

Arrows for Parallel Computations

MARTIN BRAUN

University Bayreuth, 95440 Bayreuth, Germany

PHIL TRINDER

Glasgow University, Glasgow, G12 8QQ, Scotland
and OLEG LOBACHEV

University Bayreuth, 95440 Bayreuth, Germany

Abstract

Arrows form a general interface for computation and pose therefore as an alternative to monads for API design. In the paper Hughes describes how arrows are a generalization of monads and how they are not as restrictive. We express parallelism using this concept. We define an Arrows-based interface for parallel computations and implement it using multiple parallel Haskell.

This new way of writing parallel programs has a benefit of being portable across flavours of parallel Haskell used. With our framework we define several parallel skeletons. We also introduce some syntactic sugar to provide parallelism-aware arrow combinators similar to the traditional ones. Benchmarks shows that our framework does not induce too much overhead performance-wise.

Contents

1	Introduction	2
2	Background	2
2.1	Short introduction to parallel Haskell	2
3	Related Work	5
3.1	Arrows	5
4	Parallel Arrows	7
4.1	The ArrowParallel typeclass	8
4.2	Multicore Haskell	8
4.3	ParMonad	9
4.4	Eden	9
4.5	Impact of parallel Arrows	10
4.6	Extending the Interface	10
4.7	Lazy parEvalN	10
4.8	Heterogenous tasks	11
4.9	Syntactic Sugar	12
5	Futures	12
6	Map-based Skeletons	14
6.1	Parallel map	14
6.2	Lazy parallel map	15

2	<i>M. Braun, P. Trinder and O. Lobachev</i>	
6.3	Statically load-balancing parallel map	15
6.4	farmChunk	15
6.5	parMapReduce	16
7	Topological Skeletons	16
7.1	Parallel pipe	16
7.2	Ring skeleton	18
7.3	Torus skeleton	19
8	Benchmarks	22
9	Conclusion	22
A	Utility Functions	24

Contents

1 Introduction

blablabla arrows, parallel, haskell.

Contribution HIT HERE REALLY STRONG

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. 3.1). Section 3 discusses related work. Section 4 defines Parallel Arrows and presents a basic interface. Section 5 defines futures for Parallel Arrows. Futures are a further concept that helps to define more advanced interfaces. Section 6 presents some basic algorithmic skeletons in our newly defined dialect. More advanced ones are showcased in Section 7. Section 8 shows the benchmark results. Section 9 concludes.

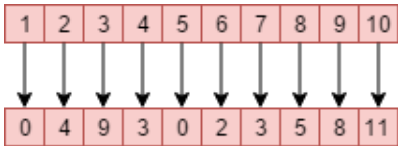
2 Background

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 $a \rightarrow b$ in parallel:

```
parEvalN :: [a -> b] -> [a] -> [b]
```

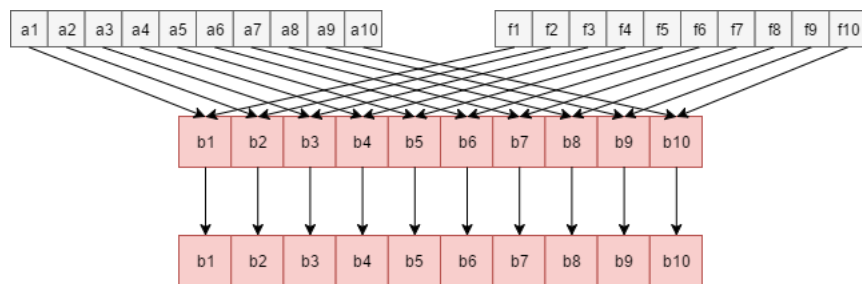


We will now implement parEvalN with the different parallel Haskells.

2.1.1 Multicore Haskell

Multicore Haskell (Marlow et al., 2009) is a 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 `$`. This lazy list `[b]` is then according to a Strategy `[b]` by the using `:: a -> Strategy a -> a` operator. This strategy can be constructed with `parList :: Strategy a -> Strategy [a]` and `rdeepseq :: NFData a => Strategy a` where the latter is a strategy which evaluates to normal form. **explain usage of pseq**

```
1 parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
2 parEvalN fs as = let bs = zipWith ($) fs as in
3   (bs 'using' parList rdeepseq) 'pseq' bs
```



2.1.2 ParMonad

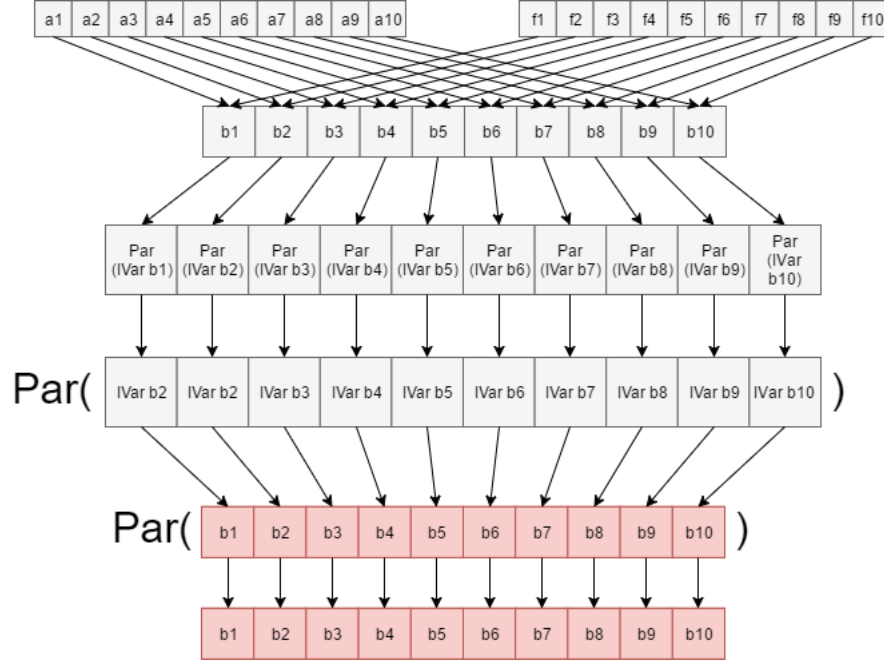
The Par monad² introduced by Marlow et al. (2011), is a monad designed for composition of parallel programs.

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 `$` 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 the `IVar b`'s inside the `Par` monad with `get`. This results in `Par [b]`. We finally execute this process with `runPar` to finally get `[b]` again.

```
1 parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
2 parEvalN fs as = runPar $
3   (sequenceA $ map (spawnP) $ zipWith ($) fs as) >>= mapM get
```

¹ See <https://hackage.haskell.org/package/parallel-3.2.1.0>.

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



2.1.3 Eden

Eden (Loogen et al., 2005; Loogen, 2012a) 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 (Berthold et al., 2009b; ?).

While Eden also comes with a monad PA for parallel evaluation, it also ships with a completely functional interface that includes

`spawnF :: (Trans a, Trans b) => [a -> b] -> [a] -> [b]`.

This allows us to define `parEvalN` directly:

```

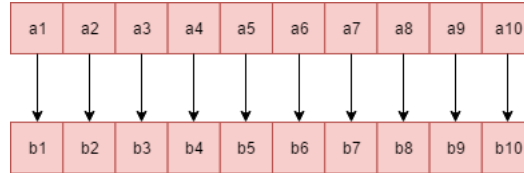
1 parEvalN :: (Trans a, Trans b) => [a -> b] -> [a] -> [b]
2 parEvalN = spawnF

```

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

Arrows for Parallel Computations

5

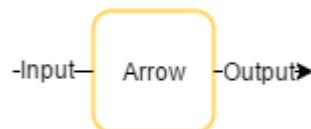


Eden TraceViewer To comprehend the efficiency and the lack thereof in a parallel program, an inspection of its execution is extremely helpful. While some large-scale solutions exist (Geimer et al., 2010), the parallel Haskell community mainly utilises the tools Threadscope (Wheeler and Thain, 2009) and Eden TraceViewer⁴ (Berthold and Loogen, 2007). 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’, i.e. 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 communication from one process to another is represented with a black arrow. A stream of communications, e.g. a transmitted list is shows as a dark shading between sender and receiver processes.

3 Related Work

3.1 Arrows



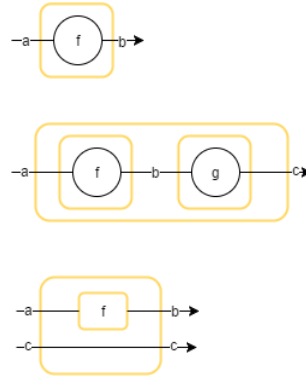
Arrows were introduced by Hughes (Hughes, 2000) as a general interface for computation. An arrow `arr a b` represents a computation that converts an input `a` to an output `b`. This is defined in the arrow typeclass:

⁴ See on hackage for the last available version of Eden TraceViewer. There was an effort to implement the TraceViewer using modern web technologies (?).

```

1 class Arrow arr where
2   arr :: (a -> b) -> arr a b
3
4
5
6   (>>>) :: arr a b -> arr b c -> arr a c
7
8
9
10
11  first :: arr a b -> arr (a,c) (b,c)

```



`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 save input across arrows.

The most prominent instances of this interface are regular functions `(->)`,

```

1 instance Arrow (->) where
2   arr f = f
3   f >>> g = g . f
4   first f = \ (a, c) -> (f a, c)

```

and the Kleisli type

```

1 data Kleisli m a b = Kleisli { run :: a -> m b }
2
3 instance Monad m => Arrow (Kleisli m) where
4   arr f = Kleisli $ return . f
5   f >>> g = Kleisli $ \a -> f a >>= g
6   first f = Kleisli $ \ (a,c) -> f a >>= \b -> return (b,c)

```

With this typeclass in place, Hughes also defined some syntactic sugar: The mirrored version of `first`, called `second`,

```

1 second :: Arrow arr => arr a b -> arr (c, a) (c, b)
2 second f = arr swap >>> first f >>> arr swap
3 where swap (x, y) = (y, x)

```

the `***` combinator which combines `first` and `second` to handle two inputs in one arrow,

```

1 (***) :: Arrow arr => arr a b -> arr c d -> arr (a, c) (b, d)
2 f *** g = first f >>> second g

```

Arrows for Parallel Computations

7

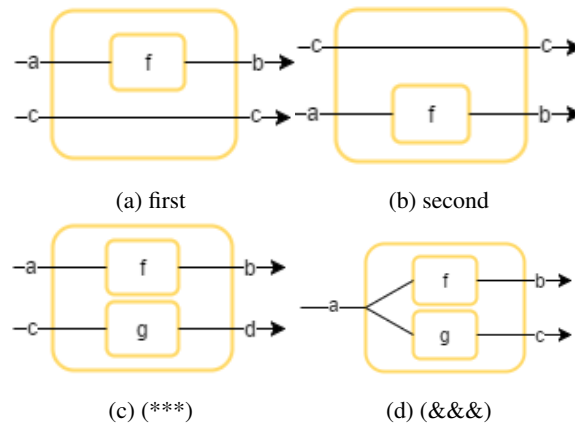


Figure 1: Syntactic sugar for arrows.

and the `&&&` combinator that constructs an arrow which outputs 2 different values like `**`, but takes only one input.

```
1 (&&&) :: Arrow arr => arr a b -> arr a c -> a a (b, c)
2 f &&& g = arr (\a -> (a, a)) >>> (f ** g)
```

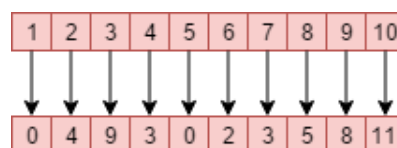
A short example given by Hughes on how to use this is add over arrows:

```
1 add :: Arrow arr => arr a Int -> arr a Int -> arr a Int
2 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.

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 existing 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 $a \rightarrow b$ in parallel:

```
1 parEvalN :: [a -> b] -> [a] -> [b]
```

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`, but does parallel instead of serial evaluation).

```
1 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 have to introduce the new typeclass `ArrowParallel` to host this combinator.

```
1 class Arrow arr => ArrowParallel arr a b where
2   parEvalN :: [arr a b] -> arr [a] [b]
```

Sometimes parallel Haskells require or allow for additional configuration parameters, e.g. an information about the execution environment or the level of evaluation (weak-head 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.

```
1 class Arrow arr => ArrowParallel arr a b conf where
2   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 `ParMonad` backend.

4.2 Multicore Haskell

The `Multicore Haskell` implementation of this class is implemented in a straightforward manner by using `listApp` from chapter 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.

```
1 instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
2   ArrowParallel arr a b () where
3   parEvalN _ fs =
4     listApp fs >>>
5     arr (withStrategy (parList rdeepseq)) &&& arr id >>>
6     arr (uncurry pseq)
```


For most cases this fully evaluating version 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, 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`. 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.

```
1 data Conf a = Conf (Strategy a)
```

We simply unwrap the strategy out of this wrapper and get the following (configurable) `ArrowParallel` instance:

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

4.3 ParMonad

The `ParMonad` implementation makes use of Haskell's laziness and `ParMonad`'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 `app` and then fork the computation away with `arr spawnP` inside a `zipWithArr` 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 `IVars` inside the `ParMonad` 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]`.

```
1 instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
2   ArrowParallel arr a b conf where
3   parEvalN _ fs =
4     (arr $ \as -> (fs, as)) >>>
5     zipWithArr (app >>> arr spawnP) >>>
6     arr sequenceA >>>
7     arr (>>= mapM get) >>>
8     arr runPar
```

4.4 Eden

For the Multicore and `ParMonad` implementation 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 like

```
1 class (Arrow arr) => ArrowUnwrap arr where
2   arr a b -> (a -> b)
```

we don't do it here, for aesthetic reasons. For now, we just implement `ArrowParallel` for normal functions

```
1 instance (Trans a, Trans b) => ArrowParallel (->) a b conf where
2   parEvalN _ fs as = spawnF fs as
```

and the Kleisli type.

```
1 instance (Monad m, Trans a, Trans b, Trans (m b)) =>
2   ArrowParallel (Kleisli m) a b conf where
3   parEvalN conf fs =
4     (arr $ parEvalN conf (map (\(Kleisli f) -> f) fs)) >>>
5     (Kleisli $ sequence)
```

4.5 Impact of parallel Arrows

We have seen that we can wrap parallel Haskells inside of the `ArrowParallel` interface, but why do we 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.

4.6 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.

4.7 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.

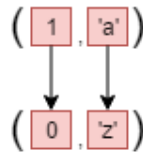
map (`arr . (+)`) `[1..]` or just because we need the arrows evaluated in chunks. `parEvalNLazy` fixes this. It works by first chunking the input from `[a]` to `[[a]]` with the given `ChunkSize` in `arr (chunksOf chunkSize)`. These chunks are then fed into a list `[arr [a] [b]]` of parallel arrows created by feeding chunks of the passed `ChunkSize` into the regular `parEvalN` by using `listApp`. The resulting `[[b]]` is lastly converted into `[b]` with `arr concat`.

```

1 parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) =>
2   conf -> ChunkSize -> [arr a b] -> (arr [a] [b])
3 parEvalNLazy conf chunkSize fs =
4   arr (chunksOf chunkSize) >>>
5   listApp fchunks >>>
6   arr concat
7   where fchunks = map (parEvalN conf) $ chunksOf chunkSize fs

```

4.8 Heterogenous tasks



We have only talked about the parallelization arrows of the same type until now. But sometimes we want to parallelize heterogenous types as well. However, we can implement such a `parEval2` combinator 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.

```

1 parEval2 :: (ArrowChoice arr,
2   ArrowParallel arr (Either a c) (Either b d) conf) =>
3   conf -> arr a b -> arr c d -> arr (a, c) (b, d)
4 parEval2 conf f g =
5   arr Left *** (arr Right >>> arr return) >>>
6   arr (uncurry (:)) >>>
7   parEvalN conf (replicate 2 (f +++ g)) >>>
8   arr partitionEithers >>>
9   arr head *** arr head

```

4.9 Syntactic Sugar

For basic arrows, we have the `***` combinator which allows us to combine two arrows `arr a b` and `arr c d` into an arrow `arr (a, c) (b, d)` which does both computations at once. This can easily be translated into a parallel version with `parEval2`, but for this we require a backend which has an implementation that does not require any configuration (hence the `()` as the `conf` parameter in the following code snippet).

```
1 (|***|) :: (ArrowChoice arr, ArrowParallel arr (Either a c) (Either b d) ()) =>
2   arr a b -> arr c d -> arr (a, c) (b, d)
3 (|***|) = parEval2 ()
```

We define the parallel `|&&&|` in a similar manner to its sequential prototype.

```
1 (|&&&|) :: (ArrowChoice arr, ArrowParallel arr (Either a a) (Either b c) ()) =>
2   arr a b -> arr a c -> arr a (b, c)
3 (|&&&|) f g = (arr $ \a -> (a, a)) >>> f |***| g
```

5 Futures

Consider the following parallel arrow combinator:

```
1 someCombinator :: (Arrow arr) => [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 = parEvalN () fs1 >>> rightRotate >>> parEvalN () fs2
```

In a distributed environment, the resulting arrow of this combinator first evaluates all `[arr a b]` in parallel, sends the results back to the master node, rotates the input once and then evaluates the `[arr b c]` in parallel to then gather the input once again on the master node. Such situations arise, e.g. in scientific computations when the data distributed across the nodes needs to be transposed. A concrete example is 2D FFT computation (Gorlatch and Bischof, 1998; Berthold et al., 2009c).

While this 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 2. This can become a serious bottleneck for larger amount of data and number of processes (showcases Berthold et al., 2009a, as, e.g.). 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 (Alt and Gorlatch, 2003; Dieterle et al., 2010). 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:

```
1 class Future fut a | a -> fut where
2   put :: (Arrow arr) => arr a (fut a)
3   get :: (Arrow arr) => arr (fut a) a
```

As RD is only type synonym for communication type that Eden uses internally, we have to use some wrapper classes to fit that definition, though:

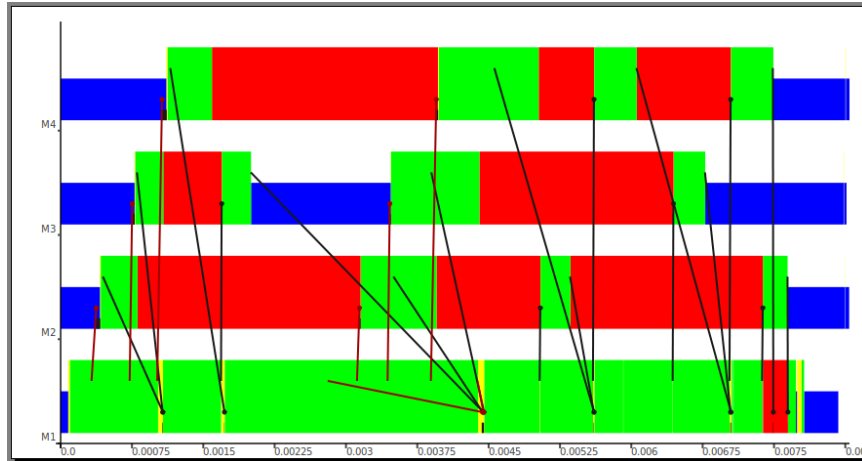


Figure 2: Communication between 4 threads without Futures

```

1 data RemoteData a = RD { rd :: RD a }
2
3 instance (Trans a) => Future RemoteData a where
4   put = arr (\a -> RD { rd = release a })
5   get = arr rd >>> arr fetch

```

For ParMonad and Multicore we can simply use MVars because we have shared memory in a single node:

```

1 {-# NOINLINE putUnsafe #-}
2 putUnsafe :: a -> MVar a
3 putUnsafe a = unsafePerformIO $ do
4   mVar <- newEmptyMVar
5   putMVar mVar a
6   return mVar
7
8 instance (NFData a) => Future MVar a where
9   put = arr putUnsafe
10  get = arr takeMVar >>> arr unsafePerformIO

```

To fit the ArrowParallel instances we gave earlier, we also have to give the necessary NFData and Trans instances - the latter only being needed in Eden. We need this implementation for our RemoteData wrapper,

```

1 instance NFData (RemoteData a) where
2   rnf = rnf . rd
3 instance Trans (RemoteData a)

```

while MVar already has a suitable NFData implementation.

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:

```

1 someCombinator :: (Arrow arr) => [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 =
3   parEvalN () (map (>>> put) fs1) >>>
4   rightRotate >>>
5   parEvalN () (map (get >>>) fs2)

```

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. 3.

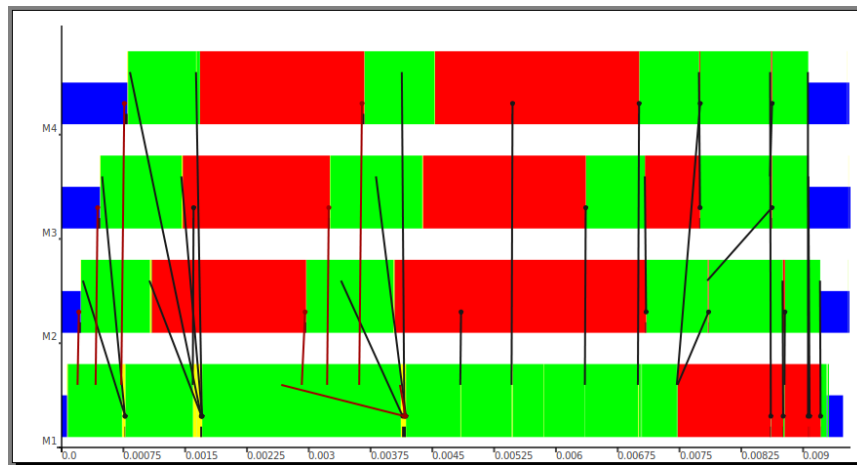
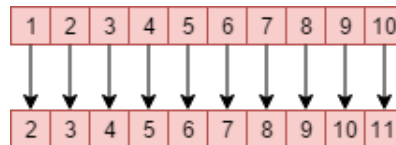


Figure 3: Communication between 4 threads with Futures

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

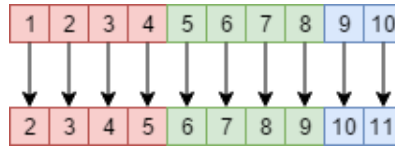


`parMap` is probably the most common skeleton for parallel programs. We can implement it with `ArrowParallel` by repeating an arrow `arr a b` and then passing it into `parEvalN` to get an arrow `arr [a] [b]`. Just like `parEvalN`, `parMap` is 100 % strict.

```

1 parMap :: (ArrowParallel arr a b conf) =>
2   conf -> (arr a b) -> (arr [a] [b])
3 parMap conf f = parEvalN conf (repeat f)

```

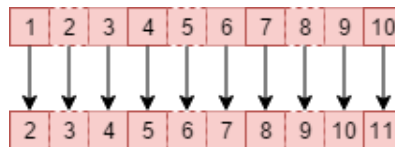
6.2 Lazy parallel map

As `parMap` is 100% strict it has the same restrictions as `parEvalN` compared to `parEvalNLazy`. So it makes sense to also have a `parMapStream` which behaves like `parMap`, but uses `parEvalNLazy` instead of `parEvalN`.

```

1 parMapStream :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) =>
2   conf -> ChunkSize -> arr a b -> arr [a] [b]
3 parMapStream conf chunkSize f = parEvalNLazy conf chunkSize (repeat f)

```

6.3 Statically load-balancing parallel map

`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 farm that equally distributes the workload over `numCores` workers (if `numCores` is greater than `actualProcessorCount`, the fastest processor(s) to finish will get more tasks) seems useful.

```

1 farm :: (ArrowParallel arr a b conf,
2   ArrowParallel arr [a] [b] conf, ArrowChoice arr) =>
3   conf -> NumCores -> arr a b -> arr [a] [b]
4 farm conf numCores f =
5   unshuffle numCores >>>
6   parEvalN conf (repeat (mapArr f)) >>>
7   shuffle

```

6.4 farmChunk

As `farm` is basically just `parMap` with a different work distribution, it is, again, 100% strict. So we define `farmChunk` which uses `parEvalNLazy` instead of `parEvalN` like this:

```

1 farmChunk :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
2   ArrowChoice arr, ArrowApply arr) =>
3   conf -> ChunkSize -> NumCores -> arr a b -> arr [a] [b]

```

```

4 farmChunk conf chunkSize numCores f =
5   unshuffle numCores >>>
6   parEvalNLazy conf chunkSize (repeat (mapArr f)) >>>
7   shuffle

```

6.5 *parMapReduce*

– this does not completely adhere to Google’s definition of Map Reduce as it – the mapping function does not allow for "reordering" of the output – The original Google version can be found at <https://de.wikipedia.org/wiki/MapReduce>

```

1 parMapReduceDirect :: (ArrowParallel arr [a] b conf,
2   ArrowApply arr, ArrowChoice arr) =>
3   conf -> ChunkSize -> arr a b -> arr (b, b) b -> b -> arr [a] b
4 parMapReduceDirect conf chunkSize mapfn foldfn neutral =
5   arr (chunksOf chunkSize) >>>
6   parMap conf (mapArr mapfn >>> foldArr foldfn neutral) >>>
7   foldArr foldfn neutral

```

7 Topological Skeletons

Even though many algorithms can be expressed by parallel maps, some problems require more sophisticated skeletons. The Eden library leverages this problem and already comes with more predefined skeletons, among them a pipe, a ring and a torus implementation (Loogen, 2012b; ede, b). These seem like reasonable candidates to be ported to our arrow based parallel Haskell to showcase that we can express such 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 loop like computations. For example this

```

1 loop (arr (\(a, b) -> (b, a:b)))

```

, which is the same as

```

1 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 by plugging in the parallel evaluation call `parEvalN conf fs` inside the second argument of `&&&` and then only picking the first element of the resulting list with `arr last`:


```

1 pipeSimple :: (ArrowLoop arr, ArrowParallel arr a a conf) =>
2   conf -> [arr a a] -> arr a a
3 pipeSimple conf fs =
4   loop (arr snd &&&
5     (arr (uncurry (:)) >>> lazy) >>> parEvalN conf fs)) >>>
6   arr last

```

where lazy is defined as:

```

1 lazy :: (Arrow arr) => arr [a] [a]
2 lazy = arr (\ ~(x:xs) -> x : lazy xs)

```

Note that here 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.

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.

```

1 pipe :: (ArrowLoop arr, ArrowParallel arr (fut a) (fut a) conf,
2   Future fut a) =>
3   conf -> [arr a a] -> arr a a
4 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

```

1 pipe2 :: (ArrowLoop arr, ArrowChoice arr,
2   ArrowParallel arr (fut ([a], [b]), [c]) (fut ([a], [b]), [c])) conf,
3   Future fut ([a], [b]), [c]) =>
4   conf -> arr a b -> arr b c -> arr a c
5 pipe2 conf f g =
6   (arr return &&& arr (const [])) &&& arr (const []) >>>
7   pipe conf (replicate 2 (unify f g)) >>>
8   arr snd >>>
9   arr head
10  where
11    unify :: (ArrowChoice arr) =>
12      arr a b -> arr b c -> arr ([a], [b]), [c] ([a], [b]), [c]
13    unify f g =
14      (mapArr f *** mapArr g) *** arr (\_ -> []) >>>
15      arr (\((a, b), c) -> ((c, a), b))

```

Note that extensive use of this combinator over pipe with a hand-written combination datatype 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 |>>>| which is semantically equivalent to >>> in a similar manner to the other parallel syntactic sugar from Section 4.9:

```

1 (|>>>|) :: (ArrowLoop arr, ArrowChoice arr,
2   ArrowParallel arr (fut ([a], [b]), [c]) (fut ([a], [b]), [c])) ()

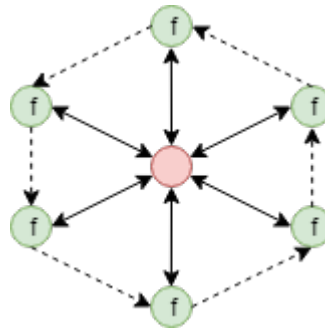
```

```

3 Future fut (([a], [b]), [c]) =>
4   arr a b -> arr b c -> arr a c
5 (|>>>|) = pipe2 ()

```

7.2 Ring skeleton



Eden comes with a ring skeleton implementation that allows the computation of a function $[i] \rightarrow [o]$ with a ring of nodes that communicate in a ring topology with each other. Its input is a node function $i \rightarrow r \rightarrow (o, r)$ in which r serves as the intermediary output that gets sent to the neighbour of each node. This data is sent over direct communication channels (remote data). They define it as (ede, b)

```

1 ringSimple :: (Trans i, Trans o, Trans r) =>
2   (i -> r -> (o,r))
3   -> [i] -> [o]
4 ringSimple f is = os
5   where (os,ringOuts) = unzip (parMap (toRD $ uncurry f)
6                                   (zip is $ lazy ringIns))
7         ringIns = rightRotate ringOuts
8
9 toRD :: (Trans i, Trans o, Trans r) =>
10  ((i, r) -> (o,r))
11  -> ((i, RD r) -> (o, RD r))
12 toRD f (i, ringIn) = (o, release ringOut)
13   where (o, ringOut) = f (i, fetch ringIn)
14
15 rightRotate :: [a] -> [a]
16 rightRotate [] = []
17 rightRotate xs = last xs : init xs

```

We can rewrite its 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, we have to feed the intermediary input into our lazy arrow here, or the evaluation would hang. The reasoning is explained by Loogen (2012b): ‘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 `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 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.

This gives us the following complete definition for the `ring` combinator:

```

1 ring :: (ArrowLoop arr, Future fut r,
2   ArrowParallel arr (i, fut r) (o, fut r) conf) =>
3   conf ->
4   arr (i, r) (o, r) ->
5   arr [i] [o]
6 ring conf f =
7   loop (second (rightRotate >>> lazy) >>>
8     arr (uncurry zip) >>>
9     parMap conf (second get >>> f >>> second put) >>>
10    arr unzip)

```

with `rightRotate`:

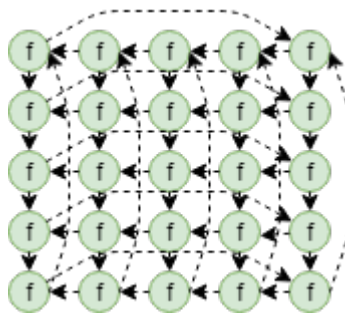
```

1 rightRotate :: (Arrow arr) => arr [a] [a]
2 rightRotate = arr $ \ list -> case list of
3   [] -> []
4   xs -> last xs : init xs

```

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 one dimension further, we get a torus. Every node sends and receives data from horizontal and vertical neighbours in each communication round.

With our parallel Arrows we implement a torus combinator 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 2 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` 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)]`.

The complete definition of the torus combinator is:

```

1 torus :: (ArrowLoop arr, ArrowChoice arr, ArrowApply arr,
2   ArrowParallel arr (c, fut a, fut b) (d, fut a, fut b) conf,
3   Future fut a, Future fut b) =>
4   conf -> arr (c, a, b) (d, a, b) -> arr [[c]] [[d]]
5 torus conf f =
6   loop (second ((mapArr rightRotate >>> lazy) ***
7     (arr rightRotate >>> lazy)) >>>
8     arr (uncurry3 (zipWith3 lazyzip3)) >>>
9     (arr length >>> arr unshuffle) &&&
10    (shuffle >>> parMap conf (ptorus f)) >>>
11    app >>>
12    arr (map unzip3) >>> arr unzip3 >>> threetotwo)

```

with `uncurry3`, `lazyzip3`, and `threetotwo` from the Appendix, and

```

1 ptorus :: (Arrow arr, Future fut a, Future fut b) =>
2   arr (c, a, b) (d, a, b) ->
3   arr (c, fut a, fut b) (d, fut a, fut b)
4 ptorus f =
5   arr (\ ~(c, a, b) -> (c, get a, get b)) >>> f >>>
6   arr (\ ~(d, a, b) -> (d, put a, put b))

```

As an example of using this skeleton Loogen (2012b) showed the matrix multiplication using the Gentleman algorithm (Gentleman, 1978). Adapting this nodefunction to our Arrow API gives us:

```

1 nodefunction :: Int ->
2   ((Matrix, Matrix), [Matrix], [Matrix]) ->
3   ([Matrix], [Matrix], [Matrix])
4 nodefunction n ((bA, bB), rows, cols) =

```

```

5  ([bSum], bA:nextAs , bB:nextBs)
6  where bSum =
7      foldl ' matAdd (matMult bA bB) (zipWith matMult nextAs nextBs)
8      nextAs = take (n-1) rows
9      nextBs = take (n-1) cols

```

If we compare the trace from a call using our arrow definition of the torus (Fig. 4) with the Eden version (Fig. 5) we can see that the behaviour of the arrow version is comparable.

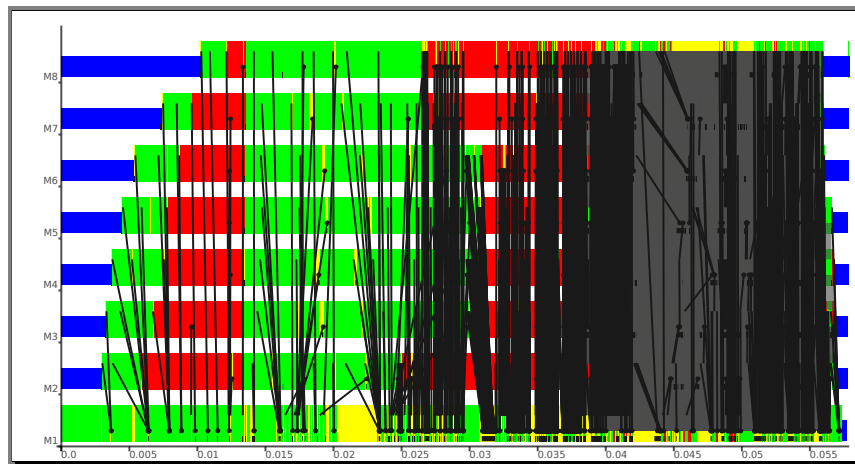


Figure 4: Matrix Multiplication with a torus (Parrows)

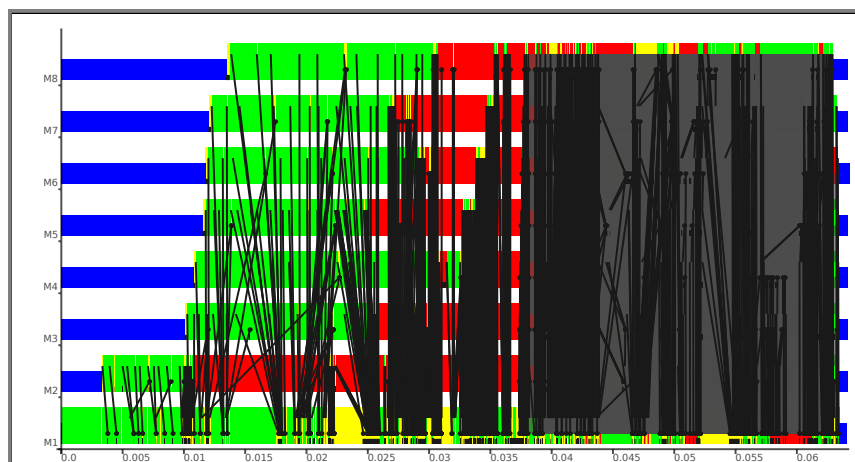
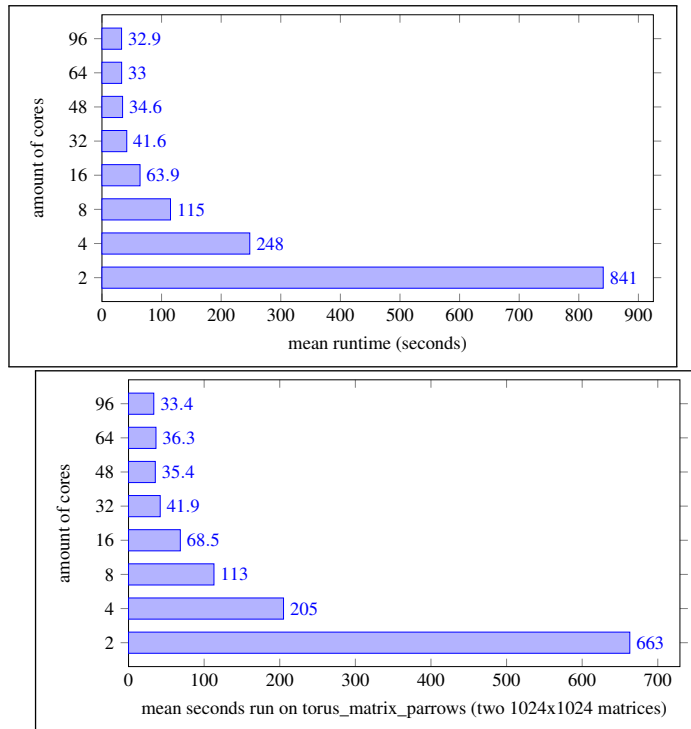


Figure 5: Matrix Multiplication with a torus (Eden)

8 Benchmarks



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.

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. Performance-wise, Parallel Arrows are on par with existing parallel Haskell, as they do not introduce any notable overhead.

Bibliography

- Eden skeletons' `control.parallel.eden.map` package source code, a. URL <https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/src/Control-Parallel-Eden-Map.html>. [Accessed on 02/12/2017].
- Eden skeletons' `control.parallel.eden.topology` package source code, b. URL <https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html>. [Accessed on 03/17/2017].

- M. Alt and S. Gorlatch. Future-Based RMI: Optimizing compositions of remote method calls on the Grid. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003*, LNCS 2790, pages 682–693. Springer-Verlag, Aug. 2003.
- J. Berthold and R. Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In *ParCo '07. Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, 2007.
- J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. In V. Malyskin, editor, *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83. Springer-Verlag, 2009a. Extended version in (Berthold et al., 2009d).
- J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. In K.-E. Großpitsch, A. Henkersdorf, S. Uhrig, T. Ungerer, and J. Hähner, editors, *Workshop on Many-Cores at ARCS '09 – 22nd International Conference on Architecture of Computing Systems 2009*, pages 47–55. VDE-Verlag, 2009b.
- J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. In V. Malyskin, editor, *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83. Springer-Verlag, 2009c. Extended version in (Berthold et al., 2009d).
- J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. Technical Report bi2009-2, Philipps-Universität Marburg, Fachbereich 12 – Mathematik und Informatik, 2009d.
- M. Dieterle, T. Horstmeyer, and R. Loogen. Skeleton composition using remote data. In M. Carro and R. Peña, editors, *PADL 2010: 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of LNCS, pages 73–87. Springer-Verlag, 2010.
- M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6), 2010.
- W. M. Gentleman. Some complexity results for matrix computations on parallel processors. *Journal of the ACM*, 25(1):112–115, 1978. .
- S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4), 1998.
- J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67 – 111, 2000. ISSN 0167-6423. . URL <http://www.sciencedirect.com/science/article/pii/S0167642399000234>.
- J. Hughes. *Programming with Arrows*, pages 73–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31872-9. . URL http://dx.doi.org/10.1007/11546382_2.
- R. Loogen. *Eden – Parallel Functional Programming with Haskell*, pages 142–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012a. ISBN 978-3-642-32096-5. . URL http://dx.doi.org/10.1007/978-3-642-32096-5_4.
- R. Loogen. Eden – parallel functional programming with haskell. 2012b. URL <http://www.mathematik.uni-marburg.de/~eden/paper/edenCEFP.pdf>.

- R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005. Special Issue on Functional Approaches to High-Performance Parallel Programming.
- S. Marlow. Sample code for ‘Parallel and Concurrent Programming in Haskell’. URL <https://github.com/simonmar/parconc-examples>. [Accessed on 01/12/2017].
- S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 44(9):65–78, 2009.
- S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, Sept. 2011. ISSN 0362-1340. . URL <http://doi.acm.org/10.1145/2096148.2034685>.
- K. B. Wheeler and D. Thain. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, 2009.

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 lifts any arrow $\text{arr } a \ b$ to an arrow $\text{arr } [a] \ [b]$ (Hughes, 2005):

```

1 mapArr :: ArrowChoice arr => arr a b -> arr [a] [b]
2 mapArr f =
3   arr listcase >>>
4   arr (const []) ||| (f *** mapArr f >>> arr (uncurry ()))
5   where listcase [] = Left ()
6         listcase (x:xs) = Right (x,xs)

```

Similarly, we can also define **foldlArr** that lifts any arrow $\text{arr } (b, a) \ b$ with a neutral element b to $\text{arr } [a] \ b$:

```

1 foldlArr :: (ArrowChoice arr, ArrowApply arr) => arr (b, a) b -> b -> arr [a] b
2 foldlArr f b =
3   arr listcase >>>
4   arr (const b) |||
5   ( first (arr (\a -> (b, a)) >>> f >>> arr (foldlArr f)) >>> app)
6   where listcase [] = Left []
7         listcase (x:xs) = Right (x,xs)

```

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

```

1 zipWithArr :: ArrowChoice arr => arr (a, b) c -> arr ([a], [b]) [c]
2 zipWithArr f = (arr $ \ (as, bs) -> zipWith (,) as bs) >>> mapArr f

```

These combinators make use of the **ArrowChoice** type class, it provides the **|||** combinator. The latter takes two arrows $\text{arr } a \ c$ and $\text{arr } b \ c$ and combines them into a new arrow $\text{arr } (\text{Either } a \ b) \ c$ which pipes all **Left** a ’s to the first arrow and all **Right** b ’s to the second arrow.

```

1 (|||) :: ArrowChoice arr a c -> arr b c -> arr (Either a b) c

```


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

```
1 listApp :: (ArrowChoice arr, ArrowApply arr) => [arr a b] -> arr [a] [b]
2 listApp fs = (arr $ \as -> (fs, as)) >>> zipWithArr app
```

This combinator also makes use of the `ArrowApply` typeclass that allows us to evaluate arrows with `app :: arr (arr a b, a) c`.

The definition of `unshuffle` is

```
1 unshuffle :: (Arrow arr) => Int -> arr [a] [[a]]
2 unshuffle n = arr (\xs -> [takeEach n (drop i xs) | i <- [0..n-1]])
3
4 takeEach :: Int -> [a] -> [a]
5 takeEach n [] = []
6 takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

while `shuffle` is defined as:

```
1 shuffle :: (Arrow arr) => arr [[a]] [a]
2 shuffle = arr (concat . transpose)
```

These functions were taken from Eden skeleton source code (`ede, a`) and lifted to Arrows.

The helper functions for `torus` are

```
1 uncurry3 :: (a -> b -> c -> d) -> (a, (b, c)) -> d
2 uncurry3 f (a, (b, c)) = f a b c
3
4 lazyzip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
5 lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)
6
7 threetotwo :: (Arrow arr) => arr (a, b, c) (a, (b, c))
8 threetotwo = arr $ \ ~(a, b, c) -> (a, (b, c))
```