Multicore programming in Haskell

Simon Marlow

Microsoft Research

A concurrent web server

```
server :: Socket -> IO ()
server sock =
  forever
    (do
       acc <- Network.accept sock
       forkIO (http acc)
    )
```

create a new thread for each new client

the client/server protocol is implemented in a single-threaded way

Concurrency = abstraction

- Threads let us implement individual interactions separately, but have them happen “at the same time”
- writing this with a single event loop is complex and error-prone
- Concurrency is for making your program *cleaner*.

More uses for threads

- for hiding latency
  - e.g. downloading multiple web pages
- for encapsulating state
  - talk to your state via a channel
- for making a responsive GUI
- fault tolerance, distribution
- ... for making your program faster?
  - are threads a good abstraction for multicore?

Parallelism

Why is concurrent programming hard?

- *non-determinism*
  - threads interact in different ways depending on the scheduler
  - programmer has to deal with this somehow: locks, messages, transactions
  - hard to think about
  - impossible to test exhaustively
- can we get *parallelism* without *non-determinism?*

What Haskell has to offer

- Purely functional by default
  - computing pure functions in parallel is deterministic
- Type system guarantees absence of side-effects
- Great facilities for abstraction
  - Higher-order functions, polymorphism, lazy evaluation
- Wide range of concurrency paradigms
- Great tools

The rest of the talk

- Parallel programming in Haskell
- Concurrent data structures in Haskell

Parallel programming in Haskell

```
par :: a -> b -> b
```

Evaluate the first argument in parallel

return the second argument

Parallel programming in Haskell

```
par  :: a -> b -> b
pseq :: a -> b -> b
```

Evaluate the first argument

Return the second argument

Using par and pseq

```
import Control.Parallel

main =
 let
    p = primes !! 3500
    q = nqueens 12
 in
    par p $ pseq q $ print (p,q)

primes = ...
nqueens = ...
```

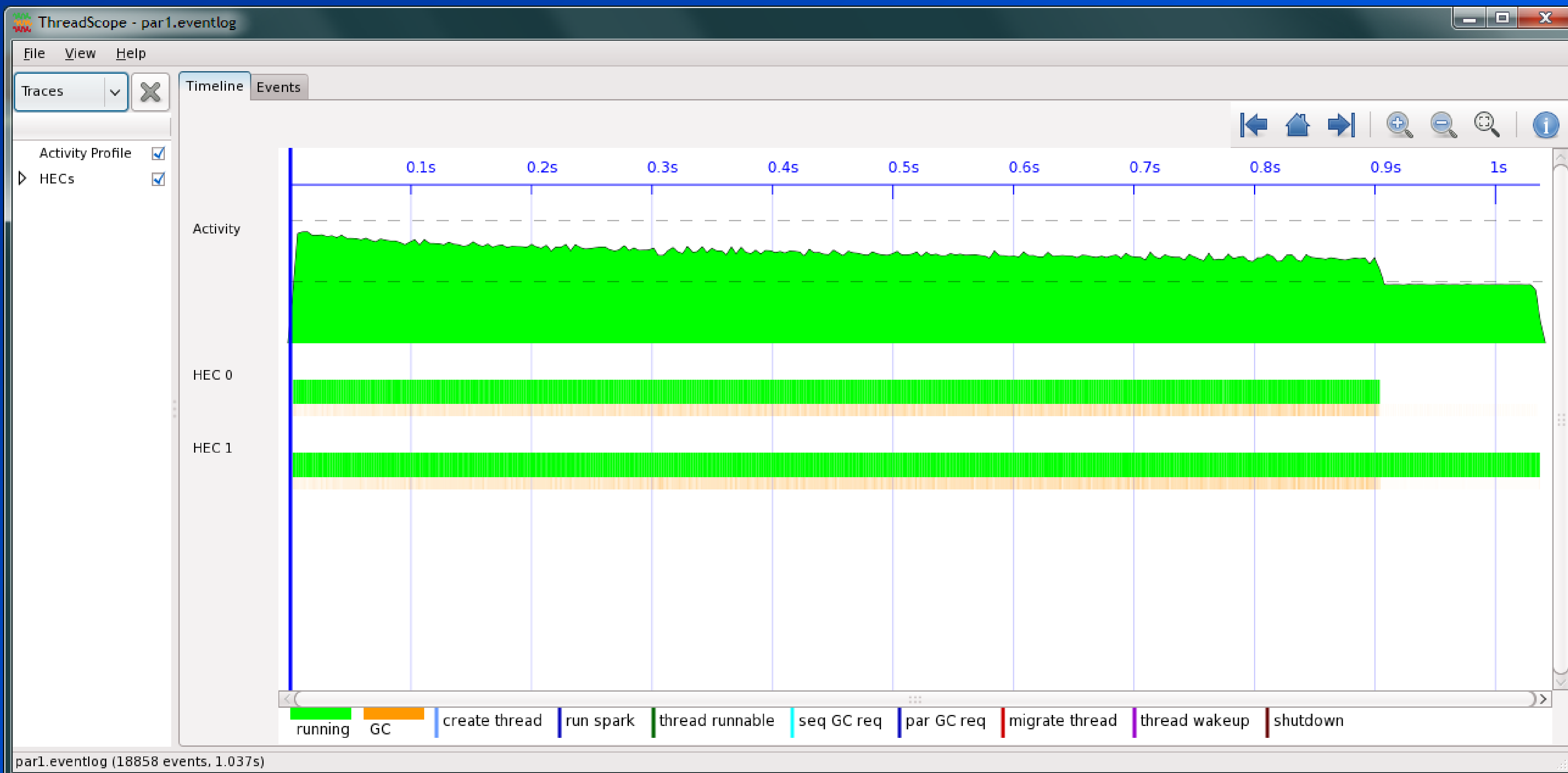
This does not

par indicates that p could be evaluated in parallel with (pseq q (print (p,q))

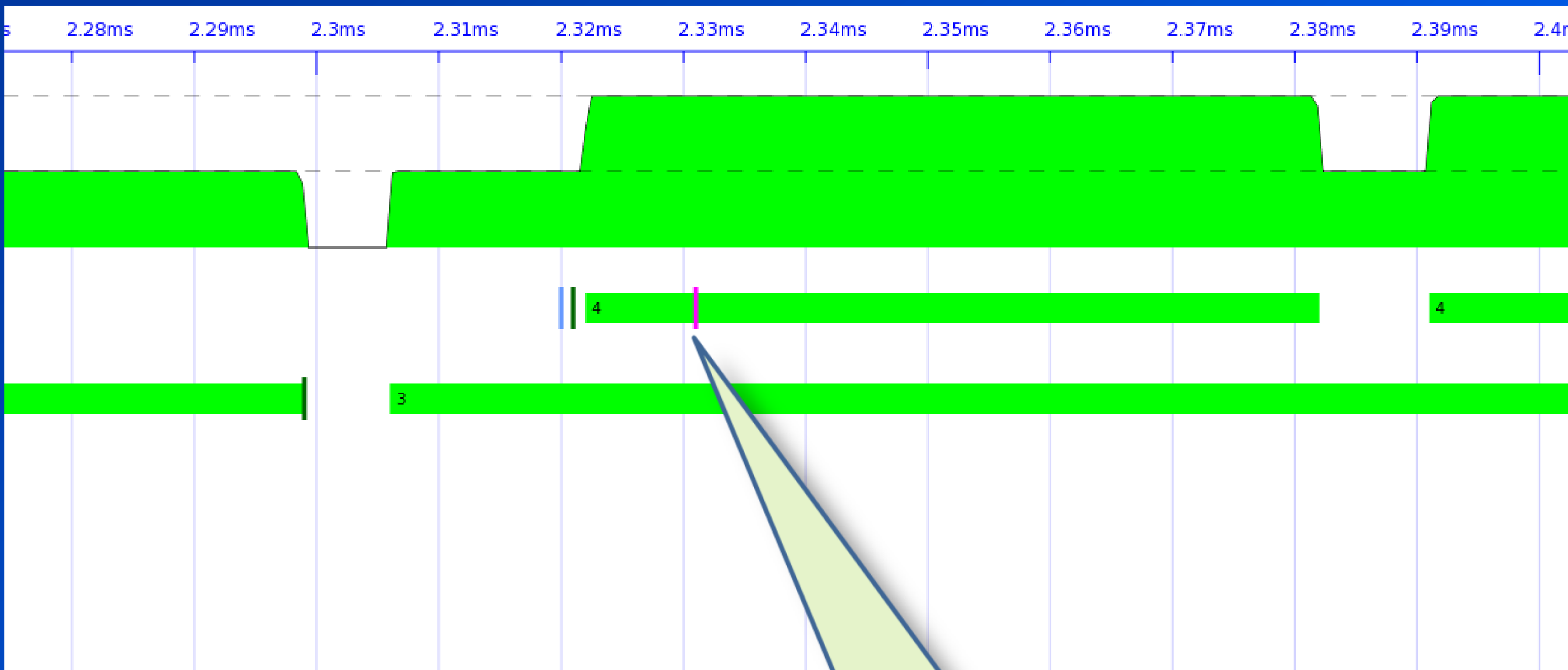
- q is evaluated by pseq
- p is *demanded* by print
- (p,q) is printed

write it like this if you want (a $ b = a b)

ThreadScope

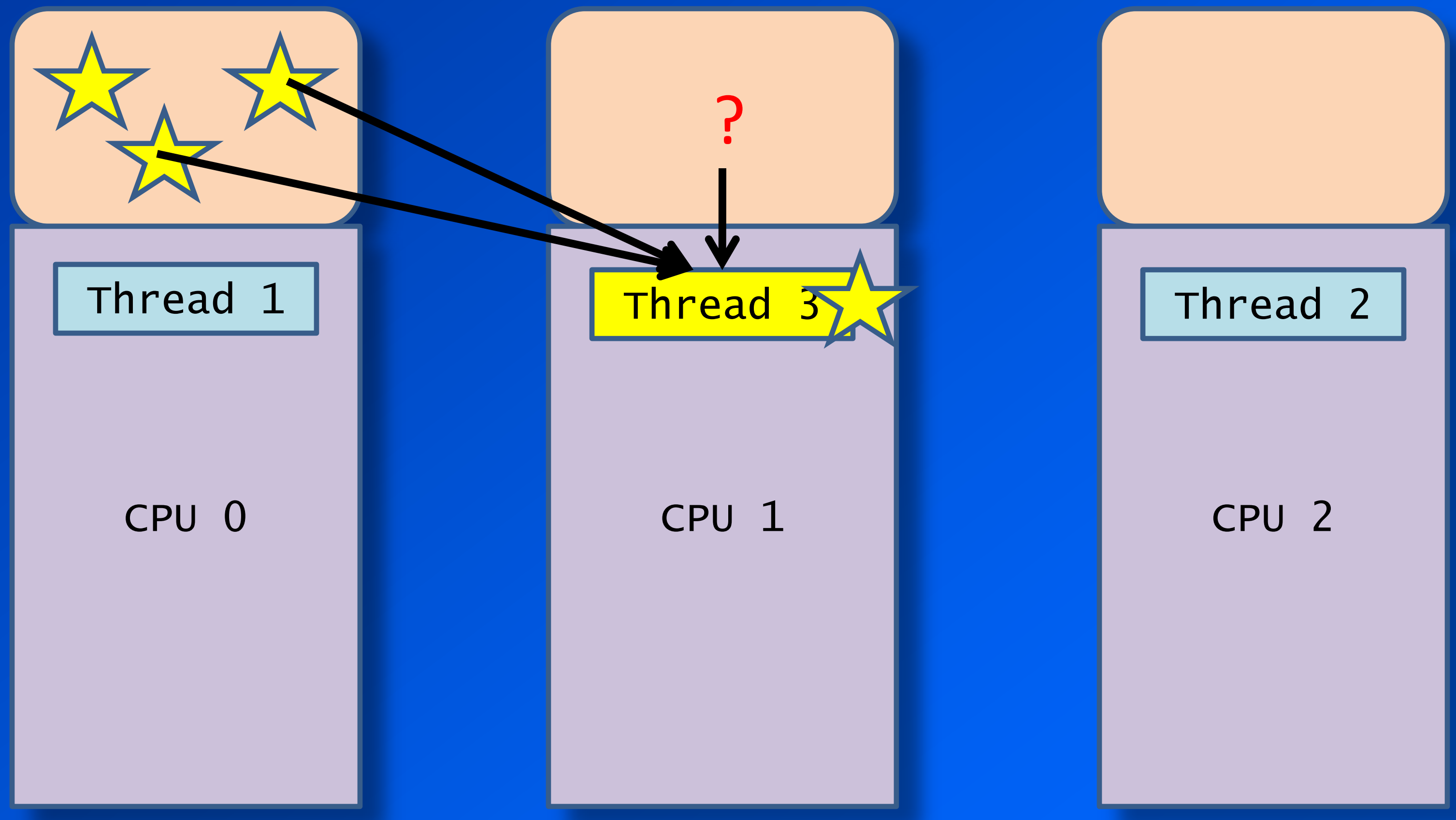


Zooming in...



The spark is picked up here

How does par actually work?



Correctness-preserving optimisation

```
par a b == b
```

- Replacing “par a b” with “b” does not change the meaning of the program
  - only its speed and memory usage
  - par cannot make the program go wrong
  - no race conditions or deadlocks, guaranteed!
- par looks like a function, but behaves like an *annotation*

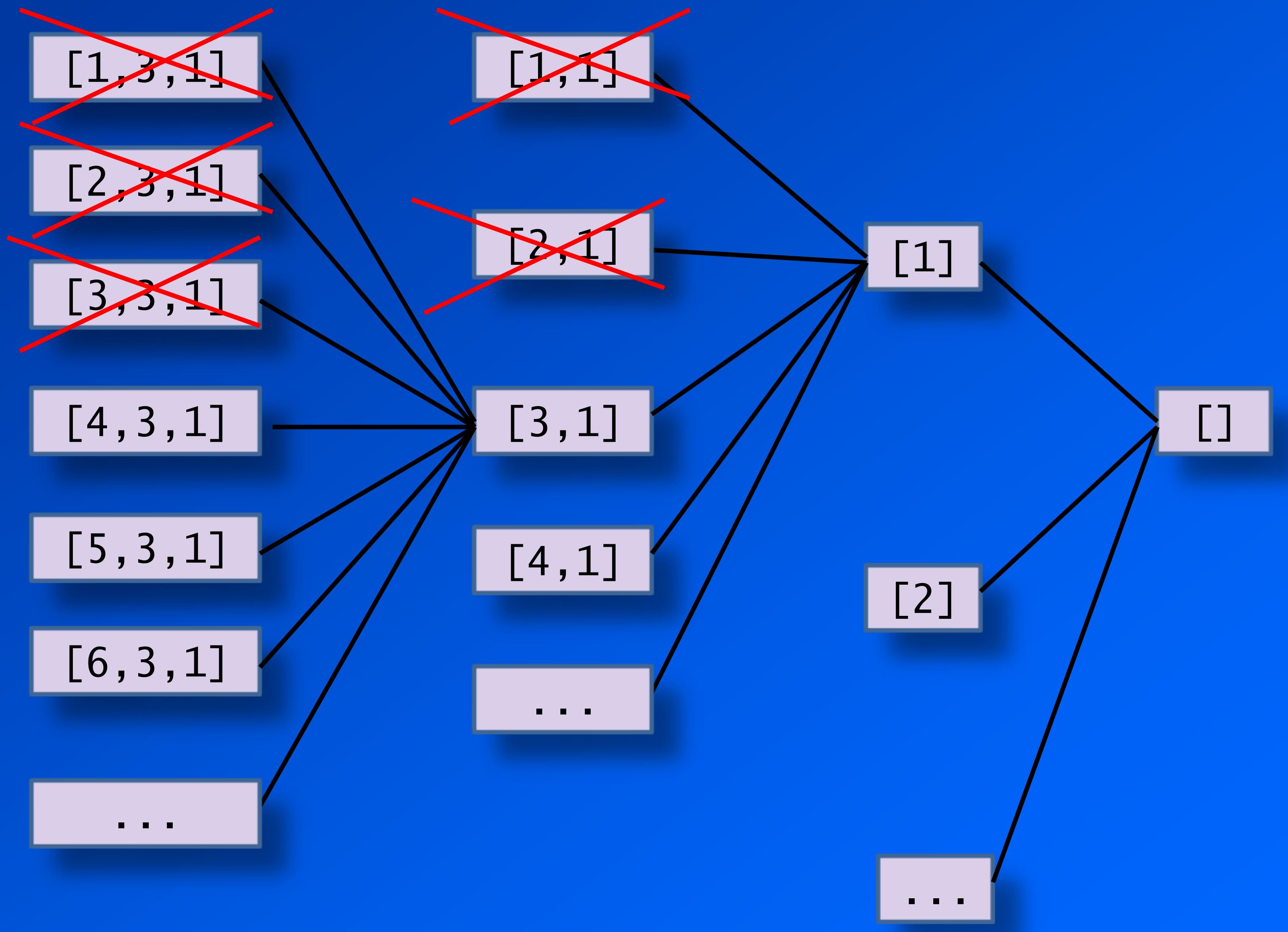
How to use par

- par is very cheap: a write into a circular buffer
- The idea is to create a lot of sparks
  - surplus parallelism doesn't hurt
  - enables scaling to larger core counts without changing the program
- par allows very fine-grained parallelism
  - but using bigger grains is still better

The N-queens problem

Place *n* queens on an *n* x *n* board such that no queen attacks any other, horizontally, vertically, or diagonally

N queens



N-queens in Haskell

```
nqueens :: Int -> [[Int]]
nqueens n = subtree n []
 where
    children :: [Int] -> [[Int]]
    children b = [ (q:b) | q <- [1..n],
                           safe q b ]

    subtree :: Int -> [Int] -> [[Int]]
    subtree 0 b = [b]
    subtree c b =
      concat $
        map (subtree (c-1)) $
          children b

    safe :: Int -> [Int] -> Bool
    ...
```

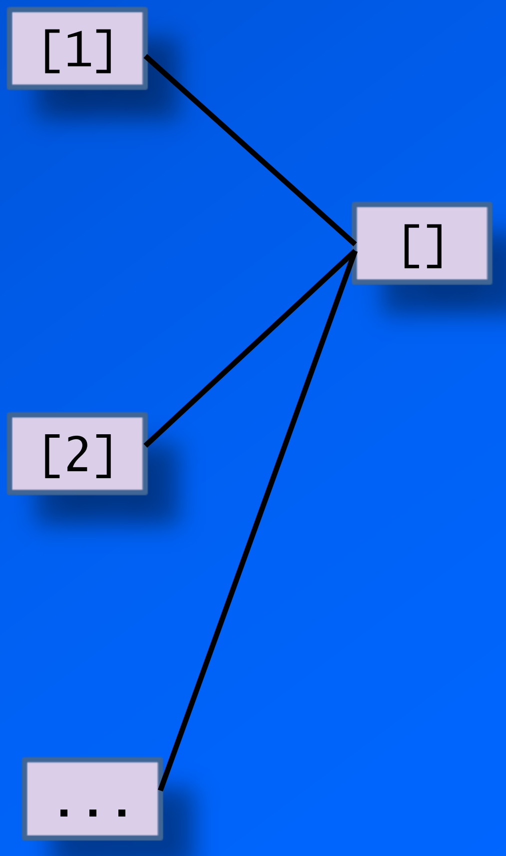
A board is represented as a list of queen rows

children calculates the valid boards that can be made by adding another queen

subtree calculates all the valid boards starting from the given board by adding *c* more columns

Parallel N-queens

- How can we parallelise this?
- *Divide and conquer*
  - aka map/reduce
  - calculate subtrees in parallel, join the results



Parallel N-queens

```
nqueens :: Int -> [[Int]]
nqueens n = subtree n []
 where
    children :: [Int] -> [[Int]]
    children b = [ (q:b) | q <- [1..n],
                           safe q b ]

    subtree :: Int -> [Int] -> [[Int]]
    subtree 0 b = [b]
    subtree c b =
      concat $
       parList $
        map (subtree (c-1)) $
          children b
```

```
parList :: [a] -> b -> b
```

parList is not built-in magic...

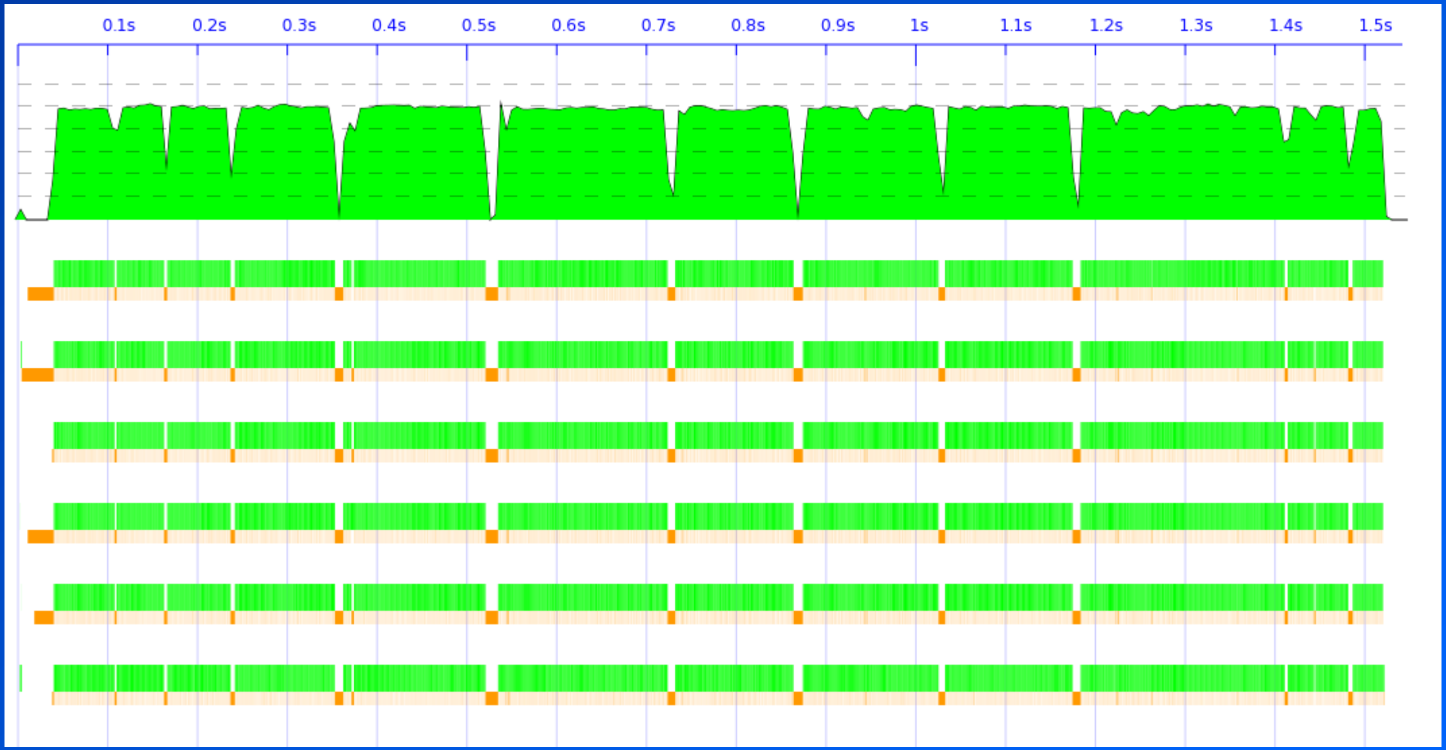
- It is defined using par:

```
parList :: [a] -> b -> b
parList []      b = b
parList (x:xs) b = par x $ parList xs b
```

- (full disclosure: in N-queens we need a slightly different version in order to fully evaluate the nested lists)

Results

- Speedup: 3.5 on 6 cores



- We can do better...

How many sparks?

```
SPARKS: 5151164 (5716 converted, 4846805 pruned)
```

- The cost of creating a spark for every tree node is high
- sparks near the leaves are cheap
- Parallelism works better when the work units are large *(coarse-grained parallelism)*
- But we don’t want to be too coarse, or there won’t be enough grains
- Solution: parallelise down to a certain depth

Bounding the parallel depth

```
subtree :: Int -> [Int] -> [[Int]]
subtree 0 b = [b]
subtree c b =
   concat $
    maybeParList c $
     map (subtree (c-1)) $
       children b

maybeParList c
   | c < threshold = id
   | otherwise     = parList
```

change parList into maybeParLIst

below the threshold, maybeParList is "id" (do nothing)

Results...

- Speedup: 4.7 on 6 cores
  - depth 3
  - ~1000 sparks

Can this be improved?

- There is more we could do here, to optimise both sequential and parallel performance
- but we got good results with only a little effort

Original sequential version

- However, we did have to change the original program... trees good, lists bad:

```
nqueens :: Int -> [[Int]]
nqueens n = gen n
 where
    gen :: Int -> [[Int]]
    gen 0 = [[]]
    gen c = [ (q:b) | b <- gen (c-1),
                      q <- [1..n],
                      safe q b]
```

- c.f. Guy Steele “Organising Functional Code for Parallel Execution”

Raising the level of abstraction

- Lowest level: par/pseq
- Next level: parList
- A general abstraction: Strategies[1]

A value of type `Strategy a` is a policy for evaluating things of type `a`

```
parPair :: Strategy a -> Strategy b -> Strategy (a,b)
```

- a strategy for evaluating components of a pair in parallel, given a Strategy for each component

[1] *Algorithm + strategy = parallelism,* **Trinder et. al., JFP 8(1),1998**

Define your own Strategies

- Strategies are just an abstraction, defined in Haskell, on top of par/pseq

```
type Strategy a = a -> Eval a
using :: a -> Strategy a -> a
```

```
data Tree a = Leaf a | Node [Tree a]

parTree :: Int -> Strategy (Tree [Int])
parTree 0 tree        = rdeepseq tree
parTree n (Leaf a)    = return (Leaf a)
parTree n (Node ts) = do
  us <- parList (parTree (n-1)) ts
  return (Node us)
```

A strategy that evaluates a tree in parallel up to the given depth

Refactoring N-queens

```
data Tree a = Leaf a | Node [Tree a]

leaves :: Tree a -> [a]

nqueens n = leaves (subtree n [])
 where
 subtree :: Int -> [Int] -> Tree [Int]
 subtree 0 b = Leaf b
 subtree c b = Node (map (subtree (c-1)) (children b))
```

Refactoring N-queens

- Now we can move the parallelism to the outer level:

```
nqueens n = leaves (subtree n [] `using` parTree 3)
```

Modular parallelism

- The description of the parallelism can be separate from the algorithm itself
  - thanks to lazy evaluation: we can build a structured computation without evaluating it, the strategy says how to evaluate it
  - don’t clutter your code with parallelism
  - (but be careful about space leaks)

Parallel Haskell, summary

- par, pseq, and Strategies let you *annotate* purely functional code for parallelism
- Adding annotations does not change what the program *means*
  - no race conditions or deadlocks
  - easy to experiment with
- ThreadScope gives visual feedback
- The overhead is minimal, but parallel programs scale
- You still have to understand how to parallelise the algorithm!
- Complements concurrency

Take a deep breath...

- ... we’re leaving the purely functional world and going back to threads and state

Concurrent data structures

- Concurrent programs often need shared data structures, e.g. a database, or work queue, or other program state
- Implementing these structures well is extremely difficult
- So what do we do?
  - let Someone Else do it (e.g. Intel TBB)
    - but we might not get exactly what we want
  - In Haskell: do it yourself...

Case study: Concurrent Linked Lists

```
newList    :: IO (List a)
```

Creates a new (empty) list

```
addToTail :: List a -> a -> IO ()
```

Adds an element to the tail of the list

```
find       :: Eq a => List a -> a -> IO Bool
```

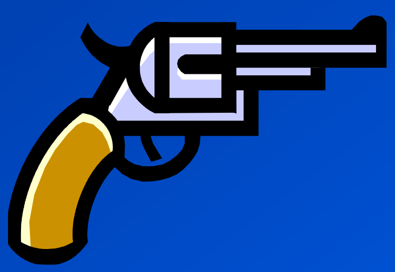
Returns True if the list contains the given element

```
delete     :: Eq a => List a -> a -> IO Bool
```

Deletes the given element from the list;
returns True if the list contained the element

Choose your weapon

CAS: atomic compare-and-swap, accurate but difficult to use



MVar: a locked mutable variable. Easier to use than CAS.

STM: Software Transactional Memory. Almost impossible to go wrong.

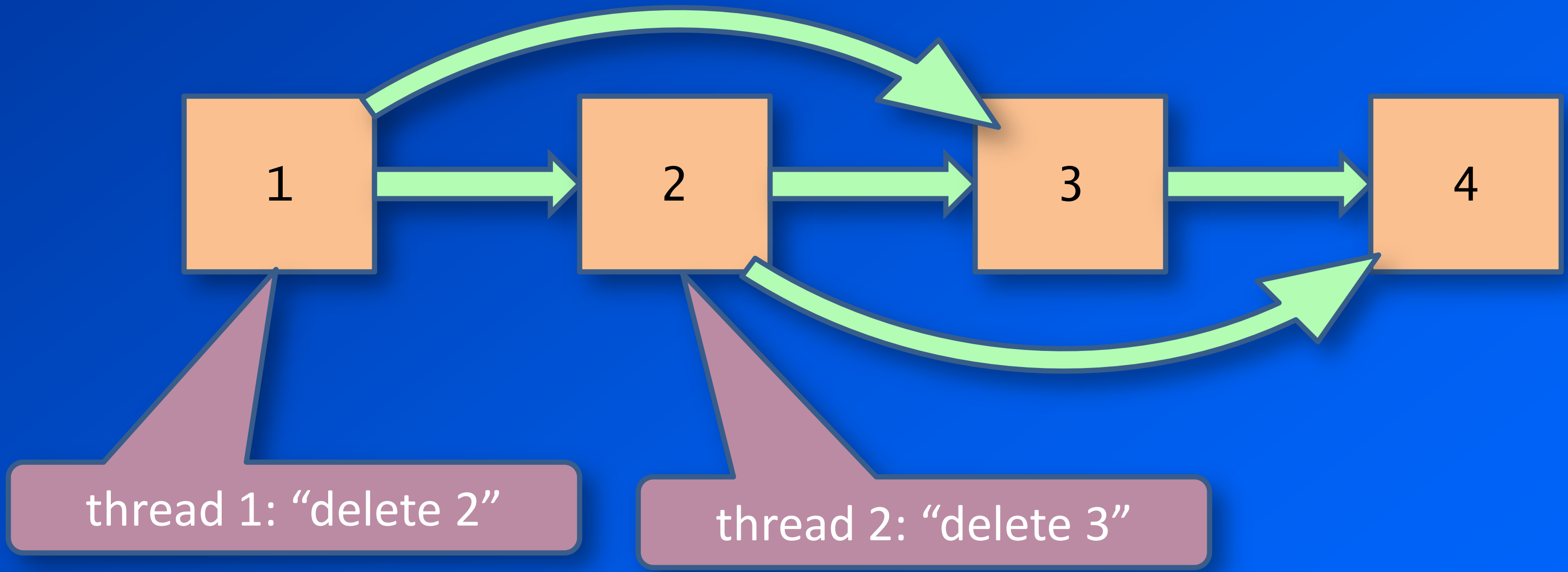
STM implementation

- Nodes are linked with transactional variables

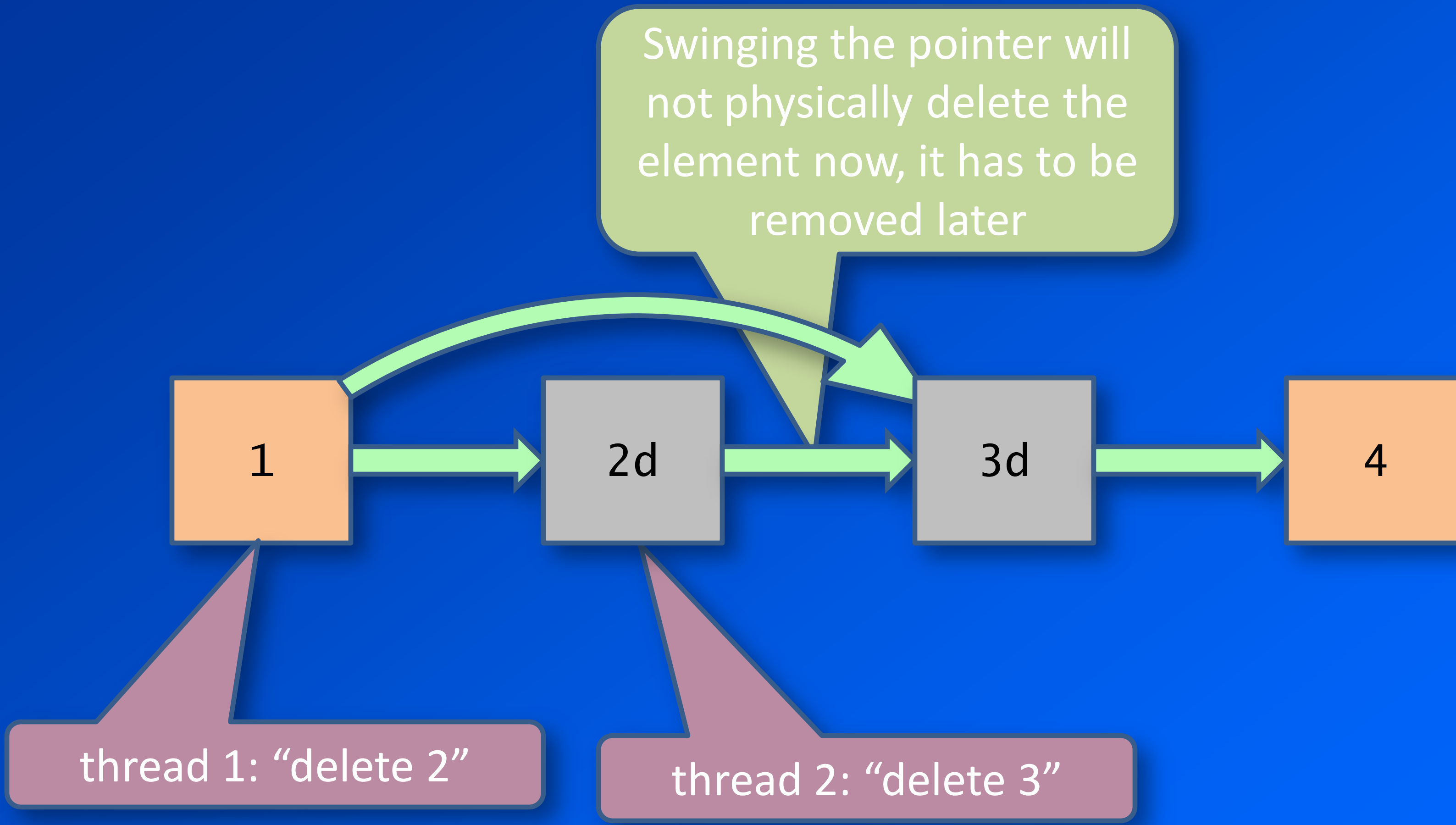
```
data List a = Null
            | Node { val :: a,
                     next :: TVar (List a) }
```

- Operations perform a transaction on the whole list: simple and straightforward to implement
- What about without STM, or if we want to avoid large transactions?

What can go wrong?



Fixing the race condition



Adding "lazy delete"

- Now we have a deleted node:

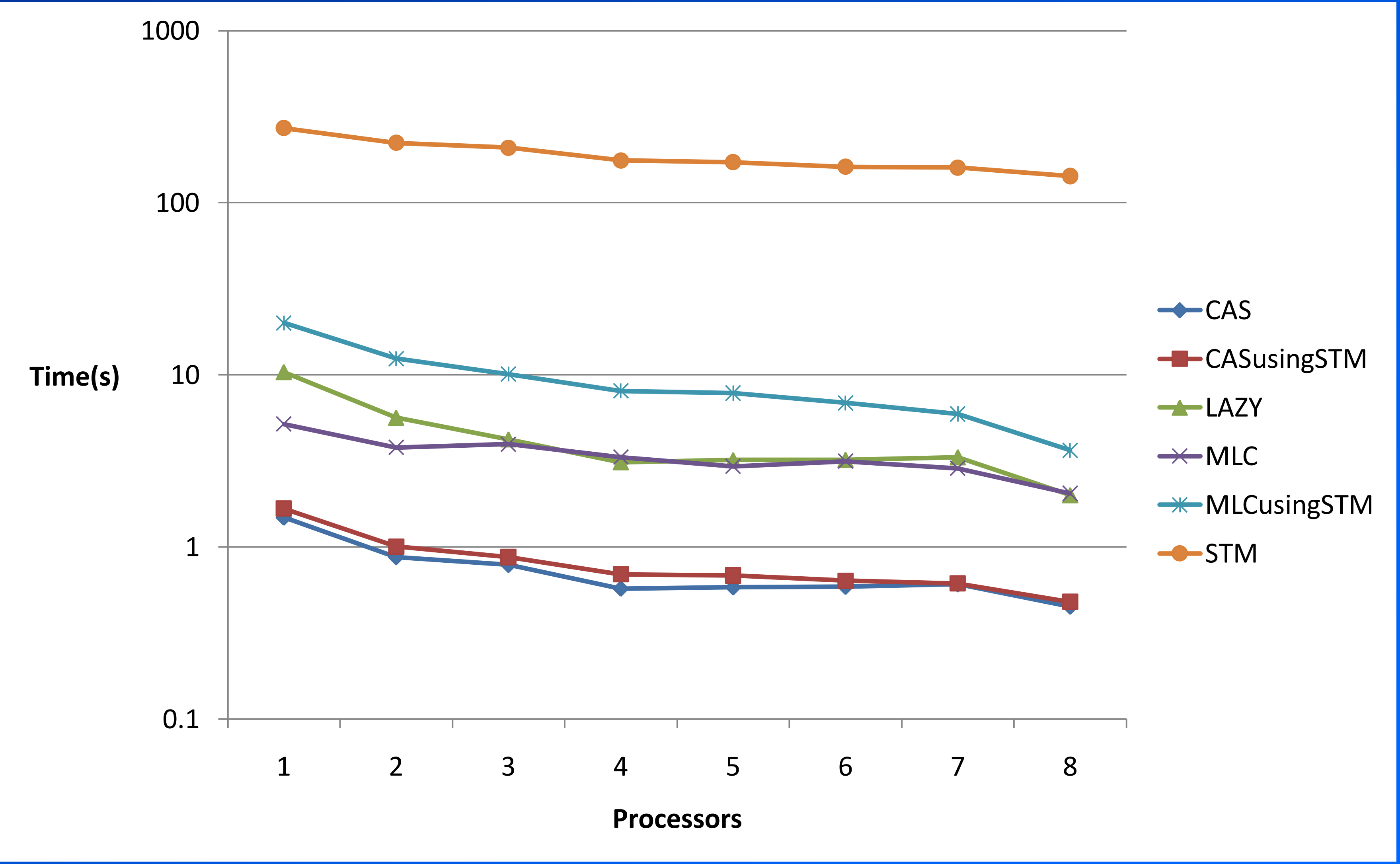
```
data List a = Null
            | Node    { val  :: a,
                        next :: TVar (List a) }
            | DelNode { next :: TVar (LIst a) }
```

- Traversals should drop deleted nodes that they find.
- Transactions no longer take place on the whole list, only pairs of nodes at a time.

We built a few implementations...

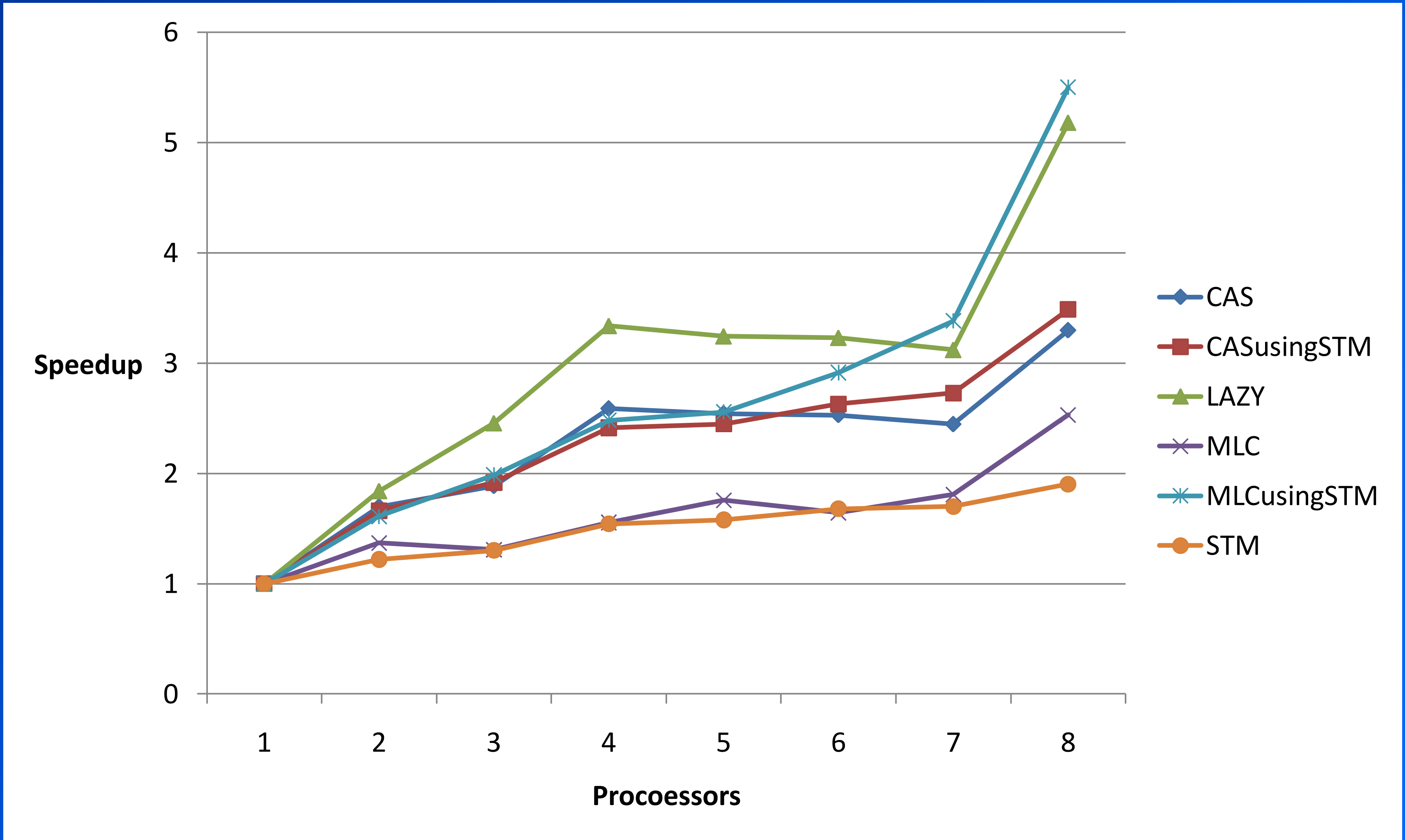
- Full STM
- Various “lazy delete” implementations:
  - STM
  - MVar, hand-over-hand locking
  - CAS
  - CAS (using STM)
  - MVar (using STM)

Results



1000
100
10
1
0.1
Time(s)
1
2
3
4
5
6
7
8
Processors
CAS
CASusingSTM
LAZY
MLC
MLCusingSTM
STM

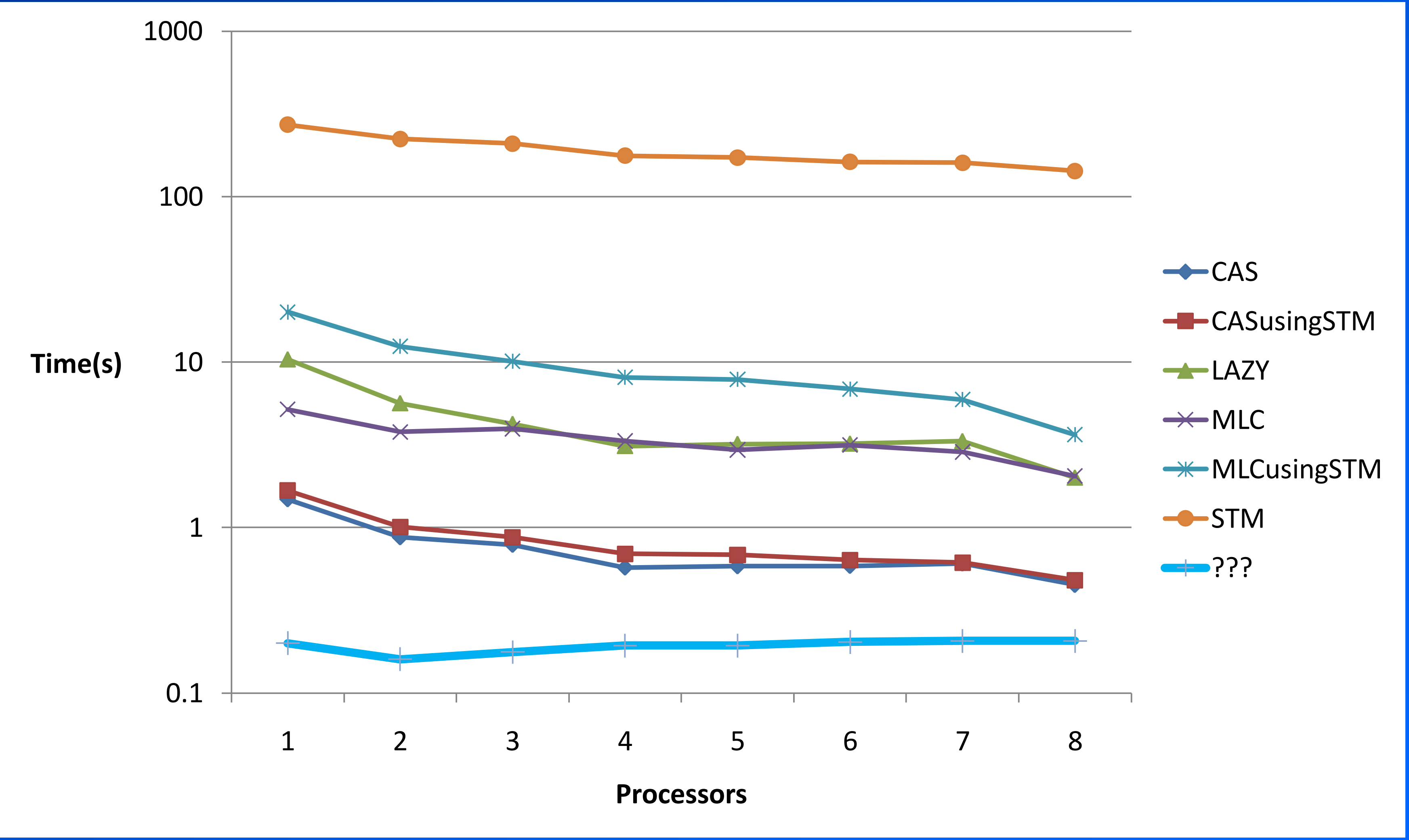
Results (scaling)



So what?

- Large STM transactions don’t scale
- The fastest implementations use CAS
- but then we found a faster implementation...

A latecomer wins the race…



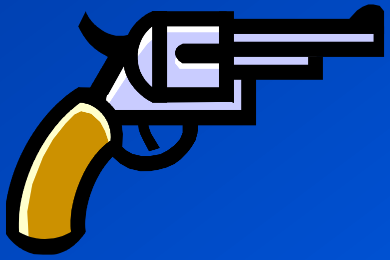
And the winner is...

```
type List a = Var [a]
```

- Ordinary immutable lists stored in a single mutable variable
- trivial to define the operations
- reads are *fast* and automatically concurrent:
  - immutable data is copy-on-write
  - a read grabs a snapshot
- but what about writes? Var = ???

Choose your weapon

IORef (unsynchronised mutable variable) ✓



MVar (locked mutable variable) ✗

TVar (STM) ✓

Built-in lock-free updates

- IORef provides this clever operation:

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

Takes a mutable variable

and a function to compute the new value (a) and a result (b)

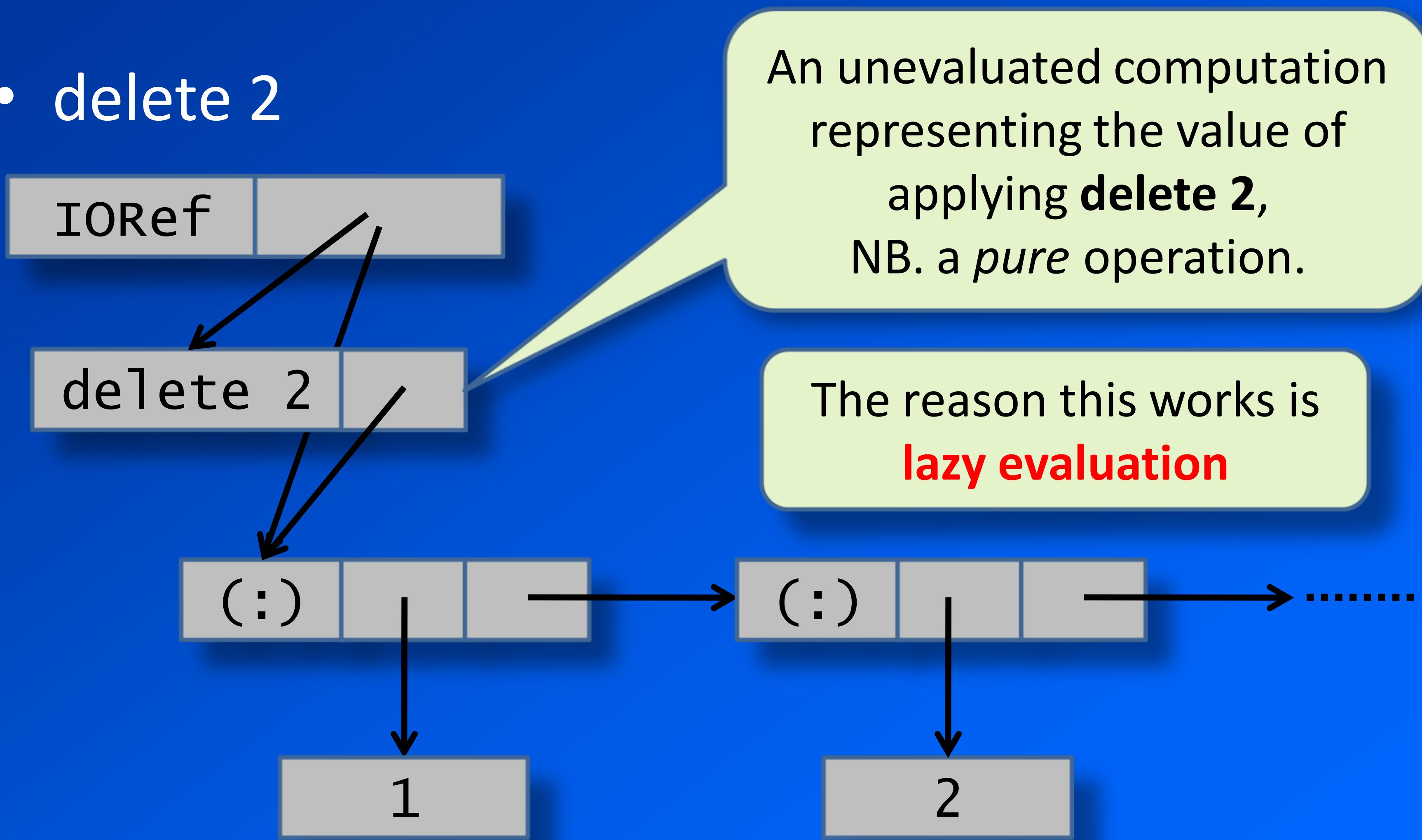
Returns the result

```
atomicModifyIORef r f = do
  a <- readIORef r
  let (new, b) = f a
  writeIORef r new
  return b
```

Lazily!

Updating the list...

- delete 2



Lazy immutable = parallel

- reads can happen in parallel with other operations, automatically
- tree-shaped structures work well: operations in branches can be computed in parallel
- lock-free: impossible to prevent other threads from making progress
- The STM variant is *composable*

Ok, so why didn't we see scaling?

- this is a shared data structure, a single point of contention
- memory bottlenecks, cache bouncing
- possibly: interactions with generational GC
- but note that we didn't see a slowdown either

A recipe for concurrent data structures

- Haskell has lots of libraries providing high-performance pure data structures
- trivial to make them concurrent:

```
type ConcSeq a    = IORef (Seq a)
type ConcTree a   = IORef (Tree a)
type ConcMap k v = IORef (Map k v)
type ConcSet a    = IORef (Set a)
```

Conclusions...

- Thinking concurrent (and parallel):
  - Immutable data and pure functions
    - eliminate unnecessary interactions
  - Declarative programming models say less about "how", giving the implementation more freedom
    - SQL/LINQ/PLINQ
    - map/reduce
    - .NET TPL: declarative parallelism in .NET
    - F# async programming
    - Coming soon: Data Parallel Haskell

Try it out...

- Haskell: http://www.haskell.org/
- GHC: http://www.haskell.org/ghc
- Libraries: http://hackage.haskell.org/
- News: http://www.reddit.com/r/haskell

- me: Simon Marlow <simonmar@microsoft.com>