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:

```
1 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`.

```
1 parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
2 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:

```
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 (->),

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

and the Kleisli type

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

as well:

```
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 and zipWith on arrows.
The mapArr combinator, that 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)
```

and 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 two 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:

```
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 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.

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

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

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

```
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

```
instance (Trans a, Trans b) => ArrowParallel (−>) a b conf where
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)
```

5.5 Benefits 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 swapping out the actual ArrowParallel instance.

6 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 up until now. But sometimes we want to paralellize heterogenous types as well. For this, we introduce a helper combinator arrMaybe first, that converts an arrow arr a b to an arrow arr (**Maybe** a) (**Maybe** b).

```
arrMaybe :: (ArrowApply arr) => (arr a b) -> arr (Maybe a) (Maybe b)
arrMaybe fn = (arr $ go) >>> app
  where
    go Nothing = (arr $ \Nothing -> Nothing, Nothing)
    go (Just a) = ((arr $ \(Just x) -> (fn, x)) >>> app >>> arr Just, (Just a))
```

With this, we can now easily write parEval2 which combines two arrows arr a b and arr c d into a new parallel arrow arr (a, c) (b, d). We start by converting both arrows with arrMaybe, combining them with *** into a new arrow arr (**Maybe** a, **Maybe** c) (**Maybe** b, **Maybe** d). This is then replicated twice and fed into parEvalN to get a arr [(**Maybe** a, **Maybe** c)] [(**Maybe** b, **Maybe** d)]. We can then apply this arrow to the input [(**Just** a, **Nothing**), (**Nothing**, **Just** c)] and then extract the resulting values with **fromJust** and the !! operator on lists in the last step.

```
parEval2 :: (ArrowParallel arr a b conf,
```

```
  ArrowParallel arr (Maybe a, Maybe c) (Maybe b, Maybe d) conf,
  ArrowApply arr) =>
  conf -> arr a b -> arr c d -> (arr (a, c) (b, d))
parEval2 conf f g =
  (arr $ \(a, c) -> (f_g, [(Just a, Nothing), (Nothing, Just c)])) >>>
  app >>>
  (arr $ \comb -> (fromJust (fst (comb !! 0)), fromJust (snd (comb !! 1))))
where
  f_g = parEvalN conf $ replicate 2 $ arrMaybe f *** arrMaybe g
```

6.3 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, ArrowApply arr) =>
  conf -> (arr a b) -> (arr [a] [b])
parMap conf f =
  (arr $ \as -> (f, as)) >>>
  (first $ arr repeat >>>
    arr (parEvalN conf)) >>>
  app
```

6.4 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 =
  (arr $ \as -> (f, as)) >>>
  (first $ arr repeat >>>
    arr (parEvalNLazy conf chunkSize)) >>>
  app
```

6.5 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, ArrowApply arr) =>
  conf -> NumCores -> arr a b -> arr [a] [b]
farm conf numCores f =
  (arr $ \as -> (f, as)) >>>
  ( first $ arr mapArr >>> arr repeat >>>
    arr (parEvalN conf)) >>>
  (second $ arr (unshuffle numCores)) >>>
  app >>>
  arr shuffle
```

The definition of unshuffle is

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

, while shuffle is defined as:

```
shuffle :: [[a]]
  -> [a]
shuffle = concat . transpose
```

(These were taken from Eden's source code. [9])

6.6 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 =
  (arr $ \as -> (f, as)) >>>
  ( first $ arr mapArr >>> arr repeat >>>
    arr (parEvalNLazy conf chunkSize)) >>>
  (second $ arr (unshuffle numCores)) >>>
  app >>>
  arr shuffle
```

7 Syntactic Sugar

To make our new API easier to use, we also introduce some syntactic sugar. We start with |>>>|, which is basically >>> on lists of arrows.

```
(|>>>|) :: (Arrow arr) => [arr a b] -> [arr b c] -> [arr a c]
(|>>>|) = zipWith (>>>)
```

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).

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

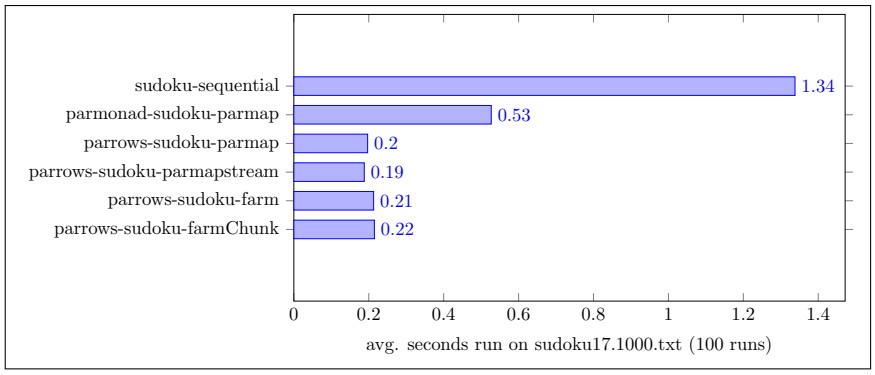
With this we can analogously to the serial &&& define the parallel |&&&|.

```
(|&&&|) :: (ArrowParallel arr a b (),
  ArrowParallel arr (Maybe a, Maybe a) (Maybe b, Maybe c) (),
  ArrowApply arr) =>
  arr a b -> arr a c -> arr a (b, c)
(|&&&|) f g = (arr $ \a -> (a, a)) >>> f |***| g
```

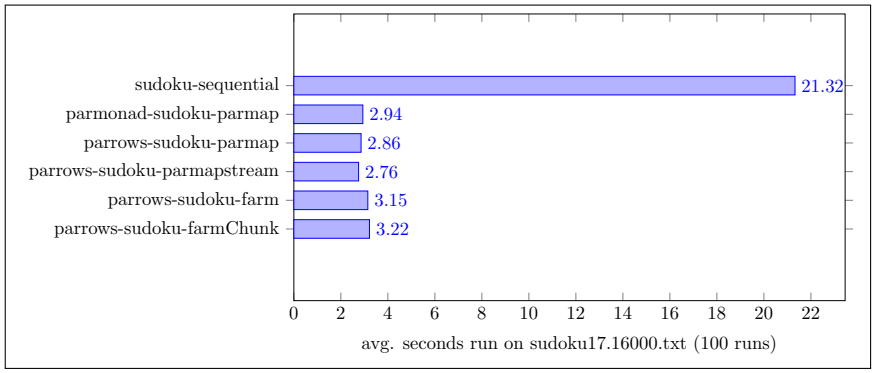
8 Benchmarks

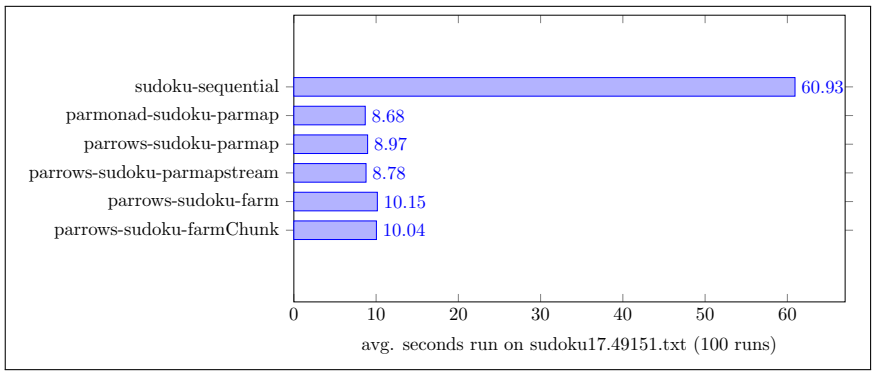
To check the performance of our new parallel arrow API, we conducted some benchmarks. These are based on a sudoku solver from the examples for Simon Marlows book "Parallel and Concurrent Programming in Haskell" which can be found on his GitHub [10]. The results are displayed in the following graphs which are ordered by problem size.

The Benchmarks were run on a Core i7-3970X CPU @ 3.5GHz with 6 cores and 12 threads. For sake of comparability with Simon Marlow's parallel version which uses the ParMonad, we use the ParMonad backend for the parallel arrow versions as well.



Note: the bad result for parmonad-sudoku-parmap seems to be a artefact as it performs much better in the other benchmarks





As we can see, the parallel arrow versions of the program all have around the same speedup as the ParMonad based version. This means that using the ArrowParallel typeclass doesn't add any real notable overhead. Also, the slightly worse results for "parrows-sudoku-farm" and "parrows-sudoku-farmChunk" can be explained by either imperfect parameters or by the fact that the benchmarks do not really require the scheduling introduced by the skeletons which in this case introduces unnecessary overhead. This would probably look different if the benchmark would use e.g. a divide and conquer approach to solve the sudoku puzzles.

9 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. While we have seen that the parallel arrows can express computation on clusters, it is noteworthy that in its current state, the parallel arrow API doesn't have as much control over data-flow as for example Eden. In its current state this will end up in the root node being a possible bottleneck on clusters when parallel arrows are combined with >>>. This can probably be remedied by introducing some sort of a Java-like Future construct that can be passed between nodes so that nodes can communicate without going through the root node. This will have to be implemented in a follow up paper, though.

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