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

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

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

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

```
1 class Arrow arr where
2   arr   ::  (a -> b) -> arr a b
3   (>>>) :: arr a b -> arr b c -> arr a c
4   first   ::  arr a b -> arr (a,c) (b,c)
```

Functions (->) are arrows:

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

## The Kleisli type

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

as well:

```
1 instance Monad m => Arrow (Kleisli m) where
2 arr f = Kleisli $ return . f
3 f >>> g = Kleisli $ \a -> f a >>= g
4 first f = Kleisli $ \(a,c) -> f a >>= \b -> return (b,c)
```

```
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)
```

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

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

## Arrow usage Example

```
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 mapArr combinator lifts any arrow arr a b to an arrow arr [a] [b] [1],

```
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
6       listcase [] = Left ()
7       listcase (x:xs) = Right (x,xs)
```

with

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

zipWithArr lifts any arrow arr (a, b) c to an arrow 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
```

```
1 listApp :: (ArrowChoice arr, ArrowApply arr) =>
2   [arr a b] -> arr [a] [b]
3 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`.

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

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

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

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

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

```
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
```

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

and the Kleisli type.

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

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

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

```
1 parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply a
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
8     fchunks = map (parEvalN conf) $ chunksOf chunkSize fs
```

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

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

```
1 parMap :: (ArrowParallel arr a b conf, ArrowApply arr) =>
2   conf -> (arr a b) -> (arr [a] [b])
3 parMap conf f =
4   (arr $ \as -> (f, as)) >>>
5   ( first $ arr repeat >>>
6     arr (parEvalN conf)) >>>
7   app
```

```
1 parMapStream :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply a
2   conf -> ChunkSize -> arr a b -> arr [a] [b]
3 parMapStream conf chunkSize f =
4   (arr $ \as -> (f, as)) >>>
5   ( first $ arr repeat >>>
6     arr (parEvalNLazy conf chunkSize)) >>>
7   app
```

```
1  farm :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
2    ArrowChoice arr, ArrowApply arr) =>
3    conf -> NumCores -> arr a b -> arr [a] [b]
4  farm conf numCores f =
5    (arr $ \as -> (f, as)) >>>
6    ( first $ arr mapArr >>> arr repeat >>>
7      arr (parEvalN conf)) >>>
8    (second $ arr (unshuffle numCores)) >>>
9    app >>>
10   arr shuffle
```

The definition of unshuffle is

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

, while shuffle is defined as:

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

(These were taken from Eden's source code. [2])

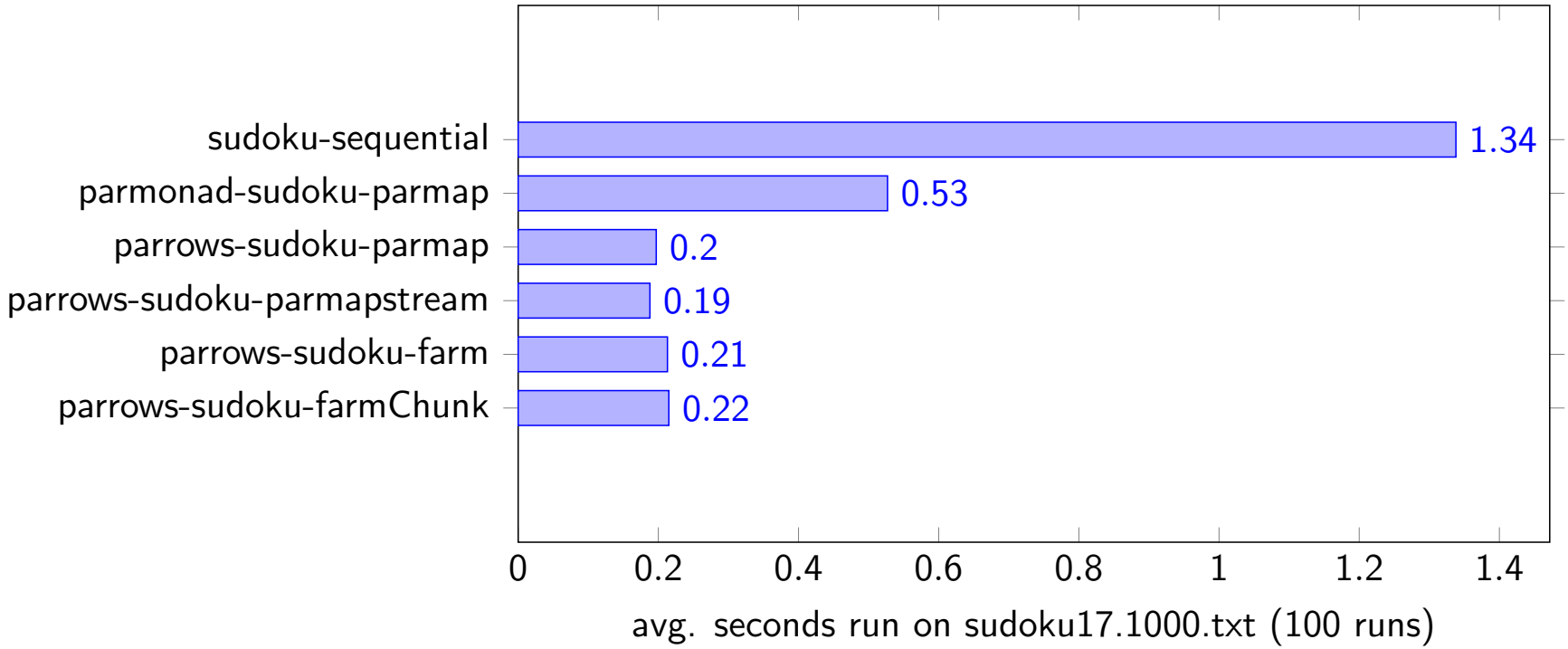
```
1  farmChunk :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
2    ArrowChoice arr, ArrowApply arr) =>
3    conf -> ChunkSize -> NumCores -> arr a b -> arr [a] [b]
4  farmChunk conf chunkSize numCores f =
5    (arr $ \as -> (f, as)) >>>
6    ( first $ arr mapArr >>> arr repeat >>>
7      arr (parEvalNLazy conf chunkSize)) >>>
8    (second $ arr (unshuffle numCores)) >>>
9    app >>>
10   arr shuffle
```

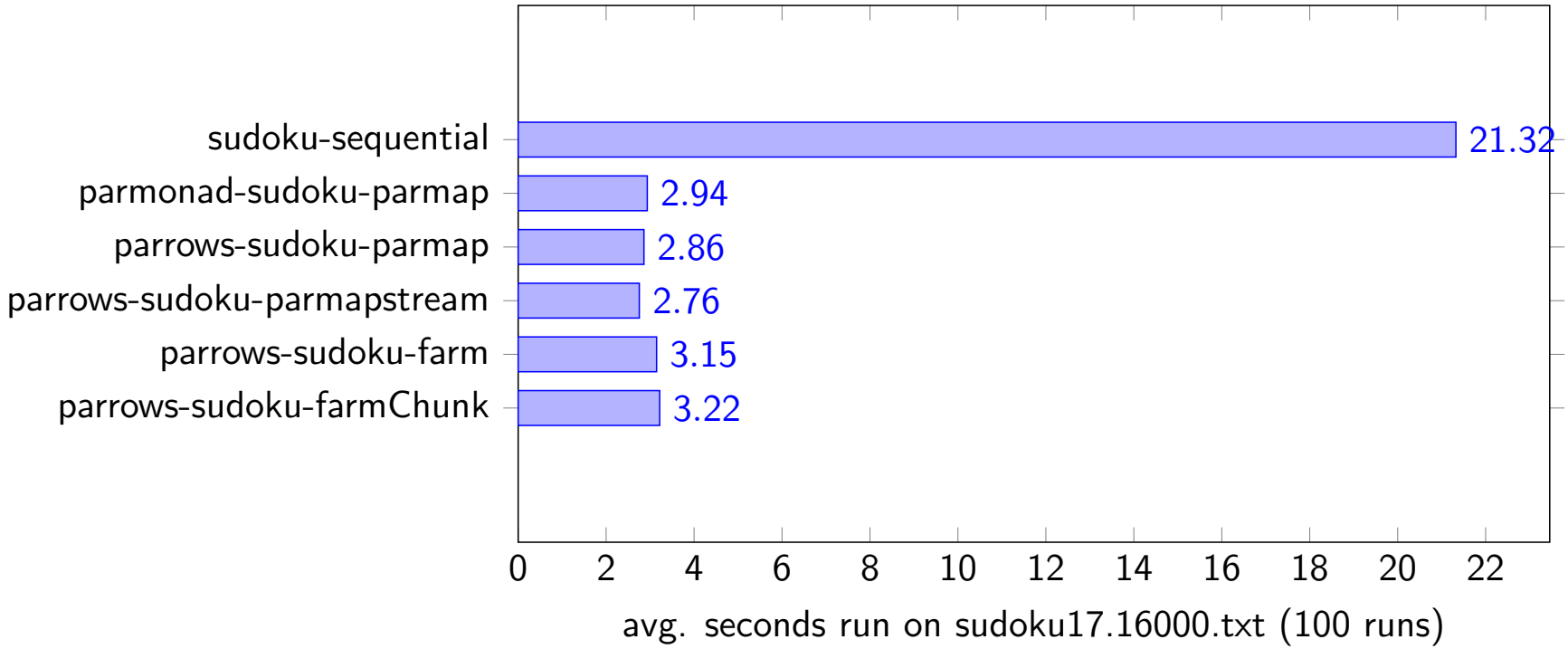
```
(|>>>|) :: (Arrow arr) => [arr a b] -> [arr b c] -> [arr a c]
(|>>>|) = zipWith (>>>)
```

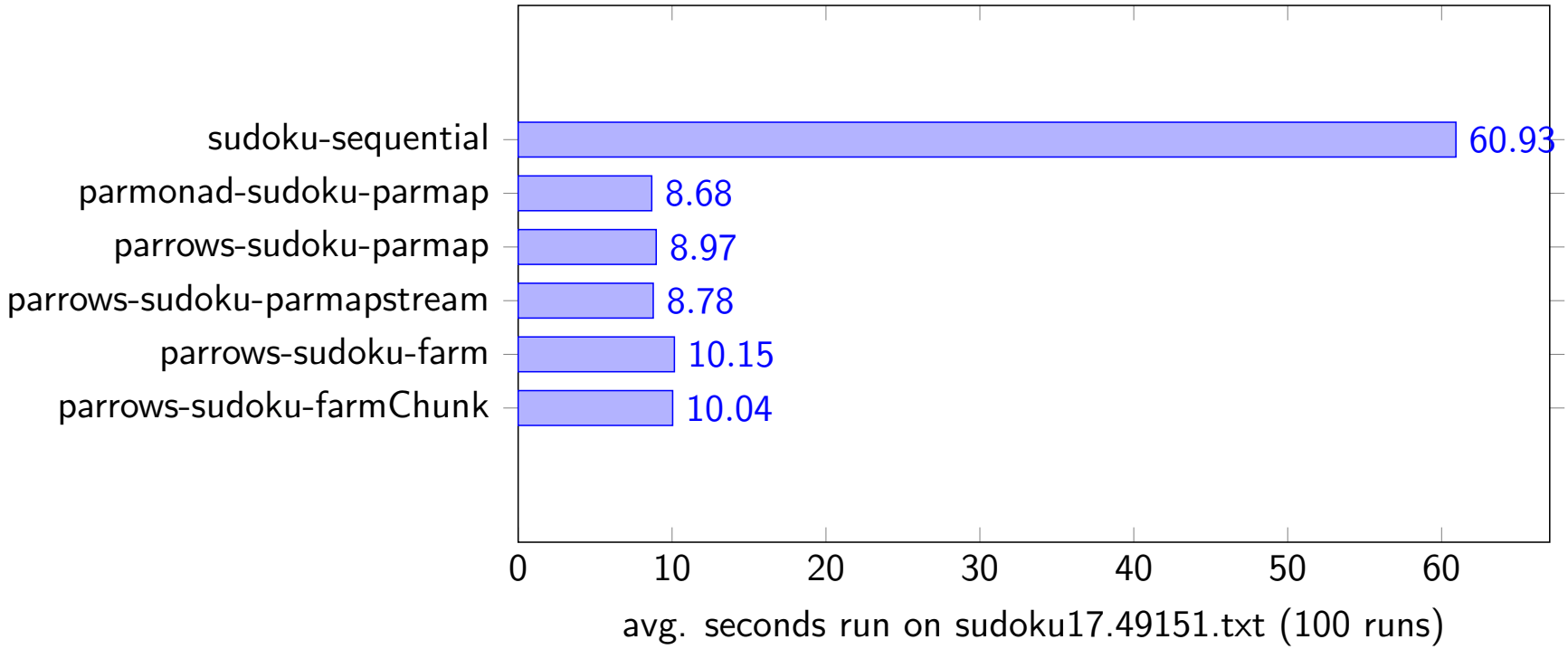
```
(|***|) :: (ArrowParallel arr a b (),
   ArrowParallel arr (Maybe a, Maybe c) (Maybe b, Maybe d) (),
   ArrowApply arr) =>
   arr a b -> arr c d -> arr (a, c) (b, d)
(|***|) = parEval2 ()
```

```
(|&&&|) :: (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
```

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.


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
1.34
0.53
0.2
0.19
0.21
0.22
0
0.2
0.4
0.6
0.8
1
1.2
1.4
avg. seconds run on sudoku17.1000.txt (100 runs)


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
21.32
2.94
2.86
2.76
3.15
3.22
0
2
4
6
8
10
12
14
16
18
20
22
avg. seconds run on sudoku17.16000.txt (100 runs)


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
60.93
8.68
8.97
8.78
10.15
10.04
0
10
20
30
40
50
60
avg. seconds run on sudoku17.49151.txt (100 runs)

[1] 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.

[2] 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].