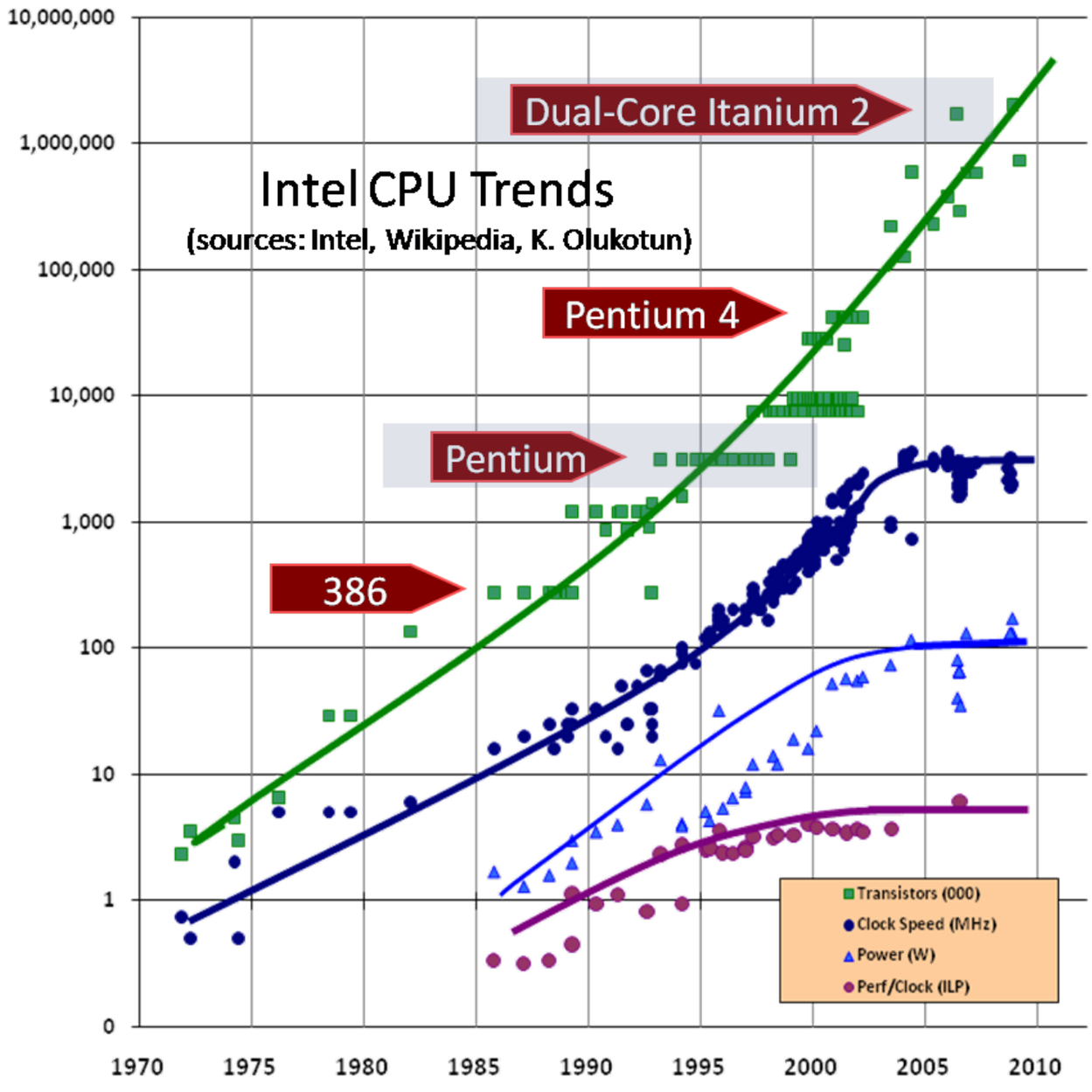
Parallel Programming

Christophe Scholliers

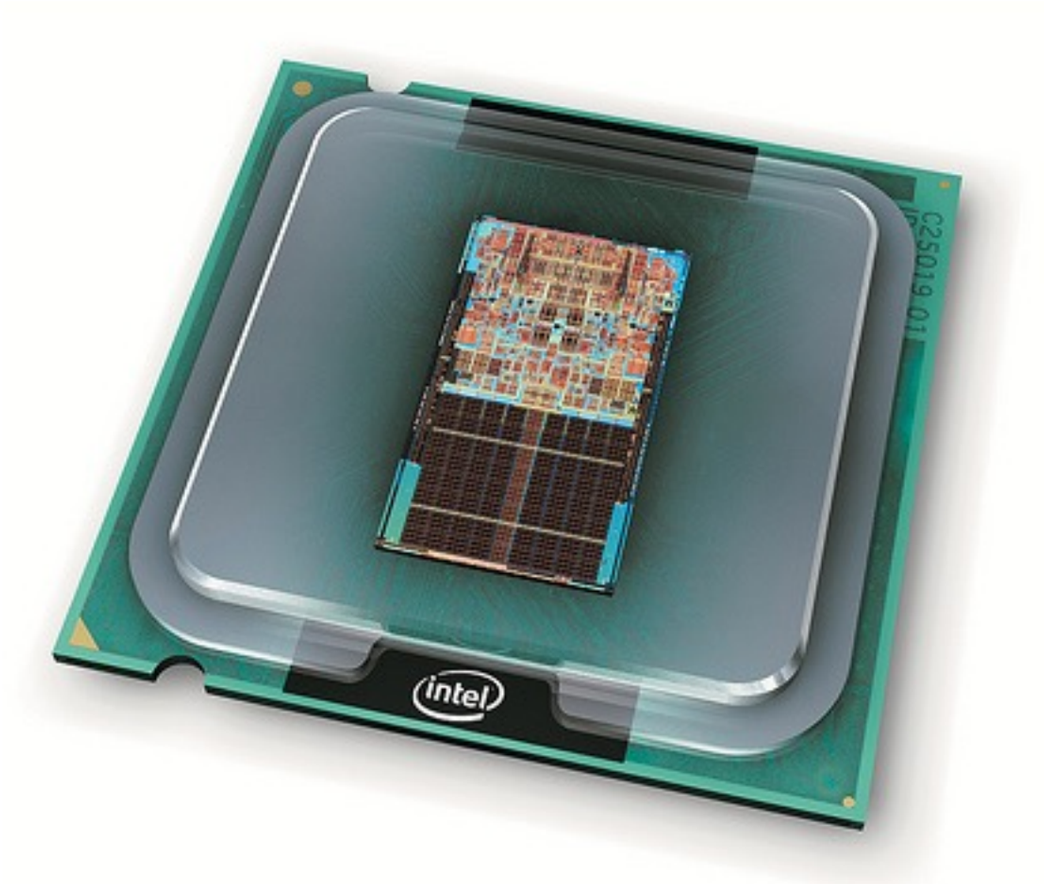
Slides are a reformulation of “Beautiful Concurrency” by Simon Peyton Jones.

Free lunch is over



The Free Lunch Is Over
A Fundamental Turn Toward Concurrency in Software
By Herb Sutter

Faster Programs



= write parallel programs !

Overview

Semi-explicit Parallelism

Software transactional Memory

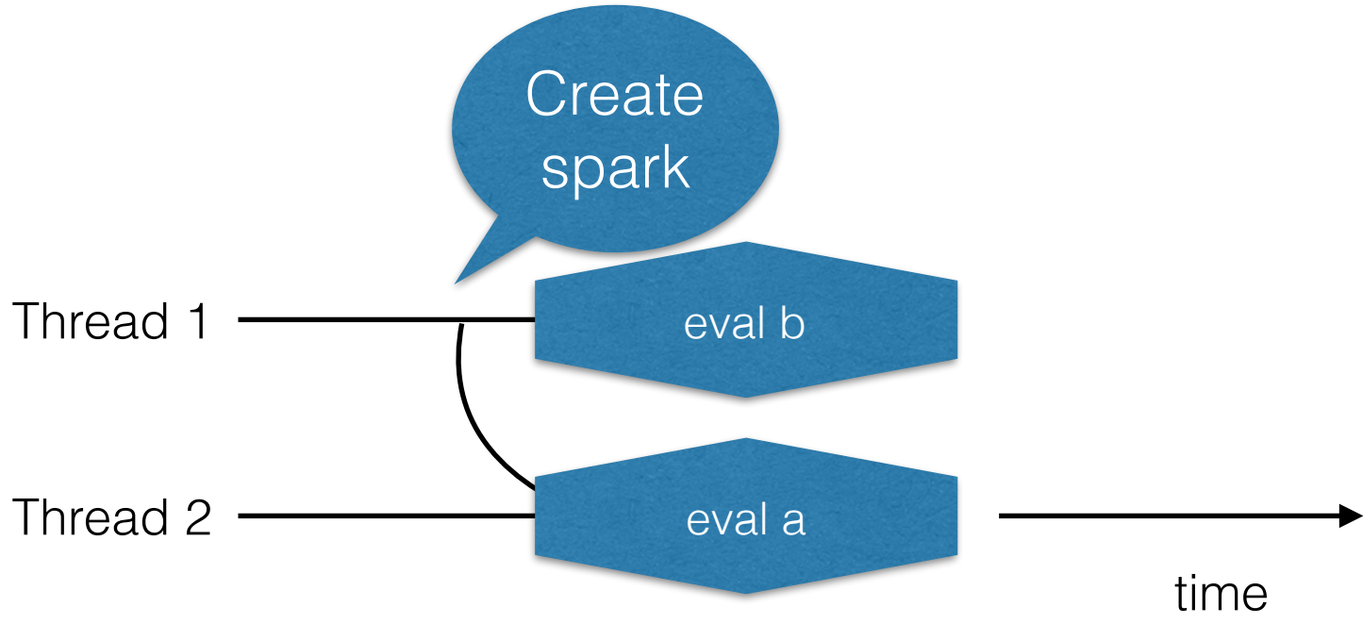
Explicit Parallelism

Semi-Explicit Parallelism

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

```
a `par` b
```

Hint



Number crunching

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

```haskell
relprime :: Int -> Int -> Bool
relprime x y = gcd x y == 1

euler :: Int -> Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int -> Int
sumEuler = sum . (map euler) . mkList
```

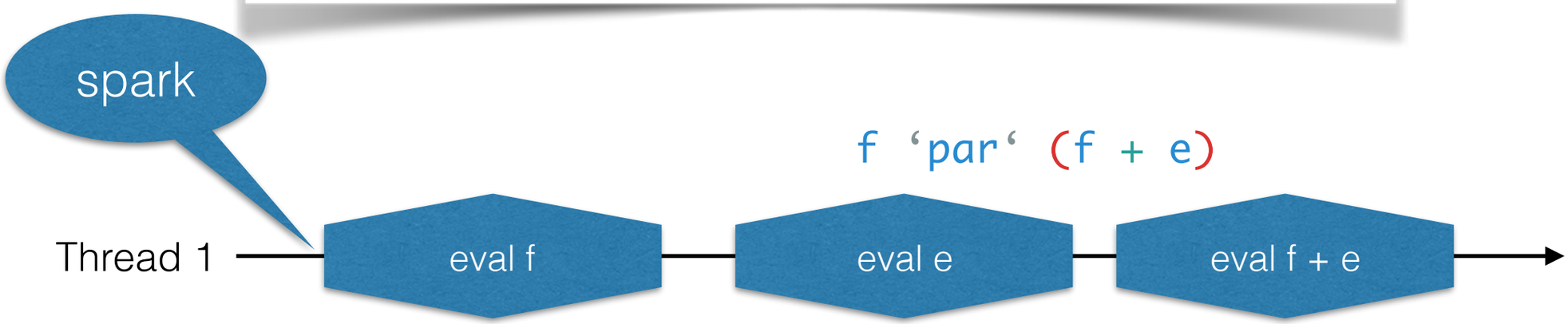
```haskell
sumFibEuler :: Int -> Int -> Int
sumFibEuler a b = fib a + sumEuler b
```

Parallel Version

```haskell
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b = f `par` (f + e)
        where
        f = fib a
        e = sumEuler b
```

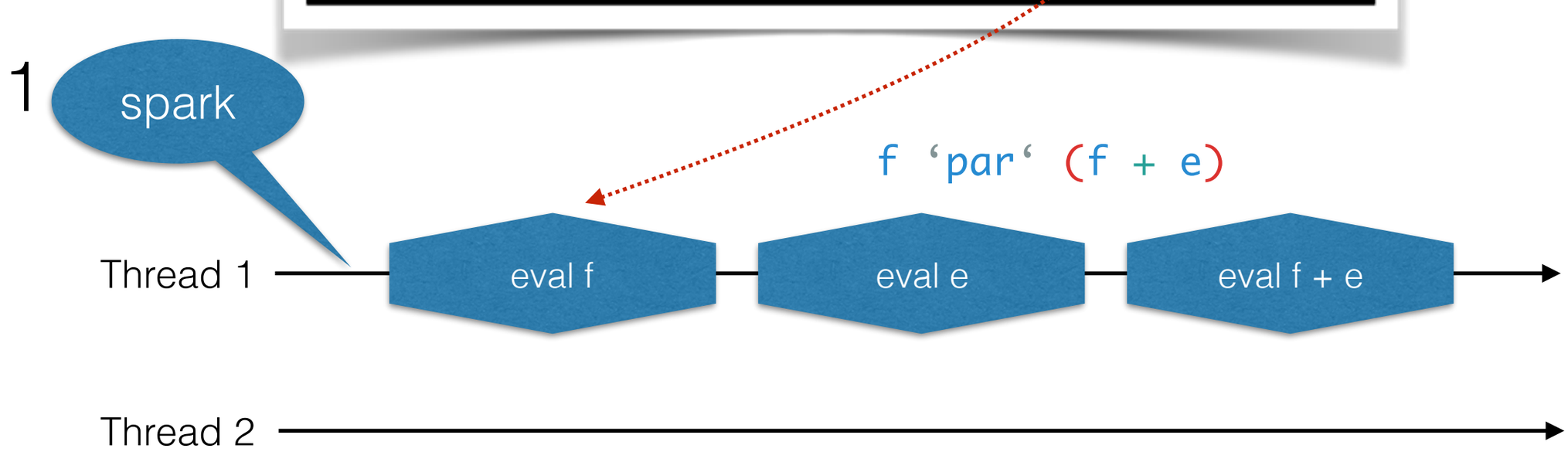
Let’s analyse

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b = f `par` (f + e)
        where
        f = fib a
        e = sumEuler b
```



Let’s analyse

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b = f `par` (f + e) 2
        where
        f = fib a
        e = sumEuler b
```



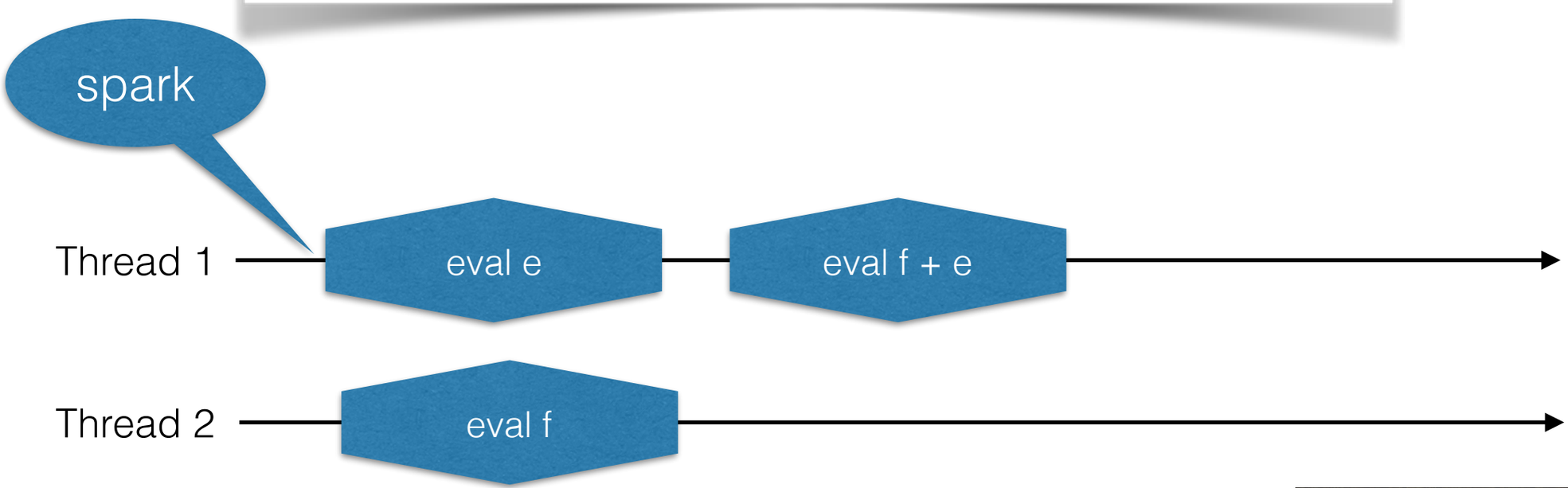
Force evaluation order

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

“evaluate a then evaluate b”

Correct Version

```haskell
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b = f `par` (e `pseq` (e + f))
        where
        f = fib a
        e = sumEuler b
```




```haskell
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b = f `par` (e `pseq` (f + e))
        where
        f = fib a
        e = sumEuler b
```

Let’s try again

```haskell
mapFib :: [Int]
mapFib = map fib [37, 38, 39, 40]

mapEuler :: [Int]
mapEuler = map sumEuler [7600, 7600]

parMapFibEuler :: Int
parMapFibEuler = mapFib `par`
                 (mapEuler `pseq` (sum mapFib + sum mapEuler))
```

WHNF

```
mapFib     =>  (fib 37) : map fib [38, 39, 40]
```

```
mapFib :: [Int]
mapFib = map fib [37, 38, 39, 40]

mapEuler :: [Int]
mapEuler = map sumEuler [7600, 7600]

parMapFibEuler :: Int
parMapFibEuler = mapFib `par`
                 (mapEuler `pseq` (sum mapFib + sum mapEuler))
```

Force evaluation of a list

```
forceList :: [a] -> ()
forceList []     = ()
forceList (x:xs) = x `pseq` forceList xs
```

```
parMapFibEuler :: Int
parMapFibEuler = (forceList mapFib) `par`
        (forceList mapEuler `pseq` (sum mapFib + sum mapEuler))
```

Overview

Semi-explicit Parallelism

Software transactional Memory

Explicit Parallelism

Explicit Parallelism

Spawn

```haskell
forkIO :: IO () -> IO ThreadId
forkOS :: IO () -> IO ThreadId

data MVar a
newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)

takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a

tryTakeMVar  :: MVar a -> IO (Maybe a)
tryPutMVar   :: MVar a -> a -> IO Bool
isEmptyMVar  :: MVar a -> IO Bool
```

Blocking

Non Blocking

Rendez-Vous

```haskell
threadA :: MVar Int -> MVar Float -> IO ()
threadA valueToSendMVar valueReceiveMVar
        = do -- some work
             -- now perform rendezvous by sending 72
             putMVar valueToSendMVar 72
             v <- takeMVar valueReceiveMVar
             putStrLn (show v)

threadB :: MVar Int -> MVar Float -> IO ()
threadB valueToReceiveMVar valueToSendMVar
        = do -- some work
             -- now perform rendezvous by waiting on value
             z <- takeMVar valueToReceiveMVar
             putMVar valueToSendMVar (1.2 * z)
             -- continue with other work

main :: IO ()
main = do aMVar <- newEmptyMVar
          bMVar <- newEmptyMVar
          forkIO (threadA aMVar bMVar)
          forkIO (threadB aMVar bMVar)
          threadDelay 1000
```

Explicit Parallel Fib

Lazy

```
fibThread :: Int -> MVar Int -> IO ()
fibThread n resultMVar = putMVar resultMVar (fib n)

s1 :: Int
s1 = sumEuler 7450

main :: IO ()
main = do putStrLn "explicit SumFibEuler"
          fibResult <- newEmptyMVar
          forkIO (fibThread 40 fibResult)
          pseq s1 (return ())
          f <- takeMVar fibResult
          putStrLn ("sum: " ++ show (s1+f))
```

Explicit Parallel Fib

```haskell
fibThread :: Int -> MVar Int -> IO ()
fibThread n resultMVar
     = do  pseq f (return ())
           putMVar resultMVar f
           where
             f = fib n

s1 :: Int
s1 = sumEuler 7450

main :: IO ()
main = do putStrLn "explicit SumFibEuler"
          fibResult <- newEmptyMVar
          forkIO (fibThread 40 fibResult)
          pseq s1 (return ())
          f <- takeMVar fibResult
          putStrLn ("sum: " ++ show (s1+f))
```

Overview

Semi-explicit Parallelism

Software transactional Memory

Explicit Parallelism

Software Transactional Memory

Implementing a bankaccount

Java style

```
class Account {
        Int balance;
        synchronized void withdraw( int n ) {
                balance = balance - n;
        }

        void deposit( int n ) {
                withdraw( -n );
        }
}
```

```
void transfer( Account from, Account to, Int amount {
        from.withdraw( amount );
        to.deposit( amount );
}
```

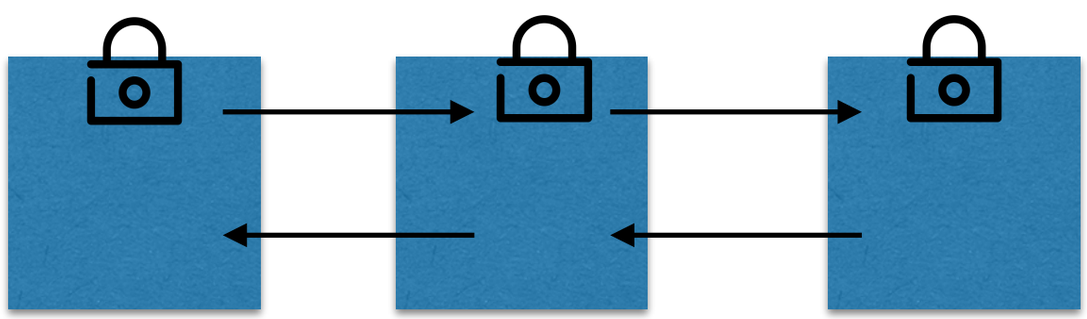
```
void transfer( Account from, Account to, Int amount ) {
        from.lock();
        to.lock();
        from.withdraw( amount );
        to.deposit( amount );
        from.unlock(); to.unlock();
}
```

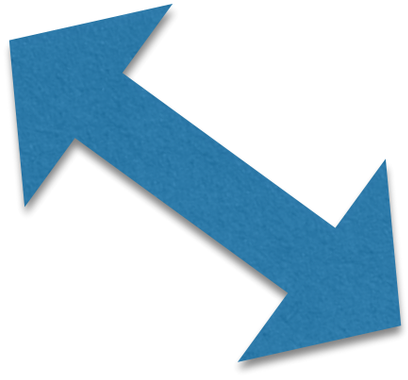
Possible Problems

- ***Taking too few locks.*** *It is easy to forget to take a lock and thereby end up with two threads that modify the same variable simultaneously.*
- ***Taking too many locks.*** *It is easy to take too many locks and thereby inhibit concurrency (at best) or cause deadlock (at worst).*
- ***Taking the wrong locks.*** *In lock-based programming, the connection between a lock and the data it protects often exists only in the mind of the programmer, and is not explicit in the program. As a result, it is all too easy to take or hold the wrong locks.*
- ***Taking locks in the wrong order.*** *In lock-based programming, one must be careful to take locks in the "right" order. Avoiding the deadlock that can otherwise occur is always tiresome and error-prone, and sometimes extremely difficult.*
- ***Error recovery*** *can be very hard, because the programmer must guarantee that no error can leave the system in a state that is inconsistent, or in which locks are held indefinitely.*
- ***Lost wake-ups and erroneous retries.*** *It is easy to forget to signal a condition variable on which a thread is waiting; or to re-test a condition after a wake-up.*

from Beautiful concurrency,Simon Peyton Jones, Microsoft Research, Cambridge, 2007

Locks are super hard





Thread 1

What to do when queue has size 0 1 or 2 !?

Thread 2

Software Transactional Memory

```
data STM a -- A monad supporting atomic memory transactions
atomically  :: STM a -> IO a           -- Perform a series of STM actions atomically
retry       :: STM a                   -- Retry current transaction from the beginning
orElse      :: STM a -> STM a -> STM a -- Compose two transactions


data TVar a -- Shared memory locations that support atomic memory operations
newTVar    :: a -> STM (TVar a)     -- Create a new TVar with an initial value
readTVar   :: TVar a -> STM a       -- Return the current value stored in a TVar
writeTVar  :: TVar a -> a -> STM () -- Write the supplied value into a TVar
```

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
           )
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
           )
```

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
          ⟶ v <- readTVar bal
            writeTVar bal (v+1)
        )
```

```
atomically (do
          ⟶ v <- readTVar bal
            writeTVar bal (v-3)
        )
```

What	Read	Written
Bal		

What	Read	Written
Bal		

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
          ⟶ v <- readTVar bal
            writeTVar bal (v+1)
        )
```

What	Read	Written
Bal		

```
atomically (do
            v <- readTVar bal
          ⟶ writeTVar bal (v-3)
        )
```

What	Read	Written
Bal	7	

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
           )
```

What	Read	Written
Bal	7	

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
           )
```

What	Read	Written
Bal	7	

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
    ⟶ )
```

What	Read	Written
Bal	7	8

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
    ⟶ )
```

What	Read	Written
Bal	7	

Software Transactional Memory

```
bal :: TVar Int
       7
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
    ⟶ )
```

What	Read	Written
Bal	7	8

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
    ⟶ )
```

What	Read	Written
Bal	7	4

Software Transactional Memory

```
bal :: TVar Int
       8
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
           )  commit
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
           )  ⟶
```

- Thread1 commits
- Shared bal variable is updated
- Transaction log is discarded

What	Read	Written
Bal	7	4

Software Transactional Memory

```
bal :: TVar Int
       8
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v+1)
           )  commit
```

```
atomically (do
              v <- readTVar bal
              writeTVar bal (v-3)
           )
```

What	Read	Written
Bal		

- Attempt to commit thread 2 fails, because value in memory is not consistent with the value in the log
- Transaction re-runs from the beginning

Bank example

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount
        = do  bal <- readTVar acc
              writeTVar acc (bal - amount)

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)


transfer :: Account -> Account -> Int -> IO ()
= atomically (do  deposit to amount
                  withdraw from amount)
```

Separation of IO & STM

```
atomically (do {  x <- readTVar xv
                ; y <- readTVar yv
                ; if x>y then
                    launchMissiles <-------- IO
                  else return () })
```

Retry

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
= do  bal <- readTVar acc
      if amount > 0 && amount > bal
         then retry
         else writeTVar acc (bal - amount)
```

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
= do { bal <- readTVar acc;
       check (amount <= 0 || amount <= bal);
       writeTVar acc (bal - amount) }
```

```
check :: Bool -> STM ()
check True = return ()
check False = retry
```

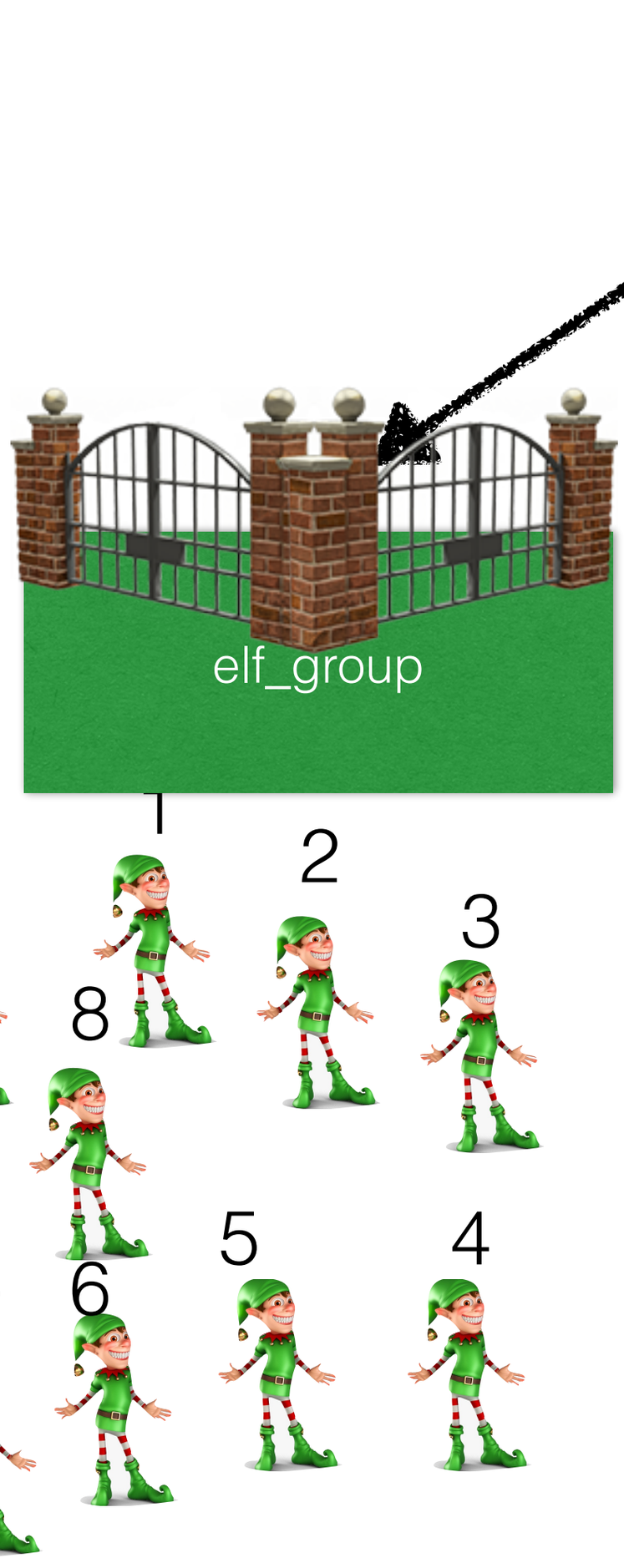
Choice

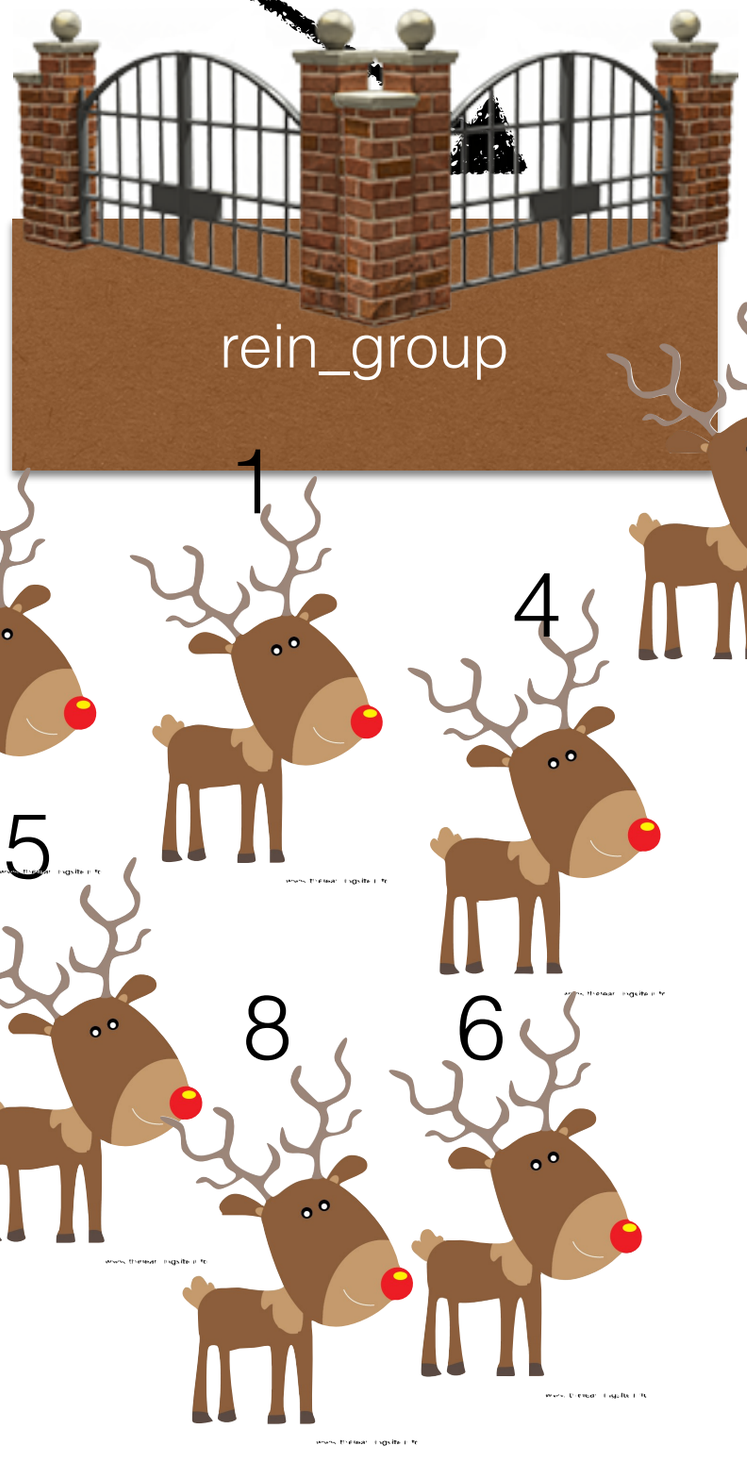
```haskell
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
limitedWithdraw2 acc1 acc2 amt
  = (limitedWithdraw acc1 amt) `orElse` (limitedWithdraw acc2 amt)
```

Santa Clause

Santa Claus Problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer,back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting


elf_group


rein_group

Main Loop

```
main = do { elf_gp <- newGroup 3
          ; sequence_ [ elf elf_gp n | n <- [1..10]]
          ; rein_gp <- newGroup 9
          ; sequence_ [ reindeer rein_gp n | n <- [1..9]]
          ; forever (santa elf_gp rein_gp) }
```

Gates

max left

```
data Gate  = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ()
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ()
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
       ; atomically (do { n_left <- readTVar tv
                        ; check (n_left == 0) }) }
```

Groups

```
newGroup   :: Int -> IO Group
joinGroup  :: Group -> IO (Gate,Gate)
```

```
awaitGroup :: Group -> STM (Gate,Gate)
```

max left

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup :: Int -> IO Group
newGroup n = atomically (do { g1 <- newGate n
                            ; g2 <- newGate n
                            ; tv <- newTVar (n, g1, g2)
                            ; return (MkGroup n tv) })

joinGroup :: Group -> IO (Gate,Gate)
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1, g1, g2)
                   ; return (g1,g2) })

awaitGroup :: Group -> STM (Gate,Gate)
awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
       ; check (n_left == 0)
       ; new_g1 <- newGate n
       ; new_g2 <- newGate n
       ; writeTVar tv (n,new_g1,new_g2)
       ; return (g1,g2) }
```

Fresh gates !

Elves & Reindeers

```haskell
elf1, reindeer1 :: Group -> Int -> IO ()
elf1      group id = helper1 group (meetInStudy id)
reindeer1 group id = helper1 group (deliverToys id)



helper1 :: Group -> IO () -> IO ()
helper1 group do_task
  = do { (in_gate, out_gate) <- joinGroup group
       ; passGate in_gate
       ; do_task
       ; passGate out_gate }

meetInStudy id = putStr ("Elf " ++ show id ++ " meeting in the study\n")
deliverToys id = putStr ("Reindeer " ++ show id ++ " delivering toys\n")
```

One Step

Elves & Reindeers

```haskell
randomDelay :: IO ()
-- Delay for a random time between 1 and 1000,000 microseconds
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))
                 ; threadDelay waitTime }

elf      gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
reindeer gp id = forkIO (forever (do { reindeer1 gp id; randomDelay }))
```

FOREVER !

Santa Clause

```
santa :: Group -> Group -> IO ()
santa elf_group rein_group
  = do { putStr "----------\n"
       ; choose [(awaitGroup rein_group, run "deliver toys"),
                 (awaitGroup elf_group,  run "meet in my study")] }
  where
    run :: String -> (Gate,Gate) -> IO ()
    run task (in_gate,out_gate)
        = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
             ; operateGate in_gate
             ; operateGate out_gate }
```

Choose

```
choose :: [(STM a, a -> IO ())] -> IO ()
```

STM action

```
awaitGroup rein_group
```

When STM action succeeds

```
run "deliver toys"
```

Choose

```
choose :: [(STM a, a -> IO ())] -> IO ()
choose choices = do { to_do <- atomically (foldr1 orElse stm_actions)
                    ; to_do }
  where
    stm_actions :: [STM (IO ())]
    stm_actions = [ do { val <- guard; return (rhs val) }
                  | (guard, rhs) <- choices ]
```

```
choices =
[(awaitGroup rein_group, run "deliver toys"),
  (awaitGroup elf_group, run "meet in my study")]
```

Running the program

```
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
----------
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

Conclusion

- Easier but still not easy
- Level of abstraction is much higher
- STM actions **compose**

- You can still write buggy programs !
- Concurrent programs are still much harder to write
- Performance is less but hardware support is on the way