Parallel Functional Programming
Lecture 3

Mary Sheeran

(with thanks to Simon Marlow and Koen Claessen
for use of slides)

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

Simon Marlow's landscape for parallel Haskell

- Parallel
  - par/pseq ①
  - Strategies ②
  - Par Monad ③
  - Repa ④
  - Accelerate
  - DPH
- Concurrent
  - forkIO
  - MVar
  - STM
  - async
  - Cloud Haskell

Haxl

Simon Marlow lecture ☺

Using par

You must

pass an unevaluated computation to par

ensure that its value will not be required by the enclosing computation for a while

ensure that the result is shared by the rest of the program

Using par

You must

pass an unevaluated computation to par

ensure that its value will not be required by the enclosi tion for a while

ensure that the result is shared by the rest of the prog

Demands an operational understanding of program execution

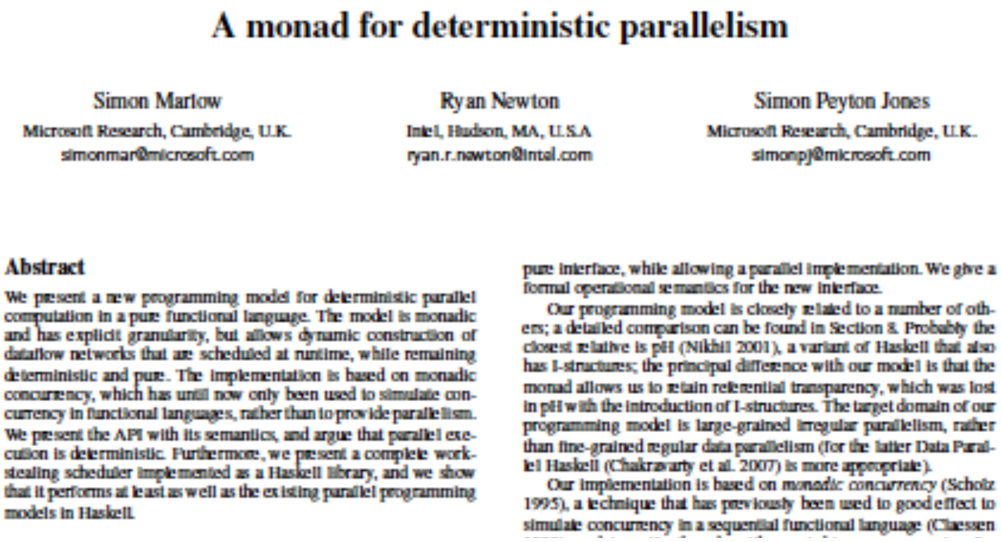
Eval monad plus Strategies

Eval monad enables expressing ordering between instances of par and pseq

Strategies separate algorithm from parallelisation

Provide useful higher level abstractions

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.

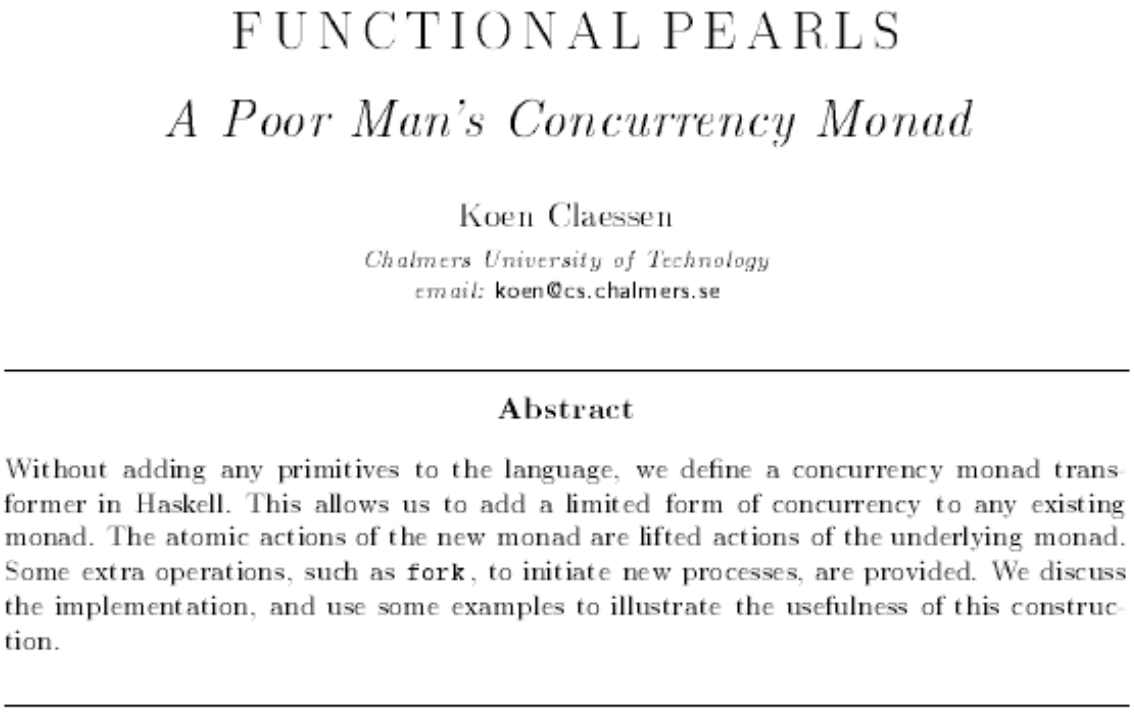
pure interface, while allowing a parallel implementation. We give a formal operational semantics for the new interface.

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 this idea



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

A Poor Man's Concurrency Monad

Koen Claessen

without adding primitives, we construct a way to lift any monad into a limited, but useful concurrent setting.

Monads

- abstraction from computation

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

- we use special notation

```
do a <- expr1     |  expr1 >>= \a ->
   expr2          |  expr2 >>= \_ ->
   b <- expr3     |  expr3 >>= \b ->
   expr4          |  expr4
```

Writer Monad

- can produce some output during computation

```
class Monad m ⇒ Writer m
  where
    write :: String → m ()
```

- An implementation could be:

-
  ```
  type W a = (a, String)
  ```

-
  ```
  instance Monad W where
    m >>= k = let (a, s) = m
                  (b, s') = k a
              in (b, s++s')
    return a = (a, "")
  ```

-
  ```
  instance Writer W where
    write s = ( (), s)
  ```

-
  ```
  output :: W a → String
  output (a, s) = s
  ```

Monad Transformer

- adds a feature to an existing monad

```
class MonadT t where
  lift :: Monad m
       => m a -> (t m) a
```

- examples:
  - state
  - exception
  - non determinism
- "compose your own monad" - LEGO

Concurrency

- interleaving actions
- atomic actions are actions in some monad
- round robin scheduler

- process has to consist of initial action + future.

Actions

We build actions from three different constructions:
atomic actions, forked actions and no-action.

```
data Action m
  = Atom (m (Action m))
  | Fork (Action m)
         (Action m)
  | Stop
```

We use constructors:

- general & simple
- expressive

See also Scholz [2].

Continuation

specifies what to do with result.

```
type C a =
  (a -> Action) -> Action
```

parametrize over a monad:

```
type C m a =
  (a -> Action m) -> Action m
```

for some type Action that stands for a process.

It is a monad:

```
instance Monad (C m) where
  m >>= k = \cont -> m
                (\a -> k a cont)
  return a = \cont -> cont a
```

Useful Operations

some functions that make life easier.

- Turn a C m a into an Action:

```
action :: C m a → Action m
action c = c (\a → Stop)
```

- Turn an m a into an (atomic) C m a:

```
atom :: m a → C m a
atom m = \cont →
   Atom (do a ← m
            return (cont a) )
```

- End a process (the empty process):

```
stop :: C m a
stop = \cont → Stop
```

Fork

Some operations on fork:

- ‘Imperative’ fork:

```
fork :: C m a -> C m ()
fork c = \cont -> Fork
  (action c) (cont ())
```

- ‘Alegebraic‘ or symmetrical fork:

```
par :: C m a -> C m a -> C m a
par c1 c2 = \cont ->
  Fork (c1 cont) (c2 cont)
```

Running a C

Ideally, we would like a function

```
run :: C m a -> m a
```

this is "not" possible, due to typing problems.
We will define a function

```
run :: C m a -> m ()
```

This means we'll only get the side-effects of the computation.

Round Robin

simple scheduler.

```
round :: [Action m] → m ()
round []     =  return ()
round (p:ps) =
   case p of
   - Atom ma →
       do p' ← ma
          round (ps ++ [p'])
   - Fork p1 p2 →
       round (ps ++ [p1, p2])
   - Stop →
       round ps
```

Using C

- We can use the scheduler to define:

```
run :: C m a -> m ()
run c = round [action c]
```

- We can construct C's with atom, fork, stop, and can run them using run.

C is a Monad Transformer

C can be made an instance of MonadTrans.

```
instance MonadTrans C
  where
    lift = atom
```

All lifted actions become atomic actions in the new setting.

Example 1: Writer

We lift every writer monad:

```
instance Writer m =>
            Writer (C m) where
  write s = lift (write s)
```

Every write action is now atomic.

```
example :: C W ()
example = do write "hej!"
             fork (loop "apa")
             fork (loop "hund")
 where
  loop s = do write s
              loop s
```

will result in :

[redacted]

```
hej! apa hund apa hund apa ....
```

Example 2: Another lifting

We can lift writers in a different way:

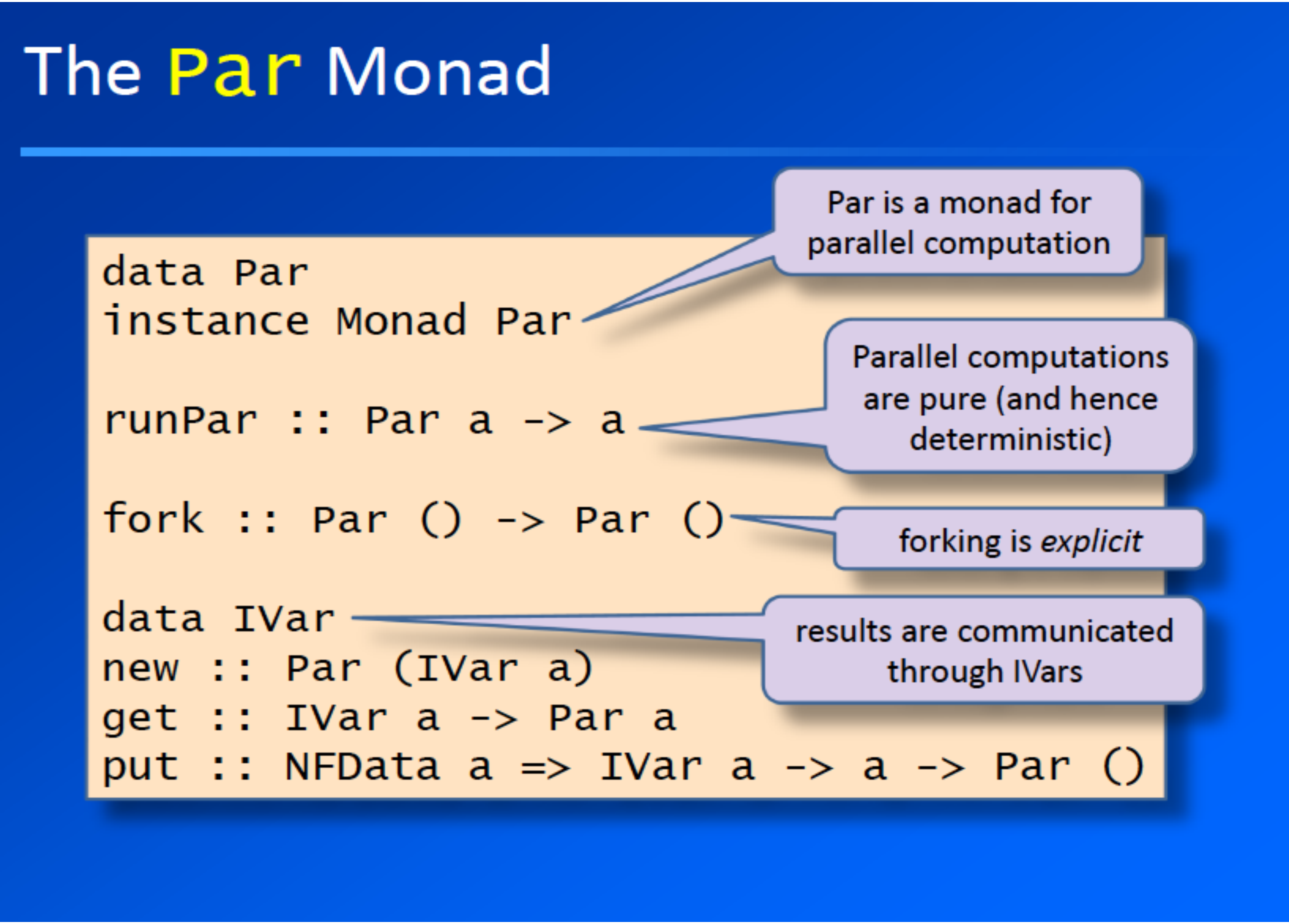
```
instance Writer m =>
              Writer (C m) where
  write ""    = return ()
  write (c:s) = do lift (write [c])
                   write s
```

a write action is now split up in atomic actions for each character.

hej!ahpuanadphaupn....

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

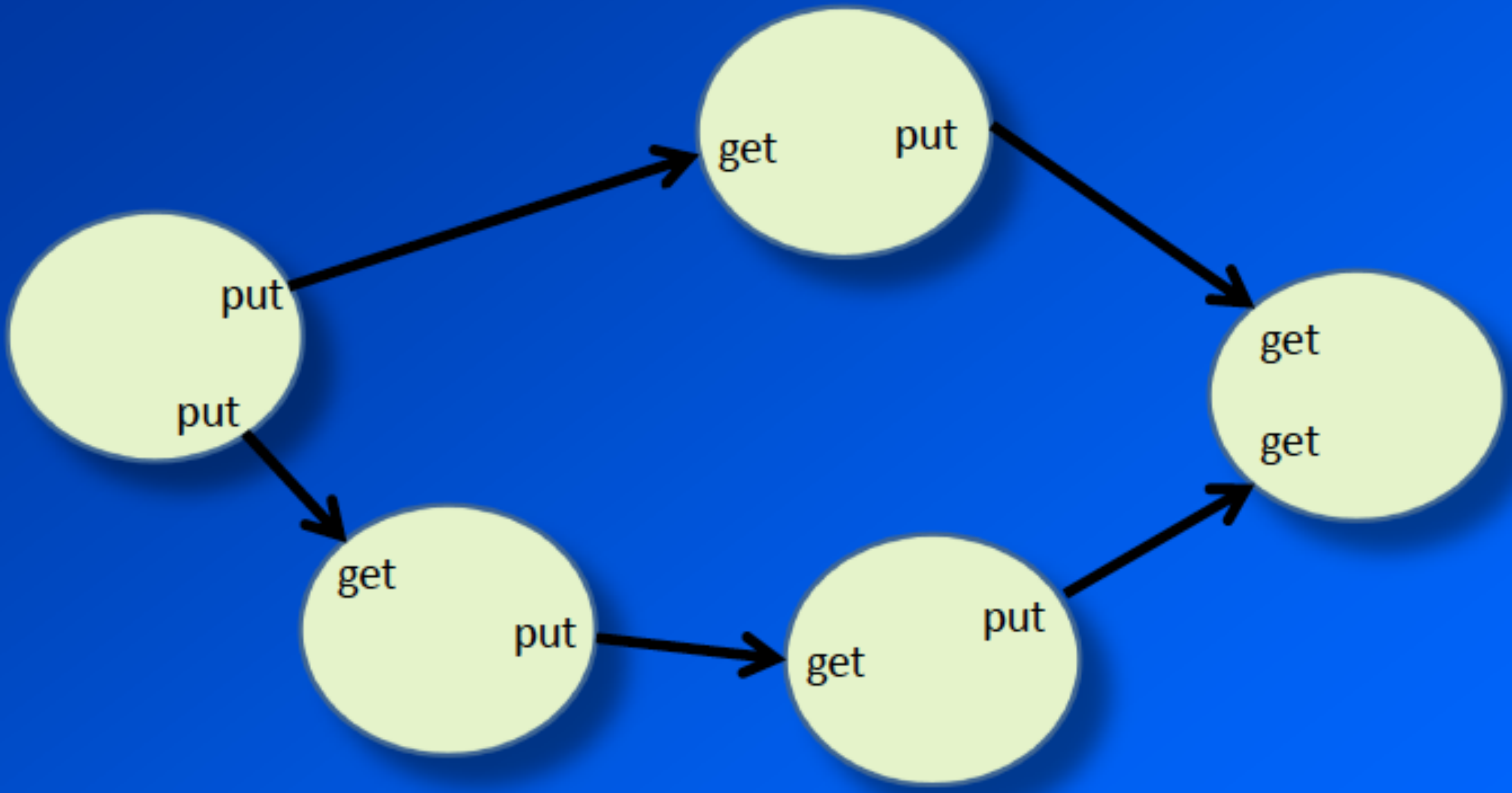
the Par Monad

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

```
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
```

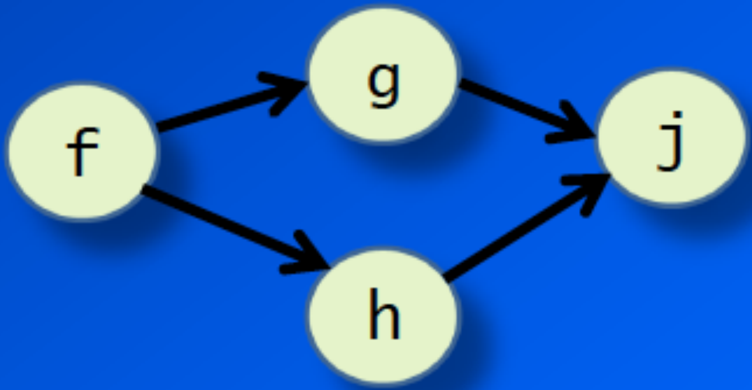
```
parfib :: Int -> Int -> Par Int
parfib n t
  | n <= 2  =  return 1
  | n <= t  =  return $ sfib n
  | otherwise = do
      x <- spawn $ parfib (n-1) t
      y <- spawn $ parfib (n-2) t
      x' <- get x
      y' <- get y
      return (x' + y’)
```

Dataflow

- Consider typechecking a set of (non-recursive) bindings:

```
f = ...
g = ... f ...
h = ... f ...
j = ... g ... h ...
```

- treat this as a dataflow graph:



Slide by Simon Marlow

```
parInfer :: [(Var,Expr)] -> [(Var,Type)]

parInfer bindings = runPar $ do
  let binders = map fst bindings
  ivars <- replicateM (length binders) new
  let env = Map.fromList (zip binders ivars)
  mapM_ (fork . infer env) bindings
  types <- mapM_ get ivars
  return (zip binders types)
```

```
parInfer :: [(Var,Expr)] -> [(Var,Type)]

parInfer bindings = runPar $ do
  let binders = map fst bindings
  ivars <- replicateM (length binders) new
  let env = Map.fromList (zip binders ivars)
  mapM_ (fork . infer env) bindings
  types <- mapM_ get ivars
  return (zip binders types)
```

Create nodes and edges and let the scheduler do the work
No dependency analysis required!
Maximum parallelism for little programmer effort   Dynamic parallelism
Very nice ☺

Divide and Conquer skeleton

```
divConq :: NFData sol => (prob -> Bool) -- indivisible?
         -> (prob -> [prob]) -- split into subproblems
         -> ([sol] -> sol) -- join solutions
         -> (prob -> sol) -- solve a subproblem
         -> (prob -> sol)
divConq indiv split join f prob
   = runPar $ go prob
       where
         go prob
           | indiv prob = return (f prob)
           | otherwise = do
               sols <- parMapM go (split prob)
               return (join sols)
```

Another D&C skeleton

```
divConq :: NFData sol
             => (prob -> Bool)        -- indivisible?
             -> (prob -> (prob,prob)) -- split into subproblems
             -> (sol -> sol -> sol)   -- join solutions
             -> (prob -> sol)         -- solve a subproblem
             -> (prob -> sol)
divConq indiv split join f prob
  = runPar $ go prob
      where
        go prob
          | indiv prob = return (f prob)
          | otherwise = do
              let (a,b) = split prob
              i <- spawn $ go a
              j <- spawn $ go b
              a <- get i
              b <- get j
              return (join a b)
```

parallel sort

```
parsort :: Int -> [Int] -> [Int]
parsort thresh xs
  = divConq indiv divide merge (List.sort . snd) (thresh,xs)
     where
       indiv (n,xs) = n == 0

       divide (n,xs) = ((n-1, as), (n-1, bs))
         where (as,bs) = halve xs


halve xs = splitAt n2 xs
  where
    n2 = div (length xs)
```

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

Traces

- A “thread” produces a lazy stream of operations:

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

Slide by Simon Marlow

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.

The currently running thread

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

type SchedState = [Trace]
```

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))

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

set of threads
blocked in **get**

Slide by Simon Marlow

Fork and Done

```
sched state Done = reschedule state
```

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

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

Slide by Simon Marlow

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
           case e of
             Full _     -> error "multiple put"
             Blocked cs -> (Full a, cs)
  let state' = map ($ a) cs ++ state
  sched state' t
```

Wake up all the blocked threads, add them to the work pool

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

Slide by Simon Marlow

Finally... runPar

```
runPar :: Par a -> a
runPar x = unsafePerformIO $ do
   rref <- newIORef (Blocked [])
   sched [] $
      runCont (x >>= put_ (IVar rref))
              (const Done)
   r <- readIORef rref
   case r of
     Full a -> return a
     _      -> error "no result"
```

rref is an IVar to hold the return value

the “main thread” stores the result in rref

if the result is empty, the main thread must have deadlocked

- that’s the complete sequential scheduler

Slide by Simon Marlow

A real parallel scheduler

- We will create one scheduler thread per core
- Each scheduler has a local work pool
  - when a scheduler runs out of work, it tries to steal from the other work pools
- The new state:

```
data SchedState = SchedState
    { no        :: Int,
      workpool  :: IORef [Trace],
      idle      :: IORef [MVar Bool],
      scheds    :: [SchedState]
    }
```

CPU number

Local work pool

Idle schedulers (shared)

Other schedulers (for stealing)

Slide by Simon Marlow

New/Get/Put

- New is the same
- Mechanical changes to Get/Put:
  - use atomicModifyIORef to operate on IVars
  - use atomicModifyIORef to modify the work pool (now an IORef [Trace], was previously [Trace]).

Slide by Simon Marlow

reschedule

```
reschedule :: SchedState -> IO ()
reschedule state@SchedState{ workpool } = do
  e <- atomicModifyIORef workpool $ \ts ->
         case ts of
           []      -> ([], Nothing)
           (t:ts') -> (ts', Just t)
  case e of
    Just t  -> sched state t
    Nothing -> steal state
```

Here’s where
we go stealing

Slide by Simon Marlow

stealing

```
steal :: SchedState -> IO ()
steal state@SchedState{ scheds, no=me } = go scheds
  where
    go (x:xs)
      | no x == me    = go xs
      | otherwise     = do
         r <- atomicModifyIORef (workpool x) $ \ ts ->
                 case ts of
                    []     -> ([], Nothing)
                    (x:xs) -> (xs, Just x)
         case r of
           Just t  -> sched state t
           Nothing -> go xs
    go [] = do
      -- failed to steal anything; add ourself to the
      -- idle queue and wait to be woken up
```

Slide by Simon Marlow

runPar

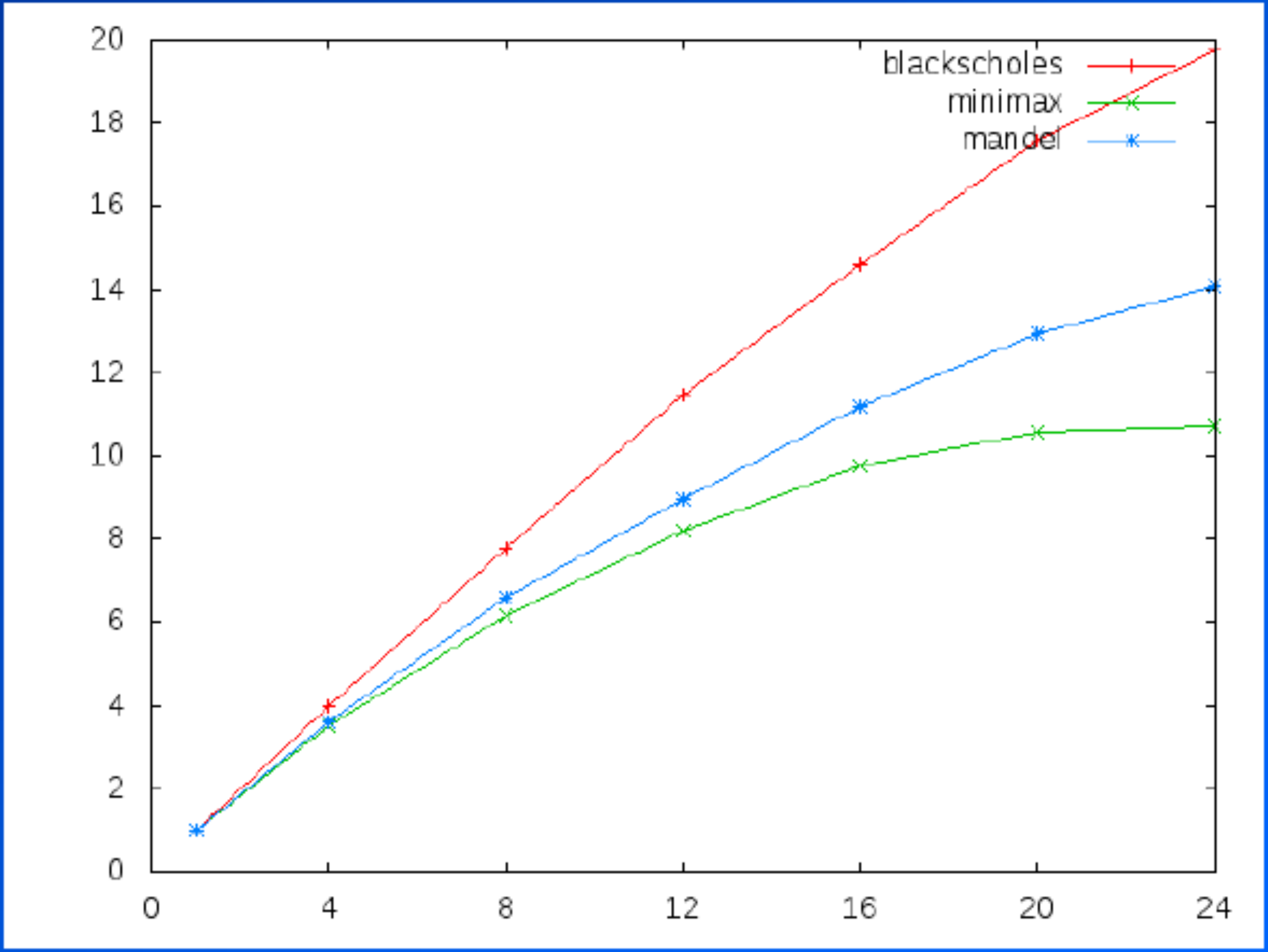
```
runPar :: Par a -> a
runPar x = unsafePerformIO $ do
   let states = ...
   main_cpu <- getCurrentCPU
   m <- newEmptyMVar
   forM_ (zip [0..] states) $ \(cpu,state) ->
     forkOnIO cpu $
       if (cpu /= main_cpu)
          then reschedule state
          else do
               rref <- newIORef Empty
               sched state $
                  runCont (x >>= put_ (IVar rref))
                          (const Done)
               readIORef rref >>= putMVar m

   r <- takeMVar m
   case r of Full a -> return a
             _ -> error "no result"
```

The "main thread" runs on the current CPU, all other CPUs run workers

An MVar communicates the result back to the caller of runPar

Slide by Simon Marlow

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 compared to Strategies

Separation of function and parallelisation done differently

Eval monad and Strategies are advisory

Par monad does not support speculative parallelism as Stategies do

Par monad supports stream processing pipelines well

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

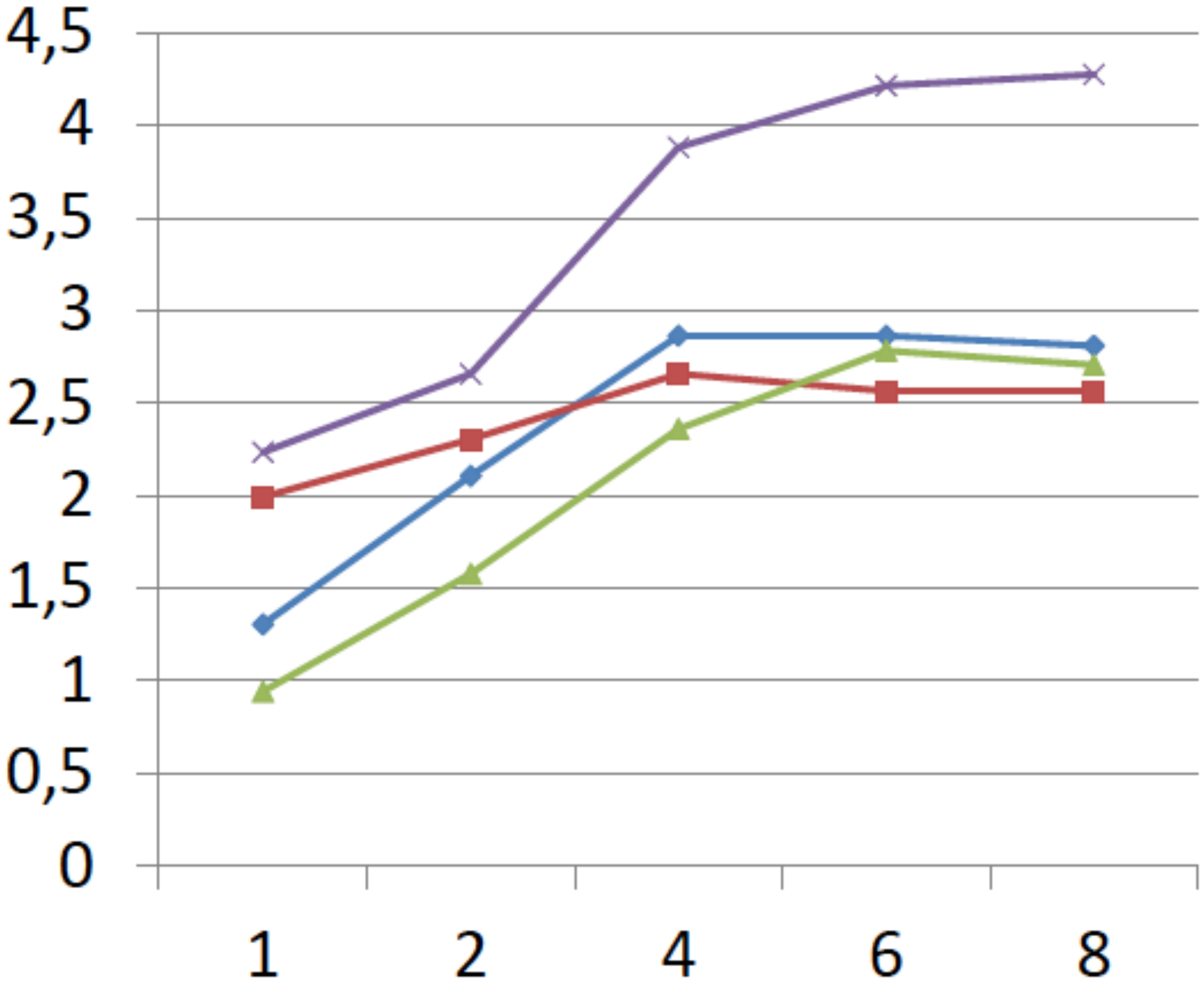
put is hyperstrict by default

Final suggestion in Par Monad paper is that maybe par is suitable for automatic parallelisation

Sorting speedups

For those curious about the Sort Challenge (from 2012), the results are

presented in this gzipped file, including slides


4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
6
8

In the meantime

Do exercise 1 (not graded)

Read papers and PCPH

Continue working on Lab A (due midnight April 6)

Note Nick’s office hours

(room 5461, wed 13-14 and fri 13-14)

Extra office hours today from 15.00

Use him! He is your best resource.