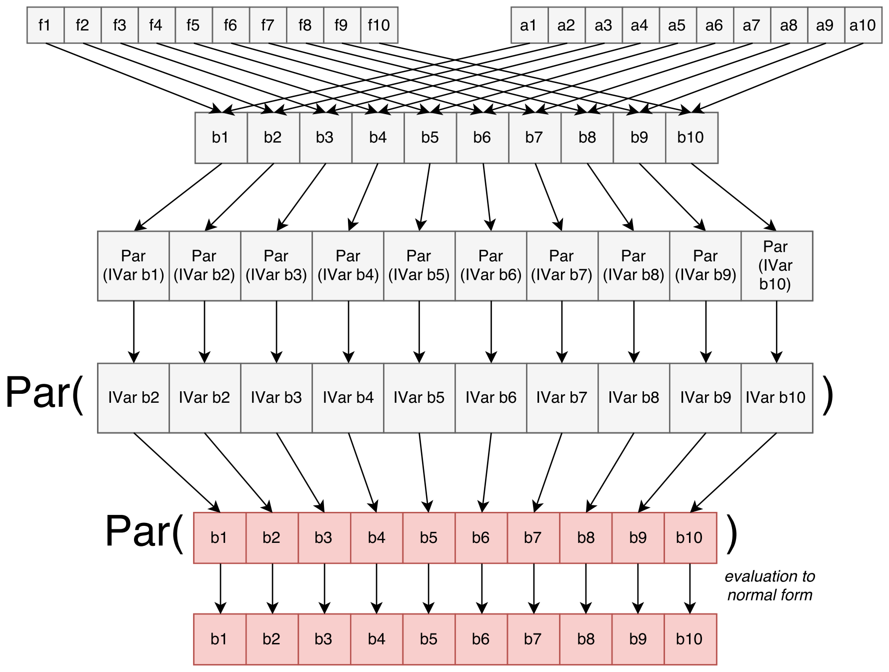

f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10
Par (IVar b1) Par (IVar b2) Par (IVar b3) Par (IVar b4) Par (IVar b5) Par (IVar b6) Par (IVar b7) Par (IVar b8) Par (IVar b9) Par (IVar b10)
Par(IVar b2 IVar b2 IVar b3 IVar b4 IVar b5 IVar b6 IVar b7 IVar b8 IVar b9 IVar b10)
Par(b1 b2 b3 b4 b5 b6 b7 b8 b9 b10)
evaluation to normal form
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10

Fig. 6. Dataflow of the *Par* Monad *parEvalN* version.

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

The trace visualizations are color-coded. In such a visualization (Fig. 13), the x axis shows the time, the y axis enumerates the machines and processes. The visualization shows a running process in green, a blocked process is red. If the process is runnable, it may run, but does not, it is yellow. The typical reason for thus is GC. An inactive machine, where no processes are started yet, or all are already terminated, shows as a blue bar. A comminication from one process to another is represented with a black arrow. A stream of communications, a transmitted list is shows as a dark shading between sender and receiver processes.

3 Related Work

Parallel Haskells. Of course, the three parallel Haskell flavours we have presented above: the GpH (Trinder *et al.*, 1996; Trinder *et al.*, 1998) parallel Haskell dialect and its multicore version (Marlow *et al.*, 2009), the *Par* Monad (Marlow *et al.*, 2011; Foltzer *et al.*, 2012),

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

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; Aljabri *et al.*, 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; Berthold *et al.*, 2009c; 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. marlow2013parallel authored a recent book in marlow2013parallel on parallel Haskells.

Algorithmic skeletons. Algorithmic skeletons were introduced by Cole1989. Early publications on this topic include (Darlington *et al.*, 1993; Botorog & Kuchen, 1996; Danelutto *et al.*, 1997; Gorlatch, 1998; Lengauer *et al.*, 1997). SkeletonBook 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; Lobachev, 2012; Janjic *et al.*, 2013), iteration skeletons (Dieterle *et al.*, 2013). The idea of scscp is to use a parallel Haskell to orchestrate further software systems to run in parallel. dieterle$_horstmeyer_loogen_berthold_2016comparethecompositionofskeleto$

Arrows. Arrows were introduced by HughesArrows, basically they are a generalised function arrow $\rightarrow$. Hughes2005 presents a tutorial on Arrows. Some theoretical details on Arrows (Jacobs *et al.*, 2009; Lindley *et al.*, 2011; Atkey, 2011) are viable. Paterson:2001:NNA:507669.507664 introduced a new notation for Arrows. Arrows have applications in information flow research (Li & Zdancewic, 2006; Li & Zdancewic, 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:2009:CCA:1631687.1596559 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.Huang2007 utilise Arrows for parallelism, but strikingly different from our approach. They basically 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 centred 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[5] (which is basically Haskell on the JVM) were quite successful. However, it is beyond the scope of this work.

achten2004arrows,achten2007arrow use an arrow implementation in Clean for better handling of typical GUI tasks. Dagand:2009:ORD:1481861.1481870 used arrows in OCaml in the implementation of a distributed system.

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 exisiting parallel Haskells. Finally, we explain why using Arrows for expressing parallelism is beneficial.

4.1 The ArrowParallel typeclass

A parallel computation (on functions) in its purest form can be seen as execution of some functions $a \to b$ in parallel, as our *parEvalN* prototype shows (Sec. 2.2). Translating this into arrow terms gives us a new operator *parEvalN* that lifts a list of arrows $[arr\ a\ b]$ to a parallel arrow $arr\ [a]\ [b]$. This combinator is similar to our utility function *listApp* from Appendix A, but does parallel instead of serial evaluation.

```
parEvalN :: (Arrow arr) ⇒ [arr a b] → 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 typeclass *ArrowParallel arr a b*:

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

Sometimes parallel Haskells require or allow for additional configuration parameters, 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]
```

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

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

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

Fig. 7. Fully evaluating *ArrowParallel* instance for the Multicore Haskell backend.

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

Fig. 8. Configurable *ArrowParallel* instance for the Multicore Haskell backend.

4.2 ArrowParallel instances

4.2.1 Multicore Haskell

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

4.2.2 Par Monad

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

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

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

Fig. 9. ArrowParallel instance for the Par Monad backend.

4.2.3 Eden

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

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

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

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

and the Kleisli type:

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

4.3 Extending the Interface

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

4.3.1 Lazy parEvalN

The function *parEvalN* is 100% strict, which means that it fully evaluates all passed arrows. Sometimes this might not be feasible, as it will not work on infinite lists of functions like e.g. $map\ (arr \circ (+))\ [1\,..]$ or just because we need the arrows evaluated in chunks. *parEvalNLazy* (Figs. 10, 11) fixes this. It works by first chunking the input from $[a]$ to $[[a]]$ with the given *ChunkSize* in $arr\ (chunksOf\ chunkSize)$. These chunks are then fed into a

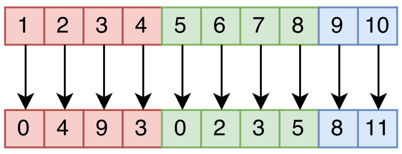


Fig. 10. Schematic depiction of *parEvalNLazy*.

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

Fig. 11. Definition of *parEvalNLazy*.

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 Heterogenous tasks

We have only talked about the paralellization arrows of the same type until now. But sometimes we want to paralellize heterogenous types as well. However, we can implement such a *parEval2* combinator (Figs. 12, B 12) which combines two arrows *arr a b* and *arr c d* into a new parallel arrow $arr\ (a,c)\ (b,d)$ quite easily with the help of the *ArrowChoice* typeclass. The idea is to use the $+++$ combinator which combines two arrows *arr a b* and *arr c d* and transforms them into $arr\ (Either\ a\ c)\ (Either\ b\ d)$ to get a common arrow type that we can then feed into *parEvalN*.

5 Futures

Consider a mock-up parallel arrow combinator:

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

In a distributed environment, a resulting arrow of this combinator first evaluates all $[arr\ a\ b]$ in parallel, sends the results back to the master node, rotates the input once and then evaluates the $[arr\ b\ c]$ in parallel to then gather the input once again on the master

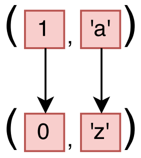


Fig. 12. Schematic depiction of *parEval2*.

node. Such situations arise, 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 properly together, it illustrates an important problem. When using a *ArrowParallel* backend that resides on multiple computers, all communication between the nodes is done via the master node, as shown in the Eden trace in Figure 13. This can become a serious bottleneck for larger amount of data and number of processes [as][showcases]Berthold2009-fft.

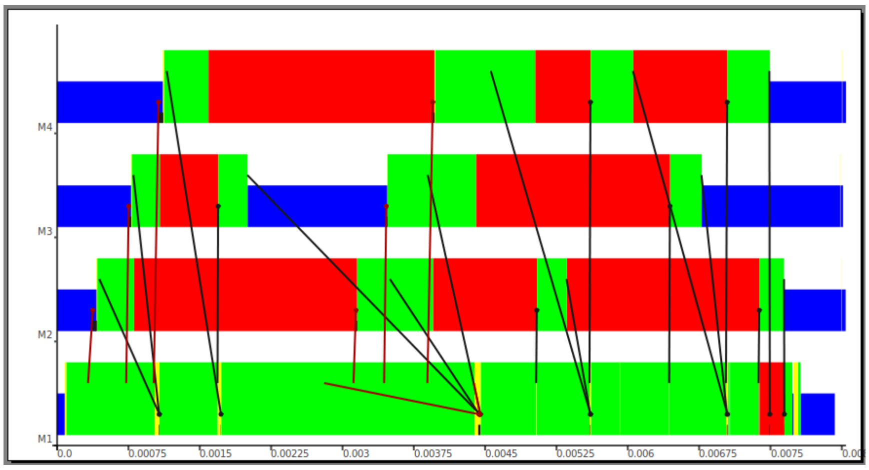


Fig. 13. Communication between 4 Eden processes without Futures. All communication goes through the master node. Each bar represents one process. Black lines between processes represent communication. Colors: blue ≙ idle, green ≙ running, red ≙ blocked, yellow ≙ suspended.
more practical and heavy-weight example! fft (I have the code)?
Depends... Are the communications easy to read in such an example?
Keep the description for the different colours, or link to the EdenTV description in 2.2.3 ok as is use the fft example (when it works)?

We should allow the nodes to communicate directly with each other. Eden already ships with "remote data" that enable this (Alt & Gorlatch, 2003; Dieterle *et al.*, 2010a). But as we want code with our DSL to be implementation agnostic, we have to wrap this context. We do this with the *Future* typeclass (Fig. 14). Since *RD* is only a type synonym for a

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

Fig. 14. Definition of the *Future* typeclass.

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

For our *Par* Monad and Multicore Haskell backends, we can simply use *BasicFuture*s (Fig. 15) which are just simple wrappers around the actual data, as in a shared memory setting we do not require Eden's sophisticated communication channels.

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

Fig. 15. The *BasicFuture* type and its *Future* instance for the *Par* Monad and Multicore Haskell backends.

In our communication example we can use this *Future* concept for direct communications between the nodes as shown in Fig. 16. In a distributed environment, this gives us a

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

Fig. 16. The mock-up combinator in parallel.

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. 17.Fig. is not really clear. Do Figs with a lot of load? — fft?

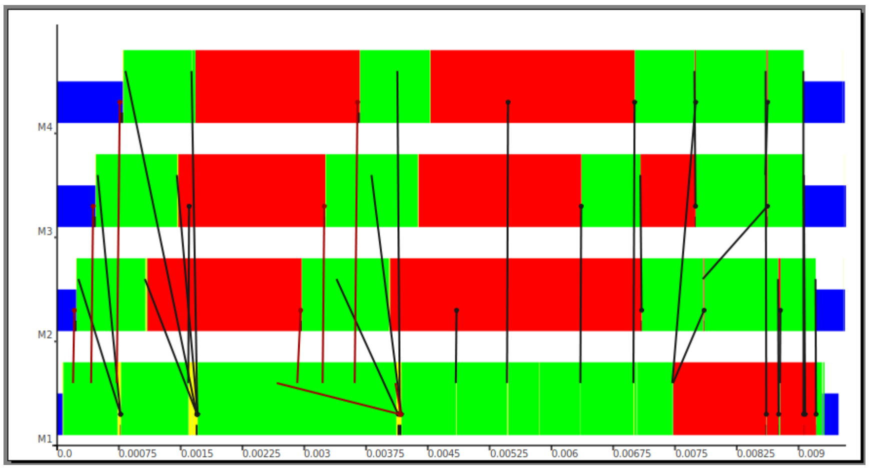


Fig. 17. Communication between 4 Eden processes with Futures. Other than in Fig. 13, processes communicate directly (black lines between the bars) instead of always going through the master node (bottom bar).

6 Map-based Skeletons

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

Parallel *map* and laziness. The *parMap* skeleton (Figs. B 2, B 3) is probably the most common skeleton for parallel programs. We can implement it with *ArrowParallel* by repeating an arrow *arr a b* and then passing it into *parEvalN* to obtain an arrow *arr* $[a]$ $[b]$. Just like *parEvalN*, *parMap* is 100% strict. As *parMap* is 100% strict it has the same restrictions as *parEvalN* compared to *parEvalNLazy*. So it makes sense to also have a *parMapStream* (Figs. B 4, B 5) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*. The code is quite straightforward, we show it in Appendix.

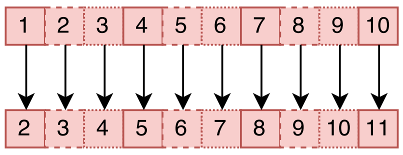


Fig. 18. Schematic depiction of a *farm*, a statically load-balanced *parMap*.

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

Fig. 19. The definition of *farm*.

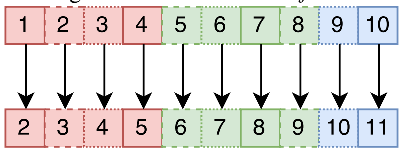


Fig. 20. Schematic depiction of *farmChunk*.

Statically load-balancing parallel *map*. Our *parMap* spawns every single computation in a new thread (at least for the instances of *ArrowParallel* we gave in this paper). This can be quite wasteful and a statically load-balancing *farm* (Figs. 18, 19) that equally distributes the workload over *numCores* workers seems useful. The definitions of the helper functions *unshuffle*, *takeEach*, *shuffle* (Fig. B 7) originate from an Eden skeleton[6].

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

7 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

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

with more predefined skeletons[7], among them a *pipe*, a *ring*, and a *torus* implementations Eden:SkeletonBookChapter02. 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 Multicore and Par Monad backends which during runtime would result in the program crashing. This materializes with the usage of *loop* of the *ArrowLoop* typeclass and is probably due to the way their respective parallelism engines work internally. okay so? As these skeletons probably do not make any practical sense besides for testing with these backends anyways (because of the shared memory between the threads), we create an extra abstraction layer for the original *parEvalN* in these skeletons called *evalN* in the *FutureEval* typeclass. This allows us for selective enabling and disabling of parallelism.

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

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

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

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

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

7.1 Parallel pipe

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

```
loop (arr (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. 21) by plugging

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

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

Fig. 21. A first implementation of the *pipe* skeleton expressed with Parallel Arrows. Note that the use of *lazy* (Fig. B 8) is essential as without it programs using this definition would never halt. We need to enforce that the evaluation of the input $[a]$ terminates before passing it into *evalN*.

```
pipe :: (ArrowLoop arr, FutureEval arr (fut a) (fut a) conf, Future fut a) ⇒
  conf → [arr a a] → arr a a
pipe conf fs = unliftFut (pipeSimple conf (map liftFut fs))
```

Fig. 22. Final definition of the *pipe* skeleton with Futures.

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

Fig. 23. Definition of *pipe2* and a parallel >>>.

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

However, using this definition directly will make the master node a potential bottleneck in distributed environments as described in Section 5. Therefore, we introduce a more sophisticated version that internally uses Futures and obtain the final definition of *pipe* in Fig. 22.

Sometimes, this *pipe* definition can be a bit inconvenient, especially if we want to pipe arrows of mixed types together, i.e. *arr a b* and *arr b c*. By wrapping these two arrows inside a common type we obtain *pipe2* (Fig. 23).

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

Another version of >>> is:

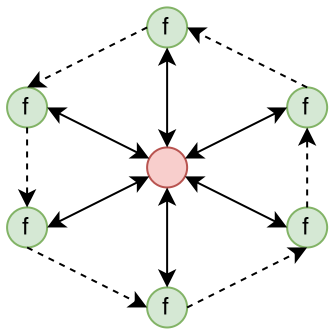


Fig. 24. Schematic depiction of the ring skeleton.

$f \mid\ggg\mid g = (f \circ put) \ggg (get \circ g)$

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

7.2 Ring skeleton

Eden comes with a ring skeleton[8] (Fig. 24) implementation that allows the computation of a function $[i] \to [o]$ with a ring of nodes that communicate in a ring topology with each other. Its input is a node function $i \to r \to (o,r)$ in which r serves as the intermediary output that gets send to the neighbour of each node. This data is sent over direct communication channels, the so called remote data. We depict it in Appendix, Fig. B 11.

We can rewrite this functionality easily with the use of *loop* as the definition of the node function, $arr\ (i,r)\ (o,r)$, after being transformed into an arrow, already fits quite neatly into the *loop*'s $arr\ (a,b)\ (c,b) \to arr\ a\ c$. In each iteration we start by rotating the intermediary input from the nodes $[fut\ r]$ with $second\ (rightRotate \ggg lazy)$ (Fig. B 8). Similarly to the *pipe* from Section 7.1 (Fig. 21), we have to feed the intermediary input into our *lazy* (Fig. B 8) arrow here, or the evaluation would fail to terminate. The reasoning is explained by Loogen2012 as a demand problem.

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

7.3 Torus skeleton

If we take the concept of a *ring* from Section 7.2 one dimension further, we obtain a *torus* skeleton (Fig. 26, 27). Every node sends ands receives data from horizontal and vertical

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

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

Fig. 25. Final definition of the *ring* skeleton.

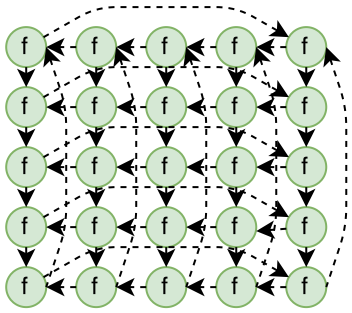


Fig. 26. Schematic depiction of the *torus* skeleton.

neighbours in each communication round. With our Parallel Arrows we re-implement the *torus* combinator[9] from Eden—yet again with the help of the *ArrowLoop* typeclass.

Similar to the *ring*, we once again start by rotating the input (Fig. B 8), but this time not only in one direction, but in two. This means that the intermediary input from the neighbour nodes has to be stored in a tuple $([[fut\ a]],[[fut\ b]])$ in the second argument (loop only allows for two arguments) of our looped arrow $arr\ ([[c]],([[fut\ a]],[[fut\ b]]))\ ([[d]],([[fut\ a]],[[fut\ b]]))$ and our rotation arrow becomes

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

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

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

from $([[c]],([[fut\ a]],[[fut\ b]]))$ to $[[(c,fut\ a,fut\ b)]]$, which we then feed into our parallel execution.

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

As an example of using this skeleton Eden:SkeletonBookChapter02 showed the matrix multiplication using the Gentleman algorithm (Gentleman1978). An adapted version can

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

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

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

Fig. 28. Adapted matrix multiplication in Eden using a the *torus* skeleton. *prMM_torus* is the parallel matrix multiplication. *mult* is the function performed by each worker. *prMMTr* calculates AB^T and is used for the (sequential) calculation in the chunks. *splitMatrix* splits the Matrix into chunks. *matAdd* calculates $A+B$. Omitted definitions can be found in B 13.

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

8 Performance results

8.1 Hardware

We have tested our parallel DSL and algorithmic skeletons implemented in it. Benchmarks were conducted both in a shared and in a distributed memory setting. All benchmarks were done on the “Glasgow grid”, consisting of 16 machines with 2 Intel® Xeon® E5-2640 v2 and 64 GB of DDR3 RAM each. Each processor has 8 cores and 16 (hyperthreaded) more nodes!!!

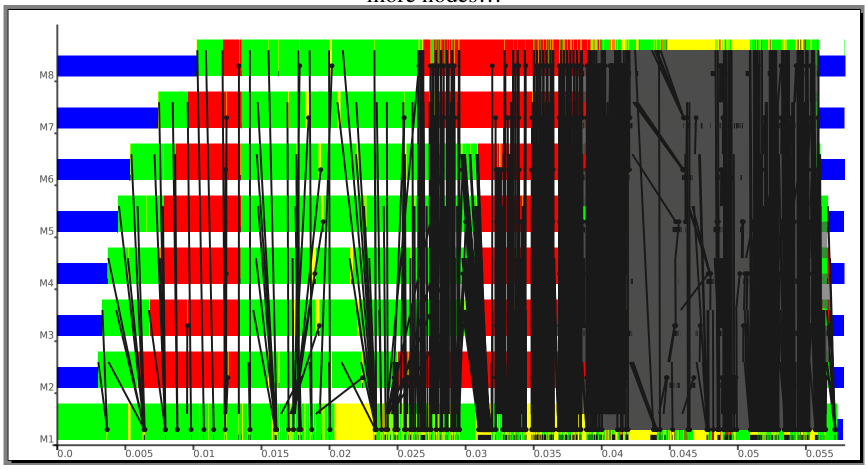


Fig. 29. Matrix Multiplication with *torus* (PArrows).

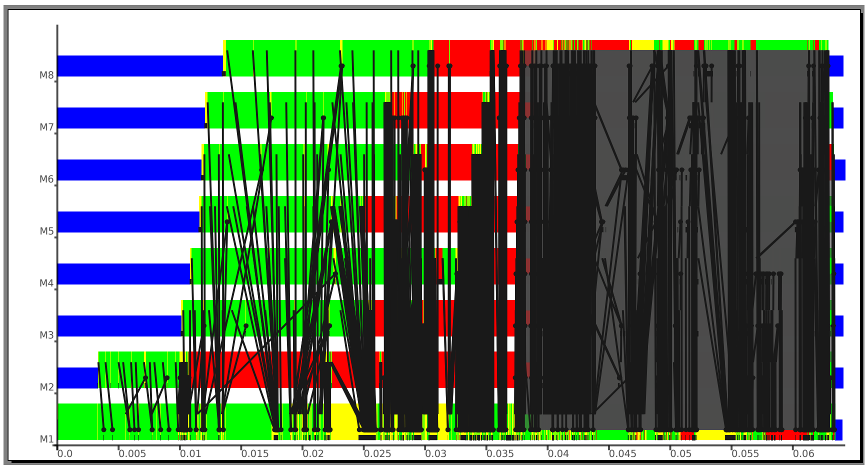


Fig. 30. Matrix Multiplication with *torus* (Eden).

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 hyperthreaded 32 cores do not behave in the same manner as real 16 cores (numbers here for a single machine). We disregarded the hyperthreading ability in most of the cases.

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

8.2 Test programs

We used multiple tests that originated from different sources. Most of them are parallel mathematical computations, initially implemented in Eden. Table 2 summarizes.

Table 2. *The benchmarks we use in this paper.*

Name	Area	Type	Origin	Citation
Rabin–Miller test	Mathematics	*parMap* + *reduce*	Eden	Lobachev2012
Jacobi sum test	Mathematics	*workpool* + *reduce*	Eden	Lobachev2012
Gentleman	Mathematics	*torus*	Eden	Eden:SkeletonBookChapter02
Sudoku	Puzzle	*parMap*	*Par* Monad	par-monad [10]

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

Jacobi sum test or APRCL is also a primality test, that however, guarantees the correctness of the result. It is probabilistic in the sense that its run time is not certain. Unlike Rabin–Miller test, the subtests of Jacobi sum test have very different durations. lobachev-phd discuss some optimisations of parallel APRCL. Generic parallel implementation of Rabin–Miller testand APRCL were presented in Lobachev2012.

“Gentleman” is a standard Eden test program, developed for their *torus* skeleton. It implements a parallel matrix multiplication Gentleman1978. We ported an Eden based version Eden:SkeletonBookChapter02 to PArrows.

A parallel Sudoku solver was used by par-monad to compare *Par* Monad to Multicore Haskell. We ported it to PArrows.

8.3 What parallel Haskells run where

The *Par* monad and Multicore Haskell can be executed on a shared memory machines only. Although GpH is available on distributed memory clusters, and newer distributed memory Haskells such as HdpH exist, current support of distributed memory in PArrows is limited to Eden. We used the MPI backend of Eden in a distributed memory setting. However, for shared memory Eden features a “CP” backend that merely copies the memory blocks between distributed heaps. In this mode Eden still operates in the “nothing shared” setting, but is adapted better to multicore machines. We label this version of Eden in the plots as “Eden CP”.

8.4 Effect of hyperthreading

The PArrows version of Rabin–Miller teston a single node of the Glasgow grid showed almost linear speedup (Fig. 31). The speedup of 64-task PArrows/Eden at 16 real cores version was 13.65, the efficiency was 85.3%. However, if we increase the number of requested cores to be 32—if we use hyperthreading on 16 real cores—the speedup does not increase that well. It is merely 15.99 for 32 tasks with PArrows/Eden. It is worse for other backends. As for 64 tasks, we obtain the speedup of 16.12 with PArrows/Eden at 32 hyperthreaded cores and only 13.55 with PArrows/Multicore Haskell. Efficiency is 50.4%

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

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 hyperthreaded cores, whereas it was 213.172 for 256 real cores. Apparently, hyperthreading in the Glasgow grid fails to execute two parallel Haskell processes with full-fledged parallelism. For this reason, we did not regard hyperthreaded cores in our speedup plots in Figs. 31-35.

8.5 Benchmark results

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

To show that PArrows induce very small overhead in a distributed context as well, we compare the original Eden versions of the benchmark to its PArrows-enabled counterpart in the Rabin–Miller test, Gentlemanand Jacobi sum testbenchmarks. We plot execution time differences between measurements for PArrows and the corresponding backend in a separate plot (Figs. 32, 34, 33). As an example, the differences range in about 0.5 seconds for the execution time of 46 seconds on 256 cores for distributed Rabin–Miller testwith PArrows and Eden. For these comparisons, the plots show absolute time differences that are not relative the total execution time. Furthermore, the error bars ends were computed from pointwise 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 above measurement error.

8.5.1 Rabin–Miller test

THE ACTUAL TEXT IS MISSING. What do we see in the plots? Why is it good? The multicore version of our parallel Rabin–Miller testbenchmark is depicted in Figure 31. We executed the test with 32 and 64 tasks. The plot shows the PArrows-enabled versions with corresponding backends. The performance of PArrows/Eden CP in shared memory is slightly better than for SMP variants such as PArrows/Multicore Haskell and PArrows/*Par* Monad but most of the time the performance is still comparable with the Multicore backend performing slightly worse than the other two in terms of speedup. In particular, the speedups for 16 cores fall behind quite noticeably with 11.69 for 32 tasks as Eden and the *Par* Monad both score 13.05 for 32 tasks. While the Multicore backend scores a bit better in the 64 tasks benchmark, its speedup of 12.70 is still behind the 13.65 and 13.33 of Eden and *Par* Monad, respectively.

Comparing the PArrows version of the Rabin–Miller testwith the original from Eden with the MPI backend in a distributed memory setting, we see an almost linear speedup of Rabin–Miller testwith 256 tasks and input $2^{444497} - 1$ in both versions. The sequential

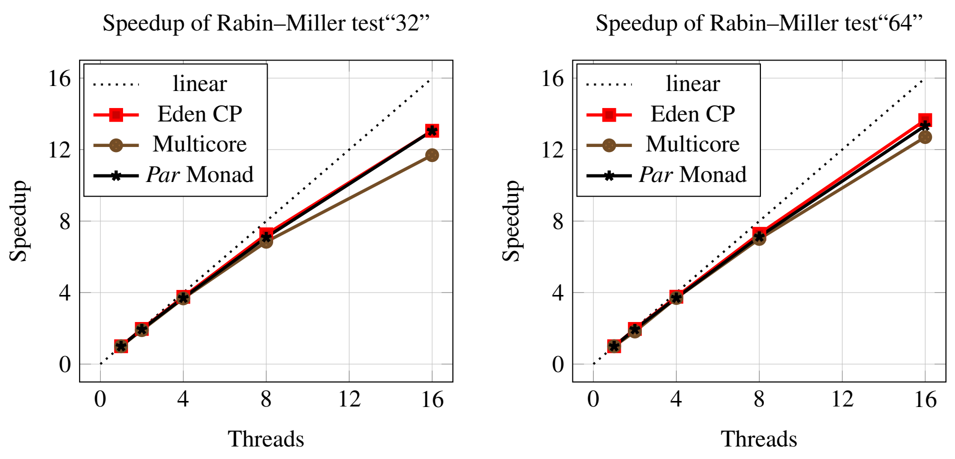


Fig. 31. Relative speedup of Rabin–Miller teston a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores. Input was $2^{11213}-1$, we used 32 (left) or 64 (right)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

run time was computed as the mean of three consecutive executions on a single core—the single run took two hours 43 minutes. The zero difference between PArrows/Eden and Eden almost always lies on the error bar of the measurement. The only exception, where the PArrows and Eden versions differ and it was it was significant, was for 64 cores, where it performed 0.49s better. This corresponds to 0.30% relative time difference. Otherwise, PArrows was 0.23s (or 0.0007%) slower for 32 cores, 0.26s (or 0.0030%) slower for 128 cores, and 0.09s faster (or 0.0019%) faster for 256 cores.

8.5.2 Jacobi sum test

Continuing, the results of the Jacobi sum testin 33 are as follows:

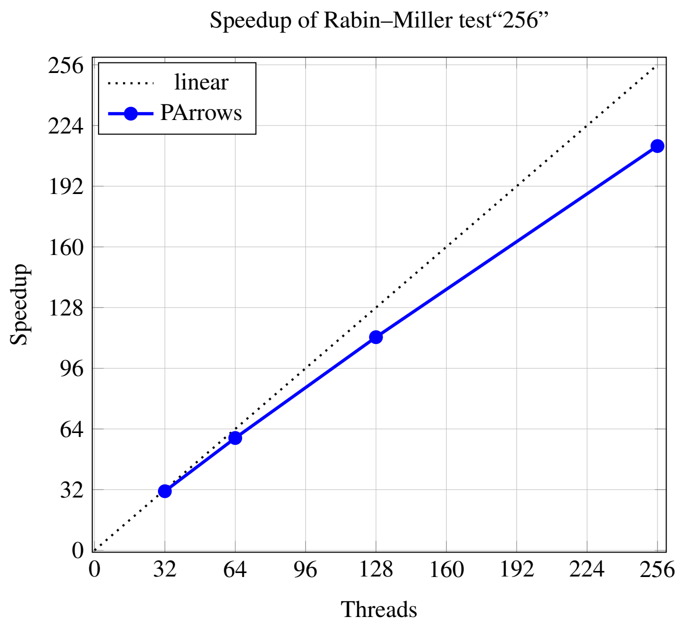
FIXME: redo this test with 2^{44497}

8.5.3 Gentleman

Next is the Gentlemanbenchmark. The results of the comparison of vanilla Eden to our PArrows-based version can be found in 34. While we do not see too much speedup for more than 16 cores, we can prove that the difference between the Eden and PArrows version are again marginal. The difference between PArrows and Eden is only significant for 16 and 64 cores where it ran 1.7% and 2.7% slower which corresponds to a real-time difference of 0.12s and 0.13s. For 256 cores PArrows performed 0.2% slower which corresponds to 0.01s overhead.

8.5.4 Sudoku

As the last Benchmark in this paper we present the Sudokuin 35 running in a shared memory setting. Here we see all three SM backends performing similarly again like in the



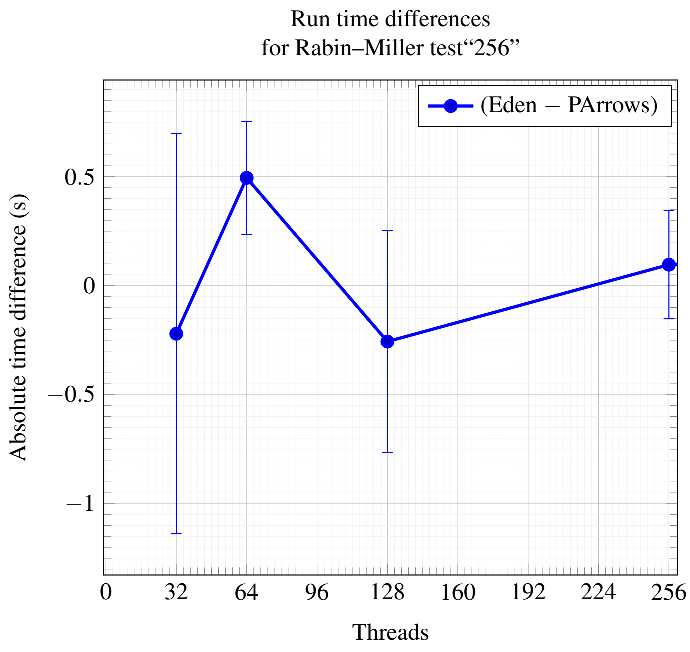


Fig. 32. Parallel performance of Rabin–Miller teston the Glasgow grid consisting of 256 cores. Input was $2^{44497}-1$, we used 256 tasks. The top plot shows absolute speedup in a distributed memory setting. The closer to linear speedup the better. Time (and hence speedup) measurements for PArrows with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

benchmark missing. in the works benchmark missing. in the works

Fig. 33. Parallel performance of Rabin–Miller teston the Glasgow grid consisting of 256 cores. Input was $2^{1279}-1$, 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 PArrowsCHECKME.

benchmark missing. in the works benchmark missing. in the works

Fig. 34. Parallel performance of Gentlemanon the Glasgow grid consisting of 256 cores. Input was a matrix size of 1024. 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 PArrowsCHECKME.

Rabin–Miller testSM benchmarks in Figs. 35-36. However, we notice that the Multicore backend seems to choke on a bigger input 36. This is due to the benchmark only using *parMap* instead of a chunking variant (however we did not change that for simplicity's sake) and 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 lack behind for $\leqslant$16 threads, but manages to pull ahead noticeably with all 32 threads of the system in use.

9 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are first to represent *parallel* computation with Arrows. Arrows turn out to be a useful tool for composing in parallel programs. We do not have to introduce new monadic types that wrap the computation. Instead, we use Arrows in the same manner

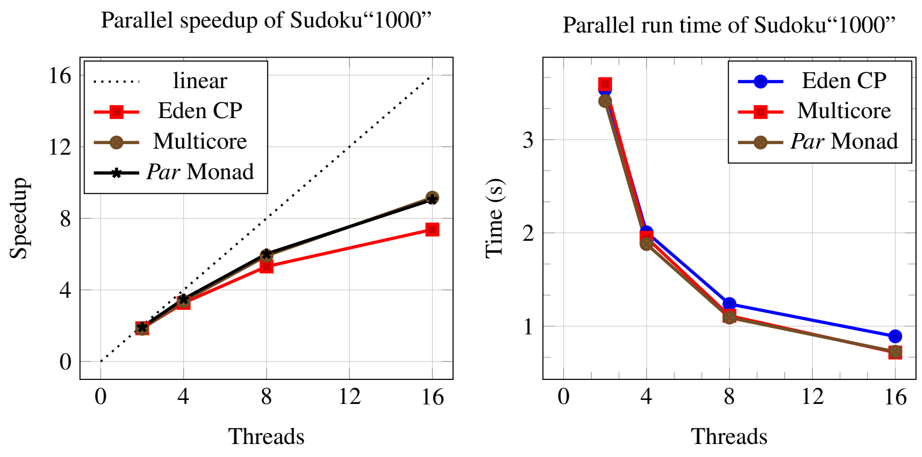


Fig. 35. Relative speedup of Sudokuon a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware and the *parMap* version from the *Par* Monad examples. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores and 32 threads.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

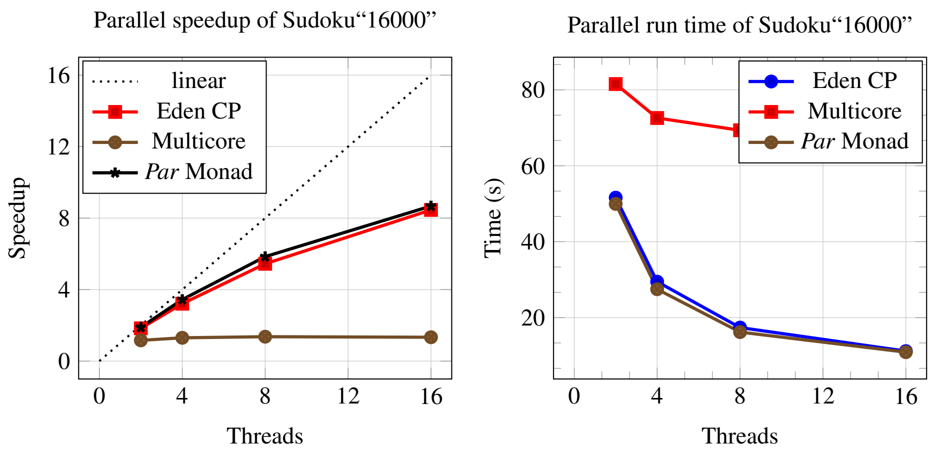


Fig. 36. Relative speedup of Sudokuon a multicore machine. We used the same PArrows-based implementation with different backends on the same hardware and the *parMap* version from the *Par* Monad examples. Measurements were performed on a single node of the Glasgow grid; it has 16 real cores and 32 threads.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

one uses sequential pure functions. This work features multiple parallel backends: the already available parallel Haskell flavours. Parallel Arrows, as presented here, feature an implementation of the *ArrowParallel* typeclass for Multicore 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 do not introduce any notable overhead. The benefit is, however, the greatly increased portability of parallel programs.

9.1 Future Work

Our PArrows DSL can be expanded to further parallel Haskells. More specifically we target HdpH (Maier *et al.*, 2014) for this future extension. HdpH is a modern distributed Haskell that would benefit from our Arrows notation. Further Future-aware versions of Arrow combinators can be defined. Existing combinators could also be improved. We would look into more transparency of our DSL, basically it should infuse as little overhead as possible.

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

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 25Symposium on Implementation and Application of Functional Languages*. IFL '13. ACM.

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

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

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

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

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

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

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

Berthold, Jost, Dieterle, Mischa, Lobachev, Oleg, & Loogen, Rita. (2009a). Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. *Pages 47–55 of:* Großpitsch, K.-E., Henkersdorf, A., Uhrig, S., Ungerer, T., & Hähner, J. (eds), *Workshop on Many-Cores at ARCS '09 – 22International 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 6Workshop 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.

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

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), *12International 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).

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 10ACM 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.

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

Hughes, John. (2005). *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: 19IEEE 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.

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, 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, 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.

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 5Conference on Computing Frontiers*. CF '08. ACM.

Russo, Alejandro, Claessen, Koen, & Hughes, John. (2008). A library for light-weight information-flow security in Haskell. *Pages 13–24 of: Proceedings of the 1ACM 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 lef out for brevity. We start with the *second* combinator from HughesArrows, which is a mirrored version of *first*, which is for example used in the definition of $***$:

```
second :: Arrow arr ⇒ arr a b → arr (c,a) (c,b)
second f = arr swap >>> first f >>> arr swap
  where swap (x,y) = (y,x)
```

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

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

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

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

Fig. A 2. The definition of *foldl* over Arrows.

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

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

Fig. 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 → arr b c → arr (Either a b) c
```

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

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

Fig. B 1. The actual default implementation of Multicore's *ArrowParallel*.

```
listApp :: (ArrowChoice arr, ArrowApply arr) ⇒ [arr a b] → arr [a] [b]
listApp fs = (arr $ λas → (fs, as)) >>> zipWithArr app
```

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

B Omitted Function Definitions

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

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

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

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

Figures B 2–B 5 show the definitions and a visualizations of two parallel *map* variants, defined using *parEvalN* and its lazy counterpart.

Arrow versions of Eden's *shuffle*, *unshuffle* and the definition of *takeEach* are in Figure B 7. Similarly, Figure B 8 contains the definition of arrow versions of Eden's *lazy* and *rightRotate* utility functions. Fig. B 9 contains Eden's definition of *lazyzip3* together with the utility functions *uncurry3* and *threetotwo*. The full definition of *farmChunk* is in Figure B 10. Eden definition of *ring* skeleton is in Figure B 11. It follows Loogen2012.

The *parEval2* skeleton is defined in Figure B 12. We start by transforming the (a, c) input into a two-element list [*Either a c*] by first tagging the two inputs with *Left* and *Right* and wrapping the right element in a singleton list with *return* so that we can combine them with *arr* (*uncurry* (:)). Next, we feed this list into a parallel arrow running on two instances of *f* +++ *g* as described above. After the calculation is finished, we convert the resulting [*Either b d*] into ([*b*], [*d*]) with *arr partitionEithers*. The two lists in this tuple

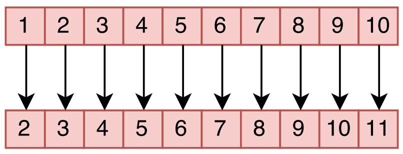


Fig. B 2. Schematic depiction of *parMap*.

$$
\begin{aligned}
&parMap :: (ArrowParallel\ arr\ a\ b\ conf) \Rightarrow conf \rightarrow (arr\ a\ b) \rightarrow (arr\ [a]\ [b])\\
&parMap\ conf\ f = parEvalN\ conf\ (repeat\ f)
\end{aligned}
$$

Fig. B 3. Definition of parMap.

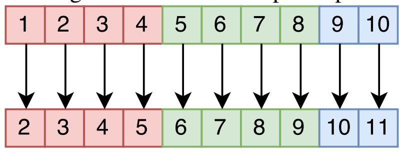


Fig. B 4. Schematic depiction of *parMapStream*.

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

Fig. B 5. Definition of *parMapStream*.

$$
\begin{aligned}
&\mathbf{data}\ RemoteData\ a = RD\ \{rd :: RD\ a\}\\
&\mathbf{instance}\ (Trans\ a) \Rightarrow Future\ RemoteData\ a\ \mathbf{where}\\
&\quad put = arr\ (\lambda a \rightarrow RD\ \{rd = release\ a\})\\
&\quad get = arr\ rd \ggg arr\ fetch
\end{aligned}
$$

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

contain only one element each by construction, so we can finally just convert the tuple to (b,d) in the last step. Furthermore, Fig. B 13 contains the ommited 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.

C Syntactic Sugar

Finally, we also give the definitions for some syntactic sugar for PArrows, namely $\ast\ast\ast$ and &&&. For basic arrows, we have the $\ast\ast\ast$ 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 $\ast\ast\ast$ 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):

$$
\begin{aligned}
&(|\ast\ast\ast|) :: (ArrowChoice\ arr, ArrowParallel\ arr\ (Either\ a\ c)\ (Either\ b\ d)\ ())) \Rightarrow\\
&\quad arr\ a\ b \rightarrow arr\ c\ d \rightarrow arr\ (a,c)\ (b,d)\\
&(|\ast\ast\ast|) = parEval2\ ()
\end{aligned}
$$

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

$$
\begin{array}{l}
\mathit{shuffle} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{arr}\ [[a]]\ [a] \\
\mathit{shuffle} = \mathit{arr}\ (\mathit{concat} \circ \mathit{transpose}) \\
\mathit{unshuffle} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{Int} \rightarrow \mathit{arr}\ [a]\ [[a]] \\
\mathit{unshuffle}\ n = \mathit{arr}\ (\lambda xs \rightarrow [\mathit{takeEach}\ n\ (\mathit{drop}\ i\ xs) \mid i \leftarrow [0\mathinner{..}n-1]]) \\
\mathit{takeEach} :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
\mathit{takeEach}\ n\ [\,] = [\,] \\
\mathit{takeEach}\ n\ (x:xs) = x : \mathit{takeEach}\ n\ (\mathit{drop}\ (n-1)\ xs)
\end{array}
$$

Fig. B 7. Definitions of *shuffle*, *unshuffle*, *takeEach*.

$$
\begin{array}{l}
\mathit{lazy} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{arr}\ [a]\ [a] \\
\mathit{lazy} = \mathit{arr}\ (\lambda{\sim}(x:xs) \rightarrow x : \mathit{lazy}\ xs) \\
\mathit{rightRotate} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow \mathit{arr}\ [a]\ [a] \\
\mathit{rightRotate} = \mathit{arr}\ \$\ \lambda \mathit{list} \rightarrow \mathbf{case}\ \mathit{list}\ \mathbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad xs \rightarrow \mathit{last}\ xs : \mathit{init}\ xs
\end{array}
$$

Fig. B 8. Definitions of *lazy* and *rightRotate*.

$$
\begin{array}{l}
(|\&\&\&|) :: (\mathit{ArrowChoice}\ \mathit{arr}, \mathit{ArrowParallel}\ \mathit{arr}\ (\mathit{Either}\ a\ a)\ (\mathit{Either}\ b\ c)\ ()) \Rightarrow \\
\quad \mathit{arr}\ a\ b \rightarrow \mathit{arr}\ a\ c \rightarrow \mathit{arr}\ a\ (b,c) \\
(|\&\&\&|)\ f\ g = (\mathit{arr}\ \$\ \lambda a \rightarrow (a,a)) \ggg f\ |{*}{*}{*}|\ g
\end{array}
$$

```
lazyzip3 :: [a] → [b] → [c] → [(a,b,c)]
lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)

uncurry3 :: (a → b → c → d) → (a,(b,c)) → d
uncurry3 f (a,(b,c)) = f a b c

threetotwo :: (Arrow arr) ⇒ arr (a,b,c) (a,(b,c))
threetotwo = arr $ λ∼(a,b,c) → (a,(b,c))
```

Fig. B 9. Definitions of *lazyzip3*, *uncurry3* and *threetotwo*.

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

Fig. B 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
```

Fig. B 11. Eden's definition of the *ring* skeleton.

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

Fig. B 12. Definition of *parEval2*.

$$
\begin{aligned}
&\mathit{prMMTr}\ \mathit{m1}\ \mathit{m2} = [[\mathit{sum}\ (\mathit{zipWith}\ (*)\ \mathit{row}\ \mathit{col}) \mid \mathit{col} \leftarrow \mathit{m2}] \mid \mathit{row} \leftarrow \mathit{m1}]\\
&\mathit{splitMatrix} :: \mathit{Int} \rightarrow \mathit{Matrix} \rightarrow [[\mathit{Matrix}]]\\
&\mathit{splitMatrix}\ \mathit{size}\ \mathit{matrix} = \mathit{map}\ (\mathit{transpose} \circ \mathit{map}\ (\mathit{chunksOf}\ \mathit{size}))\ \$\ \mathit{chunksOf}\ \mathit{size}\ \$\ \mathit{matrix}\\
&\mathit{matAdd} = \mathit{chunksOf}\ (\mathit{dimX}\ x)\ \$\ \mathit{zipWith}\ (+)\ (\mathit{concat}\ x)\ (\mathit{concat}\ y)
\end{aligned}
$$

Fig. B 13. Definition of *prMMTr*, *splitMatrix* and *matAdd*.