Parallel Functional Programming Lecture 3

Mary Sheeran

with thanks to Simon Marlow for use of slides
and to Koen Claessen for the guest appearance

http://www.cse.chalmers.se/edu/course/pfp

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

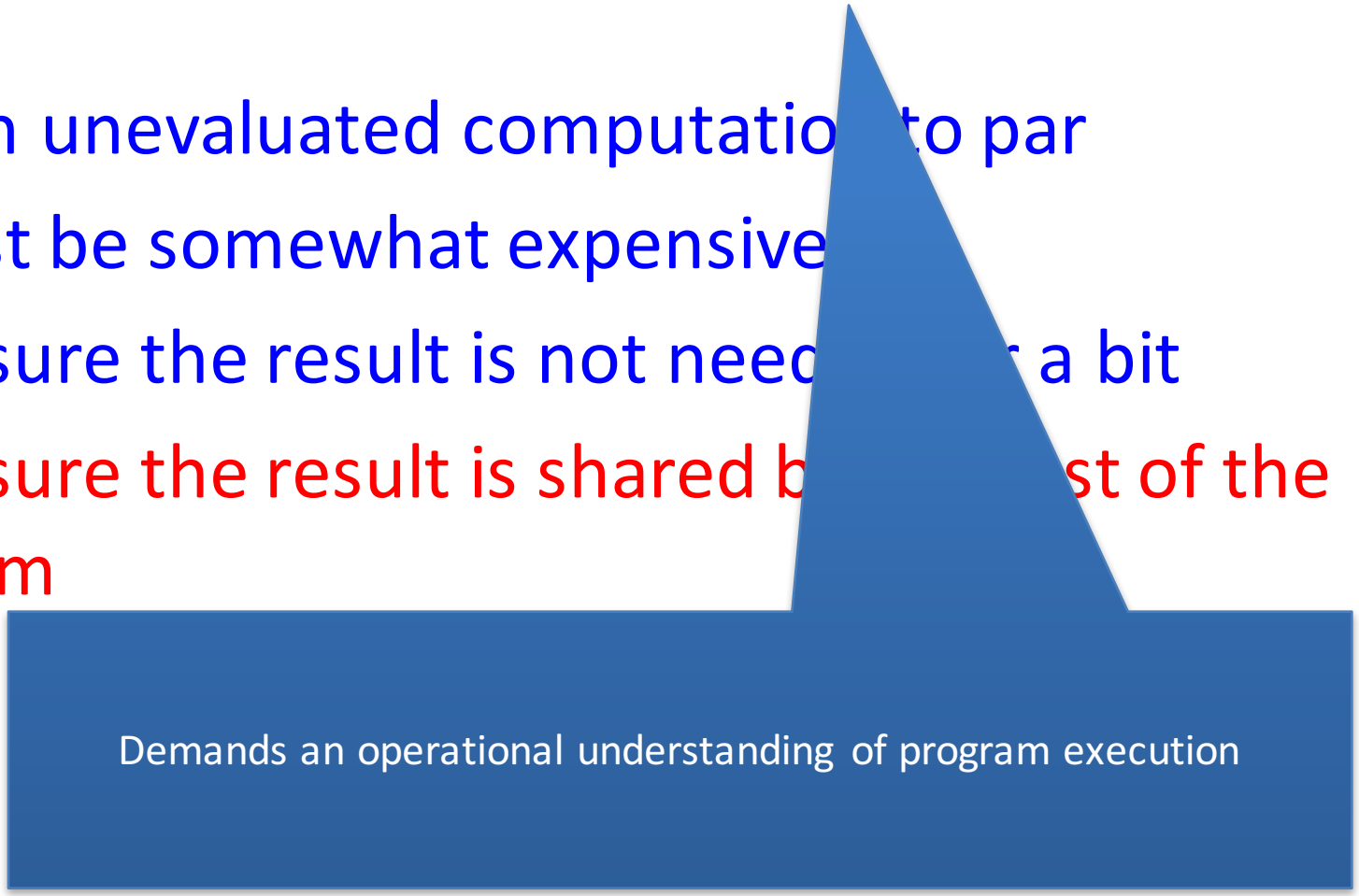
MUST

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



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

But still demand an understanding of laziness

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

Haskell'11

Builds on Koen's paper

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

Call this PMC

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.

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a -> a

fork :: Par () -> Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

forking is *explicit*

results are communicated through IVars

Slide by Simon Marlow

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, and then returns the value

the Par Monad

Implemented as a Haskell library

- surprisingly little code!
- includes a work stealing scheduler
- You get to roll your own schedulers!

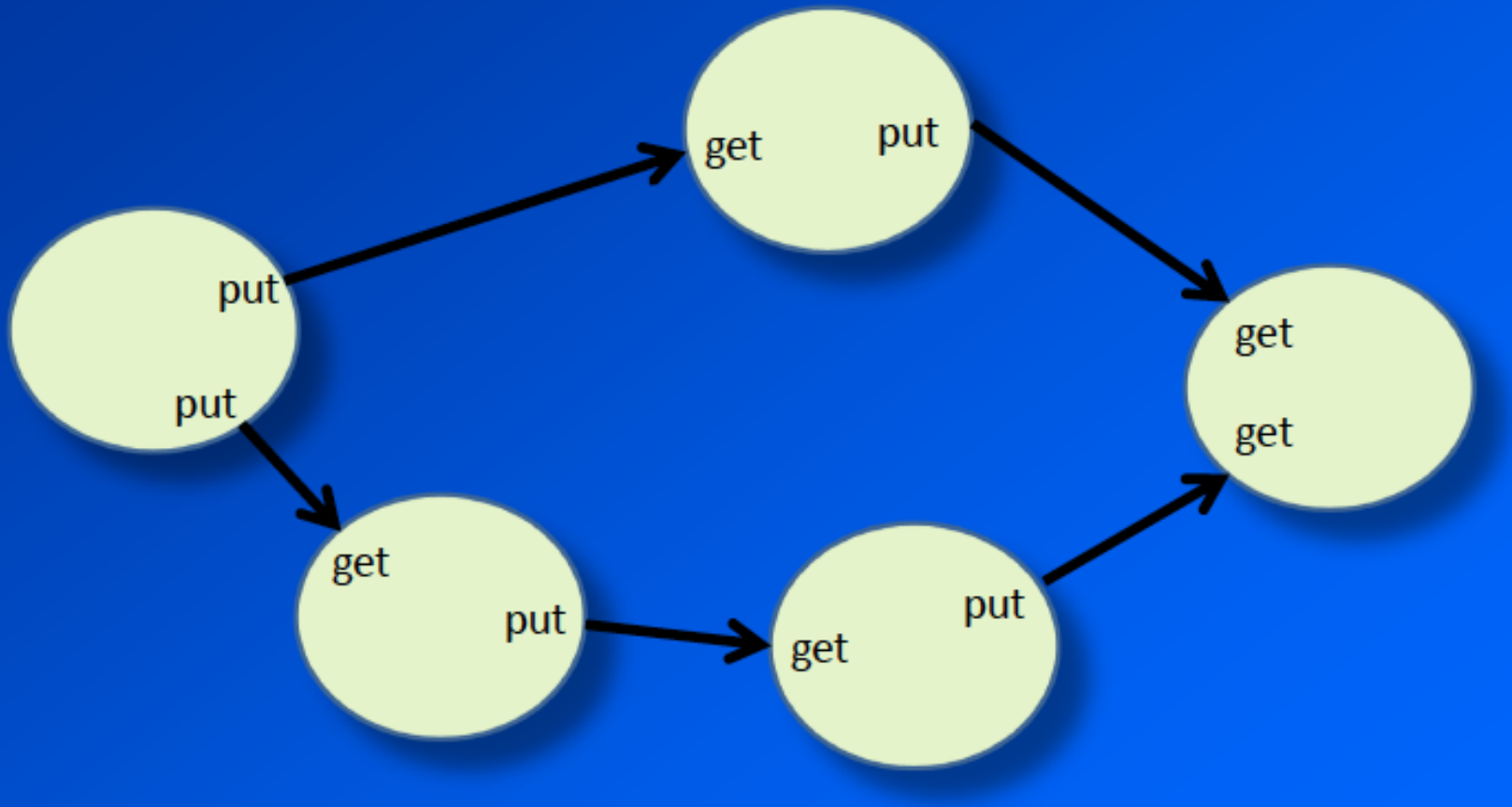
Programmer has more control than with Strategies

- => less error prone?

Good performance (comparable to Strategies)

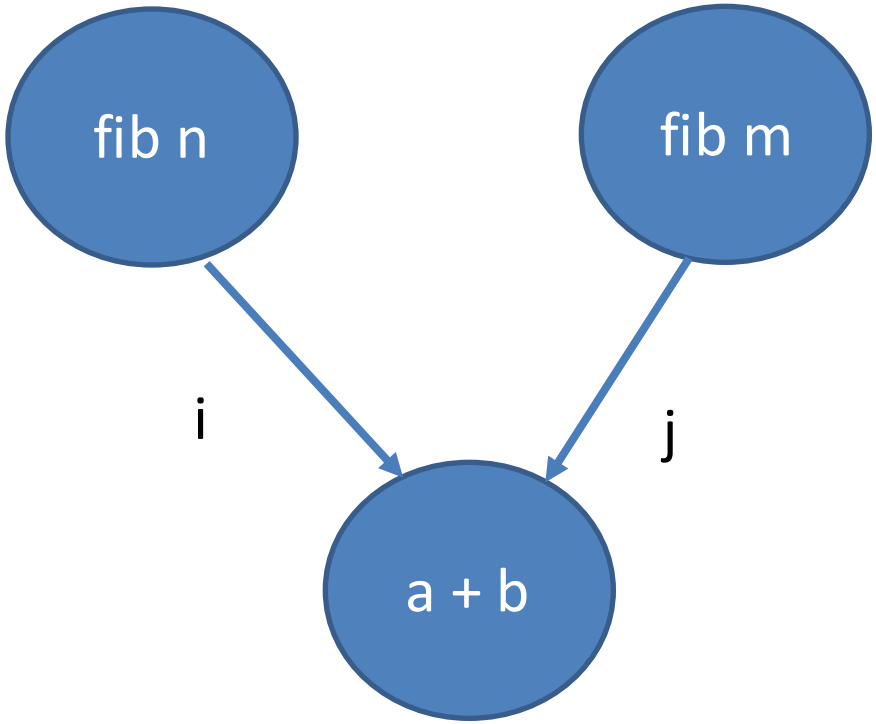
- particularly if granularity is not too small

Par expresses dynamic dataflow



Slide by Simon Marlow

```
runPar $ do
  i <- new
  j <- new
  fork (put i (fib n))
  fork (put j (fib m))
  a <- get i
  b <- get j
  return (a+b)
```


fib n
fib m
i
j
a + b

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

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

```
search :: ( partial -> Maybe solution )
        -> ( partial -> [ partial ] )
        -> partial
        -> [solution]
```

See PCPH ch. 4

```
search :: ( partial -> Maybe solution )
        -> ( partial -> [ partial ] )
        -> partial
        -> [solution]
search finished refine emptysoln = generate emptysoln
  where generate partial
    | Just soln <- finished partial = [soln]
    | otherwise = concat (map generate (refine partial))
```

```
parsearch :: NFData solution
       => ( partial -> Maybe solution )
       -> ( partial -> [ partial ] )
       -> partial
       -> [solution]
parsearch finished refine emptysoln
  = runPar $ generate emptysoln
  where
    generate partial
       | Just soln <- finished partial = return [soln]
       | otherwise = do
           solnss <- parMapM generate (refine partial)
           return (concat solnss)
```

needs granularity control

```
parsearch :: NFData solution
          => Int
          -> ( partial -> Maybe solution ) -- finished?
          -> ( partial -> [ partial ] ) -- refine a solution
          -> partial -- initial solution
          -> [solution]
parsearch maxdepth finished refine emptysoln
  = runPar $ generate 0 emptysoln
  where
    generate d partial | d >= maxdepth
      = return (search finished refine partial)
    generate d partial
      | Just soln <- finished partial = return [soln]
      | otherwise = do
          solnss <- parMapM (generate (d+1)) (refine partial)
          return (concat solnss)
```

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)
```

https://hackage.haskell.org/package/monad-par-0.3.4.8/docs/src/Control-Monad-Par-Scheds-TraceInternal.html#sched

The Par monad

- Par is a CPS monad:

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

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

Slide by Simon Marlow

Operations

```
fork :: Par () -> Par ()
fork p = Par $ \c ->
           Fork (runCont p (\_ -> Done)) (c ())

new :: Par (IVar a)
new  = Par $ \c -> New c

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

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

Slide by Simon Marlow

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

The scheduler

- First, a sequential scheduler.

```
sched :: SchedState -> Trace -> IO ()

type SchedState = [Trace]
```

The currently running thread

The work pool, "runnable threads"

Why IO? Because we're going to extend it to be a parallel scheduler in a moment.

Slide by Simon Marlow

Representation of IVar

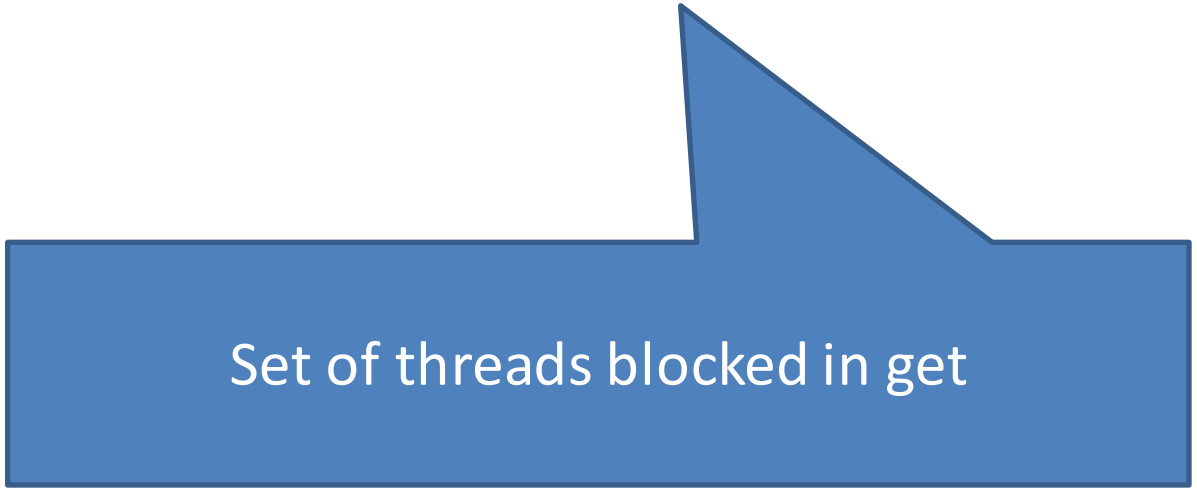
```
newtype IVar a = IVar (IORef (IVarContents a))
```

Representation of IVar

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

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

Representation of IVar

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

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



```
reschedule :: SchedState -> IO ()
reschedule []     = return ()
reschedule (t:ts) = sched ts t
```

```
sched state done = reschedule state
```

```
sched state (Fork child parent)
  = sched (child:state) parent
```

New and Get

```
sched state (New f) = do
  r <- newIORef (Blocked [])
  sched state (f (IVar r))
```

```
sched state (Get (IVar v) c) = do
  e <- readIORef v
  case e of
    Full a -> sched state (c a)
    Blocked cs -> do
      writeIORef v (Blocked (c:cs))
      reschedule state
```

Slide by Simon Marlow

Put

```
sched state (Put (IVar v) a t) = do
  cs <- modifyIORef v $ \e -> case e of
          Empty -> (Full a, [])
          Full _ -> error "multiple put"
          Blocked ds -> (Full a, ds)
  let state' = map ($ a) cs ++ state
  sched state' t
```

```
modifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

Put

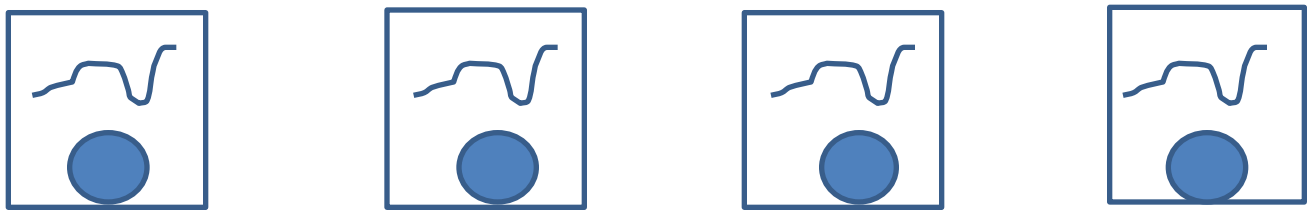
```
sched state (Put (IVar v) a t) = do
  cs <- modifyIORef v $ \e -> case e of
          Empty -> (Full a, [])
          Full _  -> error "multiple put"
          Blocked ds -> (Full a, ds)
  let state' = map ($ a) cs ++ state
  sched state' t
```

Wake up blocked threads
Add them to work pool

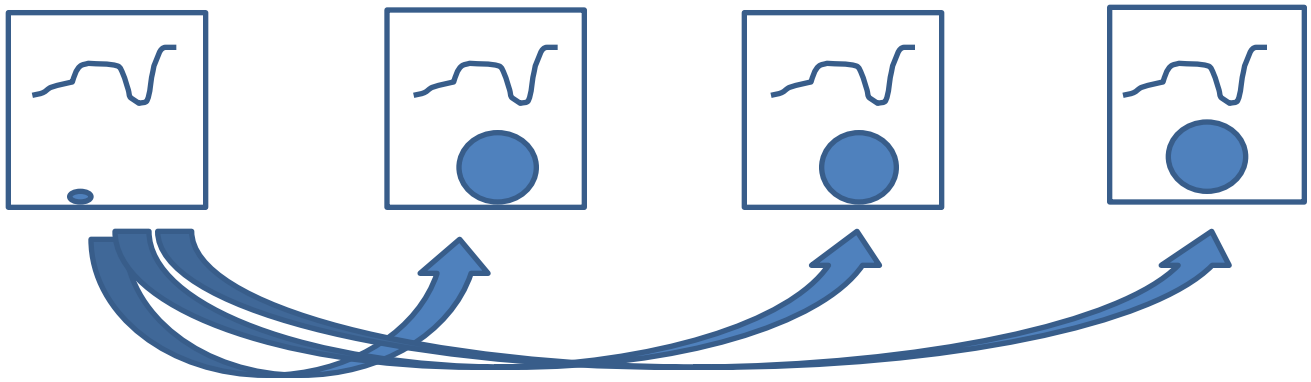
```
modifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

Parallel scheduler

One scheduler thread per core, each with a work pool

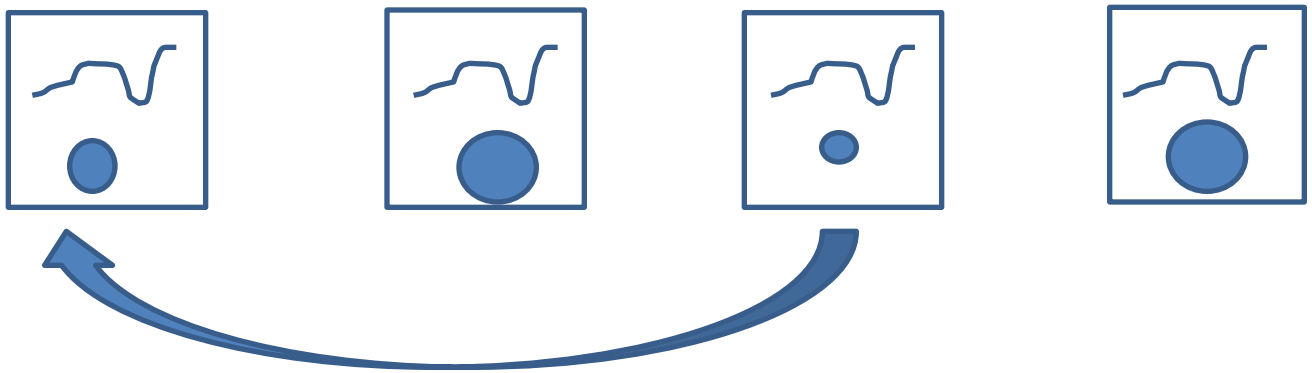


When work pool dries up attempts to steal from other work pools



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!

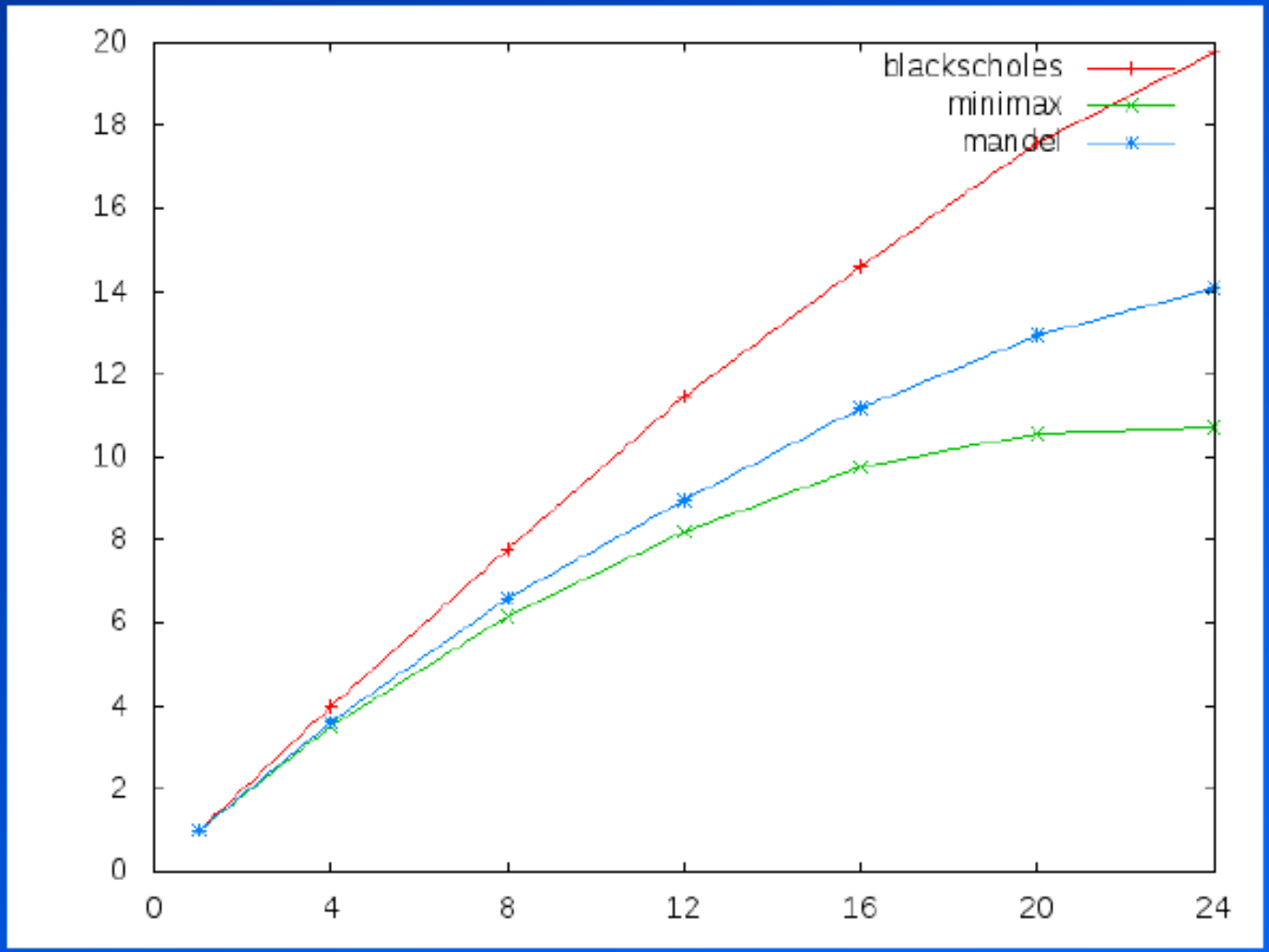
```
sched :: Sched -> Trace -> IO ()
sched queue t = loop t
  where
   loop t = case t of
    New a f -> do
     r <- newIORef a
     loop (f (IVar r))
    Get (IVar v) c -> do
     e <- readIORef v
     case e of
       Full a -> loop (c a)
       _other -> do
         r <- atomicModifyIORef v $ \e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



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

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 is suitable for automatic parallelisation

From PCPH

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