CHAIR FOR COMPILER CONSTRUCTION

# STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism

By Sebastian Ertel, Justus Adam, Norman A. Rink,
Andrés Goens and Jerónimo Castrillón-Mazo

# We want ...

- Performance
- Responsiveness

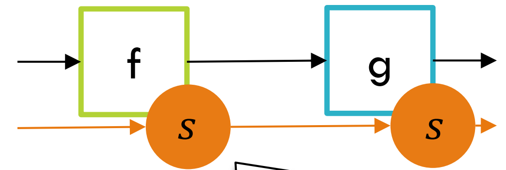
**Parallelism in the presence of State**

- Efficient
- Intuitive

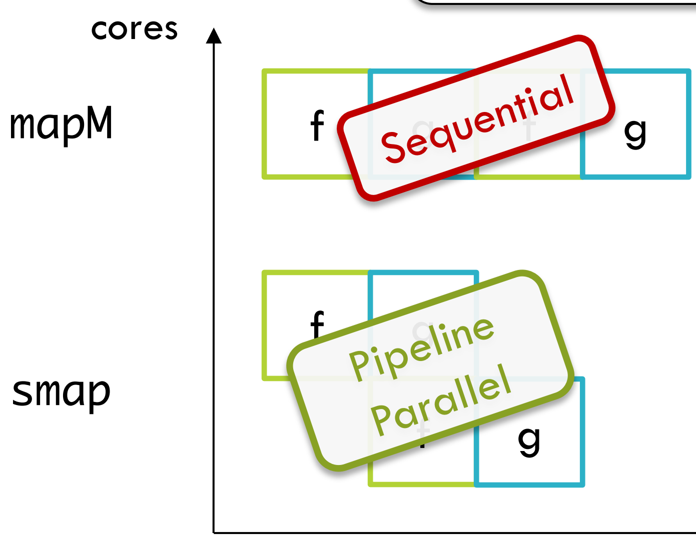
```
mapM (\x -> f x >>= g) [..]
```

Monads are a powerful and familiar abstraction, but inherently sequential

### Monad Execution



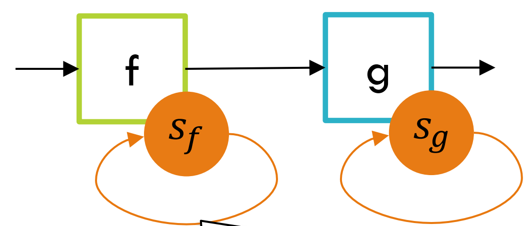

Functions depend on entire state




Parallelism arises implicitly from model

Granularity implicit from state scope[1]

### Dataflow Execution




Functions depend on local state only

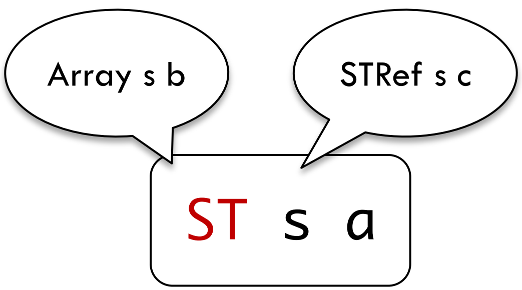
**Use Case**

- FRP
- Data Streaming

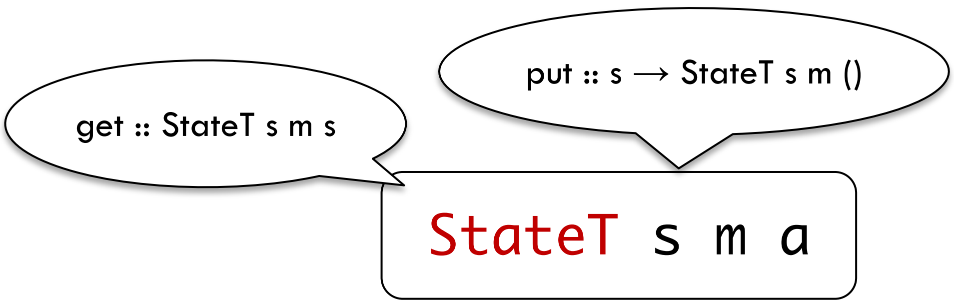
1. Tim Harris and Satnam Singh. 2007. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming* (ICFP '07). ACM, New York, NY, USA, 251-264.

# State in Haskell

Lots of references strewn about

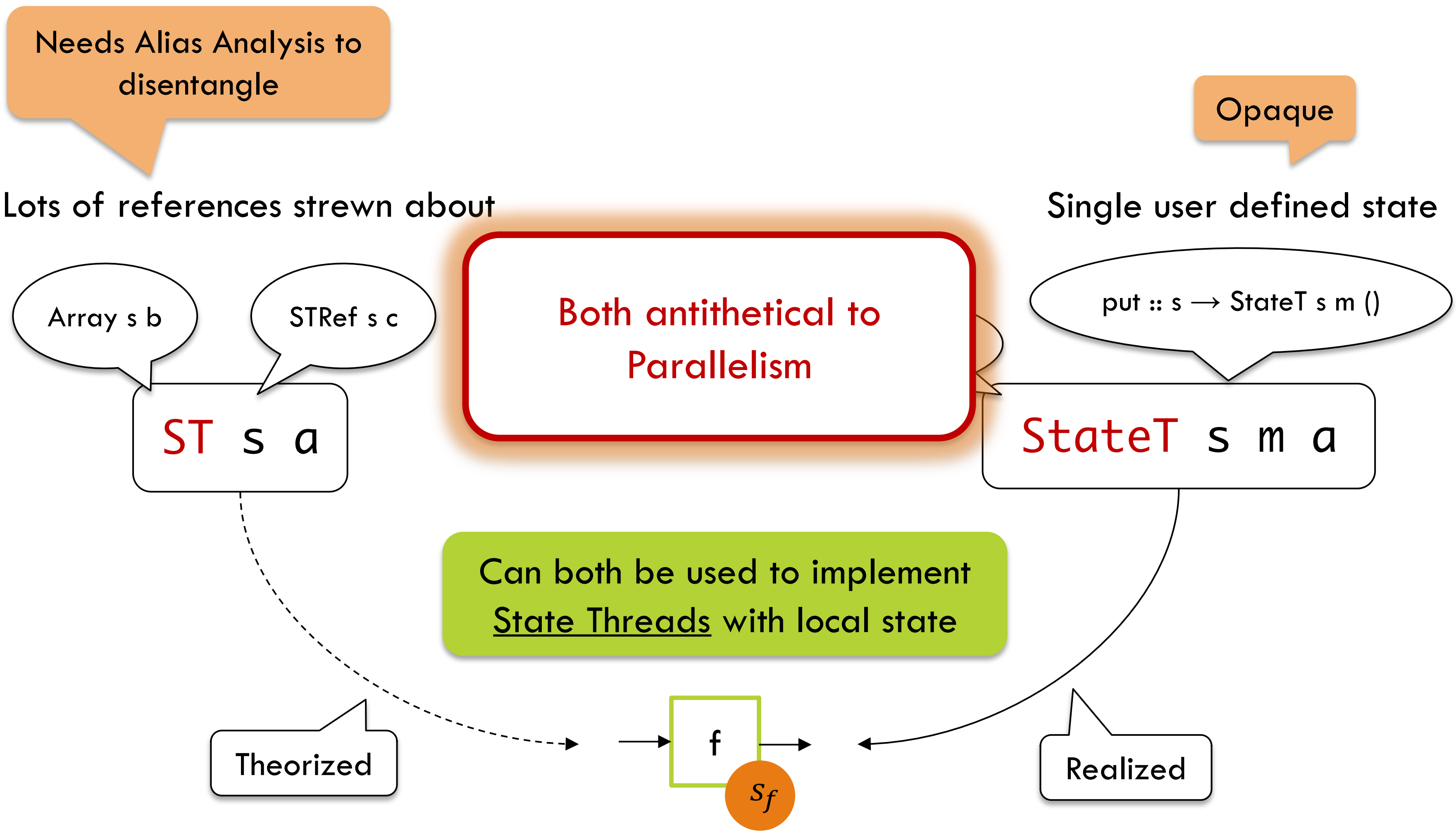



Single user defined state




1. Wadler, Philip. "The essence of functional programming." *POPL*. Vol. 92. No. 37. 1992.
2. Launchbury, John, and Simon L. Peyton Jones. "Lazy functional state threads." *ACM SIGPLAN Notices* 29.6 (1994): 24-35.

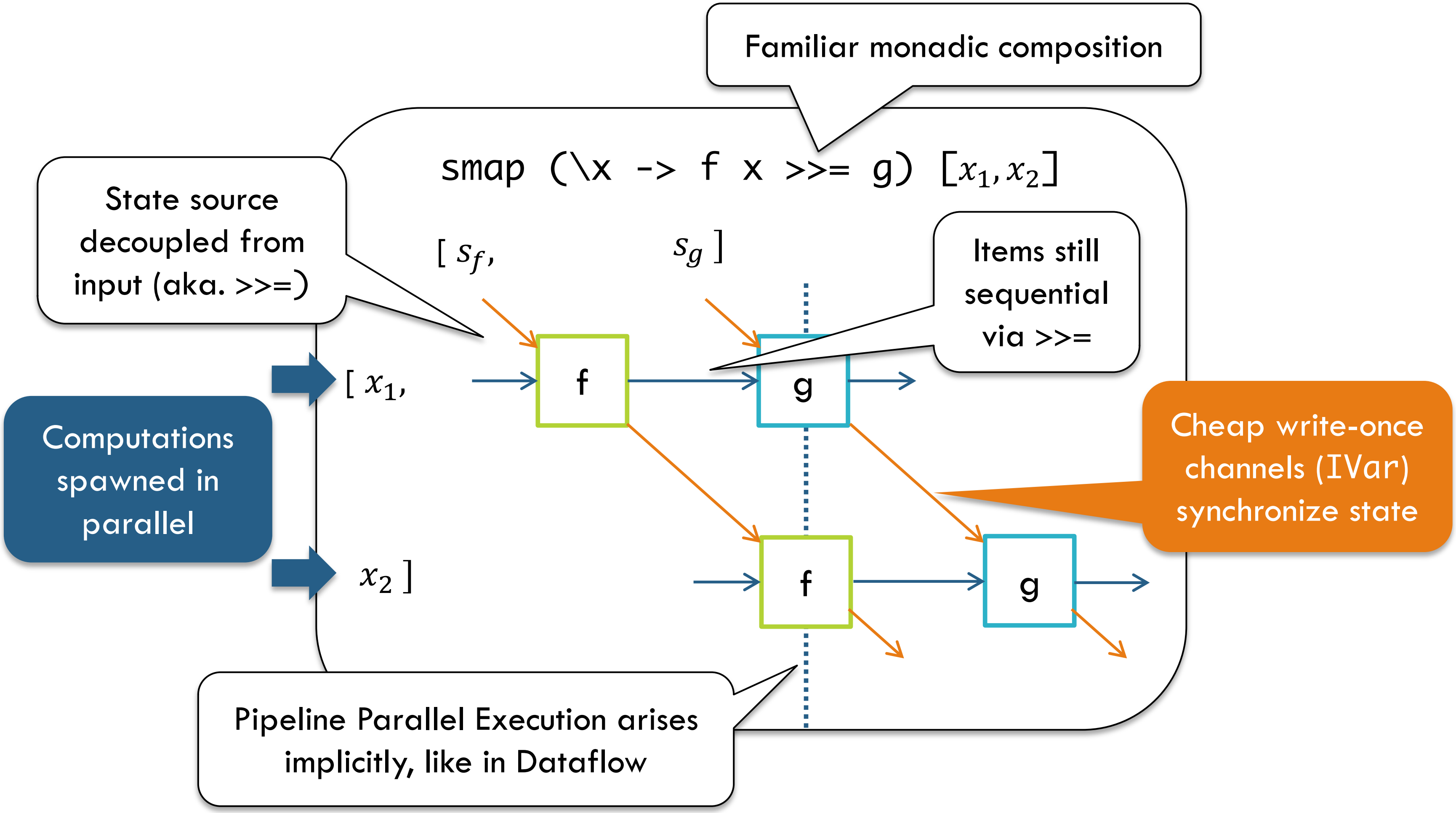
# State in Haskell





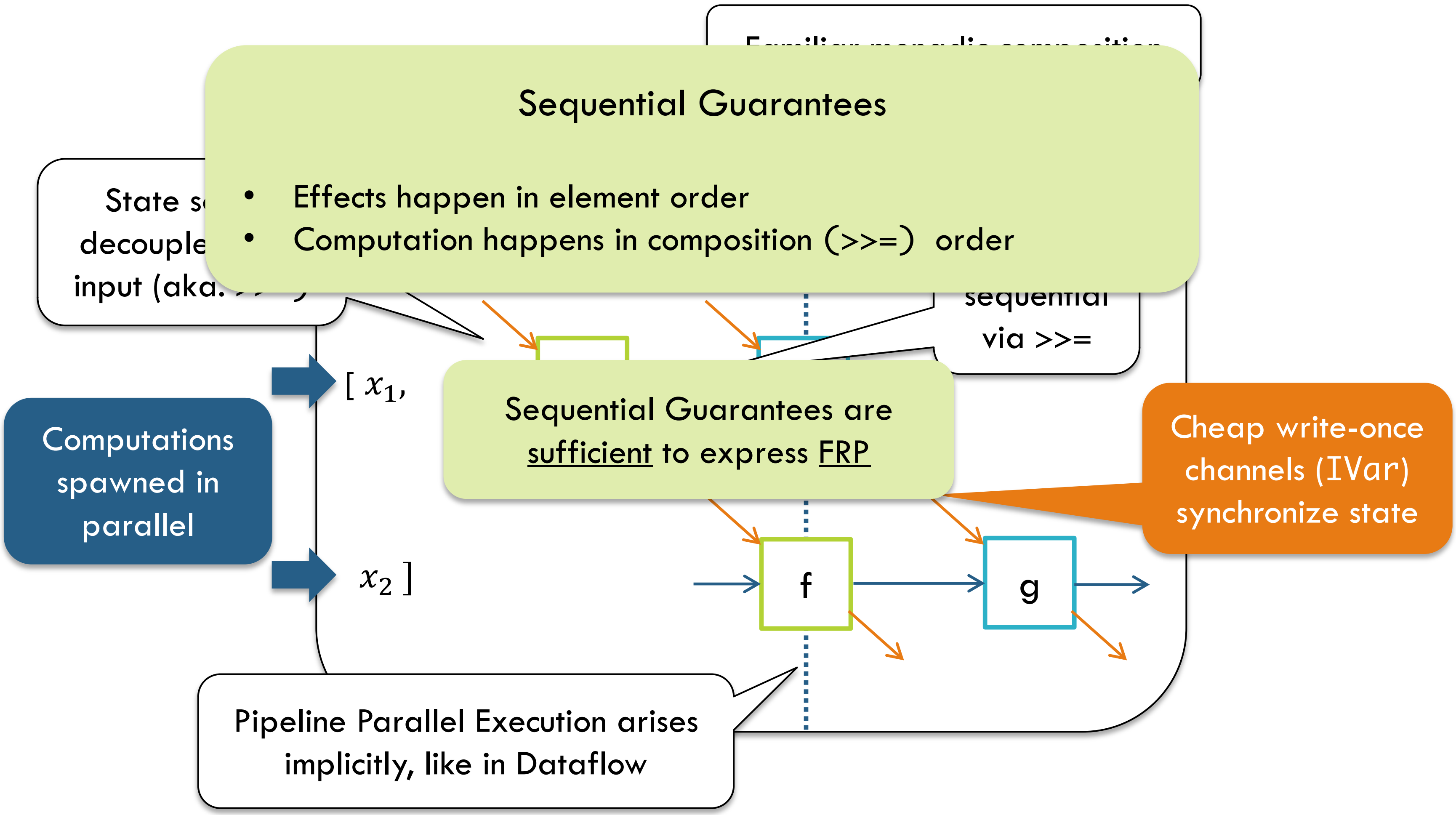
1. Wadler, Philip. "The essence of functional programming." *POPL*. Vol. 92. No. 37. 1992.
2. Launchbury, John, and Simon L. Peyton Jones. "Lazy functional state threads." *ACM SIGPLAN Notices* 29.6 (1994): 24-35.

# Smap (De)Construction



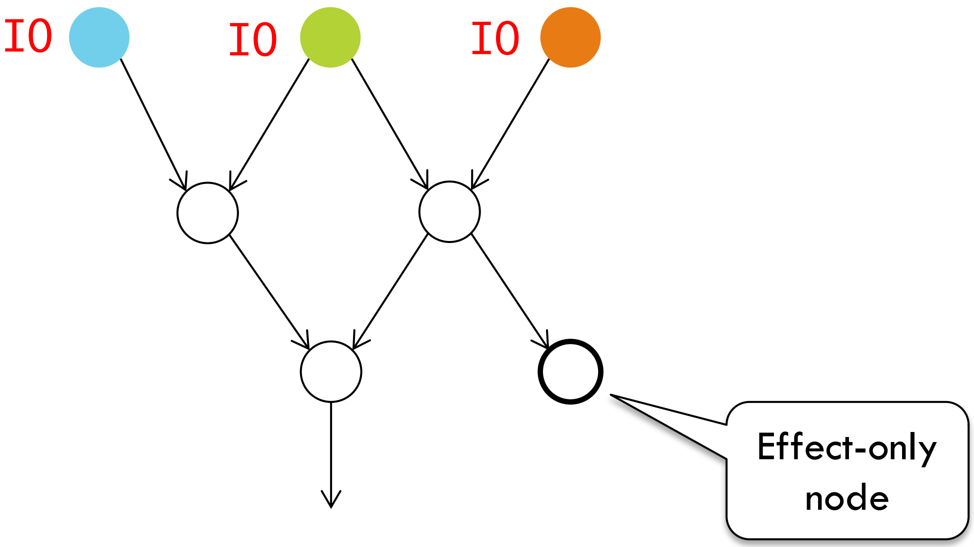

1. Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell* (Haskell '11). ACM, New York, NY, USA, 71-82.

# Smap (De)Construction

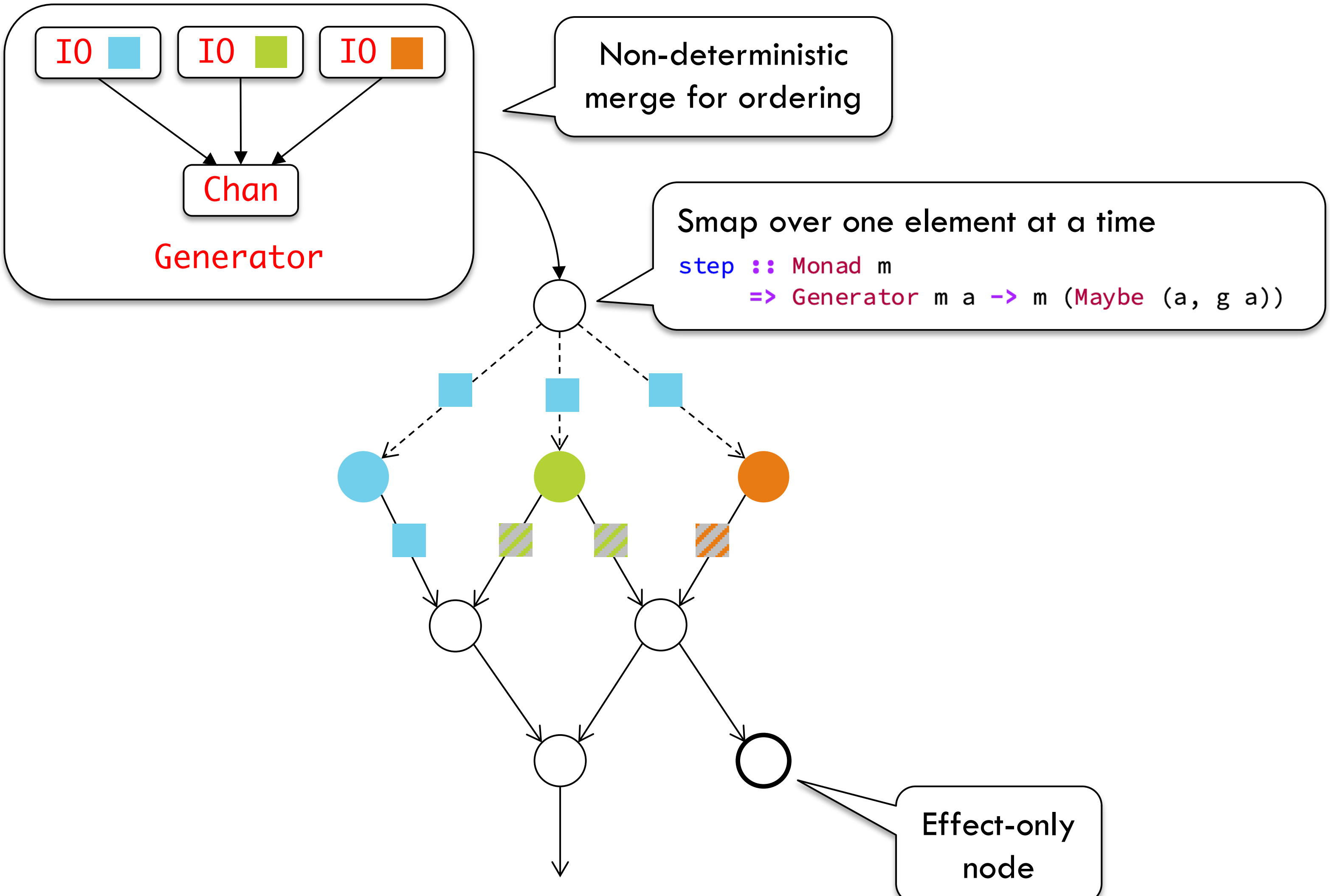



1. Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell* (Haskell '11). ACM, New York, NY, USA, 71-82.

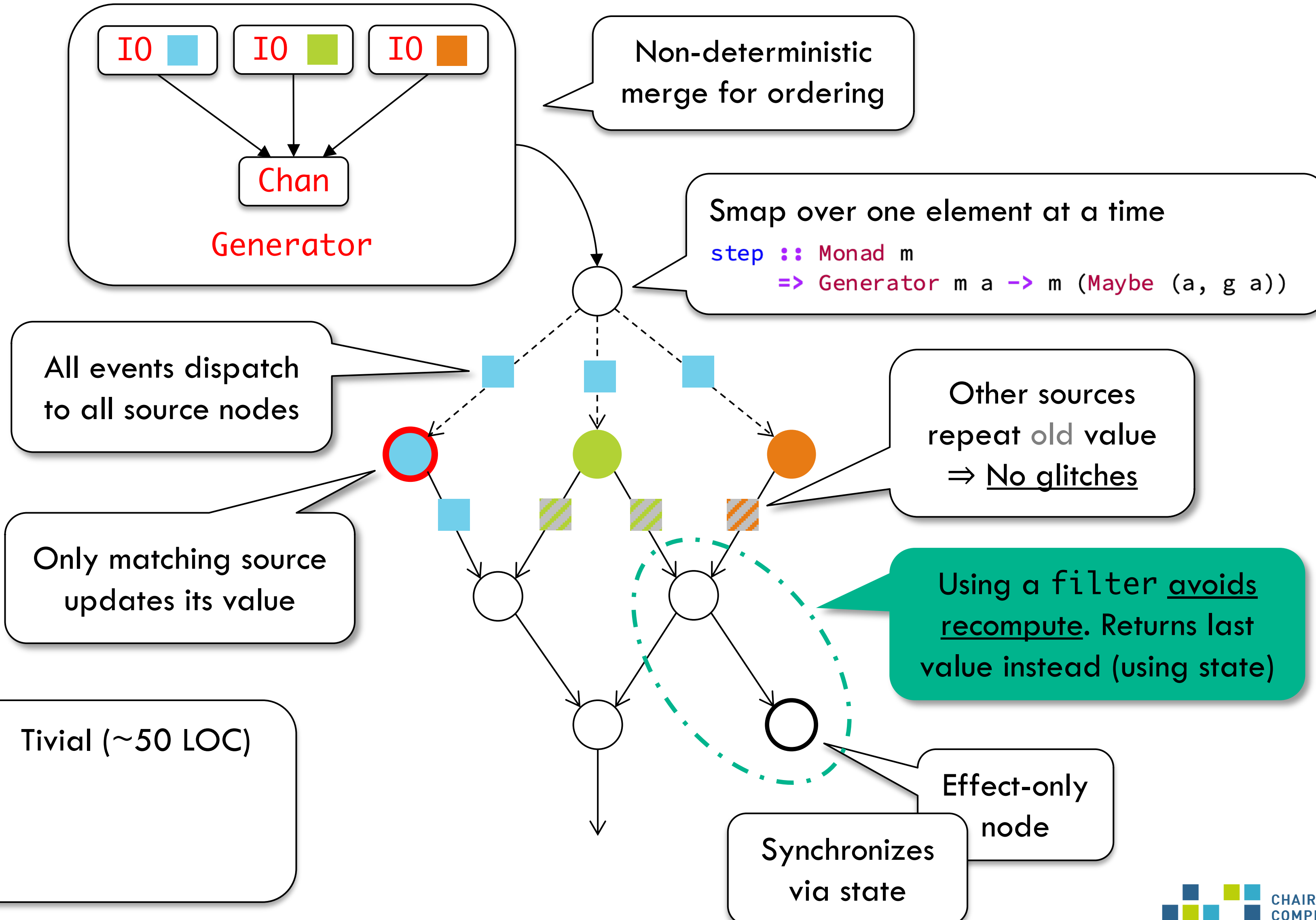
# Functional Reactive Programming



# Functional Reactive Programming



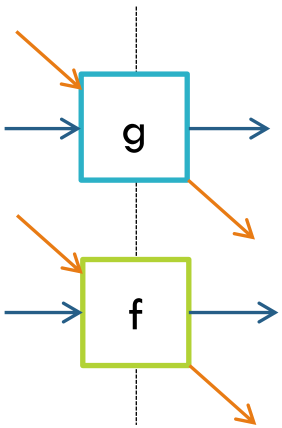
# Functional Reactive Programming




- Tivial (~50 LOC)

# Other Forms for Parallelism

## Task Parallelism




Concisely expressed with the <*> operator

1. Spawned off

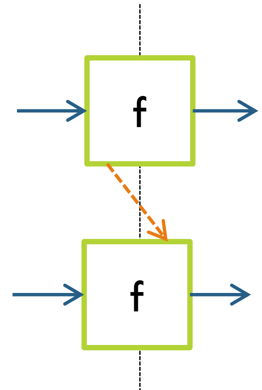
2. Compute this

```
\x -> f x <*> g x
```

3. Collect results & apply

Can be done automatically with the `ApplicativeDo` GHC Extension[1]

## Data Parallelism




Being lazy in a allows s to be sent on first

```
~(a,s) <- run f
```

```
evaluate s
send s
```

```
evaluate a
```

Node Implementation

If s unchanged, virtually free

Next instance of f can start before this point

Works for both read-only and unused state

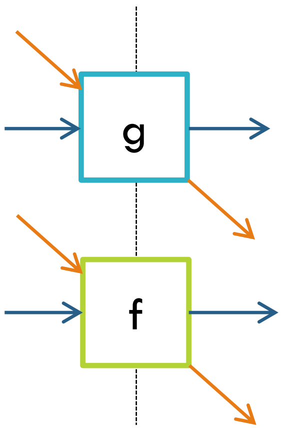
1. Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell* (Haskell 2016). ACM, New York, NY, USA, 92-104.

# Other Sources for Parallelism

- We formalized the model using category theory
- Yields a formal explanation for the forms of extracted parallelism

2. Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, Jeronimo Castrillon. 2019. Category-Theoretic Foundations of "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism". https://arxiv.org/abs/1906.12098

Tasl



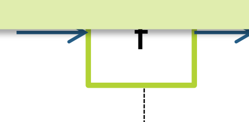

2. Compute this

`\x -> f x <*> g x`

3. Collect results & apply

Can be done automatically with the `ApplicativeDo` GHC Extension[1]

lism




lazy in a allows s be sent on first

```
~(a,s) <- run f
evaluate s
send s
evaluate a
```

Node Implementation

If s unchanged, virtually free

Next instance of f can start before this point

Works for both read-only and unused state

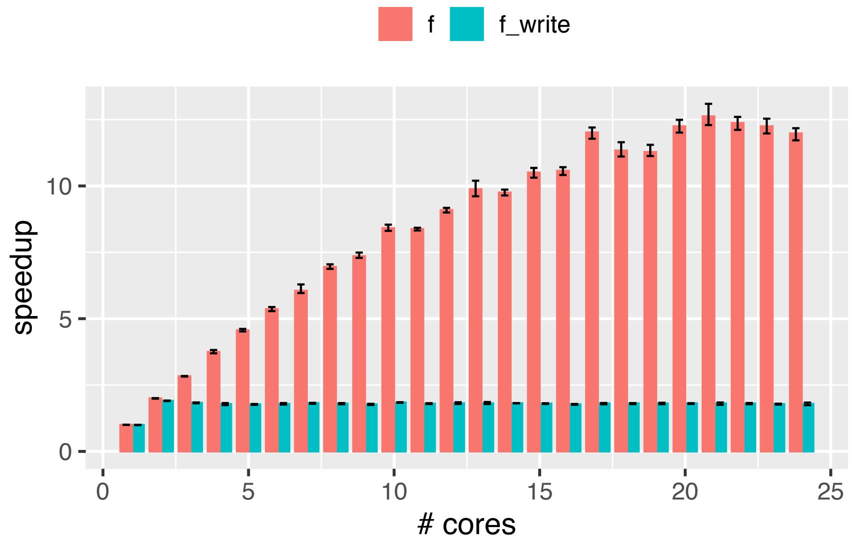
1. Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell* (Haskell 2016). ACM, New York, NY, USA, 92-104.

# Data Parallelism

Benchmark executes State Thread (f or f_write) twice, as a two stage pipeline

```
f i = (i +) <$> get

f_write i = do
    r <- f i
    put r
    pure r
```

Same as f but also writes state




f_write exploits pipeline parallelism (~2x)

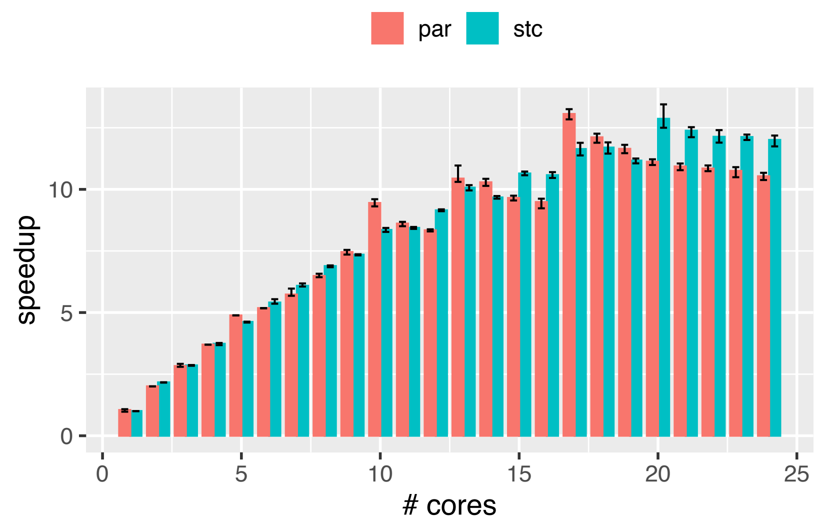
f additionally exploits data parallelism (linear scaling)



CHAIR FOR COMPILER CONSTRUCTION

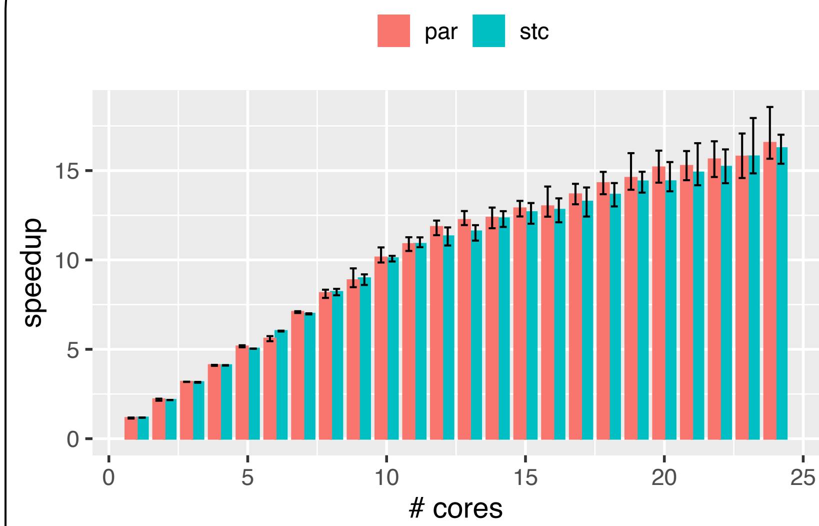
# Microbenchmarks

## Simple Compositions [1]




```
stc w = smap (w >=> w)
par w = parMap (w . w)
```

## Benchmarks from [1]



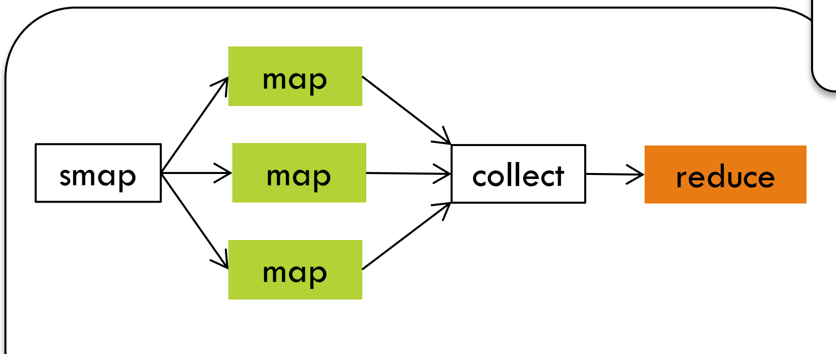

Black-Scholes [2]

STCLang is implemented using the monad-par library

Overhead over manual monad-par implementations is mostly negligible

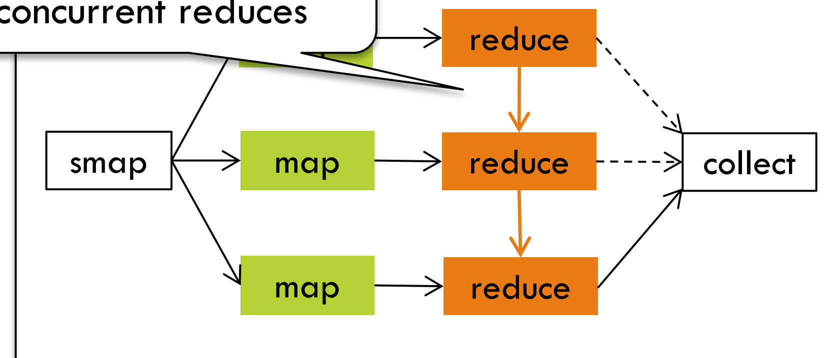
1. For simplicities sake only includes the more favorable monad-par measurement (par2 in the paper)
2. Marlow, Simon, Ryan Newton, and Simon Peyton Jones. "A monad for deterministic parallelism." *ACM SIGPLAN Notices*. Vol. 46. No. 12. ACM, 2011.

# MapReduce Benchmark

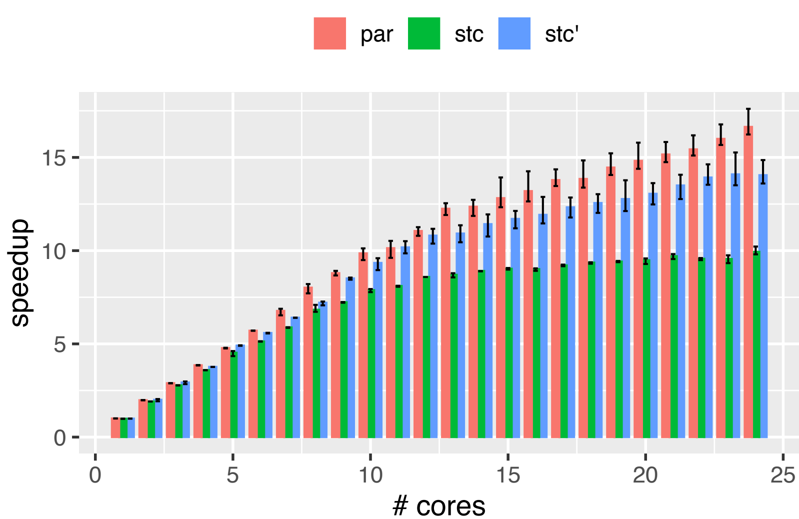



Regular MapReduce

Can express multiple concurrent reduces



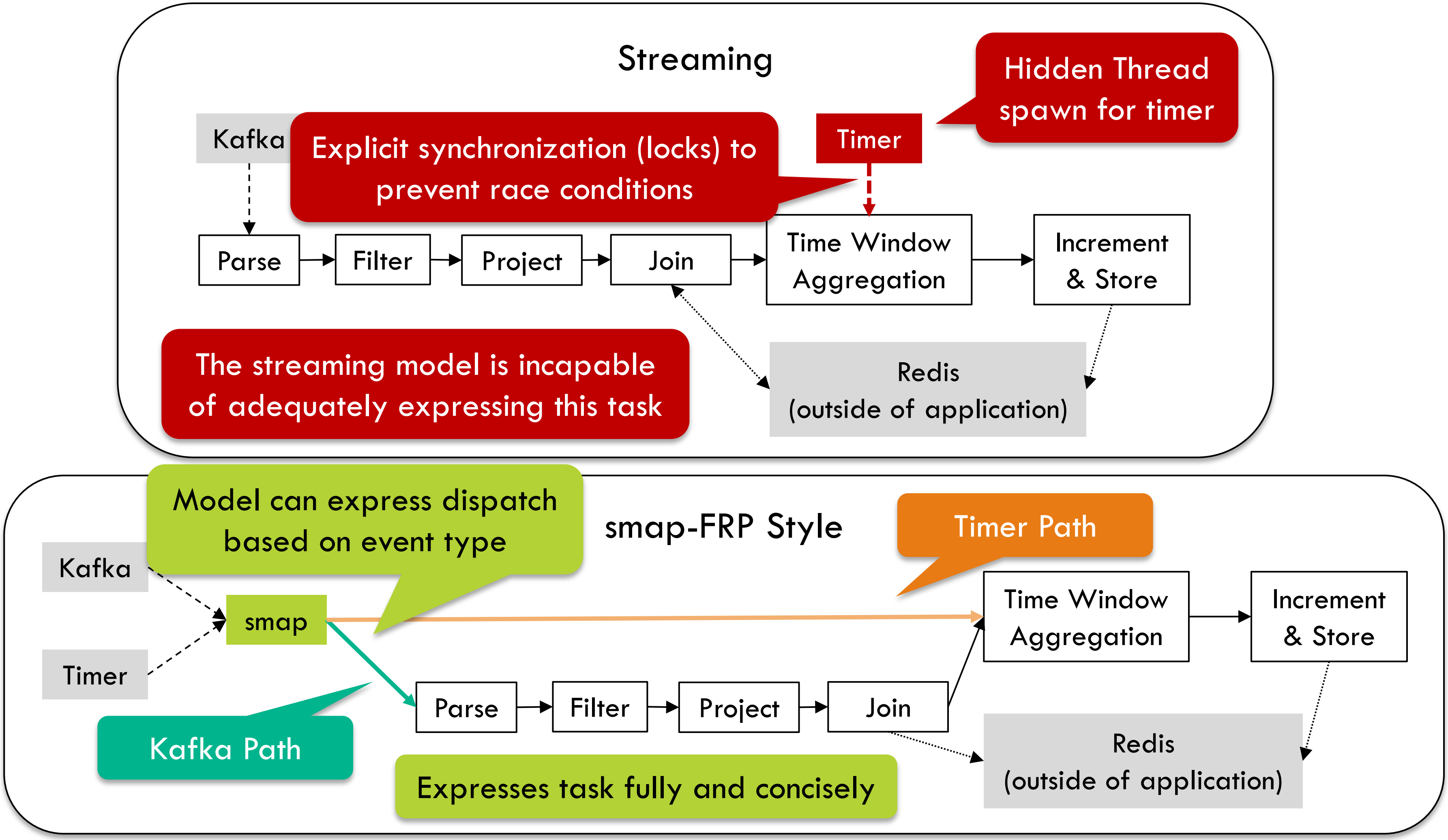

Reduce via State and Map




sum is very efficient, (+) is too cheap for the overhead

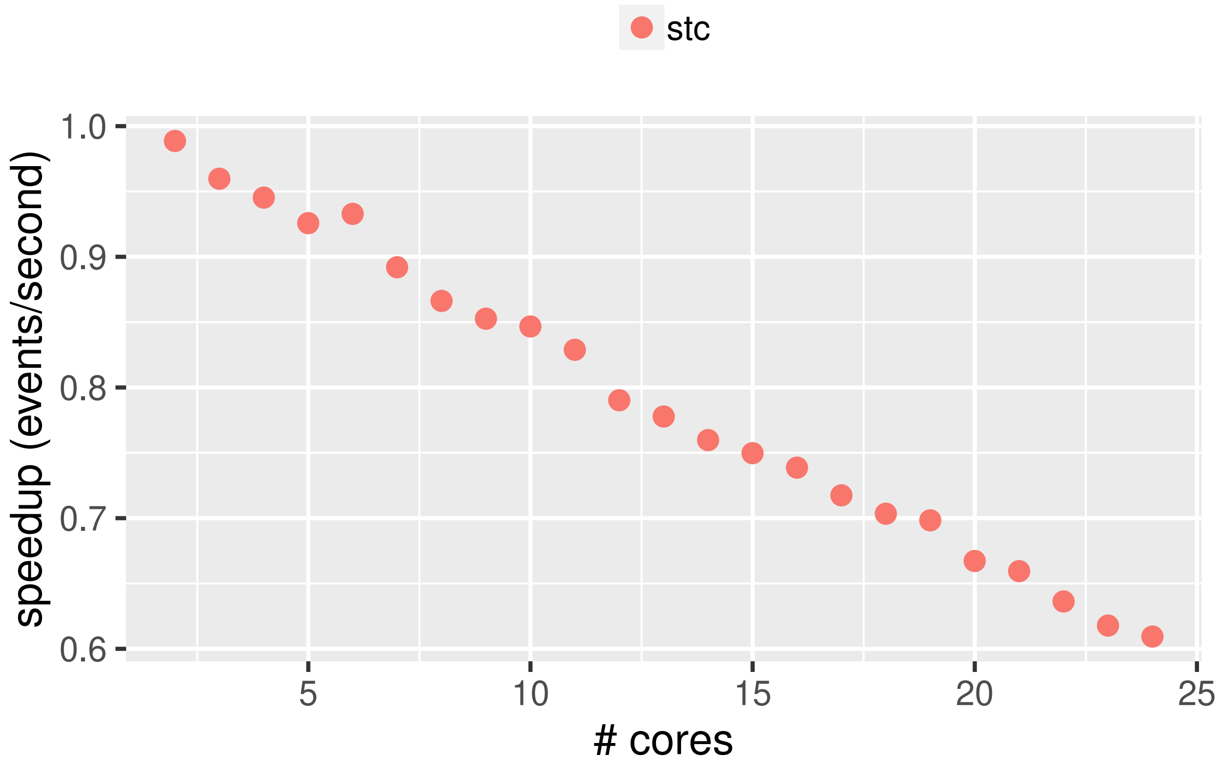
```
par  = sum <$> parMap euler
stc  = mapReduceSTC euler (+) 0
stc' = sum <$> smap euler
```

# Data Streaming Benchmark




1. Sanket Chintapalli, et al. "Benchmarking streaming computation engines: Storm, flink and spark streaming." *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016.

# Data Streaming Benchmark




Performance degrades with parallelism

Work stealing scheduler does not have a notion of output favoured scheduling


1. I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *ACM Trans. Parallel Comput.* 2, 3, Article 17 (September 2015), 42 pages

# Conclusions

**Monad**
State ✅ Parallelism ❌ Composition ✅

**Dataflow**
State ✅ Parallelism ✅ Composition ❌

$State\ Monad$
$+\ Dataflow\ Node =$

$Monad\ Composition$
$+\ Dataflow\ Execution =$

Notion of local state

State Thread

STCLang Monad/smap

Fine grained state structure

Implicit Pipeline-/Task-/Data-parallelism

Leveraging
- Laziness
- ApplicativeDo

MapReduce

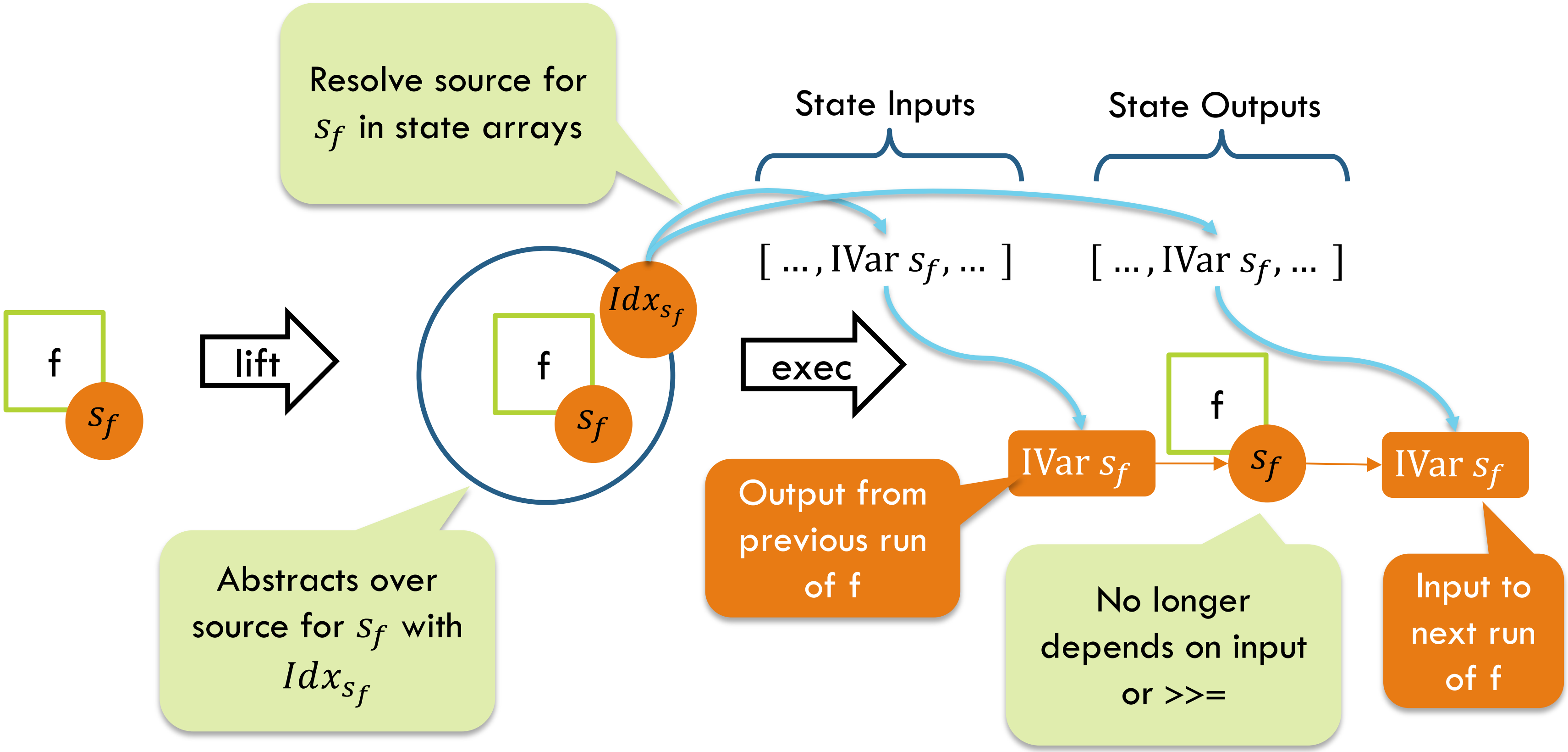
Without explicit reducer

FRP/Stream Processing

- Glitch free by default
- Straightforward

Slides https://ohua-dev.github.io/slides/haskell-19-stclang.pptx (.pdf for pdf)
Repo https://github.com/ohua-dev/stc-lang
Hackage https://hackage.haskell.org/package/stc-lang

# The STC Monad




For Sebastian: This is my backup slide to explain our monad construction. I like it, but its also dense, so I relegated it as backup.

# The Sequential Monad

`f >>= g` ⇒

Monad

- Convenient, familiar
- Inherently sequential

`g <*> f` ⇒

Applicative

- Only task parallelism
- No flow from `f` to `g` before the effects of `f`

`f >>> g` ⇒

Arrow

- Effect composition independent from data
- Does anyone use Arrows? (You should though, they're cool)

# State Thread Examples

```haskell
memoized :: Ord a
         => (a -> b) -> a
         -> State (LRUChache a b) b
memoized operation elem = do
  cache <- get
  case lookup elem cache of
    Just result -> return result
    Nothing -> do
      let res = operation elem
      put $ insert elem result cache
      return result
```

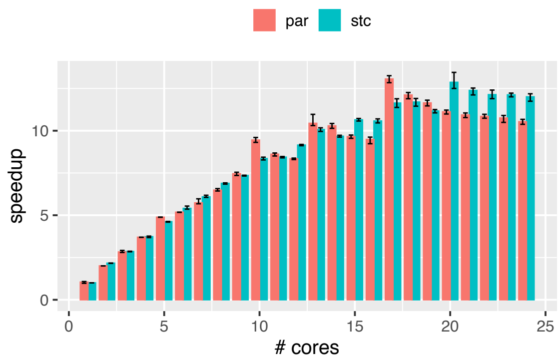
```haskell
windowedAvg :: Int -> Float
            -> State [Float] Float
windowedAvg wsize i = do
  win <- get
  let win’ = take wsize $ i : win
  put win’
  return $ sum win’ `div` realToFrac wsize
```

1. Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A monad for deterministic parallelism. In Proceedings of the 4th ACM symposium on Haskell (Haskell '11).



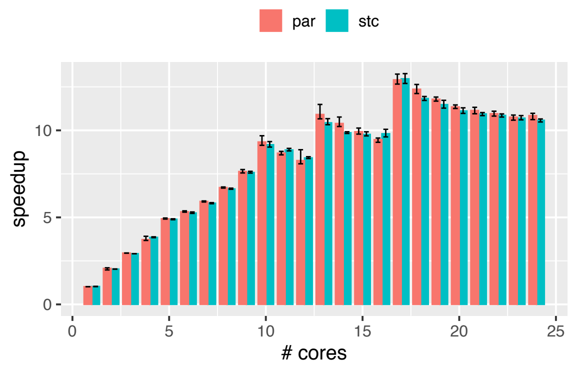
CHAIR FOR COMPILER CONSTRUCTION

# Microbenchmarks



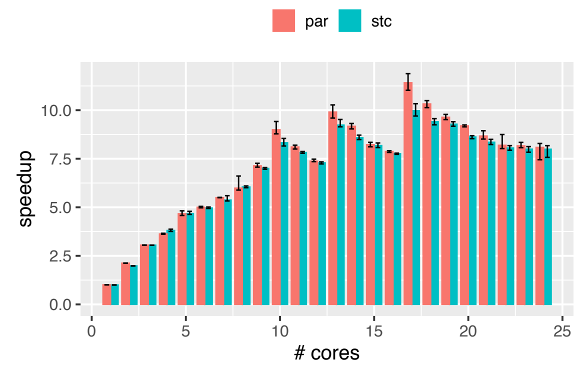

[1]

```
stc w = smap (w >=> w)
par w = parMap (w . w)
```




[2]

```
go w = (\x ->
          (,) <$> w x
              <*> w x)

stc = smap . go
par = parMapM . go
```



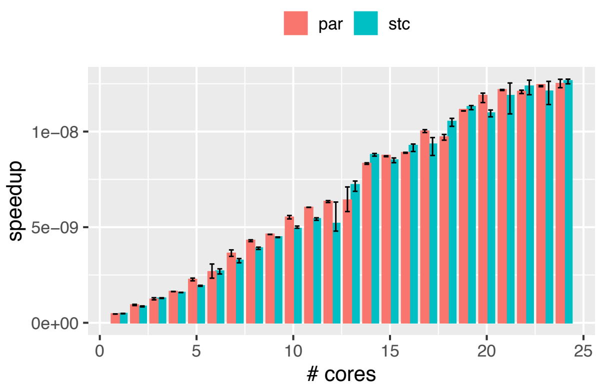

```
go cond w x =
    if cond x
       then w x
       else w x

stc = smap . go even?
par = parMap . go even?
```

1. For simplicities sake only includes the more favorable monad-par measurement (par2 in the paper)
2. NOINLINE version of the benchmark, as it is representative of what we want to test. See the paper for the complete rational

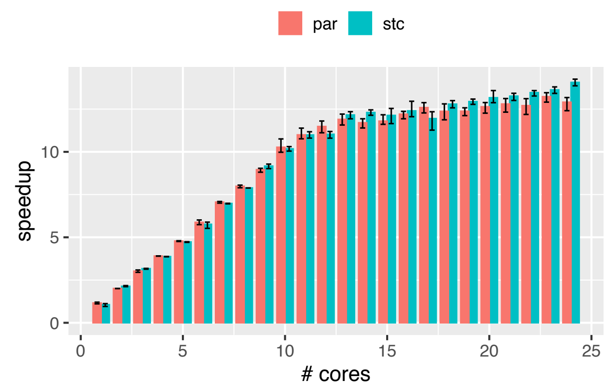


CHAIR FOR COMPILER CONSTRUCTION

# Benchmarks








Matrix Multiplication [1]

Black-Scholes [1]

Mandelbrot [1]

No slowdown
(with respect to par)

Near-linear scaling

1. Marlow, Simon, Ryan Newton, and Simon Peyton Jones. "A monad for deterministic parallelism." *ACM SIGPLAN Notices*. Vol. 46. No. 12. ACM, 2011.



CHAIR FOR COMPILER CONSTRUCTION