Parallel Functional Programming
the Par monad

Mary Sheeran

with thanks to Simon Marlow for use of slides

par and pseq

MUST

Pass an unevaluated computation to par

It must be somewhat expensive

Make sure the result is not needed for a bit

Make sure the result is shared by the rest of the program

par and pseq

MUST

Pass an unevaluated computatio to par

It must be somewhat expensive

Make sure the result is not need a bit

Make sure the result is shared b st of the program

Demands an operational understanding of program execution

Eval monad plus Strategies

Eval monad enables expressing ordering between instances of par and pseq

Strategies separate algorithm from parallelisation

Provide useful higher level abstractions

Rely on constructing a lazy data structure

Still demand an understanding of laziness

A small aside on the Eval monad

(due to Petricek, see paper on Canvas)

Haskell now (again) has monad comprehensions, as distinct from just list comprehensions

Also has parallel list comprehensions

A small aside on the Eval monad

Haskell now (again) has monad comprehensions, as distrinct from just list comprehensions

Also has parallel list comprehensions

though not in our sense!

Ordinary list comprehension

```
ghci> [a+b | a <- [1..4], b <-[1..3]]
[2,3,4, 3,4,5, 4,5,6, 5,6,7]
```

Ordinary list comprehension

```
ghci> [a+b | a <- [1..4], b <-[1..3]]
[2,3,4, 3,4,5, 4,5,6, 5,6,7]
```

How many elements in
[a+b+c | a <- [1..4] , b <-[1..3] , c <- [1..2]]
?

Ordinary list comprehension

```
ghci> [a+b | a <- [1..4], b <-[1..3]]
[2,3,4, 3,4,5, 4,5,6, 5,6,7]
```

How many elements in

```
[a+b+c | a <- [1..4] , b <-[1..3] , c <- [1..2]]
```

```
[3,4, 4,5, 5,6,
 4,5, 5,6, 6,7,
 5,6, 6,7, 7,8,
 6,7, 7,8, 8,9]
```

Parallel list comprehension

```
ghci> [a+b | a <- [1..4] , b <-[1..3]]
```

```
ghci> [a+b | a <- [1..4] | b <-[1..3]]
```

Parallel list comprehension

```
ghci> [a+b | a <- [1..4] , b <-[1..3]]
```

```
ghci> [a+b | a <- [1..4] | b <-[1..3]]
```

Answer??

Parallel list comprehension

```
ghci> [a+b | a <- [1..4] , b <-[1..3]]
```

```
ghci> [a+b | a <- [1..4] | b <-[1..3]]
```

Answer??

```
[2,4,6]
```

Parallel programs as comprehensions

Enabled by

Monad comprehensions + parallel list comprehensions

Parallel programs (in our sense) as comprehensions

Enabled by

Monad comprehensions + parallel list comprehensions

Note also Petricek's other example using Poor Man's Concurrency Monad (which also underlies the Par monad)

Type classes

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

class (Monad m) => MonadZip m where
  mzip :: m a -> m b -> m (a, b)
```

Instances for List

```
instance Monad [] where
  source >>= f = concat $ map f source
  return a = [a]

instance MonadZip [] where
  mzip = zip
```

Desugaring

```
[a+b | a <- [1..4] , b <-[1..3]]
```

Desugaring

```
[a+b | a <- [1..4] , b <-[1..3]]
```

[1..4] >>= (\a -> [1..3] >>= (\b -> return $ a+b))

Desugaring

```
[a+b | a <- [1..4] | b <-[1..3]]
```

([1..4] ‘mzip‘ [1..3]) >>= (\(a, b) -> return $ a + b)

MonadZip instance for Eval

```
instance MonadZip Eval where
  mzip ea eb = do
     a <- rpar $ runEval ea
     b <- rseq $ runEval eb
     return (a, b)
```

MonadZip instance for Eval

```
instance MonadZip Eval where
  mzip ea eb = do
     a <- rpar $ runEval ea
     b <- rseq $ runEval eb
     return (a, b)
```

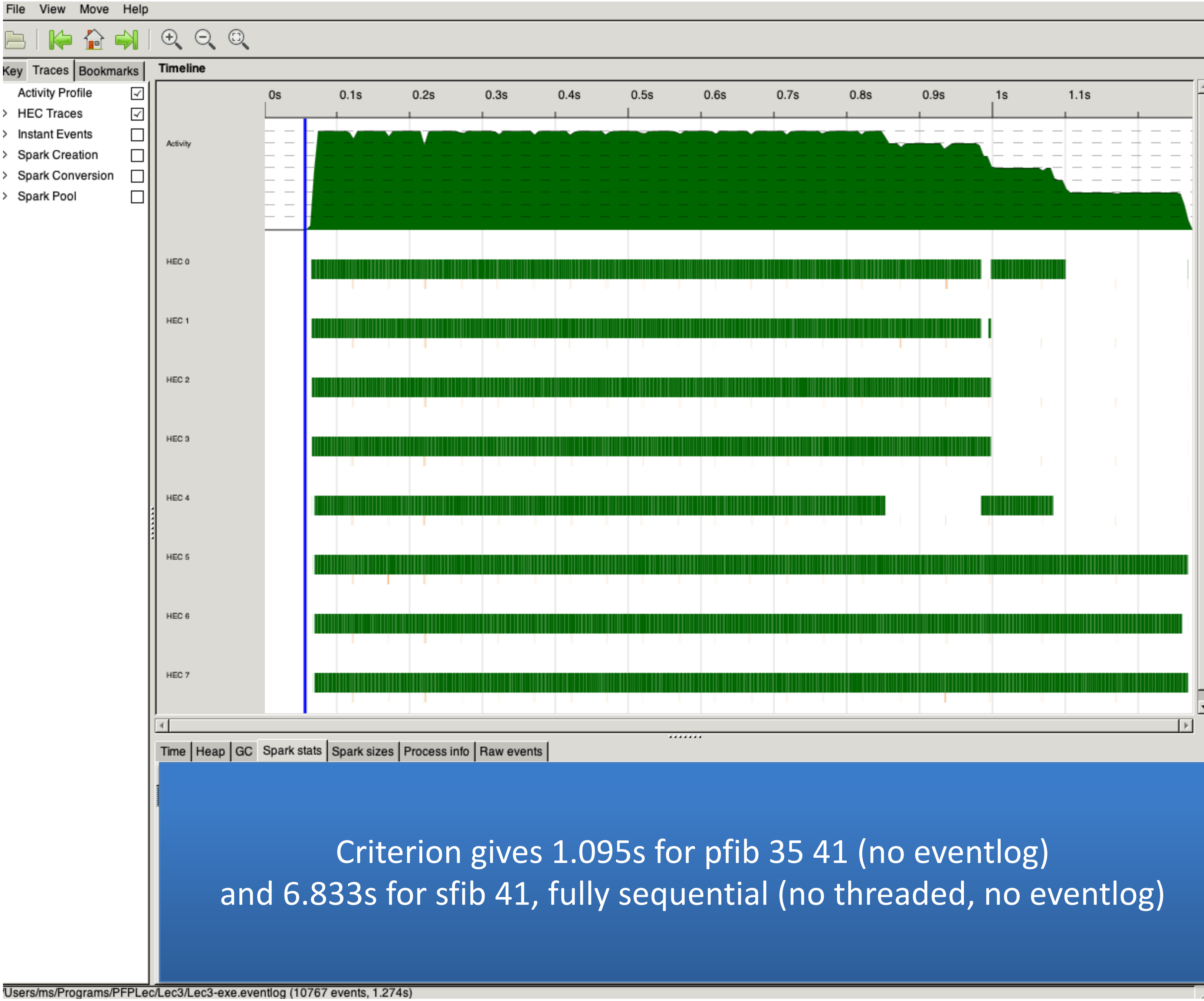
```
mzip :: Eval a -> Eval a -> Eval (a,b)
```

Sequential

```
fibs :: Integer -> Integer -> Eval Integer
fibs t n | n <= t = return $ sfib n
fibs t n  = [ a + b + 1 | a <- fibs t $ n - 1
                        , b <- fibs t $ n - 2 ]
```

Parallel

```
pfib :: Integer -> Integer -> Eval Integer
pfib t n  | n <= t = return $ sfib n
pfib t n  = [ a + b + 1 | a <- pfib t $ n - 1
                        | b <- pfib t $ n - 2 ]
```



Criterion gives 1.095s for pfib 35 41 (no eventlog)
and 6.833s for sfib 41, fully sequential (no threaded, no eventlog)

pfib 35 41

Parallel programming models

	Deterministic	Non-deterministic
Implicit	FDIP	
↕	par/pseq Strategies	
Explicit	???	Concurrent Haskell

(Slide by Simon Marlow)

A monad for deterministic parallelism

Simon Marlow
Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Ryan Newton
Intel, Hudson, MA, U.S.A
ryan.r.newton@intel.com

Simon Peyton Jones
Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com

Abstract

We present a new programming model for deterministic parallel computation in a pure functional language. The model is monadic and has explicit granularity, but allows dynamic construction of dataflow networks that are scheduled at runtime, while remaining deterministic and pure. The implementation is based on monadic concurrency, which has until now only been used to simulate concurrency in functional languages, rather than to provide parallelism. We present the API with its semantics, and argue that parallel execution is deterministic. Furthermore, we present a complete work-stealing scheduler implemented as a Haskell library, and we show that it performs at least as well as the existing parallel programming models in Haskell.

pure interface, while allowing a parallel implementation. We give a formal operational semantics for the new interface.

Our programming model is closely related to a number of others; a detailed comparison can be found in Section 8. Probably the closest relative is pH (Nikhil 2001), a variant of Haskell that also has I-structures; the principal difference with our model is that the monad allows us to retain referential transparency, which was lost in pH with the introduction of I-structures. The target domain of our programming model is large-grained irregular parallelism, rather than fine-grained regular data parallelism (for the latter Data Parallel Haskell (Chakravarty et al. 2007) is more appropriate).

Our implementation is based on *monadic concurrency* (Scholz 1995), a technique that has previously been used to good effect to simulate concurrency in a sequential functional language (Claessen

Builds on Koen Claessen's paper (PCM)

FUNCTIONAL PEARLS

A Poor Man's Concurrency Monad

Koen Claessen
Chalmers University of Technology
email: koen@cs.chalmers.se

Abstract

Without adding any primitives to the language, we define a concurrency monad transformer in Haskell. This allows us to add a limited form of concurrency to any existing monad. The atomic actions of the new monad are lifted actions of the underlying monad. Some extra operations, such as `fork`, to initiate new processes, are provided. We discuss the implementation, and use some examples to illustrate the usefulness of this construction.

JFP'99

the Par Monad

Our goal with this work is to find a parallel programming model that is expressive enough to subsume Strategies, robust enough to reliably express parallelism, and accessible enough that non-expert programmers can achieve parallelism with little effort.

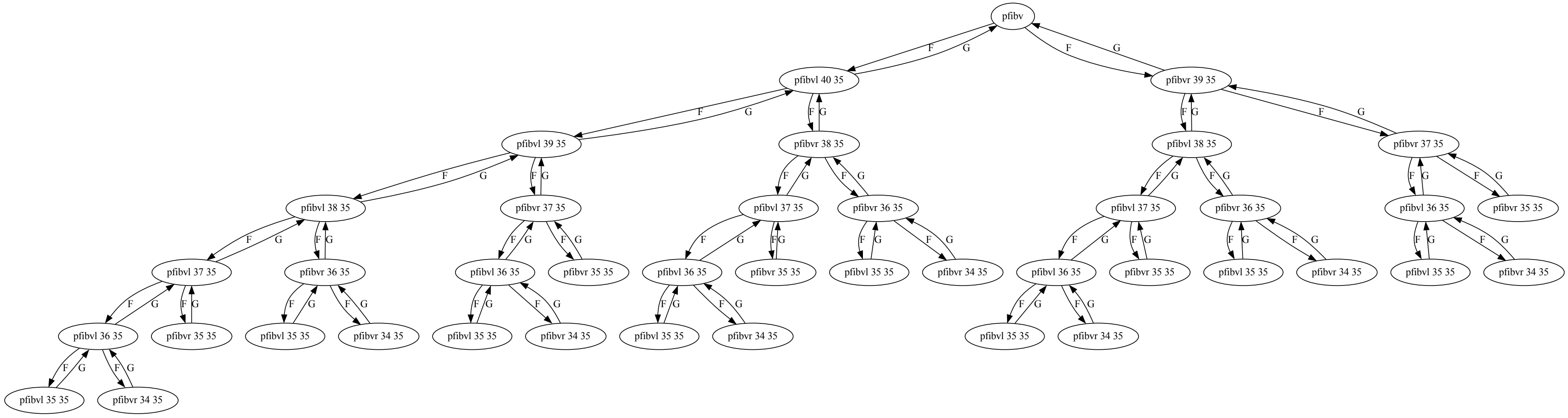
```haskell
pfib :: Integer -> Integer -> Par Integer
pfib n t | n<=t       = return $ sfib n
         | otherwise = do
             leftVar  <- spawn $ pfib (n-1) t
             rightVar <- spawn $ pfib (n-2) t
             left     <- get leftVar
             right    <- get rightVar
             return $ left + right + 1
```

```
pfib :: Integer -> Integer -> Par Integer
pfib n t | n<=t       = return $ sfib n
         | otherwise = do
             leftVar  <- spawn $ pfib (n-1) t
             rightVar <- spawn $ pfib (n-2) t
             left     <- get leftVar
             right    <- get rightVar
             return $ left + right + 1
```

Unfortunately, right now there's no way to generate a visual representation of the data flow graph from some Par monad code, but hopefully in the future someone will write a tool to do that.
[Simon Marlow, PCPH]

Max Algehed (PFP student / TA) to the rescue

```
pfibv :: Integer -> Integer -> Par Integer
pfibv n t | n <=t       = return $ sfib n
          | otherwise  = do
            leftVar    <- spawnNamed
                           (disp "pfibvl" (n-1) t) $ pfibv (n-1) t
            rightVar   <- spawnNamed
                           (disp "pfibvr" (n-2) t) $ pfibv (n-2) t
            left    <- get leftVar
            right   <- get rightVar
            return $ left + right + 1
    where
      disp s v0 v1 = s ++ " " ++ show v0 ++ " " ++ show v1
```



code https://github.com/MaximilianAlgehed/VisPar

Paper FHPC 2017 https://dl.acm.org/doi/10.1145/3122948.3122953

Nice workshop at ICFP: FHPNC

ICFP 2023 (series) / FHPNC 2023 (series) /

FHPNC 2023

About | Call for Papers

The ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing aims to bring together researchers and practitioners exploring or employing the use of functional or declarative programming languages or techniques in scientific computing, and specifically in the domains of high-performance computing and numerical programming.

The purpose of the meeting is to enable sharing of results, experiences, and novel ideas about how high-level, declarative techniques can help make high-performance, distributed/parallel, or numerically-intensive code dealing with computationally challenging problems easier to write, read, maintain, or portable to new architectures. Areas of interest include, but are not limited to, relevant compiler technologies, runtime systems (including fault tolerance mechanisms and those supporting distributed or parallel computation), domain-specific languages (embedded or otherwise), type systems, algebraic differentiation, formal methods, and libraries (e.g. for exact or interval arithmetic).

This event, now in its third instance, is originally a merger of two workshops that took place during previous editions of ICFP : FHPC (Functional High-Performance Computing) and NPFL (Numerical Programming in Functional Languages), and as such it aims to foster the exchange of ideas between the two communities.

Last time, FHPNC 2021 was held as an online only event, and we are every much looking forward to meeting in person this year!

Important Dates — AoE (UTC-12h)

Wed 31 May 2023 new
Submission deadline

Tue 18 Jul 2023 new
Camera-ready deadline

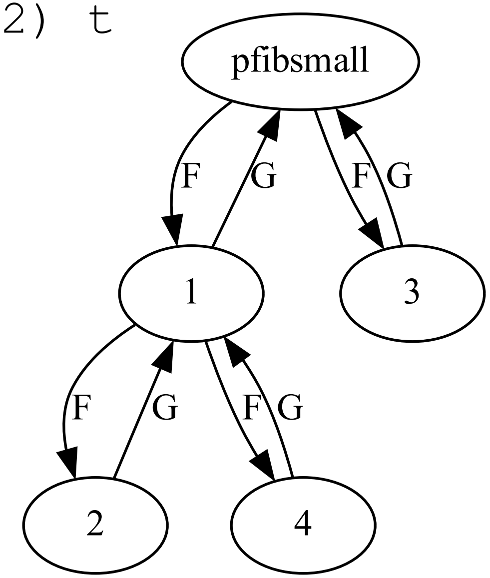
Organizing Committee

Gabriele Keller — Co-chair
Utrecht University
Netherlands

Sam Westrick — Co-chair
Carnegie Mellon University
United States

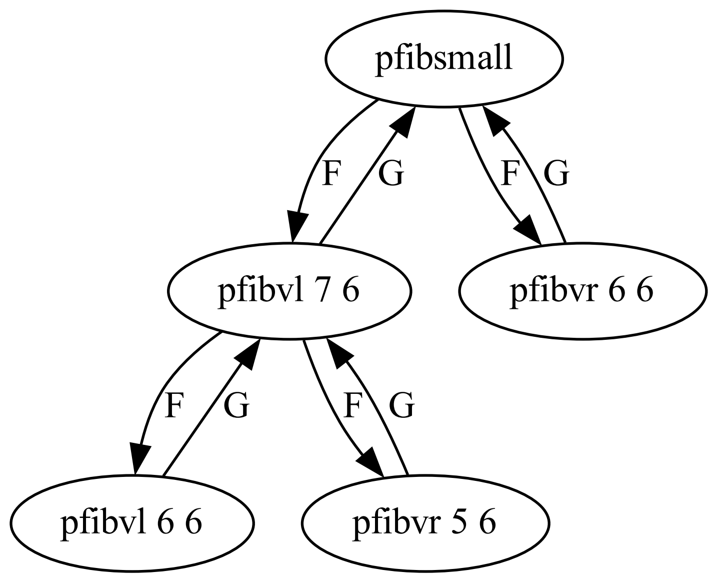
Compact graph (no explicit naming)

```
pfib :: Integer -> Integer -> Par Integer
pfib n t | n<=t       = return $ sfib n
         | otherwise = do
             leftVar  <- spawn $ pfib (n-1) t
             rightVar <- spawn $ pfib (n-2) t
             left     <- get leftVar
             right    <- get rightVar
             return $ left + right + 1
```




```
main = do
  g  <- visPar Compact "pfibsmall" (pfib 8 6)
  saveGraphPdf Vertical "pfibsmall.compact.graph.pdf" g
```

Compact graph (naming)



The Par interface

```haskell
-- Monad operations
return :: a -> Par a
(>>=)  :: Par a -> (a ->  Par b)  -> Par b
(>>)   :: Par a ->         Par b   -> Par b

-- Primitive operations
fork   :: Par () -> Par ()
new    :: Par (IVar a)
put    :: NFData a =>  IVar a -> a -> Par ()
get    :: IVar a -> Par a
runPar :: Par a -> a -- Work stealing scheduler

-- Derived operation
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do v <- new
             fork (p >>= put v)
             return v
```

(Slide by Jansson / Algehed)

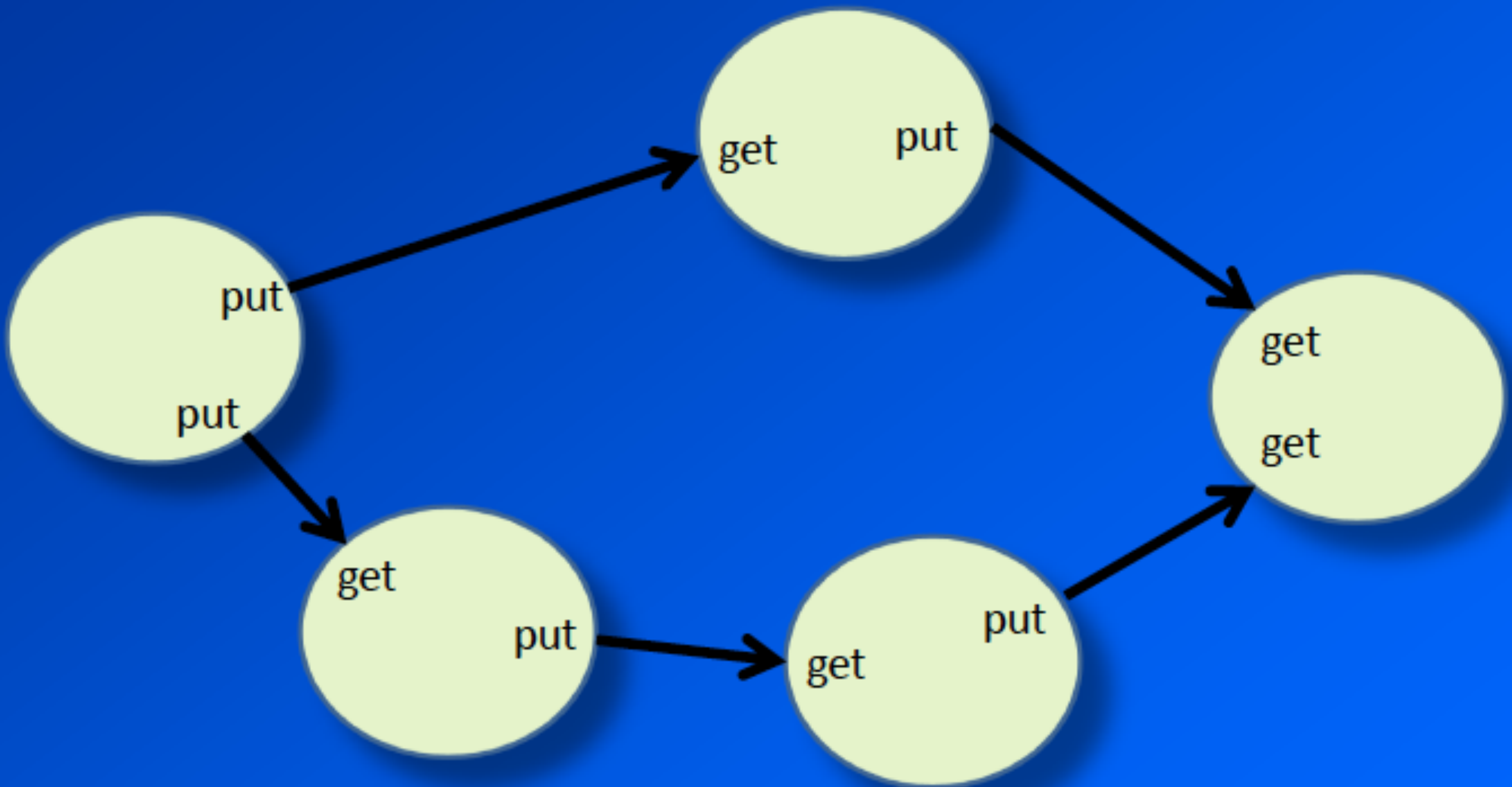
fork :: Par () -> Par ()

Fork the child to run at the same time as the current computation

takes a Par computation and runs it in parallel with the rest of the program as a light-weight “thread”

The type indicates that results from the new thread must be communicated through other means (using Ivars)

Par expresses dynamic dataflow



Slide by Simon Marlow

fork

Not so useful by itself. Need a way to communicate from child to parent

Ivar (new,put, get)

a write-once mutable reference cell

supports two operations: put and get

put assigns a value to the IVar, and may only be executed once per Ivar
Subsequent puts are an error

get waits until the IVar has been assigned a value, and then returns the value

IVar

a write-once mutable reference cell

supports two operations: put and get

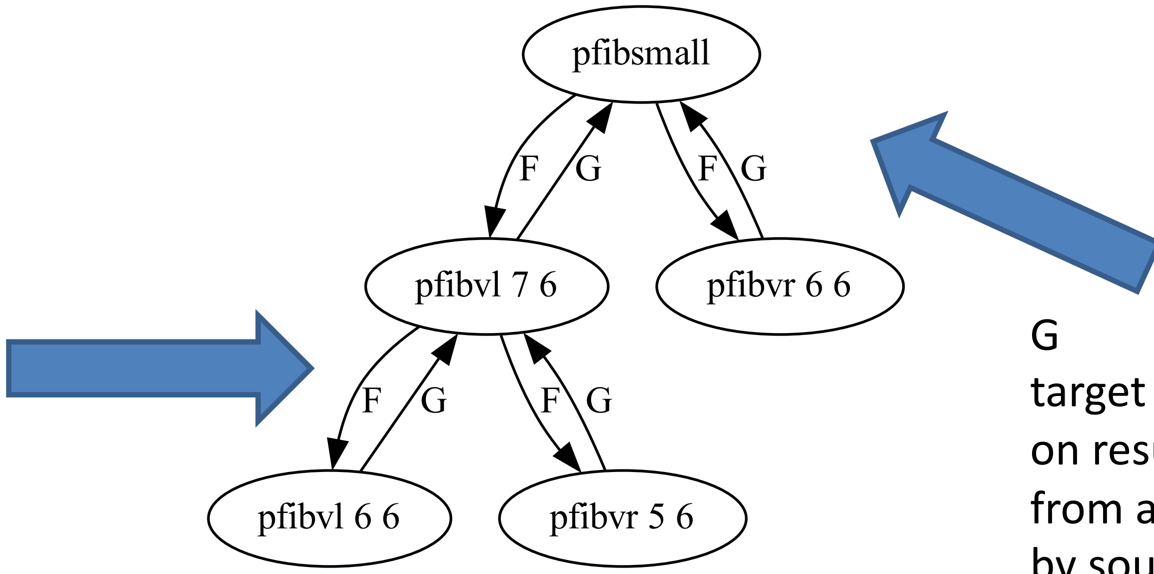
put assigns a value to the IVar, and may only be executed once per Ivar
Subsequent puts are an error

get waits until the IVar has been assigned a value, a
then returns the val

See [i-structures]

Compact graph (naming)

F
Source forked target



G
target event depends on result of a get from an IVAR filled by source

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
     i <- new
     fork (do x <- p; put i x)
     return i
```

a single child computation that returns a result

Restricting yourself to spawn (and not using fork new and put) avoids multiple-put errors on Ivars

```haskell
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
   ibs <- mapM (spawn . f) as
   mapM get ibs
```

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
   ibs <- mapM (spawn . f) as
   mapM get ibs
```

```
mapM :: (Monad m, Traversable t) => (a -> m b) -> t a -> m (t b)
```

The Par interface, extended for VisPar

```haskell
-- Primitive operations
fork   :: Par () -> Par ()
new    :: Par (IVar a)
put    :: NFData a =>  IVar a -> a -> Par ()
get    :: IVar a -> Par a
runPar :: Par a -> a -- Work stealing scheduler

-- Derived operation
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do v <- new
             fork (p >>= put v)
             return v

-- We add
forkNamed :: String -> Par () -> Par ()
visPar    :: Options -> Par a -> IO () -- Produces a PDF
```

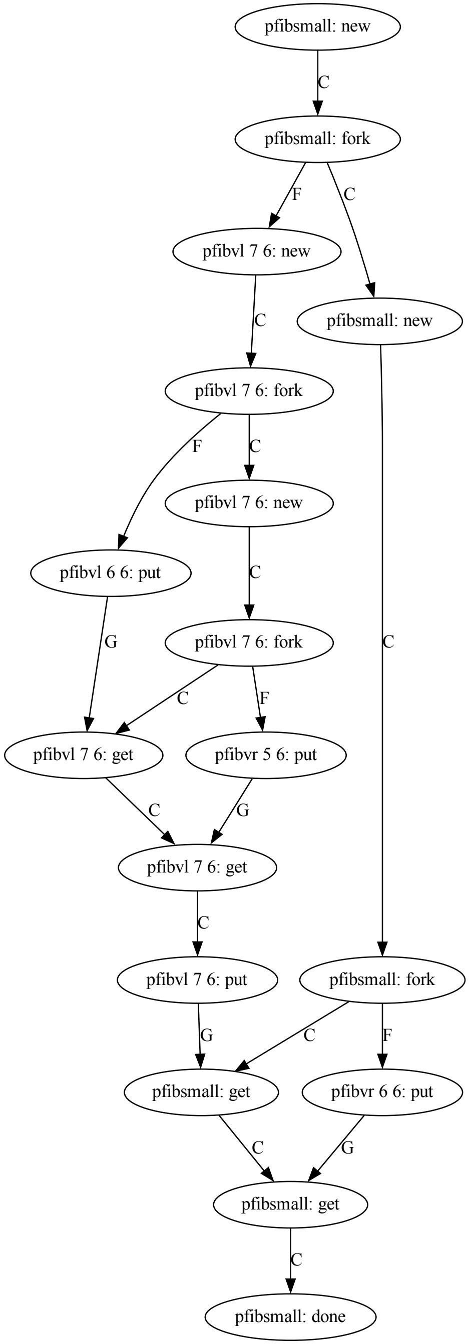
(Slide by Jansson / Algehed)

Extended graph

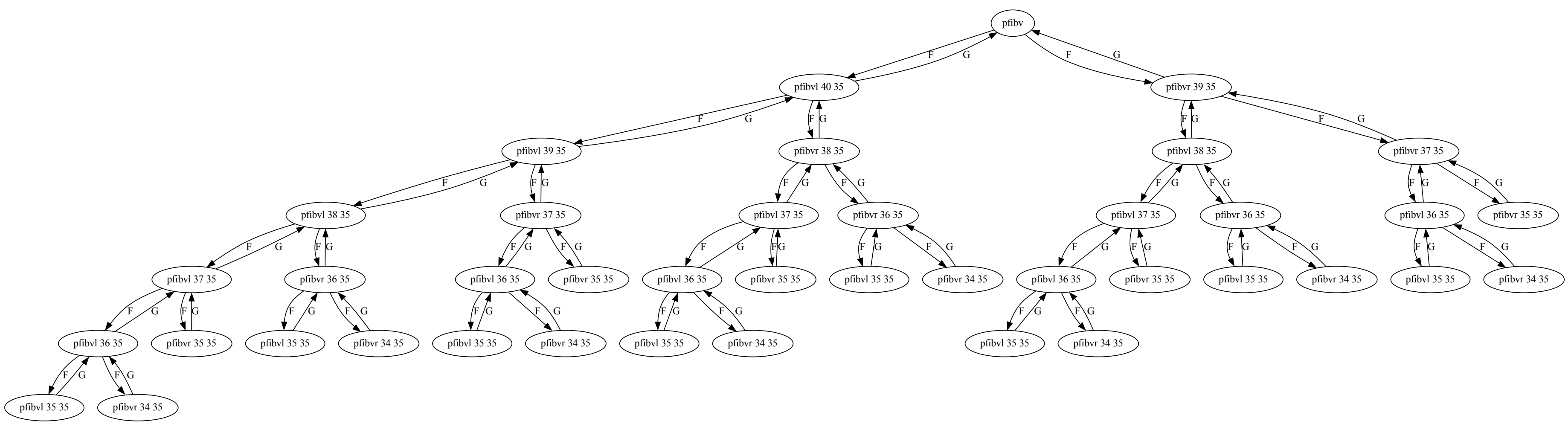
More info

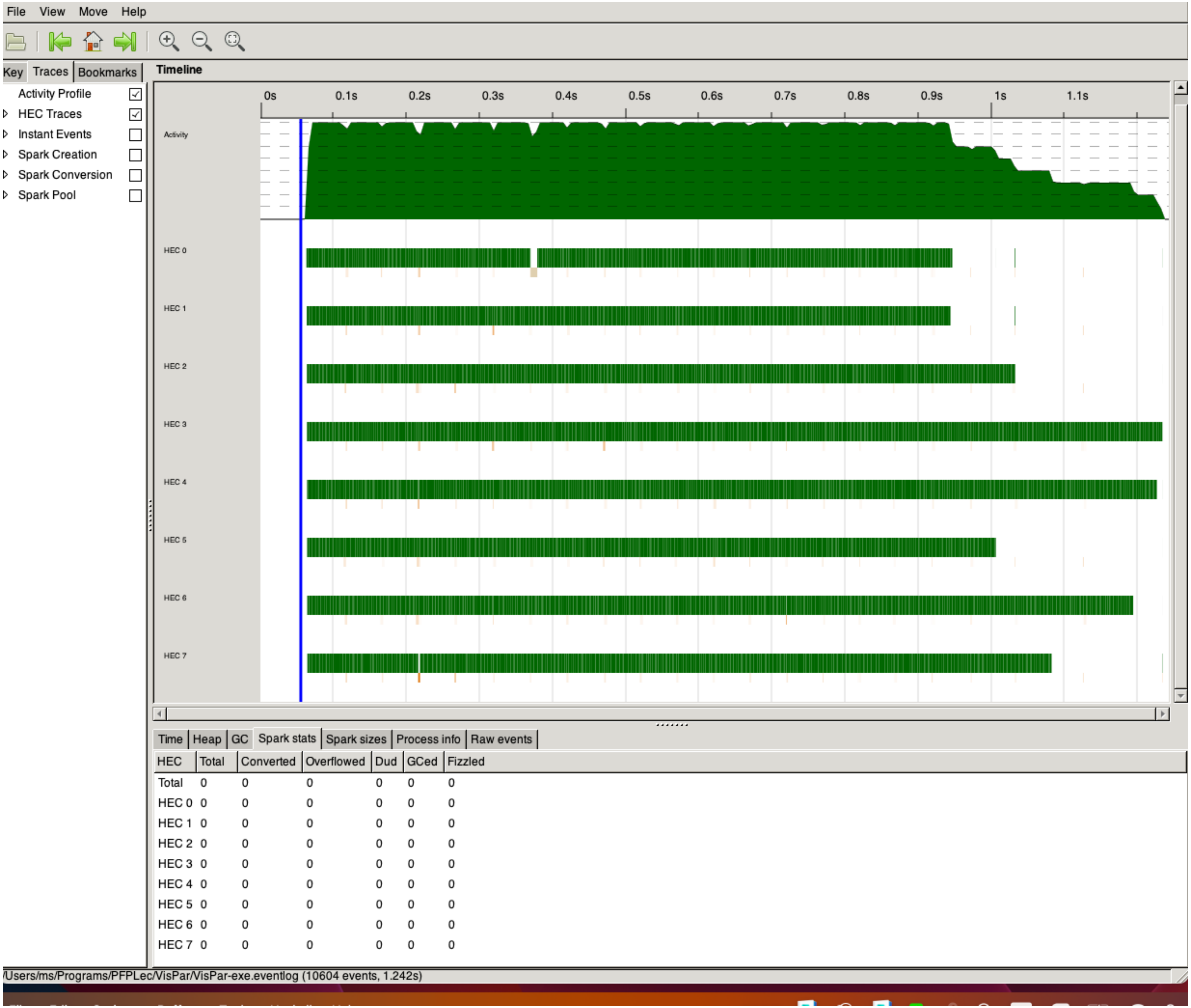
C means continuing in
same thread

Compact graphs have only F,G edges

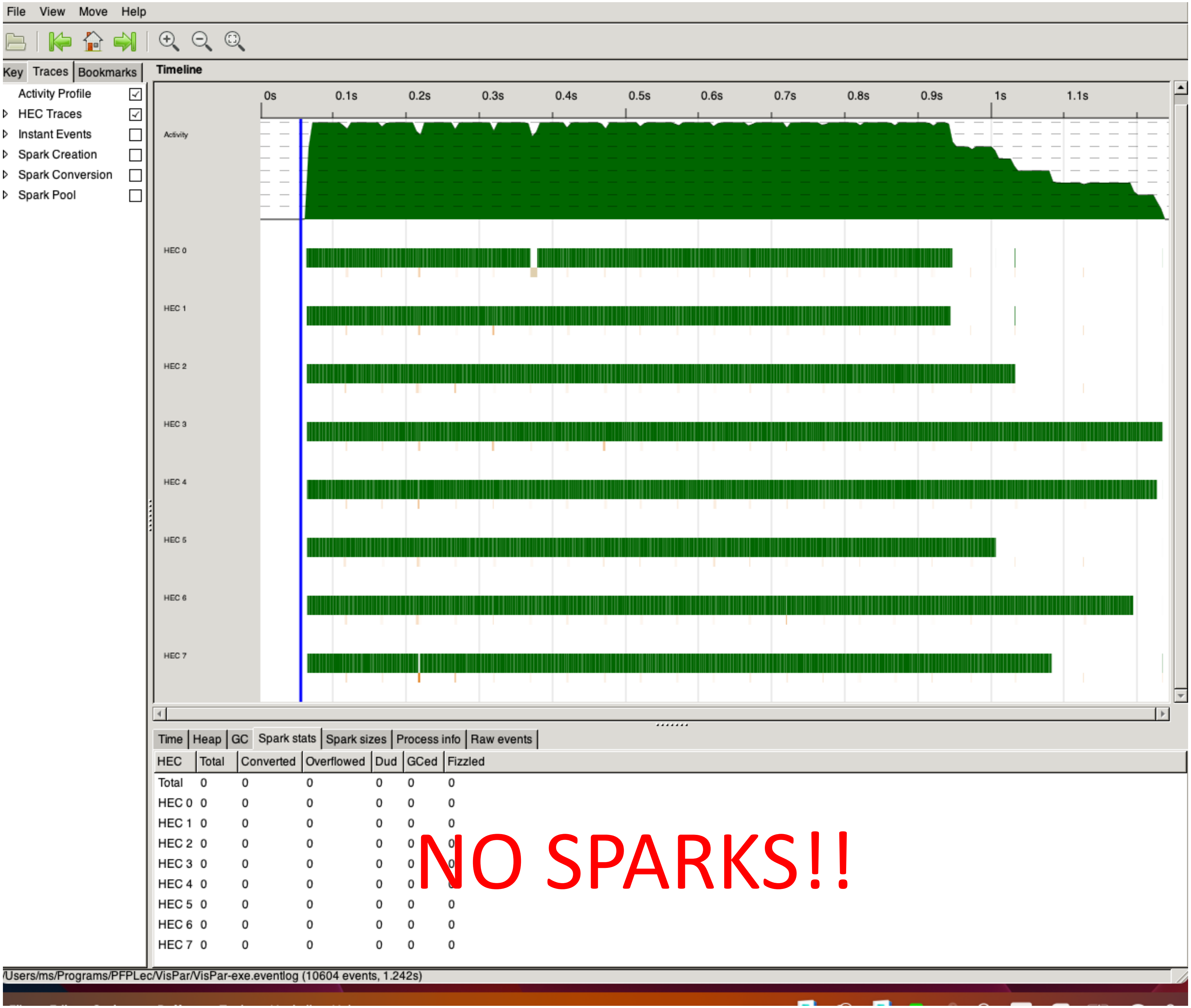


Back to pfib 41 35



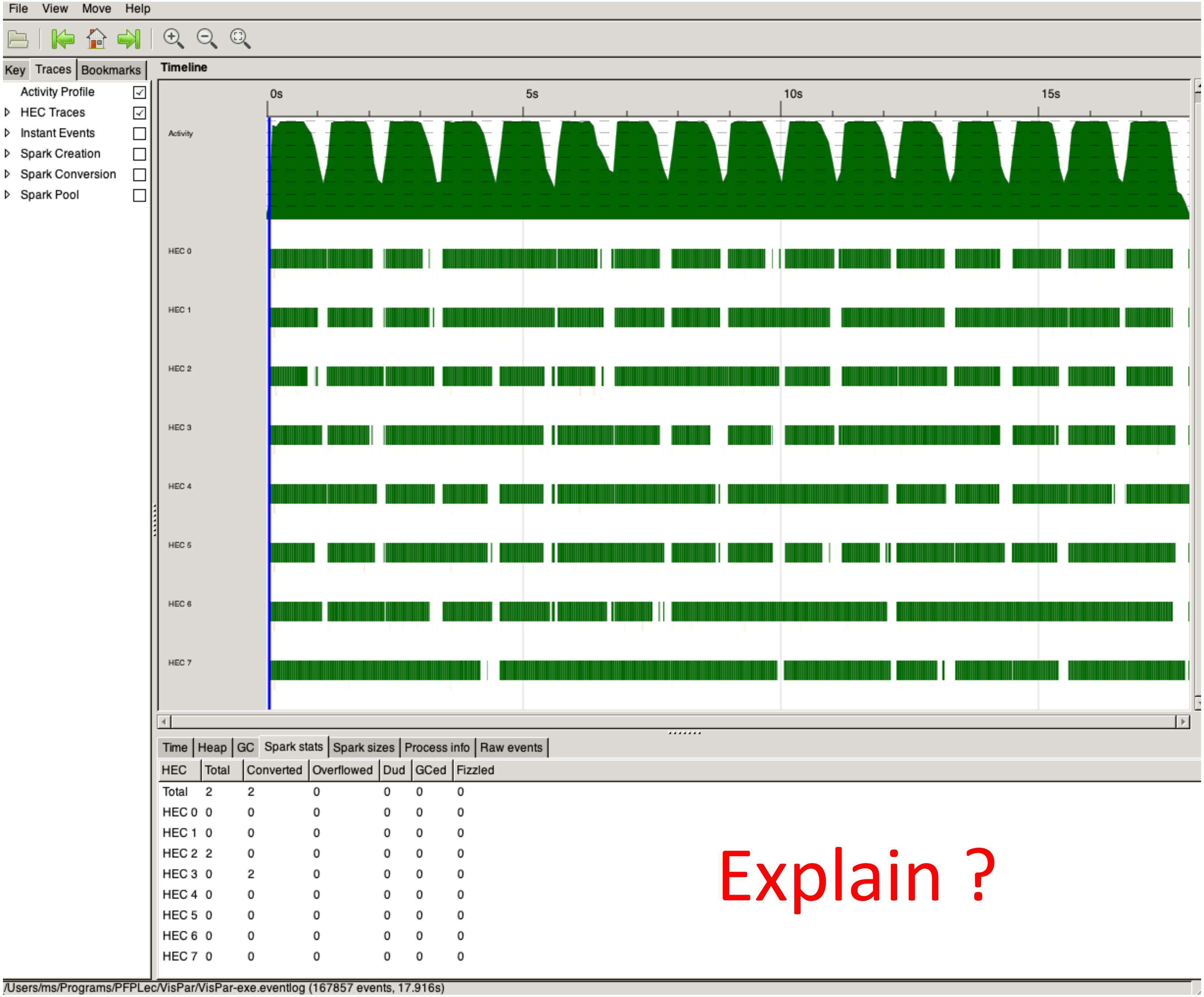


HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	0	0	0	0	0	0
HEC 0	0	0	0	0	0	0
HEC 1	0	0	0	0	0	0
HEC 2	0	0	0	0	0	0
HEC 3	0	0	0	0	0	0
HEC 4	0	0	0	0	0	0
HEC 5	0	0	0	0	0	0
HEC 6	0	0	0	0	0	0
HEC 7	0	0	0	0	0	0



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	0	0	0	0	0	0
HEC 0	0	0	0	0	0	0
HEC 1	0	0	0	0	0	0
HEC 2	0	0	0	0	0	0
HEC 3	0	0	0	0	0	0
HEC 4	0	0	0	0	0	0
HEC 5	0	0	0	0	0	0
HEC 6	0	0	0	0	0	0
HEC 7	0	0	0	0	0	0

NO SPARKS!!

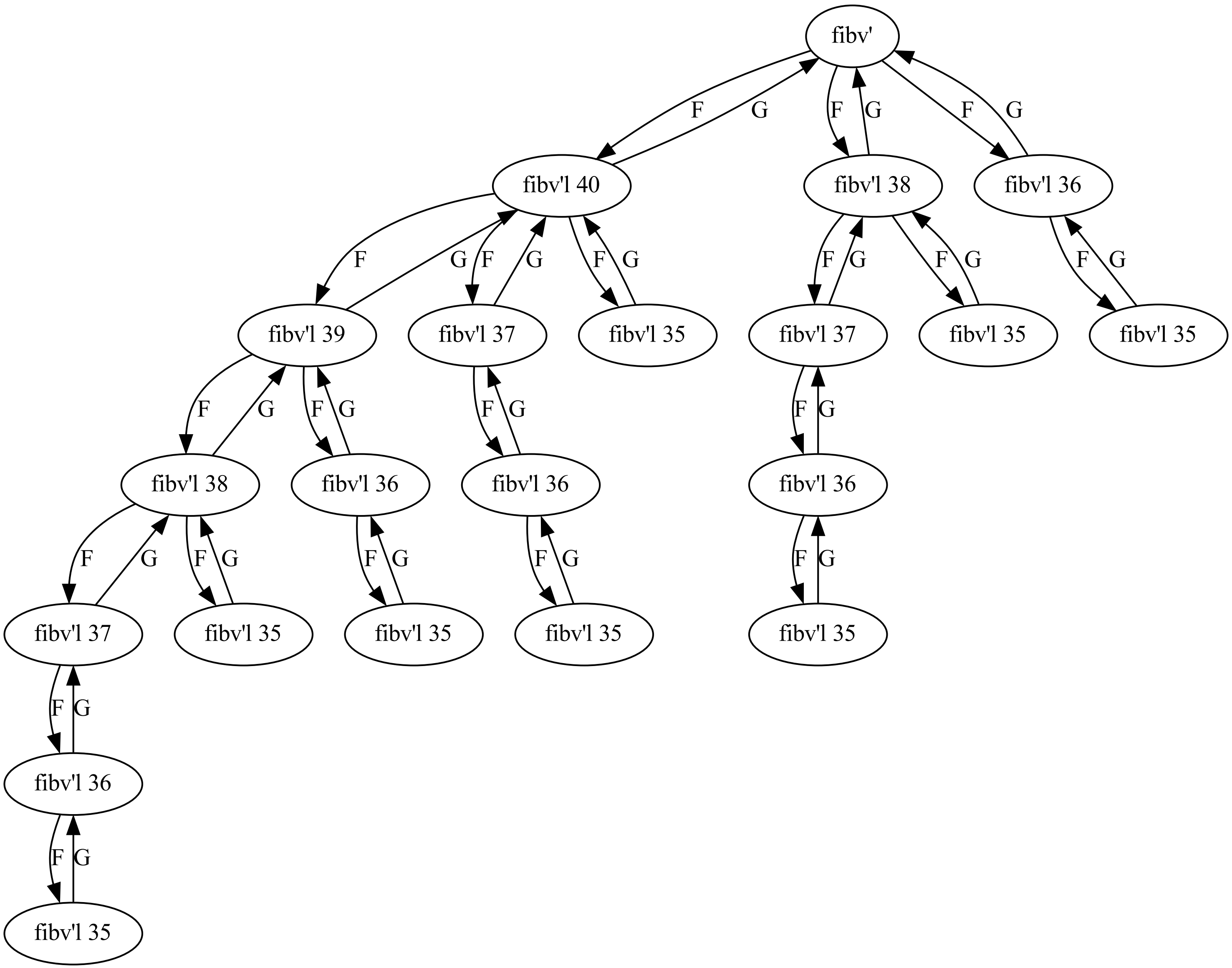


HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	2	2	0	0	0	0
HEC 0	0	0	0	0	0	0
HEC 1	0	0	0	0	0	0
HEC 2	2	0	0	0	0	0
HEC 3	0	2	0	0	0	0
HEC 4	0	0	0	0	0	0
HEC 5	0	0	0	0	0	0
HEC 6	0	0	0	0	0	0
HEC 7	0	0	0	0	0	0

Explain ?

A variant

```
fib' :: Integer -> Integer -> Par Integer
fib' n t | n<=t          = return $ sfib n
         | otherwise     = do
             leftVar  <- spawn $ fib' (n-1) t
             rightVar <-         fib' (n-2) t
             left     <- get leftVar
             right    <- get rightVar
             return $ left + right + 1
```



fibv'
fibv'l 40
fibv'l 38
fibv'l 36
fibv'l 39
fibv'l 37
fibv'l 35
fibv'l 37
fibv'l 35
fibv'l 35
fibv'l 38
fibv'l 36
fibv'l 36
fibv'l 36
fibv'l 37
fibv'l 35
fibv'l 35
fibv'l 35
fibv'l 35
fibv'l 36
fibv'l 35
F
G

```
benchmarking pfib 41 35
time                 1.109 s    (1.072 s .. 1.180 s)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 1.118 s    (1.107 s .. 1.136 s)
std dev              16.73 ms   (4.097 ms .. 22.22 ms)
variance introduced by outliers: 19% (moderately inflated)


benchmarking fib’ 41 35
time                 2.305 s    (2.245 s .. 2.352 s)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 2.304 s    (2.300 s .. 2.317 s)
std dev              8.854 ms   (108.5 μs .. 10.38 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Continuation monad (exactly as in PCM)

```
newtype Par a = Par { unPar :: (a -> Trace) -> Trace }
```

```
instance Monad Par where
    return a = Par ($ a)
    m >>= k = Par $ \c -> unPar m $ \a -> unPar (k a) c
```

the Par Monad

Implemented as a Haskell library

 surprisingly little code!

 includes a work stealing scheduler

 You get to roll your own schedulers!

Programmer has more control than with Strategies

 => less error prone?

Good performance (comparable to Strategies)

 particularly if granularity is not too small

Dataflow problems

- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

Slide by Simon Marlow

Implementation

- Starting point: A Poor Man's Concurrency Monad (Claessen JFP'99)
- PMC was used to *simulate* concurrency in a sequential Haskell implementation. We are using it as a way to implement very lightweight non-preemptive threads, with a parallel scheduler.
- Following PMC, the implementation is divided into two:
  - Par computations produce a lazy Trace
  - A scheduler consumes the Traces, and switches between multiple threads

Slide by Simon Marlow

Trace

```
data Trace = forall a . Get (IVar a) (a -> Trace)
           | forall a . Put (IVar a) a Trace
           | forall a . New (IVarContents a) (IVar a -> Trace)
           | Fork Trace Trace
           | Done
           | Yield Trace
           | forall a . LiftIO (IO a) (a -> Trace)
```

http://hackage.haskell.org/package/monad-par-0.3.5/docs/Control-Monad-Par.html

Parallel computation is pure and deterministic

```
runPar :: Par a -> a
```

Results communicated through IVars

```
newtype IVar a = IVar (IORef (IVarContents a))
```

```
data IVarContents a = Full a |  Blocked [a -> Trace]
```

Results communicated through IVars

```
newtype IVar a = IVar (IORef (IVarContents a))
```

```
data IVarContents a = Full a | Blocked [a -> Trace]
```

Set of threads blocked on get

```haskell
new :: Par (IVar a)
new = Par $ New
```

```haskell
get :: IVar a -> Par a
get v = Par $ \c -> Get v c
```

```haskell
put :: NFData a => IVar a -> a -> Par ()
put v a = deepseq a (Par $ \c -> Put v a (c ()))
```

e.g.

- This code:

```
do
   x <- new
   fork (put x 3)
   r <- get x
   return (r+1)
```

- will produce a trace like this:

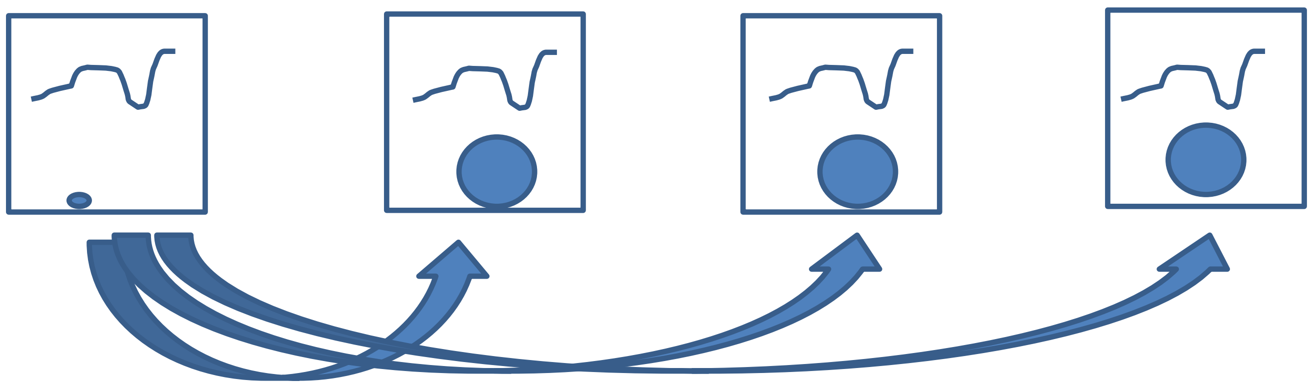
```
New (\x ->
  Fork (Put x 3 $ Done)
       (Get x (\r ->
          c (r + 1))))
```

Slide by Simon Marlow

Parallel scheduler

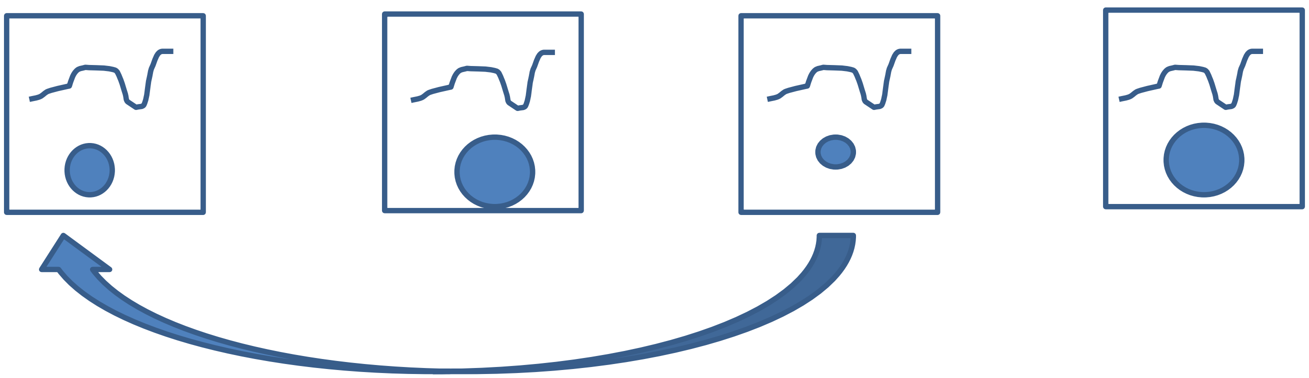
One scheduler thread per core, each with a work pool

When work pool dries up attempts to steal from other work pools



success

When work pool dries up attempts to steal from other work pools



If no work to be found, worker thread becomes idle (and is added to shared list of idle workers)

A worker thread that creates a new work item wakes up one of these idle workers

When all work pools are empty, computation is complete and **`runPar`** returns

The code is readable!

```
sched :: Sched -> Trace -> IO ()
sched queue t = loop t
  where
   loop t = case t of
    New a f -> do
     r <- newIORef a
     loop (f (IVar r))
    Get (IVar v) c -> do
     e <- readIORef v
     case e of
       Full a -> loop (c a)
       _other -> do
         r <- atomicModifyIORef v $ \e -> case e of
                    Empty -> (Blocked [c], reschedule queue)
                    Full a -> (Full a, loop (c a))
                    Blocked cs ->
                        (Blocked (c:cs), reschedule queue) r
```

```
  Put (IVar v) a t -> do
    cs <- atomicModifyIORef v $ \e -> case e of
             Empty -> (Full a, [])
             Full _ -> error "multiple put"
             Blocked cs -> (Full a, cs)
    mapM_ (pushWork queue. ($a)) cs
    loop t


. . .    -- Cases for Fork, Done, Yield, LiftIO
```

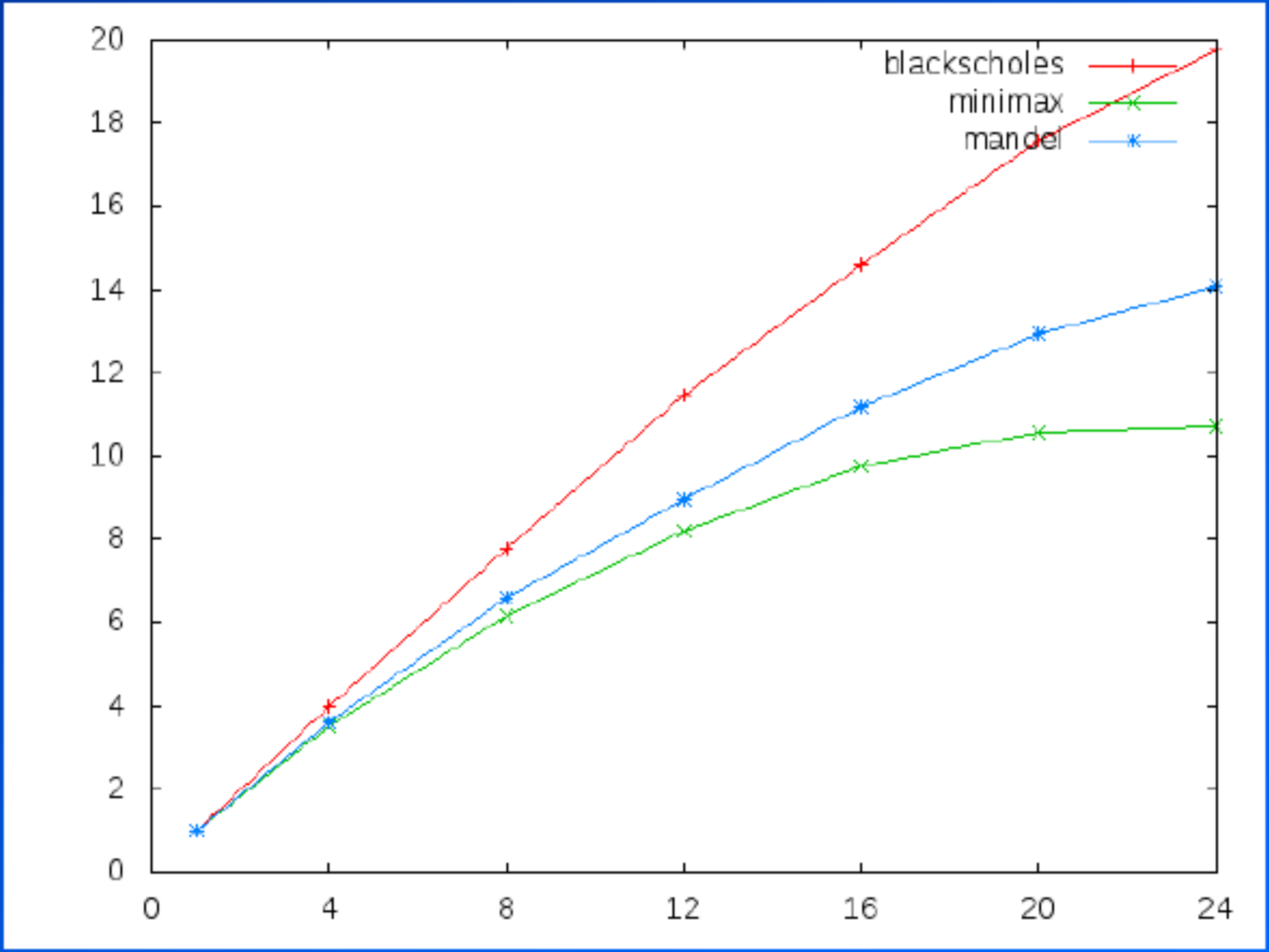
```
  Put (IVar v) a t -> do
    cs <- atomicModifyIORef v $ \e -> case e of
             Empty -> (Full a, [])
             Full _ -> error "multiple put"
             Blocked cs -> (Full a, cs)
    mapM_ (pushWork queue. ($a)) cs
    loop t

. . .    -- Cases for Fork,
```

If any worker is idle, wake one up and give it work to do

Results

speedup



99%

95%

50%

cores

Slide by Simon Marlow

Modularity

- Key property of Strategies is modularity

```
parMap f xs = map f xs `using` parList rwhnf
```

- Relies on lazy evaluation
  - fragile
  - not always convenient to build a lazy data structure
- Par takes a different approach to modularity:
  - the Par monad is for *coordination* only
  - the application code is written separately as pure Haskell functions
  - The "parallelism guru" writes the coordination code
  - Par performance is not critical, as long as the grain size is not too small

Slide by Simon Marlow

Par monad

Builds on old ideas of dataflow machines (hot topic in the 70s and 80s, reappearing in companies like [Maxeler])

Express parallelism by expressing data dependencies or using common patterns (like parMapM)
Very good match with skeletons!

Large grained, irregular parallelism is target

Par monad compared to Strategies

Separation of function and parallelisation done differently

Eval monad and Strategies are advisory

Eval monad well integrated with Threadscope

Par monad and Strategies tend to achieve similar performance

But remember

runPar is expensive and runEval is free!

Par monad compared to Strategies

Separation of function and parallelisation done differently

Eval monad and Strate... advisory

Eval monad well

Par monad and S... performance

But remember

runPar is expens

Par monad implemented as a library (and not in the runtime system)!
Scheduler is written in Haskell and you can mess with it or write your own (as Max did)

Par monad compared to Strategies

Par monad does not support speculative parallelism as Stategies do

Par monad supports stream processing pipelines well

Strategies appropriate if you are producing a lazy data structure

Note: Par monad and Strategies can be combined…

Par Monad easier to use than par?

fork creates one parallel task

Dependencies between tasks represented by Ivars

No need to reason about laziness

put is hyperstrict by default

Final suggestion in Par Monad paper is that maybe par (of par and pseq) is suitable for

automatic parallelisation

Read PCPH

Tomorrow Erlang!