Abstract

Arrows were introduced in John Hughes paper as a general interface for computation and therefore as an alternative to monads for API design [1]. In the paper Hughes describes how arrows are a generalization of monads and how they are not as restrictive. In this paper we will use this concept to express parallelism.

First, we give an introduction to some of the possible ways to add parallelism to Haskell programs. Then, we give the basic definition of Arrows, which is followed up by the introduction of some utility functions used in this paper. Next, we introduce the ArrowParallel typeclass together with backends for it written with the parallel Haskells introduced earlier, finishing up with the benefits of this new way of writing parallel programs. After this we give the definition of several parallel skeletons. Then, we introduce some syntactic sugar to mimic the sequential arrow combinators introduced by Hughes [1] but with parallelism added. We also give benchmarks of our newly created parallel Haskell. Finally we give a short conclusion of what we managed to achieve.

Contents

1 Motivation

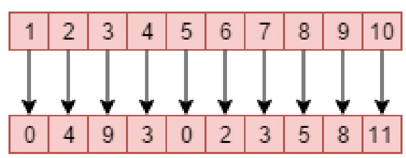
Arrows were introduced in John Hughes paper as a general interface for computation and therefore as an alternative to monads for API design [1]. In the paper Hughes describes how arrows are a generalization of monads and how they are not as restrictive. In this paper we will use this concept to express parallelism.

2 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:

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

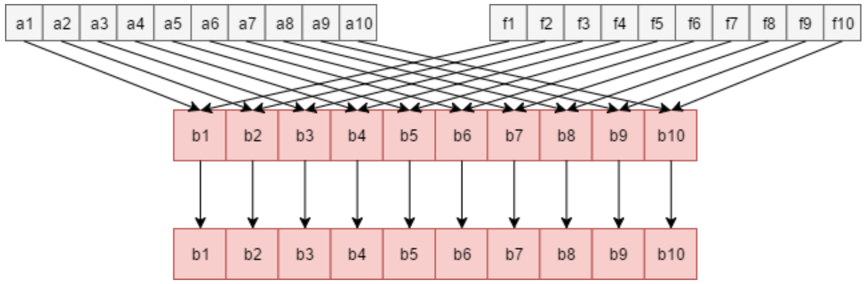


We will now implement `parEvalN` with the different parallel Haskells.

2.1 Multicore Haskell

Multicore Haskell [2] is the GHC native way to do parallel processing. 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 forcibly evaluated in parallel with the strategy `Strategy [b]` by the `using` operator. This strategy can be constructed with `parList :: Strategy a -> Strategy [a]` and `rdeepseq :: NFData a =>Strategy a`.

```
parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
parEvalN fs as = zipWith ($) fs as `using` parList rdeepseq
```

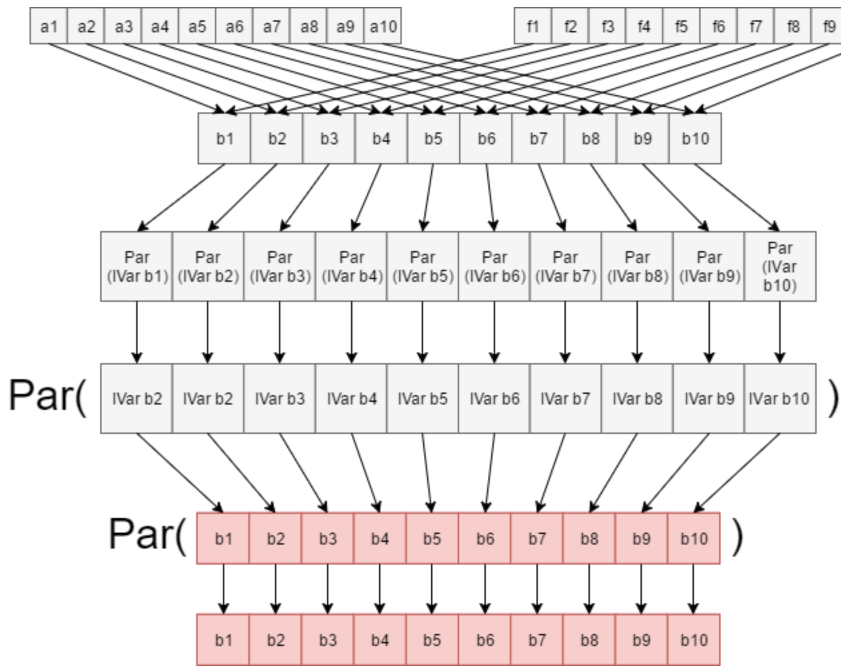


2.2 ParMonad

The Par monad introduced by Marlow et al. [3], which can be found in the monad-par package on hackage [4], 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.

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

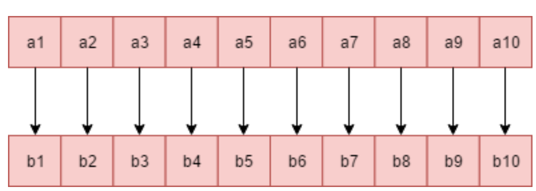


2.3 Eden

Eden is a parallel Haskell for distributed memory and comes with a MPI and a PVM backend [5–7]. This means that it works on clusters as well so it makes sense to have a Eden-based backend for our new parallel Haskell.

While it 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 quite easily:

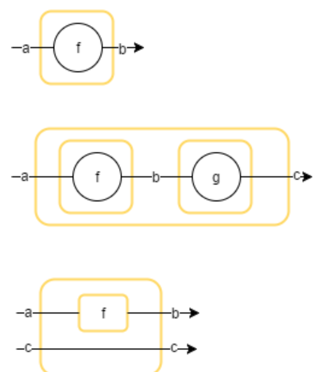
```
parEvalN :: (Trans a, Trans b) => [a -> b] -> [a] -> [b]
parEvalN fs as = spawnF fs as
```



3 Arrows

Arrows were introduced by Hughes [1] as a general interface for computation. An arrow `arr a b` can be look at as a computation that converts an input `a` to an output `b`. This is defined in the arrow typeclass:

```haskell
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)
```



`arr` is used to lift an ordinary function to an arrow type. This can be thought of as analogous to the monadic **return**. The >>> operator, in a similar way, is analogous to the monadic composition operator >>= 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`. And lastly, the `first` operator, which takes the input arrow from b to c and converts it into an arrow on pairs with the second argument untouched, is also needed for actual useful code as without it, we wouldn't have a way to save input across arrows.

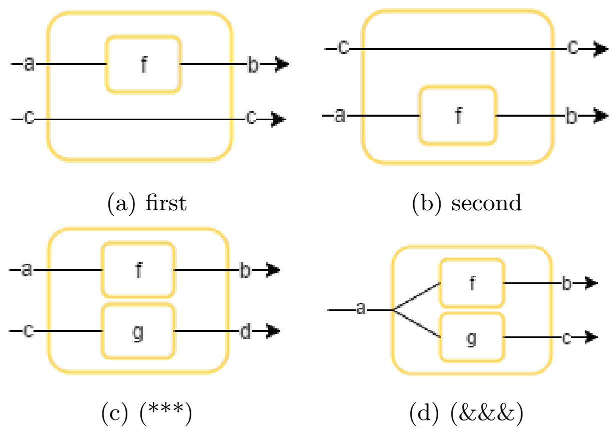
The most prominent instances of this interface are regular functions (->),

```haskell
instance Arrow (->) where
  arr  f = f
  f >>> g = g . f
   first  f  = \(a, c)  -> (f a, c)
```

and the Kleisli type

```haskell
data Kleisli  m a b =  Kleisli  { run  ::  a -> m b }
```

as well:



(a) first (b) second

(c) (***) (d) (&&&)

Figure 1: syntactic sugar

```
instance Monad m => Arrow (Kleisli m) where
  arr f = Kleisli $ return . f
  f >>> g = Kleisli $ \a -> f a >>= g
   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,

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

```
(***) :: 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 2 different values like ***, but takes only one input.

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

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

```
add :: Arrow arr => arr a Int -> arr a Int -> arr a Int
add f g = (f &&& g) >>> arr (\(u, v) -> u + v)
```

The benefit of using the Arrow typeclass is that any type which is shown to be an arrow can be used in conjunction with this newly created add combinator.

Even though this example is quite simple, the power of the arrow interface immediately is clear: If a type is an arrow, it can automatically used together with every library that works on arrows. Compared to simple monads, this enables us to write code that is more extensible, without touching the internals of the specific arrows.

Note: In the definitions Hughes gave in his paper, the notation `a b c` *for an arrow from* `b` *to* `c` *is used. We use the equivalent definition* `arr a b` *for an arrow from* `a` *to* `b` *instead, to make it easier to find the arrow type in type signatures.*

4 Utility Functions

Before we 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 `arr a b` to an arrow `arr [a] [b]` [8]:

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

Similarly, we can also define `foldlArr` that lifts any arrow `arr (b, a) b` with a neutral element `b` to `arr [a] b`:

```
foldlArr :: (ArrowChoice arr, ArrowApply arr) => arr (b, a) b -> b -> arr [a] b
foldlArr f b =
  arr listcase >>>
  arr (const b) |||
    ( first (arr (\a -> (b, a)) >>> f >>> arr (foldlArr f)) >>> app)
  where
    listcase [] = Left []
    listcase (x:xs) = Right (x,xs)
```

Finally, with the help of `mapArr`, we can define zipWithArr, which 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 (,) as bs) >>> mapArr f
```

These combinators make use of the `ArrowChoice` typeclass, which allows us to use the `|||` combinator. It takes two arrows `arr a c` and `arr b c` and combines them into a new arrow `arr (Either a b) c` which pipes all **Left** `a`'s to the first arrow and all **Right** `b`'s to the second arrow.

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

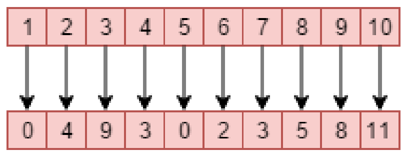
With the zipWithArr combinator we can also write a combinator `listApp`, which lifts a list of arrows `[arr a b]` to an arrow `arr [a] [b]`.

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

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

5 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 can also be looked at as 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.



5.1 The ArrowParallel typeclass

As we have seen earlier, in its purest form, parallel computation (on functions) can be looked at as the execution of some functions `a -> b` in parallel:

```
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).

```
parEvalN :: (Arrow arr) => [arr a b] -> arr [a] [b]
```

With this definition of `parEvalN`, parallel execution can be looked at as 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.

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

Sometimes parallel Haskells require additional configuration parameters as information about the execution environment. This is why 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.

```
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 want to allow the `conf` type parameter to be of any type and don't even evaluate it by blanking it in the type signature of the implemented parEvalN, as we will see in the implementation of the Multicore and the ParMonad backend.

5.2 Multicore Haskell

The Multicore Haskell implementation of this class is straightforward using listApp from chapter 4 combined with the using operator from Multicore Haskell.

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
  ArrowParallel arr a b conf where
    parEvalN _ fs = listApp fs >>> arr (flip using $ parList rdeepseq)
```

We hardcode the parList rdeepseq strategy here, as in this context it is the only one making sense, since we usually want the output list to be fully evaluated to its normal form.

5.3 ParMonad

The ParMonad implementation makes use of Haskells laziness and ParMonad's spawnP :: NFData a =>a -> Par (IVar a) function, which 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].

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

5.4 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

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

, we don't do it in this paper, as this seems too hacky. For now, we just implement ArrowParallel for normal functions

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

and the Kleisli type.

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

5.5 Benefits of parallel Arrows

Okay so?

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?

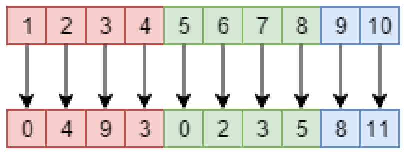
- **Arrow API benefits**: With the ArrowParallel typeclass we do not lose any benefits of using arrows as parEvalN is just yet another arrow combinator. The resulting arrow can be used in the same way a potential serial version could be used. This is a big advantage of this approach, especially compared 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 swapping out the actual ArrowParallel instance.

6 Basic Skeletons

With the ArrowParallel typeclass in place and implemented, we can now implement some basic parallel skeletons.

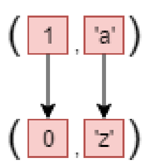
6.1 parEvalNLazy



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**.

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

6.2 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 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
```

7 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).

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

With this we can analogously to the serial &&& define the parallel |&&&|.

```
(|&&&|) :: (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, a)) >>> f |***| g
```

8 Futures

Consider the following arbitrary, but interesting, parallel arrow combinator:

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

In a distributed environment, 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. 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 in heavy threaded applications. To fix this, we have to introduce a way that enables the nodes to communicate directly with each other.

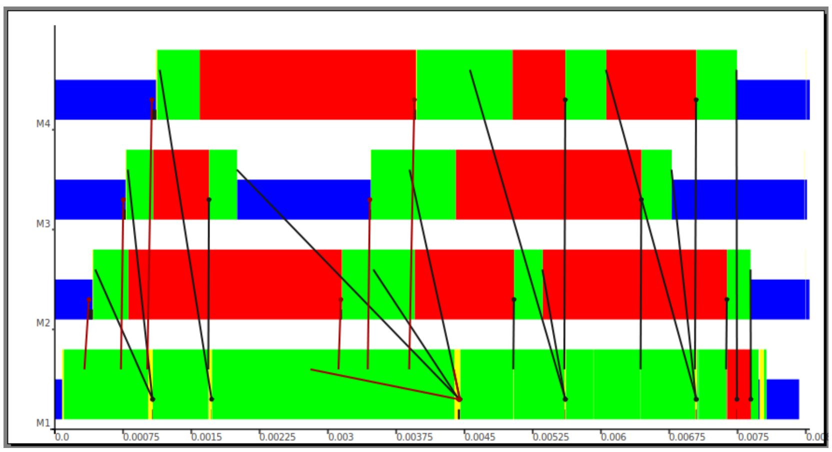


Figure 2: Communication between 4 threads without Futures

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:

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

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

```

```
instance (Trans a) => Future RemoteData a where
    put = arr (\a -> RD { rd = release a })
    get = arr rd >>> arr fetch
```

For ParMonad and Multicore we can use a basic dummy wrapper because we have shared memory in a single node:

```
data BasicFuture a = BF { val :: a }

instance (NFData a) => Future BasicFuture a where
    put = arr (\a -> BF { val = a })
    get = arr val
```

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

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

and the following for the BasicFuture dummy type:

```
instance (NFData a) => NFData (BasicFuture a) where
  rnf = rnf . val
```

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:

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

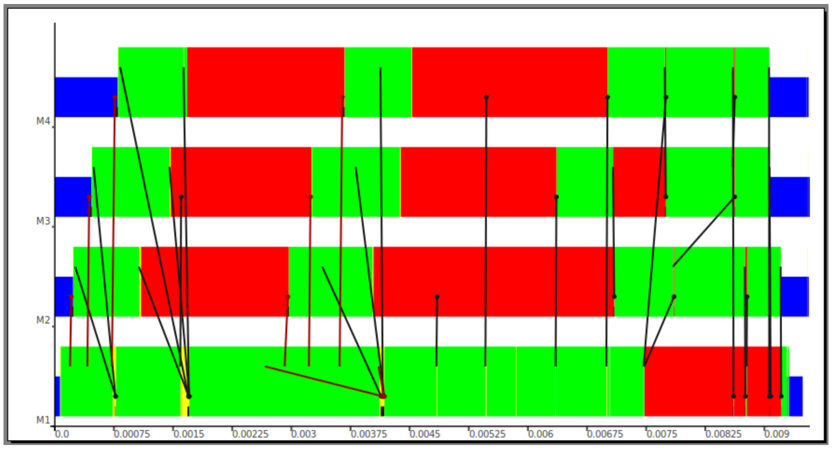
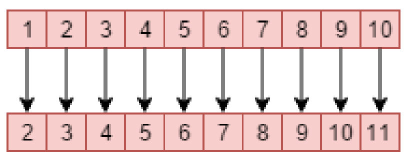


Figure 3: Communication between 4 threads with Futures

9 Map-based Skeletons

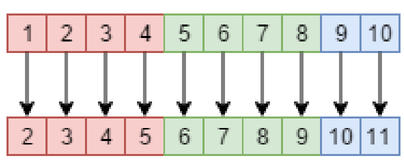
9.1 parMap



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.

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

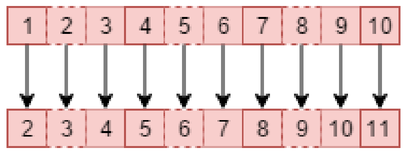
9.2 parMapStream



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.

```
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)
```

9.3 farm



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.

```
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
```

The definition of unshuffle is

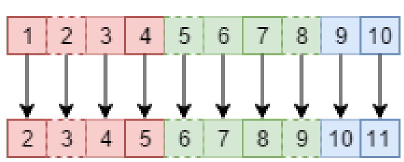
```
unshuffle :: (Arrow arr) => Int -> arr [a] [[a]]
unshuffle n = arr (\xs -> [takeEach n (drop i xs) | i <- [0..n-1]])

takeEach :: Int -> [a] -> [a]
takeEach n [] = []
takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

, while shuffle is defined as:

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

(These were taken from Eden's source code [9] and translated into Arrows.)

9.4 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:

```
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
```

9.5 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

```
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
```

10 Topology 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 are a pipe, ring and a torus implementation [10, 11]. These seem like reasonable candidates to be ported to our arrow based parallel Haskell to prove that we can express such skeletons with Arrows as well.

10.1 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 don't have to reside on the same thread/machine. We implement this skeleton using the ArrowLoop typeclass which enables us to use loop :: arr (a, b) (c, b) -> arr a c.

```
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
```

where lazy is defined as:

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

However, using this definition directly, will result in the master node becoming a potential bottleneck in distributed environments as described in chapter 8. Therefore, a more sophisticated version that uses Futures internally is a good idea:

```
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

```
pipe2 :: (ArrowLoop arr, ArrowChoice arr,
  ArrowParallel arr (fut (([a], [b]), [c])) (fut (([a], [b]), [c])) conf,
  Future fut (([a], [b]), [c])) =>
  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))
```

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 |>>>| which is semantically equivalent to >>> similar to the other parallel syntactic sugar from chapter 7:

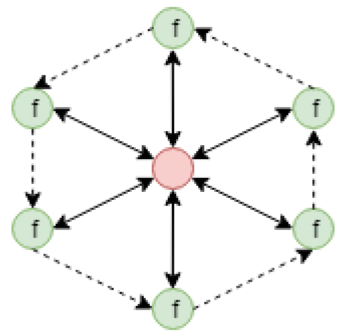
```
(|>>>|) :: (ArrowLoop arr, ArrowChoice arr,
  ArrowParallel arr (fut (([a], [b]), [c])) (fut (([a], [b]), [c])) (),
  Future fut (([a], [b]), [c])) =>
  arr a b -> arr b c -> arr a c
(|>>>|) = pipe2 ()
```

10.2 ring

Eden's ring implementation allows a ring of functions to communicate over direct channels. We can rewrite its functionality easily again with the use of loop. In each loop we start by rotating the intermediary input from the nodes [fut r] with second (rightRotate >>> lazy). We have to feed the intermediary input into lazy or the evaluation would hang. The reasoning was explained by [10]:

> 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 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.




```
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)
```

with rightRotate:

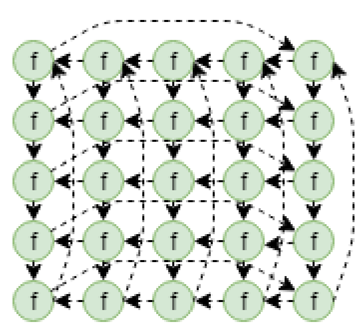
```
rightRotate :: (Arrow arr) => arr [a] [a]
rightRotate = arr $ \list -> case
  list of [] -> []
      xs -> last xs : init xs
```

and lazy:

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

This combinator can be used for example to calculate the shortest paths in a graph using Warshall's algorithm. Further details on this can be found in [10].

10.3 torus



If we take the concept of a ring one dimension further, we get a torus. Every node sends ands receives data from horizontal and vertical neighbours in each communication round. This gives us . With our parallel Arrows we implement a torus combinator yet again with the help of the ArrowLoop typeclass.

Inside the loop 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 input `([[c]], ([[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. We once again zip the inputs for the nodefunction 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 this 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:

```haskell
torus :: (ArrowLoop arr, ArrowChoice arr, ArrowApply arr,
  ArrowParallel arr (c, fut a, fut b) (d, fut a, fut b) conf,
  Future fut a, Future fut b) =>
  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)
```

with uncurry3,

```haskell
uncurry3 :: (a -> b -> c -> d) -> (a, (b, c)) -> d
uncurry3 f (a, (b, c)) = f a b c
```

lazyzip3,

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

ptorus,

```
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))
```

and threetotwo.

```
threetotwo :: (Arrow arr) => arr (a, b, c) (a, (b, c))
threetotwo = arr $ \ ~(a, b, c) -> (a, (b, c))
```

As an example of using this skeleton [10] gave the matrix multiplication using the gentlemans algorithm. Adapting this nodefunction to our arrow API gives us:

```
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
```

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.

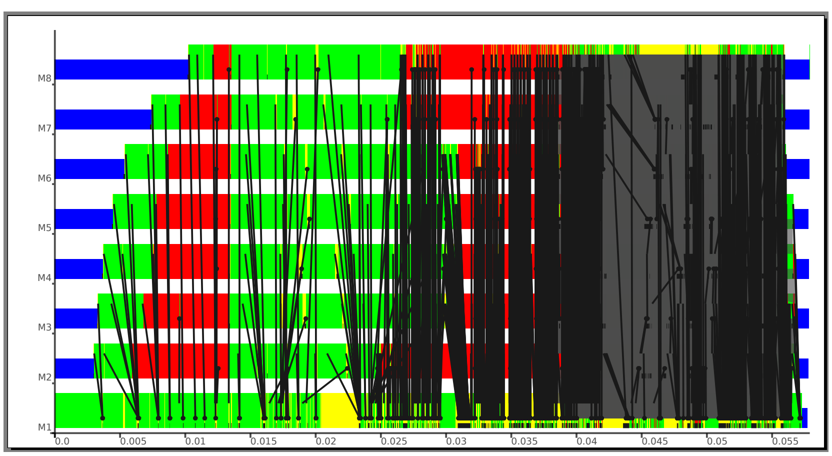


Figure 4: Matrix Multiplication with a torus (Parrows)

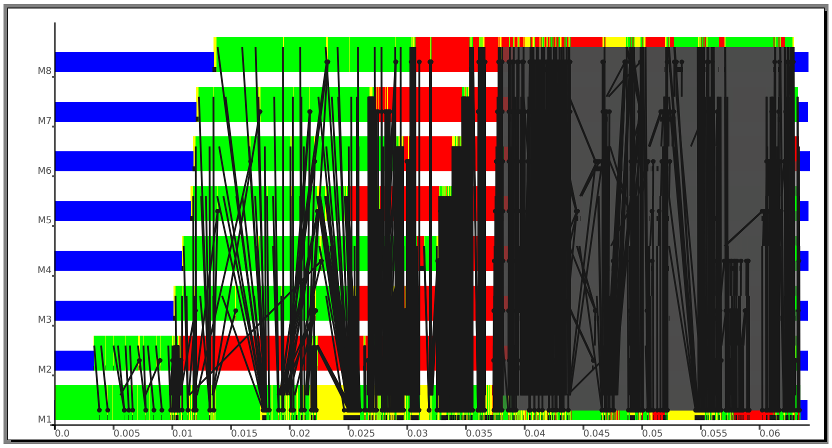
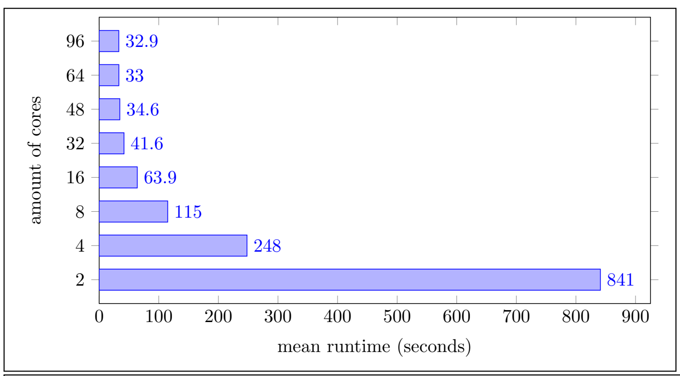
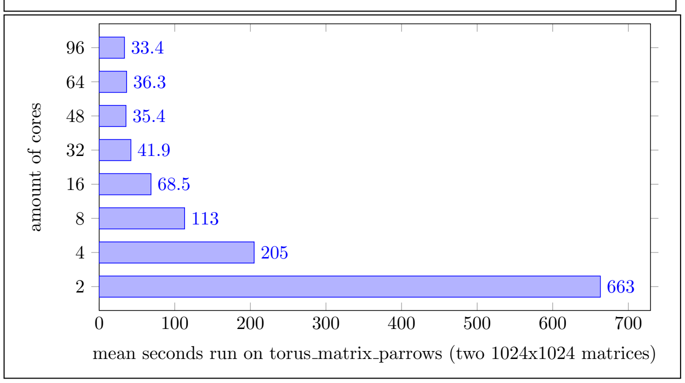


Figure 5: Matrix Multiplication with a torus (Eden)





11 Conclusion

As we have seen in this paper, arrows are a useful tool for composing parallel programs. By basing our parallel Haskell on them, we do not have to introduce new monadic types that wrap the computation and instead can use them just like they were regular sequential pure code. Performancewise, parallel arrows are on par with existing parallel Haskells, as they do not introduce any notable overhead.

ArrowApply (or equivalent) are needed because we basically want to be able to produce intermediary resu

References

[1] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67 – 111, 2000. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(99)00023-4. URL http://www.sciencedirect.com/science/article/pii/S0167642399000234.

[2] Multicore's parallel package on hackage. URL https://hackage.haskell.org/package/parallel-3.2.1.0. [Accessed on 01/12/2017].

[3] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, September 2011. ISSN 0362-1340. doi: 10.1145/2096148.2034685. URL http://doi.acm.org/10.1145/2096148.2034685.

[4] Parmonad (control.monad.par) on hackage. URL https://hackage.haskell.org/package/monad-par-0.3.4.8/. [Accessed on 01/12/2017].

[5] Eden modules on hackage, . URL https://hackage.haskell.org/package/edenmodules-1.2.0.0/. [Accessed on 01/12/2017].

[6] Rita Loogen. *Eden – Parallel Functional Programming with Haskell*, pages 142–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32096-5. doi: 10.1007/978-3-642-32096-5_4. URL http://dx.doi.org/10.1007/978-3-642-32096-5_4.

[7] Eden homepage, . URL http://www.mathematik.uni-marburg.de/~eden/. [Accessed on 01/12/2017].

[8] John Hughes. *Programming with Arrows*, pages 73–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31872-9. doi: 10.1007/11546382_2. URL http://dx.doi.org/10.1007/11546382_2.

[9] Eden skeletons' control.parallel.eden.map package source code, . URL https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/src/Control-Parallel-Eden-Map.html. [Accessed on 02/12/2017].

[10] Rita Loogen. Eden - parallel functional programming with haskell. URL http://www.mathematik.uni-marburg.de/~eden/paper/edenCEFP.pdf. [Accessed on 03/17/2017].

[11] Eden skeletons' control.parallel.eden.topology package source code, . URL https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html. [Accessed on 03/17/2017].

[12] Simon Marlow. Sample code for "Parallel and Concurrent Programming in Haskell". URL https://github.com/simonmar/parconc-examples. [Accessed on 01/12/2017].