*Under consideration for publication in J. Functional Programming*

# *Arrows for Parallel Computations*

Submission ID xxxxxx

## Contents

## Contents

## 1 Introduction

todo, reuse 5.5, and more

blablabla arrows, parallel, haskell.

**Contribution** HIT HERE REALLY STRONG

We wrap parallel Haskells inside of our —ArrowParallel— interface, but why do we aim to abstract parallelism this way and what does this approach do better than the other parallel Haskells?

- **Arrow API benefits**: With the —ArrowParallel— typeclass we do not lose any benefits of using arrows as —parEvalN— is just yet another arrow combinator. The resulting arrow can be used in the same way a potential serial version could be used. This is a big advantage of this approach, especially compared to the monad solutions as we do not introduce any new types. We can just plug in parallel parts into our sequential programs without having to change anything.
- **Abstraction**: With the —ArrowParallel— typeclass, we abstracted all parallel implementation logic away from the business logic. This gives us the beautiful situation of being able to write our code against the interface the typeclass gives us without being bound to any parallel Haskell. So as an example, during development, we can run the code on the simple Multicore version and afterwards deploy it on a cluster by converting it into an Eden version, by just replacing the actual —ArrowParallel— instance.

**Structure** The remaining text is structures as follows. Section 2 briefly introduces known parallel Haskell flavours and gives an overview of Arrows to the reader (Sec. 2.2). Section 3 discusses related work. Section 4 defines Parallel Arrows and presents a basic interface. Section 5 defines Futures for Parallel Arrows, this concept enables better communication. Section 6 presents some basic algorithmic skeletons (parallel —map— with and without load balancing, —map-reduce—) in our newly defined dialect. More advanced ones are showcased in Section 7 (—pipe—, —ring—, —torus—). Section 8 shows the benchmark results. Section 9 discusses future work and concludes.

## 2 Background

### *2.1 Short introduction to parallel Haskells*

There are already several ways to write parallel programs in Haskell. As we will 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 -¿ b— in parallel, as also Figure **??** symbolically shows:

parEvalN :: [a -¿ b] -¿ [a] -¿ [b]

Before we go into detail on how we can use this idea of parallelism for parallel Arrows, as a short introduction to parallelism in Haskell we will now implement —parEvalN— with several different parallel Haskells.

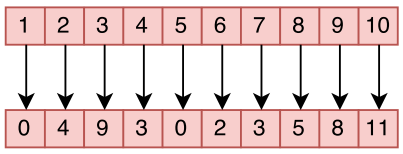


Fig. 1. Schematic illustration of `parEvalN`.

### *2.1.1 Multicore Haskell*

Multicore Haskell (**?**; **?**) is way to do parallel processing found in standard GHC.[1] It ships with parallel evaluation strategies (**?**; **?**) for several types which can be applied with —using :: a -¿ Strategy a -¿ a—. For —parEvalN— this means that we can just apply the list of functions —[a -¿ b]— to the list of inputs —[a]— by zipping them with the application operator —$|.Wethenevaluatethislazylist|[b]|accordingtoa|Strategy[b]|withthe|using :: a- >$ $Strategya- > a|operator.Weconstructthisstrategywith|parList :: Strategya- > Strategy[a]|and|rdeepseq ::$ $NFDataa => Strategya|wherethelatterisastrategywhichevalutestonormalform.Toensurethatprogramsthatuse|parEv$ $a- > b- > b|whichforcesthecompilertonotreordermultiple|parEvalN|computations.Thisisparticularlynecessaryinc$

parEvalN :: (NFData b) =¿ [a -¿ b] -¿ [a] -¿ [b] parEvalN fs as = let bs = zipWith $()fsasin(bs`using`parListrdeepseq)`pseq`bs$

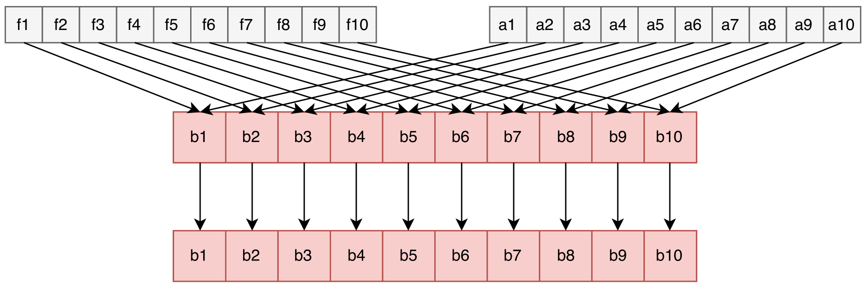


Fig. 2. Dataflow of the Multicore Haskell parEvalN version
Evaluation step explicitly shown?

### *2.1.2 ParMonad*

The —Par— monad[2] introduced by monad$_{par_paper_2}011, isamonaddesignedforcompositionofparallelprograms.$

Our parallel evaluation function —parEvalN— can be defined by zipping the list of —[a -¿ b]— with the list of inputs —[a]— with the application operator —DOLLAR— just like with Multicore Haskell. Then, we map over this not yet evaluated lazy list of results —[b]— with —spawnP :: NFData a =¿ a -¿ Par (IVar a)— to transform them to a list of not yet evaluated forked away computations —[Par (IVar b)]—, which we convert to —Par [IVar b]— with —sequenceA—. We wait for the computations to finish by mapping over

[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.

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

the —IVar b— values inside the —Par— monad with —get—. This results in —Par [b]—. We finally execute this process with —runPar— to finally get —[b]— again.

explain problems with laziness here. Problems with torus

parEvalN :: (NFData b) =¿ [a -¿ b] -¿ [a] -¿ [b] parEvalN fs as = runPar (*sequenceA* map (spawnP) *zipWith*() fs as) ¿¿= mapM get

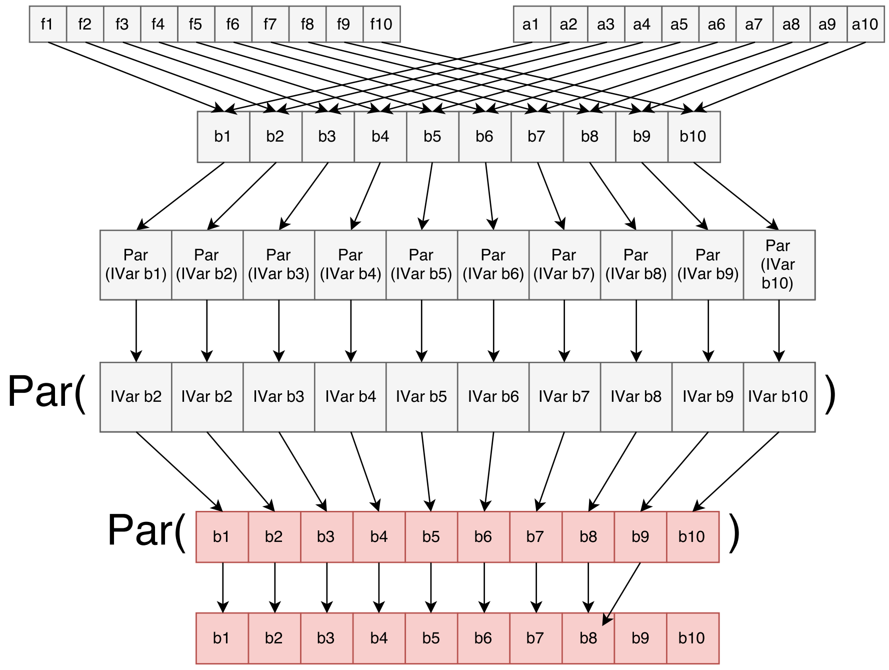


Fig. 3. Dataflow of the Par Monad parEvalN version

### 2.1.3 Eden

Eden (**?**; **?**) is a parallel Haskell for distributed memory and comes with a MPI and a PVM backends.[3] This means that it works on clusters as well so it makes sense to have a Eden-based backend for our new parallel Haskell flavour.

Eden was designed to work on clusters, but with a further simple backend it operates on multicores. 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 (**?**; **?**).

While Eden also 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:

parEvalN :: (Trans a, Trans b) =¿ [a -¿ b] -¿ [a] -¿ [b] parEvalN = spawnF

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

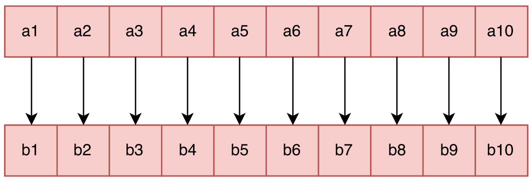


Fig. 4. Dataflow of the Eden parEvalN version

**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 (**?**), the parallel Haskell community mainly utilises the tools Threadscope (**?**) and Eden TraceViewer[4] (**?**). In the next sections we will present some *traces*, the post-mortem process diagrams of Eden processes and their activity.

In a trace, the *x* axis shows the time, the *y* axis enumerates the machines and processes. A trace 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 then is GC. An inactive machine where no processes are started yet, or all are already terminated, is 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.

show example trace or refer to a trace in later figures

### *2.2 Arrows*

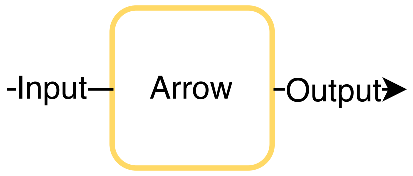


Fig. 5. Schematic depiction of an arrow

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 typeclass:

—arr— is used to lift an ordinary function to an arrow type, similarly to the monadic —return—. The —¿¿¿— 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 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.

The most prominent instances of this interface are regular functions —(-¿)—: instance

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

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)

Fig. 6. Arrow class definition

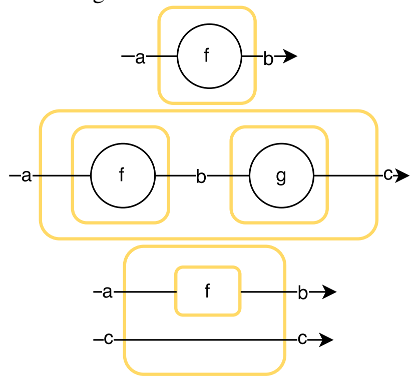
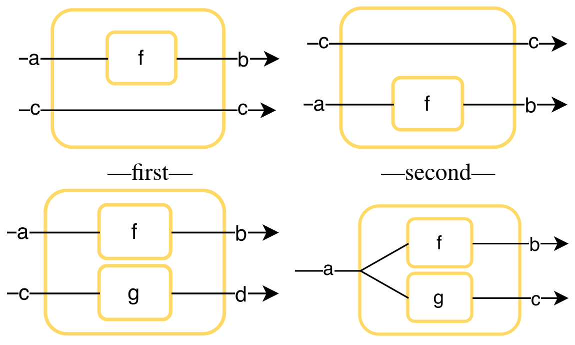


Fig. 7. The schematic depiction of —Arrow— combinators

Arrow (-¿) where arr f = f f ¿¿¿ g = g . f first f = $a,c)->(fa,c)$ and the Kleisli type: 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 (¿ f a ¿¿= g) first f = Kleisli $(a,c)->fa>>=\_->return(b,c))$ With this typeclass in



—first— —second—

—(***)— —(

)—

Fig. 8. Visual depiction of syntactic sugar for Arrows.

place, Hughes also defined some syntactic sugar: the functions —second—, —***— and ——. The mirrored version of —first—, called —second— (Fig. 2.2) is: 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) the —***— combinator that combines —first— and —second— to handle two inputs in one arrow, (Fig.2.2) is defined as (***) :: Arrow arr =¿ arr a b -¿ arr c d -¿ arr (a, c) (b, d) f *** g = first f ¿¿¿ second g and the —— combinator that constructs an arrow which outputs two different values like —***—, but takes only one input (Fig. 2.2) is: () :: Arrow arr =¿ arr a b -¿ arr a c -¿ a a (b, c) f g = arr (¿ (a, a)) ¿¿¿ (f *** g) A short example given by Hughes on how to use this is addition with arrows: add :: Arrow arr =¿ arr a Int -¿ arr a Int -¿ arr a Int add f g = (f  g) ¿¿¿ arr $(u,v)- > u+v)$

The more restrictive interface of arrows (a monad can be *anything*, an arrow is a process of doing something, a *computation*) allows for more elaborate composition and transformation combinators. One of the major problems in parallel computing is composition of parallel processes.

## 3 Related Work

arrows or Arrows?

### *3.1 Parallel Haskells*

Of course, the three parallel Haskell flavours we have presented above: the GpH (**?**; **?**) parallel Haskell dialect and its multicore version (**?**), the —Par— monad (**?**; **?**), and Eden (**?**; **?**) are related to this work. We use these languages as backends: our library can switch from one to other at user's command.

HdpH (**?**; **?**) is an extension of —Par— monad to heterogeneous clusters. LVish (**?**) is a communication-centred extension of —Par— monad. Further parallel Haskell approaches include pH (**?**), research work done on distributed variants of GpH (**?**; **?**; **?**) and low-level Eden implementation (**?**; **?**). Skeleton composition (**?**), communication (**?**), and generation of process networks (**?**) are recent in-focus research topics in Eden. This also includes the definitions of new skeletons (**?**; **?**; **?**; **?**; **?**; **?**; **?**; **?**).

More different approaches include data parallelism (**?**; **?**;

), GPU-based approaches (**?**; **?**), software transactional memory (**?**; **?**). The Haskell–GPU bridge Accelerate (**?**; **?**; **?**) deserves a special mention. Accelerate is completely orthogonal to our approach. marlow2013parallel authored a recent book marlow2013parallel on parallel Haskells.

### *3.2 Algorithmic skeletons*

Algorithmic skeletons were introduced by Cole1989. Early efforts include (**?**; **?**; **?**; **?**; **?**). SkeletonBook consolidated early reports on high-level programming approaches. The effort is ongoing, including topological skeletons (**?**), special-purpose skeletons for computer algebra (**?**; **?**; **?**; **?**), iteration skeletons (**?**). The idea of scscp is to use a parallel Haskell to orchestrate further software systems to run in parallel. dieterle$_h$*orstmeyer*$_l$*oogen*$_b$*erthold*$_2$016*comparethecompositionof*

### *3.3 Arrows*

Arrows were introduced by HughesArrows, basically they are a generalised function arrow —-¿—. Hughes2005 is a tutorial on arrows. Some theoretical details on arrows (**?**; **?**; **?**) are viable. Paterson:2001:NNA:507669.507664 introduced a new notation for arrows. Arrows have applications in information flow research (**?**; **?**; **?**), invertible programming (**?**), and quantum computer simulation (**?**). But perhaps most prominent application of arrows is functional reactive programming (**?**).cite more!

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 a 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.ugh, take care! Huang2007 utilise arrows for parallelism, but strikingly different from our approach. They basically use arrows to orchestrate several tasks in robotics. We propose a general interface for parallel programming, remaining completely in Haskell.

### *3.4 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. Our experiments with our approach in Frege language (which is basically Haskell on the JVM) were quite successful, we were able to use typical Java libraries for parallelism. 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

We have seen what Arrows are and how they can be used as a general interface to computation. In the following section we will discuss how Arrows constitute a general interface not only to computation, but to **parallel computation** as well. We start by introducing the interface and explaining 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*

As we have seen earlier, in its purest form, parallel computation (on functions) can be seen as the execution of some functions —a -¿ b— in parallel, —parEvalN— (Fig. **??**, 1). 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— to host this combinator: 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 normalform vs. normalform). 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—. So we add it to the type signature of the typeclass as well and get —ArrowParallel arr a b conf—: —ArrowParallel arr a b conf— or —ArrowParallel conf arr a b—? class Arrow arr =¿ ArrowParallel arr a b conf where parEvalN :: conf -¿ [arr a b] -¿ arr [a] [b] Note that we don't 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—, as we will see in the implementation of the Multicore and the —Par— Monad backend.

### *4.2 ArrowParallel instances*

#### *4.2.1 Multicore Haskell*

The Multicore Haskell implementation of this class is implemented in a straightforward manner by using —listApp— from Appendix A combined with the —withStrategy :: Strategy a -¿ a -¿ a— and —pseq :: a -¿ b -¿ b— combinators from Multicore Haskell, where —withStrategy— is the same as —using :: a -¿ Strategy a -¿ a— but with flipped parameters. For most cases a fully evaluating version like in Fig. 9 would probably suffice,

instance (NFData b, ArrowApply arr, ArrowChoice arr) =¿ ArrowParallel arr a b () where parEvalN $_f s = listApp fs >>> arr(withStrategy(parList rdeepseq)) arr id >>> arr(uncurry pseq)$

Fig. 9. Fully evaluating ArrowParallel instance for the Multicore Haskell backend

but as the Multicore Haskell interface allows the user to specify the level of evaluation to be done via the —Strategy— interface, we want to the user not to lose this ability because of using our API. We therefore introduce the —Conf a— data-type that simply wraps a —Strategy a— (Fig. **??**). We can't directly use the —Strategy a— type here as GHC (at least in the versions used for development in this paper) does not allow type synonyms in type class instances: data Conf a = Conf (Strategy a) To get our configurable —ArrowParallel— instance, we simply unwrap the strategy and pass it to —parList— like in the fully evaluating version (Fig. 10).

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. 10. Configurable ArrowParallel instance for the Multicore Haskell backend

#### *4.2.2 —Par— Monad*

introduce a newcommand for par-monad, "arrows", "parrows" and replace all mentions to them to ensure uniform typesetting The —Par— monad implementation (Fig. 11) makes use of Haskells laziness and —Par— monad's —spawnP :: NFData a =¿ a -¿ 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 —listApp— (Fig. **??**) and then fork the computation away with —arr spawnP— inside a —zipWith-Arr— 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 (¿¿= 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 $_f s = (arr$ -¿ (fs, as)) ¿¿¿ zipWithArr (app ¿¿¿ arr spawnP) ¿¿¿ arr sequenceA ¿¿¿ arr (¿¿= mapM get) ¿¿¿ arr runPar

Fig. 11. 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) We don't do it here for aesthetic resons, though. For now, we just implement —ArrowParallel— for normal functions: instance (Trans a, Trans b) =¿ ArrowParallel (-¿) a b conf where parEvalN $_f sas = spawnF f sas$ 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 $parEvalNconf(map(Kleislif)->f)fs)) >>> (Kleisli$ 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—*

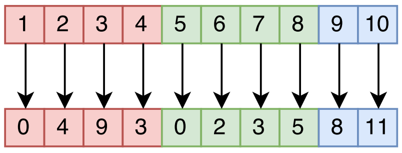


Fig. 12. Schematic depiction of parEvalNLazy

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 . (+)) [1..]— or just because we need the arrows evaluated in chunks. —parEvalNLazy— (Fig. 12, 13) 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— (Fig. **??**). The resulting —[[b]]— is lastly converted into —[b]— with —arr concat—.

parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) =¿ conf -¿ ChunkSize -¿ [arr a b] -¿ (arr [a] [b]) parEvalNLazy conf chunkSize fs = arr (chunksOf chunkSize) ¿¿¿ listApp fchunks ¿¿¿ arr concat where fchunks = map (parEvalN conf) $chunksOfchunkSizefs$
Fig. 13. Definition of —parEvalNLazy—.

### *4.3.2 Heterogenous tasks*

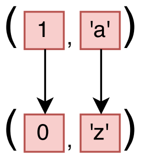


Fig. 14. Schematic depiction of —parEval2—.

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 (Fig. 14, 15) 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.

We start by transforming the —(a, c)— input into a 2-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 2 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 1 element each by construction, so we can finally just convert the tuple to —(b, d)— in the last step.

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. 15. Definition of parEval2

#### *4.3.3 Syntactic Sugar*

For basic arrows, we have the —***— combinator (Fig. 2.2, **??**) 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 —parstar— (Fig. **??**) with the use of —parEval2—, but for this we require a backend which has an implementation that does not require any configuration (hence the —()— as the —conf— parameter): (—***—) :: (ArrowChoice arr, ArrowParallel arr (Either a c) (Either b d) ())) =¿ arr a b -¿ arr c d -¿ arr (a, c) (b, d) (—***—) = parEval2 () We define the parallel —parand— in a similar manner to its sequential pendant —— (Fig. 2.2): (——) :: (ArrowChoice arr, ArrowParallel arr (Either a a) (Either b c) ()) =¿ arr a b -¿ arr a c -¿ arr a (b, c) (——) f g = (arr $>(a,a)) >>> f|***|g$

## 5 Futures

Consider the parallel arrow combinator in Fig. 16 In a distributed environment, the result-

someCombinator :: (Arrow arr) =¿ [arr a b] -¿ [arr b c] -¿ arr [a] [c] someCombinator fs1 fs2 = parEvalN () fs1 ¿¿¿ rightRotate ¿¿¿ parEvalN () fs2

Fig. 16. An example parallel Arrow combinator without Futures

ing arrow of this combinator first evaluates all —[arr a b]— in parallel, sends the results back to the master node, rotates the input once and then evaluates the —[arr b c]— in parallel to then gather the input once again on the master node. Such situations arise, in scientific computations when the data distributed across the nodes needs to be transposed. A concrete example is 2D FFT computation (**?**; **?**).

While the example in Fig. 16 could be rewritten into only one —parEvalN— call by directly wiring the arrows properly together, this example 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 17. This can become a serious bottleneck for larger amount of data and number of processes [showcases][as, ]Berthold2009-fft.

This motivates for an approach that allows the nodes to communicate directly with each other. Thankfully, Eden, the distributed parallel Haskell we have used in this paper so far, already ships with the concept of —RD— (remote data) that enables this behaviour (**?**; **?**).

But as we want code written against our API to be implementation agnostic, we have to wrap this context. We do this with the —Future— typeclass (Fig. 18). Since —RD— is only a type synonym for communication type that Eden uses internally, we have to use some wrapper classes to fit that definition, though, as seen in Appendix in Fig. B 1. This is due to the same reason we had to introduce a wrapper for —Strategy a— in the Multicore Haskell implementation of —ArrowParallel— in Section 4.2.1.

For our Par Monad and Multicore Haskell backends, we can simply use —MVar—s (**?**) (Fig. 19), because we have shared memory in a single node and don't require Eden's sophisticated communication channels. explain MVars

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— in-

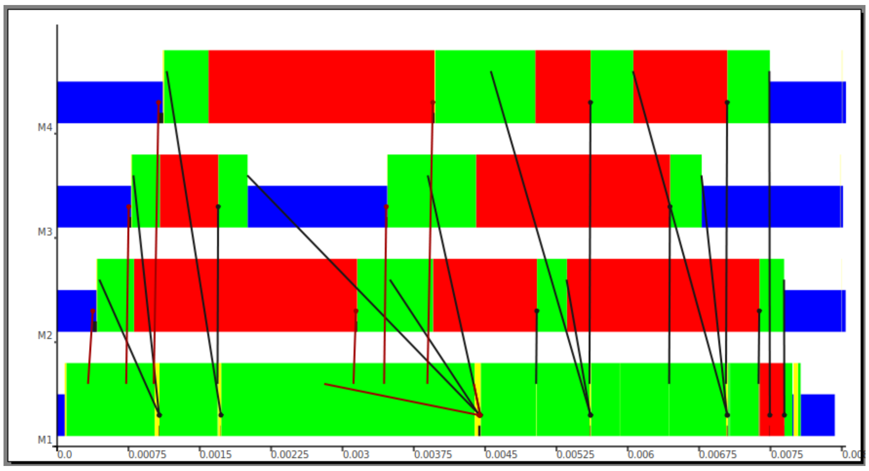


Fig. 17. Communication between 4 threads without Futures. All communication goes through the master node. Each bar represents one process. Black lines between processes represent communication. Colors: blue ≙ idle, green ≙ running, red ≙ blocked, yellow ≙ suspended.
more practical and heavy-weight example! fft (I have the code)?

class Future fut a @—@ a -¿ fut where put :: (Arrow arr) =¿ arr a (fut a) get :: (Arrow arr) =¿ arr (fut a) a
Fig. 18. Definition of the Future typeclass

stances, the latter are only needed in Eden. Because —MVar— already ships with a —NFData— instance, we only have to supply two simple instances for our —RemoteData— type: instance NFData (RemoteData a) where rnf = rnf . rd instance Trans (RemoteData a) The —Trans— instance does not have any functions declared as the default implementation suffices here.

Going back to our communication example we can use this —Future— concept in order to enable direct communications between the nodes in the following way: In a distributed environment, this gives us a communication scheme with messages going through the master node only if it is needed - similar to what is shown in the trace in Fig. 21.Fig. 3 is not really clear. Do Figs 2-3 with a lot of load?

## 6 Map-based Skeletons

Now we have developed Parallel Arrows far enough to define some algorithmic skeletons useful to an application programmer.

### *6.1 Parallel map*

The —parMap— skeleton (Fig. 22, 23) 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 get an arrow —arr [a] [b]—. Just like —parEvalN—, —parMap— is 100% strict.

- NOINLINE putUnsafe - putUnsafe :: a -¿ MVar a putUnsafe a = unsafePerformIO
$domVar <- newEmptyMVarputMVarmVarareturnmVar$
instance (NFData a) =¿ Future MVar a where put = arr putUnsafe get = arr takeMVar ¿¿¿ arr unsafePerformIO

Fig. 19. MVar instance of the Future typeclass for the Par Monad and Multicore Haskell backends

someCombinator :: (Arrow arr) =¿ [arr a b] -¿ [arr b c] -¿ arr [a] [c] someCombinator fs1 fs2 = parEvalN () (map (¿¿¿ put) fs1) ¿¿¿ rightRotate ¿¿¿ parEvalN () (map (get ¿¿¿) fs2)

Fig. 20. The combinator from Fig. 16 in parallel

### *6.2 Lazy parallel map*

As —parMap— (Fig. 22, 23) is 100% strict it has the same restrictions as —parEvalN— compared to —parEvalNLazy—. So it makes sense to also have a —parMapStream— (Fig. 24, 25) which behaves like —parMap—, but uses —parEvalNLazy— instead of —parEvalN—.

### *6.3 Statically load-balancing parallel map*

A —parMap— (Fig. 22, 23) 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 —farm— (Fig. 26, 27) that equally distributes the workload over —numCores— workers (if numCores is greater than the actual processor count, the fastest processor(s) to finish will get more tasks) seems useful. The definitions of helper functions —unshuffle—, —takeEach—, —shuffle— (shown in Appendix) originate from an Eden skeleton[5].

### *6.4 The —farmChunk— Skeleton*

Since a —farm— (Fig. 26, 27) is basically just —parMap— with a different work distribution, it is, again, 100% strict. So we can define —farmChunk— (Fig. 28, B 2) which uses —parEvalNLazy— instead of —parEvalN—. It is basically the same definition as for —farm—, with —parEvalN— replaced with —parEvalNLazy—, as Appendix shows.

### *6.5 Map and reduce*

A simple —map———reduce— can be written like in Figure 29. Notice that the performance of the —¿¿¿— combinator is essential for the performance of the skeleton. A definitive version would use Futures.

it appears STRANGE. are the data really left alone and noded after map and taken from there by reduce? It makes sense only is no communication through master takes place. ELSE: CUT!

this requires some work. Either change this to use futures or cut, yes.

now rewritten as motivation for futures. maybe still cut?

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

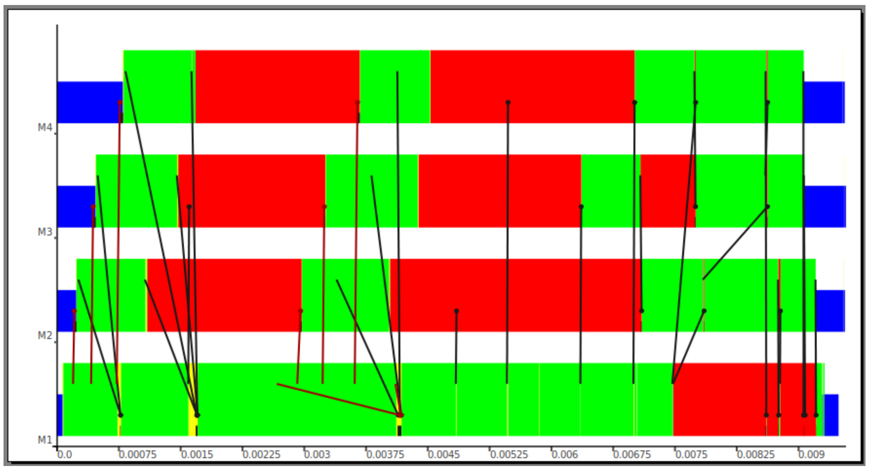


Fig. 21. Communication between 4 threads with Futures. Other than in Fig. 17, threads communicate directly (black lines between the bars) instead of always going through the master node (bottom bar).

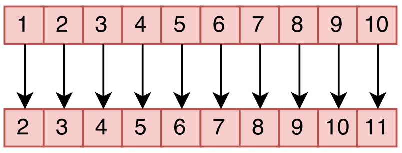


Fig. 22. Schematic depiction of —parMap—.

## 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] with more predefined skeletons, among them a —pipe—, a —ring—, and a —torus— implementations (**?**). 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.

### *7.1 Parallel pipe*

The parallel —pipe— skeleton is semantically equivalent to folding over a list —[arr a a]— of arrows with —¿¿¿—, but does this in parallel, meaning that the arrows do not have to reside on the same thread/machine. We implement this skeleton using the —ArrowLoop— typeclass 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 das kann man hier so lassen, oder?sicherlich! loop (arr $(a,b)->(b,a:b))$) 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. 30) by plugging in the parallel evaluation call

[6] Available on Hackage: https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html.

parMap :: (ArrowParallel arr a b conf) =¿ conf -¿ (arr a b) -¿ (arr [a] [b]) parMap conf f = parEvalN conf (repeat f)

Fig. 23. Definition of parMap

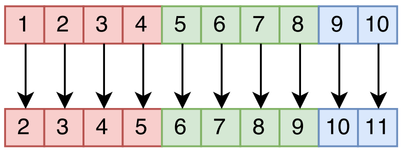


Fig. 24. Schematic depiction of parMapStream

—parEvalN 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 get the final definition of —pipe—: pipe :: (ArrowLoop arr, ArrowParallel 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))

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. 31).

Note that extensive use of this combinator 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. Nonetheless, we can define a parallel piping operator —parcomp— (Fig. 32, which is semantically equivalent to —¿¿¿— similarly to other parallel syntactic sugar from Section 4.3.3.

### *7.2 Ring skeleton*

Eden comes with a ring skeleton[7] (Fig. 33) implementation that allows the computation of a function —[i] -¿ [o]— with a ring of nodes that communicate in a ring topology 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 the neighbour of each node. This data is sent over direct communication channels, so called remote data. We depict it in Appendix, Fig. B 3.

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) -¿ arr a c—. In each iteration we start by rotating the intermediary input from the nodes —[fut r]— with —second (rightRotate ¿¿¿ lazy)—. Similarly to the —pipe— (Fig. 30, **??**), we have to feed the intermediary input into our —lazy— arrow here, or the evaluation would hang.meh, wording The reasoning is explained by Loogen2012:

> *Note that the list of ring inputs —ringIns— is the same as the list of ring outputs —ringOuts— rotated by one element to the right using the auxiliary function*

[7] Available on Hackage: https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html

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)

Fig. 25. Definition of —parMapStream—.

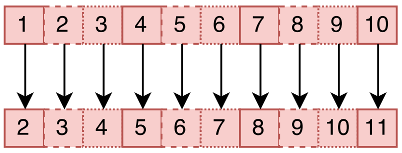


Fig. 26. Schematic depiction of a —farm—, a statically load-balanced —parMap—.

*—rightRotate—. Thus, the program would get stuck without the lazy pattern, because the ring input will only be produced after process creation and process creation will not occur without the first input.*

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. 34). This combinator can, for example, be used to calculate the shortest paths in a graph using Warshall's algorithm. let's do it?

### *7.3 Torus skeleton*

If we take the concept of a ring from 7.2 one dimension further, we get a torus (Fig. 35, 36). Every node sends ands receives data from horizontal and vertical neighbours in each communication round. With our Parallel Arrows we re-implement the —torus— combinator[8] from Eden—yet again with the help of the —ArrowLoop— typeclass.

Similar to the —ring—, we once again start by rotating the input, 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, however, is 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 —parMap conf (ptorus f)— which yields us —[(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 ::

[8] Available on Hackage: https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html.

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
unshuffle :: (Arrow arr) =¿ Int -¿ arr [a] [[a]] unshuffle n = arr (-¿ [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)
shuffle :: (Arrow arr) =¿ arr [[a]] [a] shuffle = arr (concat . transpose)
Fig. 27. The definition of —farm—.

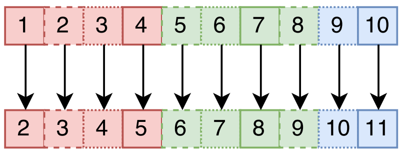


Fig. 28. Schematic depiction of farmChunk

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 Loogen2012 showed the matrix multiplication using the Gentleman algorithm Gentleman1978. Their instantiation of the skeleton —node-function— can be adapted as shown in Fig. 37. If we compare the trace from a call using our arrow definition of the torus (Fig. 38) with the Eden version (Fig. 39) we can see that the behaviour of the arrow version is comparable.much more details on this!

## 8 Benchmarks

[thick, scale=0.7] [ xbar, xmin=0, bar width=0.4cm, width=12cm, height=7cm, enlarge y limits=0.1, ylabel=amou

[thick, scale=0.7] [ xbar, xmin=0, bar width=0.4cm, width=12cm, height=7cm, enlarge y limits=0.1, ylabel=am

## 9 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are the first ones to represent parallel computation with arrows.that strange arrows-based robot interaction paper from 1993 or so! clearly discuss in related work

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 use arrows just like regular sequential pure functions. This work features multiple parallel backends: the already available parallel Haskell flavours. Parallel Arrows feature an implementation of the —ArrowParallel— interface for Multicore Haskell, —Par— Monad, and Eden. With our approach parallel programs can be ported across these flavours with no effort. Performancewise, Parallel Arrows are on par with existing parallel Haskells, as they do not introduce any notable overhead.PROOFS. Many proofs in benchmarks!

ArrowApply (or equivalent) are needed because we basically want to be able to produce intermediary results, this is by definition of the parallel evaluation combinators

Remove websites from citations, put them into footnotes!

Parrows + accelerate = love? Metion port to Frege. Mention the Par monad troubles.

parMapReduceDirect :: (ArrowParallel arr [a] b conf, ArrowApply arr, ArrowChoice arr) =¿ conf -¿ ChunkSize -¿ arr a b -¿ arr (b, b) b -¿ b -¿ arr [a] b parMapReduceDirect conf chunkSize mapfn foldfn neutral = arr (chunksOf chunkSize) ¿¿¿ parMap conf (mapArr mapfn ¿¿¿ foldlArr foldfn neutral) ¿¿¿ foldlArr foldfn neutral

Fig. 29. Definition of parMapReduceDirect

pipeSimple :: (ArrowLoop arr, ArrowParallel arr a a conf) =¿ conf -¿ [arr a a] -¿ arr a a pipeSimple conf fs = loop (arr snd (arr (uncurry (:) ¿¿¿ lazy) ¿¿¿ parEvalN conf fs)) ¿¿¿ arr last lazy :: (Arrow arr) =¿ arr [a] [a] lazy = arr ( (x:xs) -¿ x : lazy xs)

Fig. 30. A first implementation of the —pipe— skeleton expressed with Parallel Arrows. Note that the use of —lazy— 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 —parEvalN—.

### *9.1 Future Work*

Our PArrows interface can be expanded to futher parallel Haskells. More specifically we target HdpH (**?**) a modern distributed Haskell that would benefit from our Arrows notation. Future-aware special versions of Arrow combinated can be extended and further improved. We would look into more transparency of the API, it should basically infuse as little overhead as possible.

Of course, definitions of further skeletons are viable and needed. We are looking into more experiences with seamless porting of parallel PArrow-based programs across the backends.

Accelerate (**?**) is not related to our approach. It would be interesting to see a hybrid of both APIs.

replace API with DSL globally?

## A Utility Functions

To be able to go into detail on parallel arrows, we introduce some utility combinators first, that will help us later: —map—, —foldl— and —zipWith— on arrows.

The —mapArr— combinator (Fig. A 1) lifts any arrow —arr a b— to an arrow —arr [a] [b]— (**?**). Similarly, we can also define —foldlArr— (Fig. A 2) that lifts any arrow —arr (b, a) b— with a neutral element —b— to —arr [a] b—.

Finally, with the help of —mapArr— (Fig. A 1), we can define —zipWithArr— 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 $as,bs) -> zipWith(,)asbs) >>> mapArrf$ These combinators make use of the —ArrowChoice— type class which provides the CHECK! 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]—. listApp :: (ArrowChoice arr, ArrowApply arr) =¿ [arr a b] -¿ arr [a] [b] listApp fs = (arr $- > (fs,as)) >>> zipWithArrapp$ Note that this additionally makes use of the —ArrowApply— typeclass that allows us to evaluate arrows with —app :: arr (arr a b, a) c—.

I swapped the type classes here: pipe2 :: (ArrowLoop arr, ArrowChoice arr, Future fut (([a], [b]), [c]), ArrowParallel 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 $((a,b),c)- > ((c,a),b))$
Fig. 31. Definition of —pipe2—.

swapped type classes (—¿¿¿—) :: (ArrowLoop arr, ArrowChoice arr, Future fut (([a], [b]), [c]), ArrowParallel arr (fut (([a], [b]), [c])) (fut (([a], [b]), [c])) ()) =¿ arr a b -¿ arr b c -¿ arr a c (—¿¿¿—) = pipe2 ()
Fig. 32. Definition of —parcomp—.

## B Omitted Funtion Definitions

We have omitted some function definitions in the main text for brevity, and redeem this here. We warp Eden's build-in Futures in PArrows as in Figure B 1. The full definition of —farmChunk— is in Figure B 2. Eden definition of —ring— skeleton following Loogen2012 is in Figure B 3.

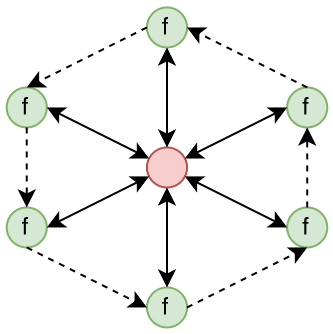


Fig. 33. Schematic depiction of the ring skeleton

ring :: (ArrowLoop arr, Future fut r, ArrowParallel 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) ¿¿¿ parMap conf (second get ¿¿¿ f ¿¿¿ second put) ¿¿¿ arr unzip)
rightRotate :: (Arrow arr) =¿ arr [a] [a] rightRotate = arr >
case list of [] -¿ [] xs -¿ last xs : init xs lazy :: (Arrow arr) =¿ arr [a] [a] lazy = arr ( (x:xs) -¿ x : lazy xs)
Fig. 34. Final definition of the —ring— skeleton.

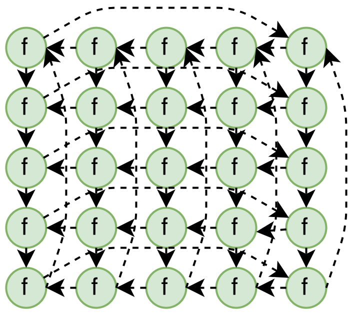


Fig. 35. Schematic depiction of the —torus— skeleton.

swapped type classes torus :: (ArrowLoop arr, ArrowChoice arr, ArrowApply arr, Future fut a, Future fut b, ArrowParallel 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 ¿¿¿ parMap conf (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 ( (c, a, b) -¿ (c, get a, get b)) ¿¿¿ f ¿¿¿ arr ( (d, a, b) -¿ (d, put a, put b))
uncurry3 :: (a -¿ b -¿ c -¿ d) -¿ (a, (b, c)) -¿ d uncurry3 f (a, (b, c)) = f a b c
lazyzip3 :: [a] -¿ [b] -¿ [c] -¿ [(a, b, c)] lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)
threetotwo :: (Arrow arr) =¿ arr (a, b, c) (a, (b, c)) threetotwo = arr $(a,b,c)->(a,(b,c))$
Fig. 36. Definition of the —torus— skeleton.

nodefunction :: Int -¿ ((Matrix, Matrix), [Matrix], [Matrix]) -¿ ([Matrix], [Matrix], [Matrix])
nodefunction n ((bA, bB), rows, cols) = ([bSum], bA:nextAs , bB:nextBs) where bSum = foldl' matAdd (matMult bA bB) (zipWith matMult nextAs nextBs) nextAs = take (n-1) rows nextBs = take (n-1) cols
Fig. 37. Adapted —nodefunction— for matrix multiplication with the —torus— from Fig. 36

more nodes!!!

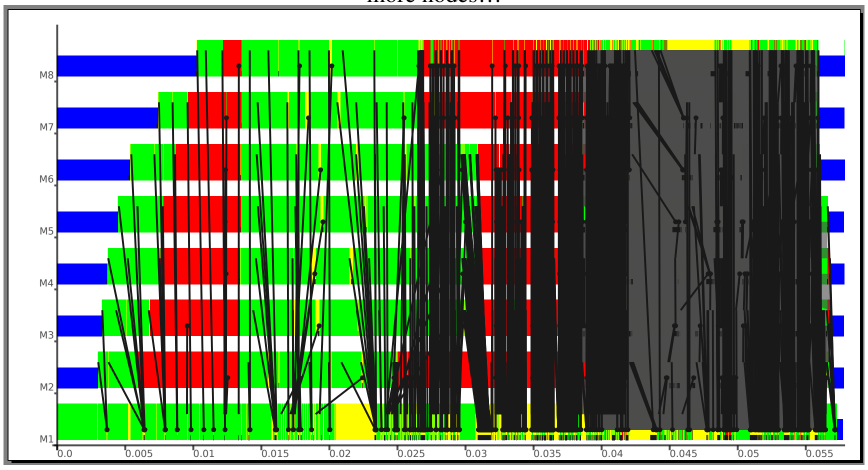


Fig. 38. Matrix Multiplication with a torus (Parrows)

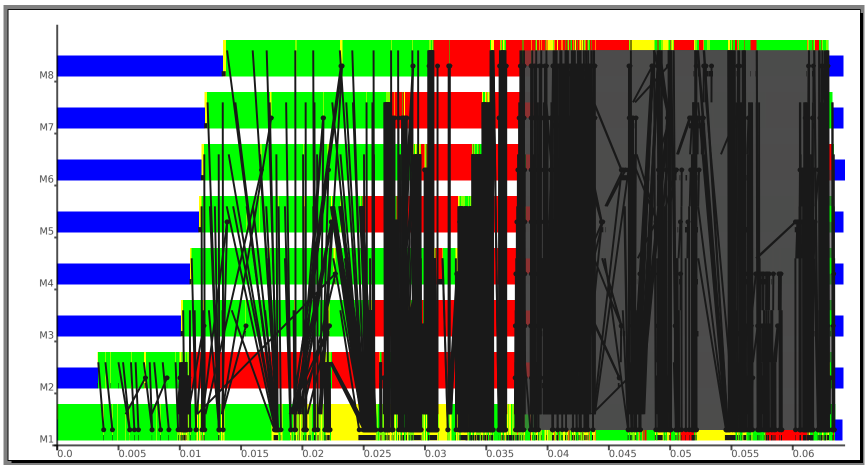


Fig. 39. Matrix Multiplication with a torus (Eden)

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 (¿ (b, a)) ¿¿¿ f ¿¿¿ arr (foldlArr f)) ¿¿¿ app)
Fig. A 2. The definition of —foldl— over Arrows.

data RemoteData a = RD  rd :: RD a
instance (Trans a) =¿ Future RemoteData a where put = arr (¿ RD  rd = release a ) get = arr rd ¿¿¿ arr fetch
Fig. B 1. —RD—-based —RemoteData— version of —Future— for the Eden backend.

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 2. 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 3. Eden's definition of the —ring— skeleton.