Parallelism and Concurrency

Advanced Haskell

Andres Löh

10–11 October 2013 —

Parallelism

Parallelism

Running (parts of) programs in parallel on multiple cores (or nodes), in order to speed up the program.

Parallelism

Parallelism

Running (parts of) programs in parallel on multiple cores (or nodes), in order to speed up the program.

The goal is speed, by better utilizing the hardware we have.

Parallelism in Haskell

Haskell is a so-called pure functional language, and

- explicit about side effects,
- non-strict evaluation,
- a strong type system,
- (in GHC) a great run-time system supporting light-weight threading.

Parallelism in Haskell

Haskell is a so-called pure functional language, and

- explicit about side effects,
- non-strict evaluation,
- a strong type system,
- (in GHC) a great run-time system supporting light-weight threading.

So parallelism should be easy in Haskell . . . ?

Parallelism in Haskell

Haskell is a so-called pure functional language, and

- explicit about side effects,
- non-strict evaluation,
- a strong type system,
- (in GHC) a great run-time system supporting light-weight threading.

So parallelism should be easy in Haskell . . . ? And safe?

Parallelism in Haskell

Haskell is a so-called pure functional language, and

- explicit about side effects,
- non-strict evaluation,
- a strong type system,
- (in GHC) a great run-time system supporting light-weight threading.

So parallelism should be easy in Haskell . . . ? And safe? And automatic?

The basic situation

In Haskell, function application is free of side effects, and evaluation is non-strict:

The basic situation

In Haskell, function application is free of side effects, and evaluation is non-strict:

```
f x
```

In principle, we can run `f` in parallel with `x`:

- `f` might not need `x` at all, but no harm is done,
- `f` might need `x` immediately, then no harm is done,
- `f` might not need `x` immediately, then time is saved!

The basic situation

In Haskell, function application is free of side effects, and evaluation is non-strict:

```
f x
```

In principle, we can run `f` in parallel with `x`:

- `f` might not need `x` at all, but no harm is done,
- `f` might need `x` immediately, then no harm is done,
- `f` might not need `x` immediately, then time is saved!

(The final case looks particularly attractive if `x` produces a data structure lazily that is consumed by `f`.)

However . . .

The enemies of parallelism:

- there is overhead in running things in parallel,
- garbage collection is difficult to parallelize,
- non-strictness can not only be helpful, but also tricky:
  - we might run too many things we don't need,
  - it's unclear how far to evaluate speculatively,
  - we have to make clear how it interacts with GC.

However . . .

The enemies of parallelism:

- there is overhead in running things in parallel,
- garbage collection is difficult to parallelize,
- non-strictness can not only be helpful, but also tricky:
  - we might run too many things we don't need,
  - it's unclear how far to evaluate speculatively,
  - we have to make clear how it interacts with GC.

Conclusion

Fully automatic parallelism is still a future goal. For now, we need to help the compiler.

So what about safety?

This is where Haskell shines . . .

So what about safety?

This is where Haskell shines . . .

In other languages, parallelism is often implemented using concurrency:

Concurrency

Language constructs that support structuring a program as if it has many independent threads of control.

Concurrency vs. Parallelism

Concurrency:

- is a goal in its own (program structure),
- usually rather low-level (shared memory, message passing, communication problems, deadlocks, race conditions),
- does not require parallel hardware at all (can be simulated by multitasking on a single core),
- while supported in Haskell, is not an attractive choice for parallelism.

Concurrency vs. Parallelism

Concurrency:

- is a goal in its own (program structure),
- usually rather low-level (shared memory, message passing, communication problems, deadlocks, race conditions),
- does not require parallel hardware at all (can be simulated by multitasking on a single core),
- while supported in Haskell, is not an attractive choice for parallelism.

Parallelism:

- the goal is speed,
- using several cores is the main point,
- there's conceptually no need for low-level effects or IO,
- we would like deterministic results.

Deterministic parallelism

We call a parallel algorithm deterministic if its result is independent of the number of cores it is being run on, and the individual run of the program (scheduling decisions etc.).

Deterministic parallelism

We call a parallel algorithm deterministic if its result is independent of the number of cores it is being run on, and the individual run of the program (scheduling decisions etc.).

Deterministic parallelism is quite unique to Haskell (due to its relative purity), but it removes a significant source of errors and is an extremely cool feature.

Deterministic parallelism

We call a parallel algorithm deterministic if its result is independent of the number of cores it is being run on, and the individual run of the program (scheduling decisions etc.).

Deterministic parallelism is quite unique to Haskell (due to its relative purity), but it removes a significant source of errors and is an extremely cool feature.

Haskell supports multiple approaches to (deterministic) parallelism.

The Haskell landscape

Deterministic approaches:

- nested data parallelism (Data-Parallel Haskell, dph),
- flat data parallelism (`repa`),
- evaluation strategies (`parallel`),
- safe dataflow specification (`monad-par`).

The Haskell landscape

Deterministic approaches:

- nested data parallelism (Data-Parallel Haskell, dph),
- flat data parallelism (`repa`),
- evaluation strategies (`parallel`),
- safe dataflow specification (`monad-par`).

Non-deterministic approaches:

- concurrency primitives (`forkIO` , `MVar`),
- software transactional memory (`stm`),
- high-level asynchronous computations (`async`).

Why so many approaches?

- Parallelism and concurrency are “hot”.
- Parallelising programs (even explicitly) is not trivial.

Why so many approaches?

- Parallelism and concurrency are “hot”.
- Parallelising programs (even explicitly) is not trivial.
- Different forms of parallelism have different demands:
  - data parallelism is about doing the same operations for many pieces of data; a particular common form that warrants dedicated support (dph, `repa`)

Why so many approaches?

- Parallelism and concurrency are “hot”.
- Parallelising programs (even explicitly) is not trivial.
- Different forms of parallelism have different demands:
  - data parallelism is about doing the same operations for many pieces of data; a particular common form that warrants dedicated support (dph, `repa`)
  - task or control parallelism is about dividing the overall work into many parts – these approaches can be used for data parallelism, too (`parallel`, `monad-par`).

The plan

We will focus on a few aspects:

- deterministic task parallelism using strategies;
- using the Par monad instead;
- basic concurrency, threads and communication;
- lock-free concurrency using software transactional memory.

The plan

We will focus on a few aspects:

- deterministic task parallelism using strategies;
- using the `Par` monad instead;
- basic concurrency, threads and communication;
- lock-free concurrency using software transactional memory.

Regardless of the details of the approaches, you'll learn about the major concepts that play a role when writing parallel and concurrent programs.

Strategies

The main idea

A strategy is a description of how and when a value of a particular type should be evaluated.

Strategies can in particular specify:

- that evaluation should be more eager than by default,
- that parts of the value should potentially be evaluated in parallel.

The main idea

A strategy is a description of how and when a value of a particular type should be evaluated.

Strategies can in particular specify:

- that evaluation should be more eager than by default,
- that parts of the value should potentially be evaluated in parallel.

Strategies can then be applied to terms of that type and turn them into annotated terms.

The main idea

A strategy is a description of how and when a value of a particular type should be evaluated.

Strategies can in particular specify:

- that evaluation should be more eager than by default,
- that parts of the value should potentially be evaluated in parallel.

Strategies can then be applied to terms of that type and turn them into annotated terms.

Annotated terms can be run, which means that the strategy will be used in the actual evaluation of the term.

The interface

```
data Eval a                          -- (abstract), annotated terms
instance Monad Eval                  -- we can combine such terms
runEval :: Eval a → a                -- we can execute annotations
type Strategy a = a → Eval a         -- a strategy annotates a term
```

The interface

```
data Eval a                          -- (abstract), annotated terms
instance Monad Eval                  -- we can combine such terms
runEval :: Eval a → a                -- we can execute annotations
type Strategy a = a → Eval a         -- a strategy annotates a term
```

How do we build strategies?

Basic strategies

```
                                                -- evaluation:
r0          ::                    Strategy a    -- none
rseq        ::                    Strategy a    -- WHNF
rdeepseq :: NFData a ⇒ Strategy a               -- NF
rpar        ::                    Strategy a    -- WHNF in parallel
```

Names start with “r”: think “reduce”.

```
r0 = return
```

The first three strategies determine how much of a value is evaluated. The `rpar` strategy introduces parallelism.

Partially defined values

```
undefined :: a    --  undefined  inhabits every type
```

Partially defined values

```
undefined :: a    --  undefined  inhabits every type
```

In practice, we have lots of partially defined values in a structured type:

```
undefined                 :: [Bool]
undefined : undefined     :: [Bool]
undefined : []            :: [Bool]
True      : undefined     :: [Bool]
...
```

Distinguishing values using strategies

Testing values:

```
test :: Strategy a → a → ()
test s x = runEval $ do
                        _ ← s x
                        return ()
```

Even more conveniently than by using `seq` and `deepseq` directly, we can use `test` with suitable strategies to distinguish the values from the previous slide.

Parallelizing an example

Running example – a somewhat expensive function

Collatz function

```
collatz :: Integer → Int
collatz 0                = 0
collatz 1                = 0
collatz n | even n       = 1 + collatz (n `div` 2)
          | otherwise    = 1 + collatz (3 * n + 1)
```

Running example – a somewhat expensive function

Collatz function

```
collatz :: Integer → Int
collatz 0                = 0
collatz 1                = 0
collatz n | even n       = 1 + collatz (n `div` 2)
          | otherwise    = 1 + collatz (3 * n + 1)
```

The sequence is more interesting than one might expect:

```
GHCi>  map collatz [1..10]
[0,1,7,2,5,8,16,3,19,6]
```

Running example – a somewhat expensive function

Collatz function

```
collatz :: Integer → Int
collatz 0                = 0
collatz 1                = 0
collatz n | even n       = 1 + collatz (n `div` 2)
          | otherwise    = 1 + collatz (3 * n + 1)
```

The sequence is more interesting than one might expect:

```
GHCi>  map collatz [1..10]
[0,1,7,2,5,8,16,3,19,6]
```

9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Example, continued

We're interested in the maximum of the Collatz function in a certain interval:

```
maxC :: Int → Int → Int
maxC lo hi = maximum (map (collatz ∘ fromIntegral) [lo . . hi])
```

Example, continued

We're interested in the maximum of the Collatz function in a certain interval:

```
maxC :: Int → Int → Int
maxC lo hi = maximum (map (collatz ∘ fromIntegral) [lo . . hi])
```

```
GHCi> maxC 1 10
19
```

A first attempt at parallelism

```
partest1 n = runEval $ do
                          r1 ← rpar $ maxC 1         h
                          r2 ← rpar $ maxC (h + 1) n
                          return (r1 `max` r2)
  where
    h = n `div` 2
```

How do we run this program in parallel?

Practicalities

We need a main program:

```
import Control.Parallel.Strategies
main = print $ partest1 500000
```

Practicalities

We need a main program:

```
import Control.Parallel.Strategies
main = print $ partest1 500000
```

We compile it with a number of flags:

```
$ ghc -O ParTest1.hs -threaded -rtsopts -eventlog
```

Practicalities

We need a main program:

```
import Control.Parallel.Strategies
main = print $ partest1 500000
```

We compile it with a number of flags:

```
$ ghc -O ParTest1.hs -threaded -rtsopts -eventlog
```

We run with yet more flags:

```
$ ./ParTest1 +RTS -N2
```

Flags

Linker flags for `ghc` invocation:

`-threaded`	Link with the threaded (multi-core) runtime system.
`-rtsopts`	Make the resulting program accept RTS options.
`-eventlog`	For debugging, allow creation of event logs during program runs.

Runtime system (RTS) flags for program invocation:

`+RTS`	Signals that subsequent flags are for the RTS.
`-N2`	Run on two cores. Without numeric argument, run on maximum number of cores available.
`-s`	Print lots of useful performance statistics.
`-lf`	Create an event log.

Speedup!

```
speedup n = sequentialTime / parallelTime n
```

Note that to be entirely correct, we have to compare with the time of the sequential program, not the parallel program run with `-N1`.

Speedup!

```
speedup n = sequentialTime / parallelTime n
```

Note that to be entirely correct, we have to compare with the time of the sequential program, not the parallel program run with `-N1`.

We quickly write `ParTest0` for that purpose, then:

```
$ ./ParTest0 +RTS -s     2>&1 | grep Total
  Total   time    3.50s  (  3.51s elapsed)
$ ./ParTest1 +RTS -s -N2 2>&1 | grep Total
  Total   time    4.10s  (  2.06s elapsed)
```

Speedup of 1.7.

A minor modification

Let's return not only the maximum, but also the sum (for example to compute an average):

```
maxC :: Int → Int → (Int, Int)
maxC lo hi = let cs = map (collatz ∘ fromIntegral) [lo . . hi]
             in  (maximum cs, sum cs)
```

```
partest2 n =
  runEval $ do
    r1 ← rpar $ maxC 1         h
    r2 ← rpar $ maxC (h + 1) n
    let c (m1, s1) (m2, s2) = (m1 `max` m2, s1 + s2)
    return $ c r1 r2
  where
    h = n `div` 2
```

An (unfortunately) not uncommon situation

```
$ ./ParTest2 +RTS -N1 -s 2>&1 | grep Total
  Total   time    3.72s  (  3.77s elapsed)
$ ./ParTest2 +RTS -N2 -s 2>&1 | grep Total
  Total   time    4.50s  (  3.93s elapsed)
```

No speedup – slowdown. What happened?

An (unfortunately) not uncommon situation

```
$ ./ParTest2 +RTS -N1 -s 2>&1 | grep Total
  Total   time    3.72s  (  3.77s elapsed)
$ ./ParTest2 +RTS -N2 -s 2>&1 | grep Total
  Total   time    4.50s  (  3.93s elapsed)
```

No speedup – slowdown. What happened?

```
$ ./ParTest2 +RTS -N2 -s 2>&1 | grep SPARKS
  SPARKS: 2 (0 converted, ..., 2 fizzled)
```

An (unfortunately) not uncommon situation

```
$ ./ParTest2 +RTS -N1 -s 2>&1 | grep Total
  Total   time    3.72s  (  3.77s elapsed)
$ ./ParTest2 +RTS -N2 -s 2>&1 | grep Total
  Total   time    4.50s  (  3.93s elapsed)
```

No speedup – slowdown. What happened?

```
$ ./ParTest2 +RTS -N2 -s 2>&1 | grep SPARKS
  SPARKS: 2 (0 converted, ..., 2 fizzled)
```

2 sparks, 0 converted doesn't sound good . . .

Sparks

- A spark is created whenever a computation is marked for parallel execution using `rpar`.
- For every capability (or HEC, think core), the RTS maintains a spark queue.
- If a capability is idle, it looks at all the spark queues for work to steal. It then converts the spark and executes the computation.

Creation and conversion

When sparks are created, creation can fail because

- the queue is full (overflow),
- the expression is already evaluated (dud).

Creation and conversion

When sparks are created, creation can fail because

- the queue is full (overflow),
- the expression is already evaluated (dud).

When sparks are in the queue, they can

- be run (converted),
- become evaluated independently (fizzle),
- be garbage collected if nothing else needs them.

Creation and conversion

When sparks are created, creation can fail because

- the queue is full (overflow),
- the expression is already evaluated (dud).

When sparks are in the queue, they can

- be run (converted),
- become evaluated independently (fizzle),
- be garbage collected if nothing else needs them.

These statistics are reported by the RTS (more detailed for recent GHC version).

Debugging parallel programs

Introducing ThreadScope

ThreadScope is a graphical debugging tool that visualizes event logs that can be generated from Haskell program runs:

- compile with `-eventlog`,
- run with RTS option `-ls`,
- get useful info about the (in)activity of capabilities and the garbage collector and more.

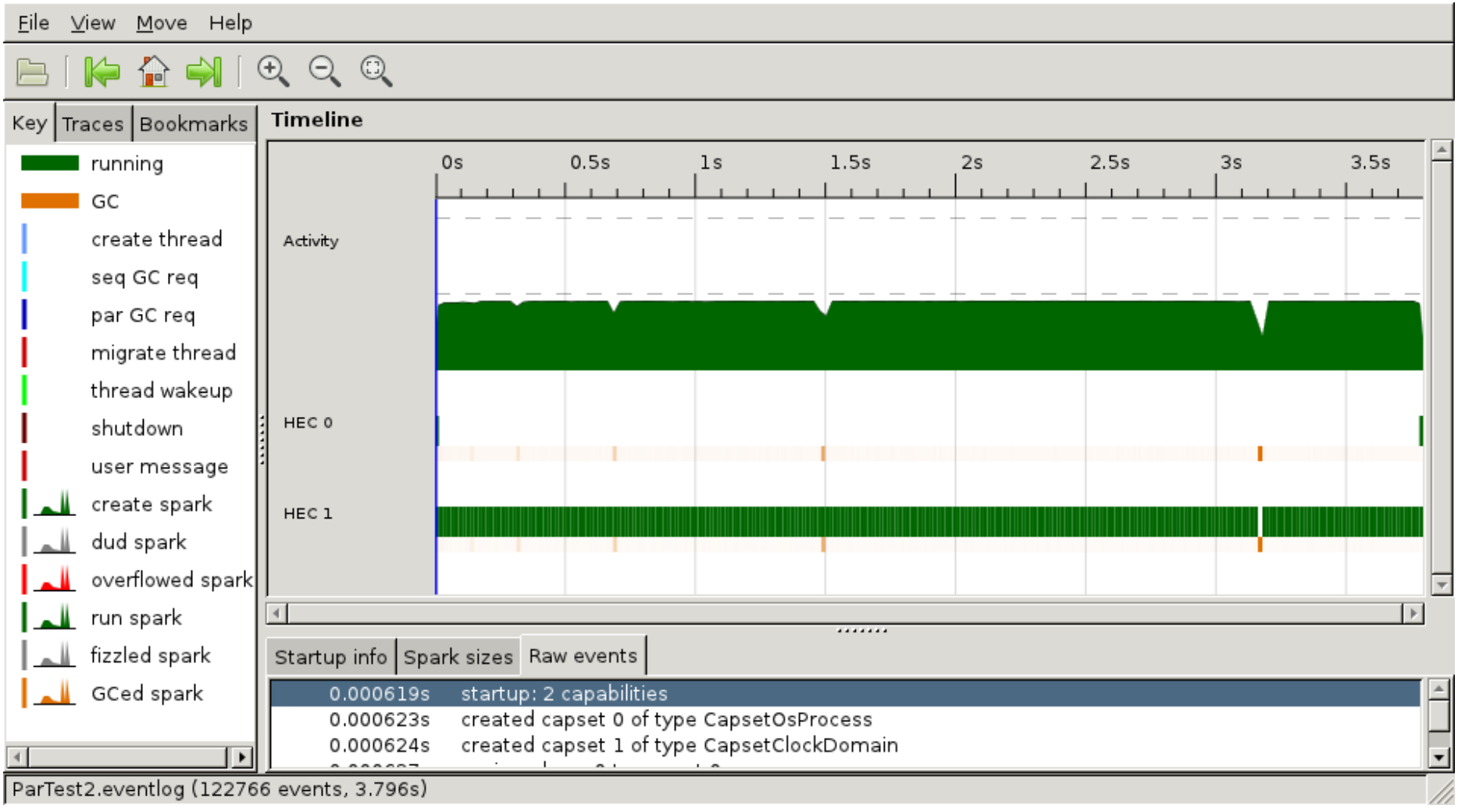
Introducing ThreadScope

ThreadScope is a graphical debugging tool that visualizes event logs that can be generated from Haskell program runs:

- compile with `-eventlog`,
- run with RTS option `-ls`,
- get useful info about the (in)activity of capabilities and the garbage collector and more.

Get ThreadScope:

```
$ cabal install threadscope
```


File View Move Help
Key Traces Bookmarks
running
GC
create thread
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
user message
create spark
dud spark
overflowed spark
run spark
fizzled spark
GCed spark
Timeline
0s 0.5s 1s 1.5s 2s 2.5s 3s 3.5s
Activity
HEC 0
HEC 1
Startup info Spark sizes Raw events
0.000619s startup: 2 capabilities
0.000623s created capset 0 of type CapsetOsProcess
0.000624s created capset 1 of type CapsetClockDomain
ParTest2.eventlog (122766 events, 3.796s)

So what happened?

We are calling

```
r1 ← rpar $ maxC 1          h
r2 ← rpar $ maxC (h + 1) n
```

but

```
maxC lo hi = let cs = map (collatz ∘ fromIntegral) [lo . . hi]
             in  (maximum cs, sum cs)
```

returns immediately in WHNF.

So what happened?

We are calling

```
r1 ← rpar $ maxC 1         h
r2 ← rpar $ maxC (h + 1) n
```

but

```
maxC lo hi = let cs = map (collatz ∘ fromIntegral) [lo . . hi]
             in  (maximum cs, sum cs)
```

returns immediately in WHNF.

So actually, `c` will force the top-level constructors of `r1` and `r2` almost immediately, causing both sparks to fizzle.

How can we fix it?

We must make sure that the work we intend to parallelize is actually performed within the parallel computation:

```
partest3 n =
  runEval $ do
    r1 ← rpar ‘dot‘ rdeepseq $ maxC 1       h
    r2 ← rpar ‘dot‘ rdeepseq $ maxC (h + 1) n
    let c (m1, s1) (m2, s2) = (m1 ‘max‘ m2, s1 + s2)
    return $ c r1 r2
  where
    h = n ‘div‘ 2
```

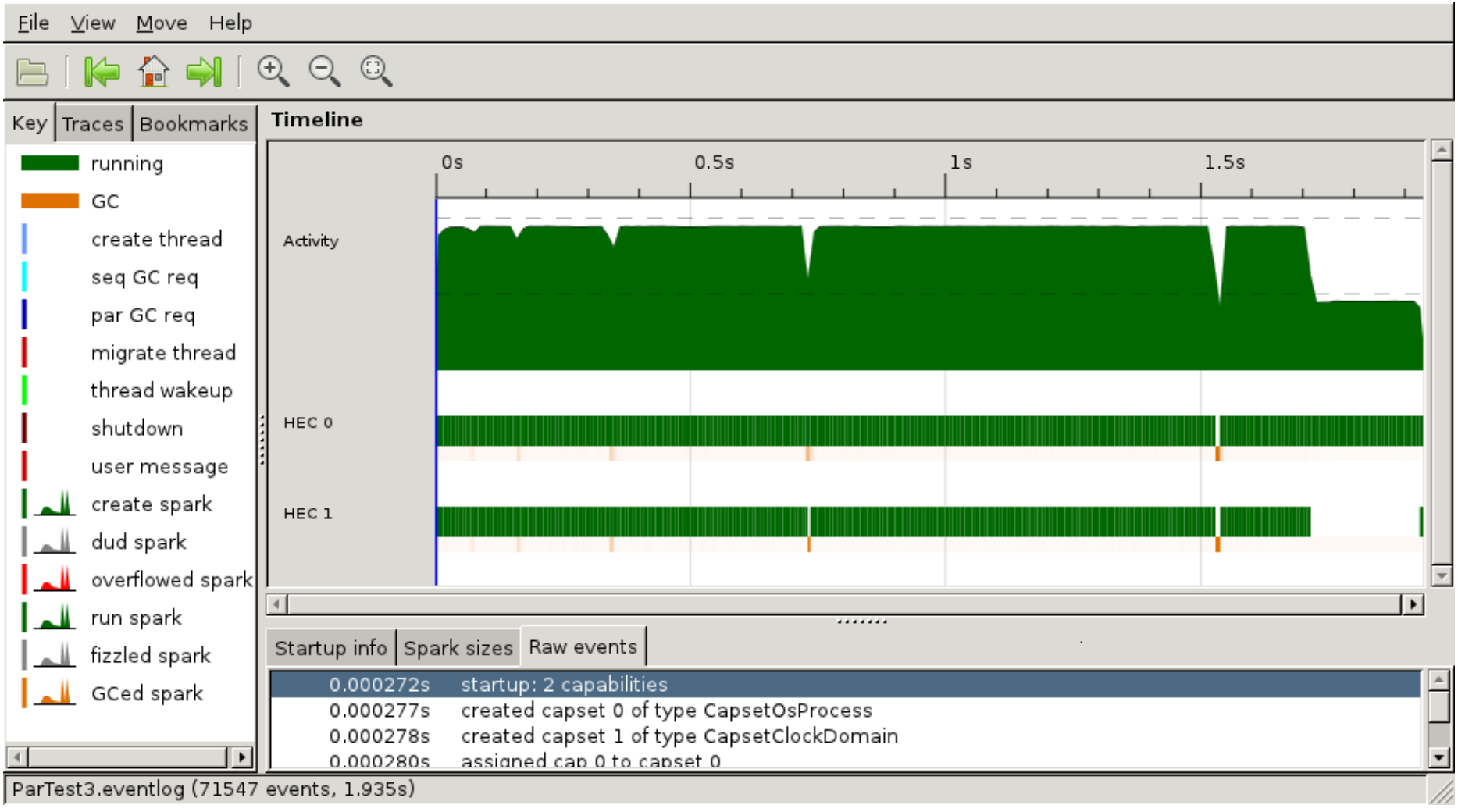
How can we fix it?

We must make sure that the work we intend to parallelize is actually performed within the parallel computation:

```
partest3 n =
  runEval $ do
    r1 ← rpar `dot` rdeepseq $ maxC 1       h
    r2 ← rpar `dot` rdeepseq $ maxC (h + 1) n
    let c (m1, s1) (m2, s2) = (m1 `max` m2, s1 + s2)
    return $ c r1 r2
  where
    h = n `div` 2
```

The `dot` function composes strategies:

```
dot :: Strategy a → Strategy a → Strategy a
s1 `dot` s2 = s1 ∘ runEval ∘ s2
```


File View Move Help
Key Traces Bookmarks
running
GC
create thread
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
user message
create spark
dud spark
overflowed spark
run spark
fizzled spark
GCed spark
Timeline
0s
0.5s
1s
1.5s
Activity
HEC 0
HEC 1
Startup info
Spark sizes
Raw events
0.000272s startup: 2 capabilities
0.000277s created capset 0 of type CapsetOsProcess
0.000278s created capset 1 of type CapsetClockDomain
0.000280s assigned cap 0 to capset 0
ParTest3.eventlog (71547 events, 1.935s)

Problem overview

Not enough parallelism:

- parallel tasks return partially evaluated expressions,
- parallel tasks are demanded by the main thread too soon,
- some tasks can't be interrupted while waiting for GC,
- parallel tasks are too large,
- too few sparks,
- too many sparks (overflow).

Too much overhead:

- memory requirements, leading to too many GCs,
- parallel tasks are too small (and usually too many),
- tasks duplicate work or perform work that is not needed,
- algorithms might become more complicated.

Current example

We statically partition the task in two subtasks:

- no further speedups for more than two cores,
- bad work distribution (although Collatz is relatively forgiving here).

Let's explore other, preferably more dynamic, partitioning options.

Strategies for lists

Our example program is a typical example of MapReduce. We map a function over a large list, and then reduce with an associative operator.

Strategies for lists

Our example program is a typical example of MapReduce. We map a function over a large list, and then reduce with an associative operator.

We can capture the idea of traversing a list in parallel:

```
evalList, parList :: Strategy a → Strategy [a]
evalList s []       = return []
evalList s (x : xs) = do
                        r  ← s x
                        rs ← evalList s xs
                        return (r : rs)
parList s = evalList (rpar `dot` s)
```

Strategies for lists

Our example program is a typical example of MapReduce. We map a function over a large list, and then reduce with an associative operator.

We can capture the idea of traversing a list in parallel:

```
evalList, parList :: Strategy a → Strategy [a]
evalList s []       = return []
evalList s (x : xs) = do
                        r  ← s x
                        rs ← evalList s xs
                        return (r : rs)
parList s = evalList (rpar `dot` s)
```

It is easy to define corresponding strategy combinators for other datatypes, or one can use `evalTraversable` / `parTraversable`.

Testing

```
partest4 :: Int → (Int, Int)
partest4 n = let cs = map (collatz ∘ fromIntegral) [1 . . n]
                          `using` parList rdeepseq
             in  (maximum cs, sum cs)
```

Testing

```
partest4 :: Int → (Int, Int)
partest4 n = let cs = map (collatz ∘ fromIntegral) [1 . . n]
                          `using` parList rdeepseq
             in  (maximum cs, sum cs)
```

The combinator `using` allows us to mostly separate programs from strategies:

```
using :: a → Strategy a → a
using x s = runEval (s x)
```

Results

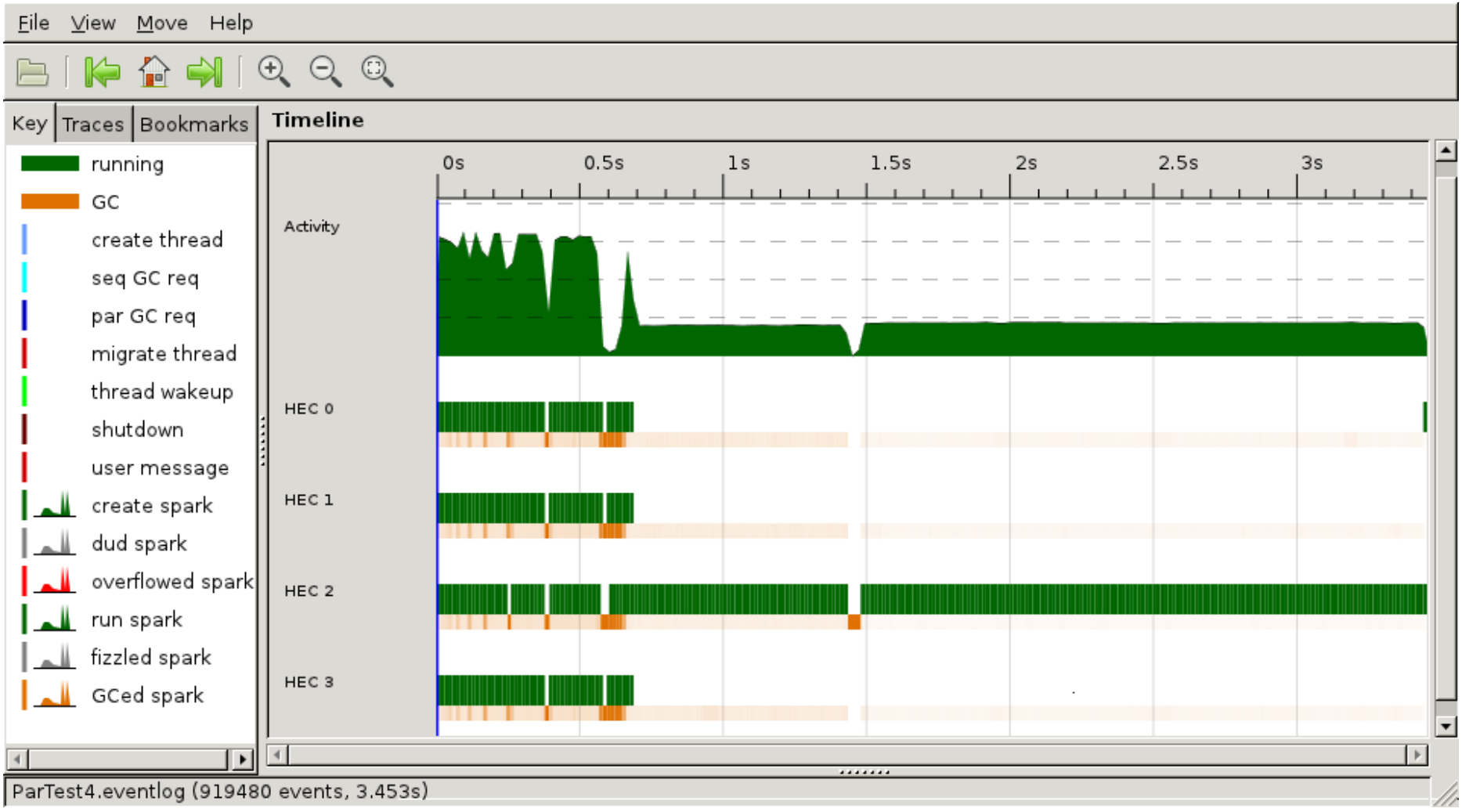
```
$ ./ParTest4 +RTS -N1 -s 2>&1 | grep Total
  Total   time    4.61s  (  4.38s elapsed)
$ ./ParTest4 +RTS -N2 -s 2>&1 | grep Total
  Total   time    6.68s  (  4.73s elapsed)
$ ./ParTest4 +RTS -N3 -s 2>&1 | grep Total
  Total   time   10.76s  (  4.62s elapsed)
$ ./ParTest4 +RTS -N4 -s 2>&1 | grep Total
  Total   time   15.44s  (  4.81s elapsed)
```

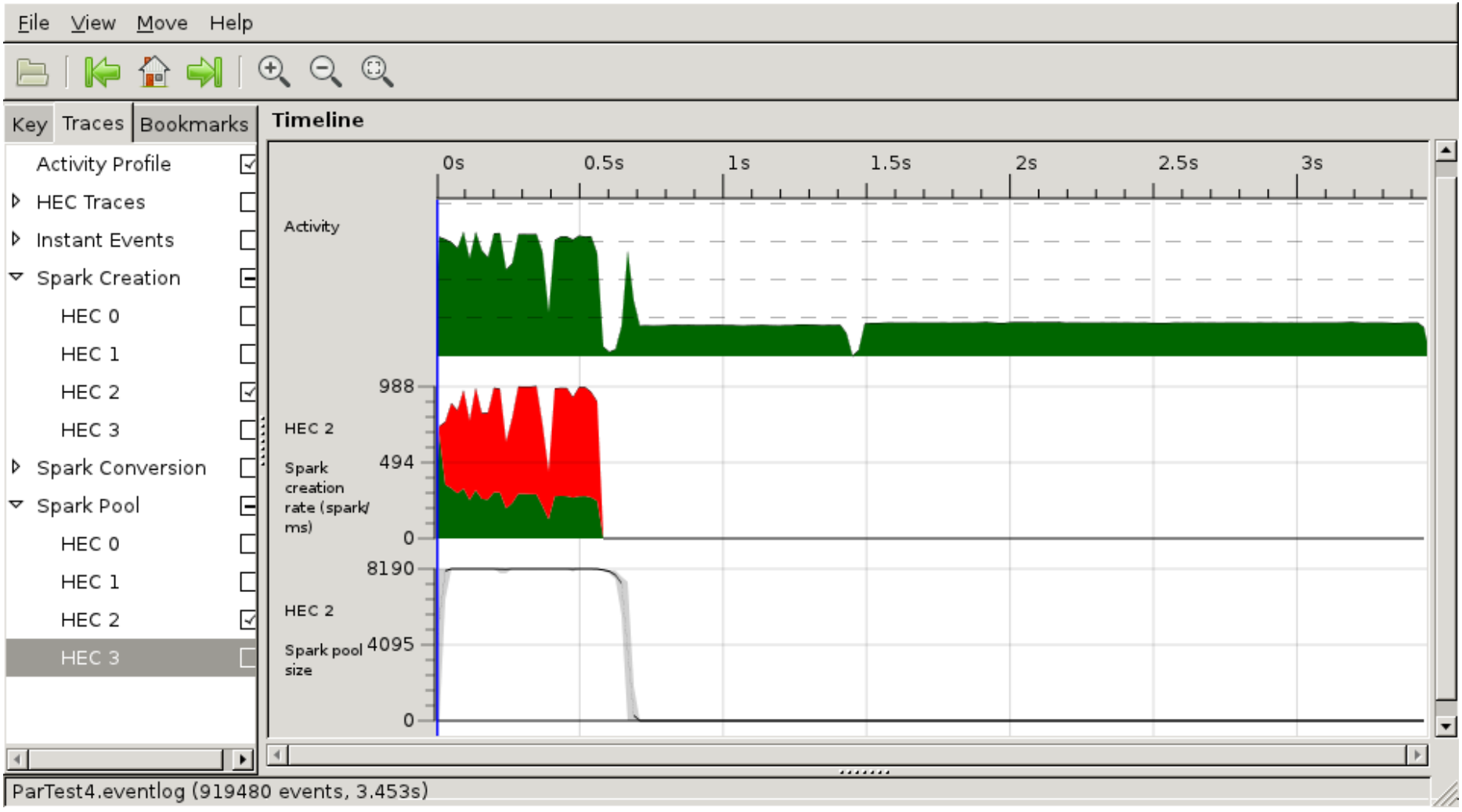
Results

```
$ ./ParTest4 +RTS -N1 -s 2>&1 | grep Total
  Total   time    4.61s  (  4.38s elapsed)
$ ./ParTest4 +RTS -N2 -s 2>&1 | grep Total
  Total   time    6.68s  (  4.73s elapsed)
$ ./ParTest4 +RTS -N3 -s 2>&1 | grep Total
  Total   time   10.76s  (  4.62s elapsed)
$ ./ParTest4 +RTS -N4 -s 2>&1 | grep Total
  Total   time   15.44s  (  4.81s elapsed)
```

```
$ ./ParTest4 +RTS -N4 -s 2>&1 | grep SPARKS
  SPARKS: 500000 (17743 converted, 482257 overflowed, ...)
```

A whole lot of sparks, but most of them overflow!


File View Move Help
Key Traces Bookmarks
running
GC
create thread
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
user message
create spark
dud spark
overflowed spark
run spark
fizzled spark
GCed spark
Timeline
0s
0.5s
1s
1.5s
2s
2.5s
3s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
ParTest4.eventlog (919480 events, 3.453s)


File View Move Help
Key Traces Bookmarks
Timeline
Activity Profile
HEC Traces
Instant Events
Spark Creation
HEC 0
HEC 1
HEC 2
HEC 3
Spark Conversion
Spark Pool
HEC 0
HEC 1
HEC 2
HEC 3
0s
0.5s
1s
1.5s
2s
2.5s
3s
Activity
988
494
0
HEC 2
Spark creation rate (spark/ms)
8190
4095
0
HEC 2
Spark pool size
ParTest4.eventlog (919480 events, 3.453s)

Analysis

```
parList :: Strategy a → Strategy [a]
parList s []       = return []
parList s (x : xs) = do
                       r  ← rpar ‘dot‘ s $ x
                       rs ← parList s xs
                       return (r : rs)
```

The `parList` strategy forces the entire spine of the list. It creates one spark per element.

Analysis

```
parList :: Strategy a → Strategy [a]
parList s []       = return []
parList s (x : xs) = do
                       r  ← rpar ‘dot‘ s $ x
                       rs ← parList s xs
                       return (r : rs)
```

The `parList` strategy forces the entire spine of the list. It creates one spark per element.

For long lists, these are too many and too small sparks in a too short amount of time.

Chunking

```
chunk :: Int → [a] → [[a]]
chunk n [ ]  = [ ]
chunk n xs = case splitAt n xs of
                    (ys, zs) → ys : chunk n zs
```

Chunking

```
chunk :: Int → [a] → [[a]]
chunk n [] = []
chunk n xs = case splitAt n xs of
                  (ys, zs) → ys : chunk n zs
```

```
parListChunk :: Int → Strategy a → Strategy [a]
parListChunk n s xs =
  do
    rss ← parList (evalList s) (chunk n xs)
    return (concat rss)
```

Only runs the chunks in parallel, thereby generating fewer sparks. Still forces the entire spine of the list.

General MapReduce abstraction

```
mapReduce :: Int →                -- threshold
             Int → Int →          -- bounds
             Strategy a →         -- strategy
             (Int → a) →          -- map
             ([a] → a) →          -- reduce
             a
mapReduce n lo hi s f c = runEval $ go lo hi
  where go lo hi | m < n      = rpar `dot` s $ c (map f [lo .. hi])
                 | otherwise = do
                                 r1 ← go lo          m2
                                 r2 ← go (m2 + 1) hi
                                 return $ c [r1, r2]
        where m  = hi − lo
              m2 = lo + m `div` 2
```

Having a threshold is important.

Applying the abstraction

```
combine (!m1, !s1) (!m2, !s2) = (m1 `max` m2, s1 + s2)

partest6 n = mapReduce threshold 1 n rdeepseq
                       ((λx → (x, x)) ∘ collatz ∘ fromIntegral)
                       (foldl1′ combine)

threshold = 1000
```

Adapting the threshold

The module `GHC.Conc` exports

```
numCapabilities :: Int
```

This is the number of capabilities the RTS has been started with (returns `1` in case of the non-threaded RTS).

Adapting the threshold

The module `GHC.Conc` exports

```
numCapabilities :: Int
```

This is the number of capabilities the RTS has been started with (returns `1` in case of the non-threaded RTS).

We can use it to automatically determine thresholds for parallelism:

```
autoMapReduce lo hi =
  mapReduce ((hi − lo) `div` (numCapabilities ∗ 5)) lo hi
```

Results

```
$ ./ParTest7 +RTS -N1 -s 2>&1 | grep Total
  Total   time    4.13s  (  4.13s elapsed)
$ ./ParTest7 +RTS -N2 -s 2>&1 | grep Total
  Total   time    4.30s  (  2.16s elapsed)
$ ./ParTest7 +RTS -N4 -s 2>&1 | grep Total
  Total   time    5.54s  (  1.30s elapsed)
$ ./ParTest7 +RTS -N8 -s 2>&1 | grep Total
  Total   time    6.80s  (  0.91s elapsed)
```

Lessons

- Most “problems” can be traced back to memory behaviour and / or evaluation order – understanding evaluation is the key to understanding parallel programming.
- Do not create too many sparks.
- Pay attention to the overhead of parallelisation: remember, the goal is speed.

Lessons

- Most “problems” can be traced back to memory behaviour and / or evaluation order – understanding evaluation is the key to understanding parallel programming.
- Do not create too many sparks.
- Pay attention to the overhead of parallelisation: remember, the goal is speed.
- Try to get rid of lists, and move towards proper tree traversals.
- Similarly, avoid linear traversals – but tree traversals are ok.
- Use the strengths of Haskell: abstract common patterns.
- Predefined strategies get you quite far, but many real-life problems require skeletons: predefined program patterns such as our `mapReduce`.
- Use ThreadScope and GHC’s RTS statistics for debugging.

The Par monad

The idea of `monad-par`

A more recent approach to deterministic parallel programming:

- an interface with explicit forking of subcomputations,
- communication via write-once variables ensured deterministic results,
- one can explicitly wait for results.

Interface

From `Control.Monad.Par`:

```
data Par a    -- abstract
instance Monad Par
data IVar a   -- abstract

spawn  :: NFData a ⇒ Par a → Par (IVar a)
spawnP :: NFData a ⇒ a → Par (IVar a)

get    :: IVar a → Par a

runPar :: Par a → a
```

Note that spawning causes evaluation to NF by default.
(There's a variant that does not force full evaluation.)

Static partitioning using the Par monad

```
partest1 n = runEval $ do
                         r1 ← rpar $ maxC 1       h
                         r2 ← rpar $ maxC (h + 1) n
                         return (r1 `max` r2)
  where
    h = n `div` 2
```

This is how we wrote the program using strategies.

Static partitioning using the Par monad

```
partest1 n = runPar $ do
                        v1 ← spawnP $ maxC 1         h
                        v2 ← spawnP $ maxC (h + 1) n
                        r1 ← get v1
                        r2 ← get v2
                        return (r1 `max` r2)
  where
    h = n `div` 2
```

This is the same in the Par monad. We explicitly extract the results from the IVar s.

Data dependency graphs can easily be transcribed into Par monad computations.

Dynamic partitioning using the Par monad

It is easy to write a parMap in the Par monad:

```
parMap :: (a → b) → [a] → Par [b]
parMap f xs = do
  vs ← mapM (spawnP ∘ f) xs
  mapM get vs
```

Dynamic partitioning using the Par monad

It is easy to write a parMap in the Par monad:

```
parMap :: (a → b) → [a] → Par [b]
parMap f xs = do
  vs ← mapM (spawnP ∘ f) xs
  mapM get vs
```

```
partest4 :: Int → (Int, Int)
partest4 n =
  let cs = runPar $ parMap (collatz ∘ fromIntegral) [1 . . n]
  in  (maximum cs, sum cs)
```

There is still a granularity problem that can – again – be fixed using chunking.

Parallel mapping in chunks

```
chunk :: Int → [a] → [[a]]    -- as before
parMapChunk :: Int → (a → b) → [a] → Par [b]
parMapChunk n f xs = concat <$> parMap (map f) (chunk n xs)
```

Parallel mapping in chunks

```
chunk :: Int → [a] → [[a]]    -- as before
parMapChunk :: Int → (a → b) → [a] → Par [b]
parMapChunk n f xs = concat <$> parMap (map f) (chunk n xs)
```

```
partest5 :: Int → (Int, Int)
partest5 n =
  let cs = runPar $
             parMapChunk (n `div` (10 * numCapabilities))
                         (collatz ∘ fromIntegral) [1 .. n])
  in (maximum cs, sum cs)
```

And again, it is also possible to define map-reduce as well as other high-level abstractions.

Concurrency

Forking lightweight threads

The central function to write concurrent programs is:

```
data ThreadId   -- abstract
forkIO → IO () → IO ThreadId
```

- defined in `Control.Concurrent`;
- the given computation is started in a separate thread;
- threads are lightweight Haskell threads;
- you get a `ThreadId` back that can be used to kill the thread or send exceptions to the thread.

Forking lightweight threads

The central function to write concurrent programs is:

```
data ThreadId   -- abstract
forkIO → IO () → IO ThreadId
```

- defined in `Control.Concurrent`;
- the given computation is started in a separate thread;
- threads are lightweight Haskell threads;
- you get a `ThreadId` back that can be used to kill the thread or send exceptions to the thread.

As with parallel programs, you want to make sure that concurrent programs are linked using the threaded run-time system via `-threaded`.

Interleaved execution

By using several threads, you write programs where several different sequences of `IO` actions are executed in an unspecified order:

```
thread :: Int → IO ()
thread n = forever $ print n

main :: IO ()
main = do
  mapM_ (forkIO ∘ thread) [1 .. 10]
  thread 0
```

Interleaved execution

By using several threads, you write programs where several different sequences of `IO` actions are executed in an unspecified order:

```
thread :: Int → IO ()
thread n = forever $ print n

main :: IO ()
main = do
  mapM_ (forkIO ∘ thread) [1 .. 10]
  thread 0
```

Note that all threads are killed if the main thread is killed.

Waiting for a while

We can also delay a thread:

```
second :: Int
second = 1000000    -- delays measured in microseconds

thread :: Int → IO ()
thread n = forever $ print n ≫ threadDelay (second `div` 10)

main :: IO ()
main = do
  mapM_ (forkIO ∘ thread) [1 .. 10]
  threadDelay (30 ∗ second)
```

Communicating via IORefs is dangerous

```
thread :: IORef Int → Int → IO ()
thread var n = forever $ do
  writeIORef var n
  x ← readIORef var
  when (x ≠ n) $ print (x, n)
main :: IO ()
main = do
  var ← newIORef 0
  mapM_ (forkIO ∘ thread var) [1 . . 10]
  threadDelay (15 ∗ second)
```

At least when run with -N2 or higher, this will produce some output.

Using MVars for communication

An MVar is a variation of an IORef defined by Control.Concurrent, specifically designed for synchronized access:

```
data MVar a    -- abstract

newEmptyMVar :: IO (MVar a)          -- after: empty
newMVar      :: a → IO (MVar a)      -- after: full

putMVar  :: MVar a → a → IO ()       -- before: empty, after: full
readMVar :: MVar a → IO a            -- before: full, after: full
takeMVar :: MVar a → IO a            -- before: full, after: empty
```

- An MVar is either empty or full.
- If a read/write operation expects a different status than the MVar is in, then the thread blocks until the status changes.

An MVar-based version of the example

```
thread :: IORef Int → Int → IO ()
thread var n = forever $ do
  writeIORef var n
  x ← readIORef var
  when (x ≠ n) $ print (x, n)

main :: IO ()
main = do
  var ← newIORef 0
  mapM_ (forkIO ∘ thread var) [1 . . 10]
  threadDelay (15 ∗ second)
```

Waiting for threads

Haskell threads are deliberately lightweight, so there's no built-in way to query the status of a thread and determine if it is still running.

```
thread :: MVar () → Int → IO ()
thread finished n = do
  replicateM_ 50 (print n)
  putMVar finished ()

main :: IO ()
main = do
  let nThreads = 10
  vars ← replicateM_ nThreads newEmptyMVar
  zipWithM_ (λv i → forkIO (thread v i)) vars [1 . . nThreads]
  mapM_ takeMVar vars
```

The final `mapM_` waits for all threads to finish now.

Thread managers

Based on this idea, one can implement thread systems with slightly more information available.

Inspired by Real World Haskell, package `threadmanager`, `Control.Concurrent.ThreadManager`:

```
data ThreadManager    -- abstract
data ThreadStatus =
   Running | Finished | Threw SomeException
make      :: IO ThreadManager
fork      :: ThreadManager → IO () → IO ThreadId
getStatus :: ThreadManager → ThreadId
                 → IO (Maybe ThreadStatus)
waitFor   :: ThreadManager → ThreadId
                 → IO (Maybe ThreadStatus)
```

FIFO channels

On top of MVar s, it is possible to build other concurrency abstractions, such as FIFO channels:

```
data Chan a    -- abstract

newChan  :: IO (Chan a)
writeChan :: Chan a → a → IO ()
readChan :: Chan a → IO a
```

- Reading from an empty channel blocks until contents are available.
- Typical uses include distributing work to several threads, or collecting results from several threads.
- Note that channels are typed, and all channel entries have the same type.

Simple networking

Network interface

Provided by `network` package, module `Network`:

```
type HostName = String
data PortID =
   Service String | PortNumber PortNumber | UnixSocket String
data Socket    -- abstract

withSocketsDo :: IO a → IO a    -- initialization

   -- server
listenOn :: PortID → IO Socket
accept   :: Socket → IO (Handle, HostName, PortNumber)

   -- client
connectTo ::    HostName → PortID → IO Handle
```

A “shouting” server

```
main :: IO ()
main = withSocketsDo $ do
    s ← listenOn (PortNumber 8765)
    forever $ do
        (h, _, _) ← accept s
        forkIO $ handleClient h

handleClient :: Handle → IO ()
handleClient h = do
    hSetBuffering h LineBuffering
    forever $ do
        line ← hGetLine h
        hPutStrLn h (map toUpper line)
```

A “shouting” server

```
main :: IO ()
main = withSocketsDo $ do
    s ← listenOn (PortNumber 8765)
    forever $ do
        (h, _, _) ← accept s
        forkIO $ handleClient h
handleClient :: Handle → IO ()
handleClient h = do
    hSetBuffering h LineBuffering
    forever $ do
        line ← hGetLine h
        hPutStrLn h (map toUpper line)
```

Typical server pattern: endless `accept` loop, fork a new thread for every client.

A corresponding client

```
main :: IO ()
main = withSocketsDo $ do
  h ← connectTo "localhost" (PortNumber 8765)
  hSetBuffering h LineBuffering
  forkIO $ copyByLine h stdout    -- receiving lines
  copyByLine stdin h    -- sending lines

copyByLine :: Handle → Handle → IO ()
copyByLine from to = forever $ do
  line ← hGetLine from
  hPutStrLn to line
```

Software Transactional Memory

A lock-free approach to concurrency

Haskell's `stm` package offers an appealing approach to concurrency:

- transactions are guaranteed to be run atomically;
- the type system guarantees that transactions can be safely restarted;
- there are no locks, hence no danger of deadlocks;
- transactional computations are more easy to compose than computations based on `MVar`s and locks.

Control.Concurrent.STM interface

```
data STM a    -- abstract
instance Monad STM
data TVar a   -- abstract

   -- transactional variables
newTVar    :: a → STM (TVar a)
newTVarIO  :: a → IO (TVar a)
readTVar   :: TVar a → STM a
writeTVar  :: TVar a → a → STM ()

   -- running a transaction
atomically :: STM a → IO a
```

Note that `STM` is a restricted `IO` monad.

Classic example: transfer money

Library helper function:

```
modifyTVar :: TVar a → (a → a) → STM ()
modifyTVar var f = do
  x ← readTVar var
  writeTVar (f x)
```

Transfer function:

```
transfer :: Num a ⇒ TVar a → TVar a → a → STM ()
transfer from to amount = do
  modifyTVar from (λx → x − amount)
  modifyTVar to   (λx → x + amount)
```

Stress-testing the example

```
main :: IO ()
main = do
  manager ← make
  accs ← mapM newTVarIO [1000, 2500]
  printTotal accs
  replicateM_ 1000 (fork manager (randomTransfer accs))
  waitForAll manager
  printTotal accs
```

We use a thread manager here to wait for all forked threads.

Stress-testing the example (contd.)

```
printTotal :: [TVar Integer] → IO ()
printTotal accs =
  print (sum <$> atomically (mapM readTVar accs))
```

```
randomTransfer :: [TVar Integer] → IO ()
randomTransfer xs = do
  let maxIndex = length xs − 1
  from   ← randomRIO (0, maxIndex)
  to     ← randomRIO (0, maxIndex)
  amount ← randomRIO (− 100, 100)
  atomically $ transfer (xs !! from) (xs !! to) amount
```

Associating IO with transactions

We cannot do IO within a transaction, but we can perform IO after a transaction:

- Compute the data necessary to perform the IO within the transaction and return that from the transaction.

Associating IO with transactions

We cannot do IO within a transaction, but we can perform IO after a transaction:

- Compute the data necessary to perform the IO within the transaction and return that from the transaction.
- Because `IO` is first-class data in Haskell, we can even compute the action itself.

Example

```
transfer :: Num a ⇒ TVar a → TVar a → a → STM (IO ())
transfer from to amount = do
    current ← readTVar from
    if current < amount
       then return $ putStrLn "not ok"
       else do
          modifyTVar from (λx → x − amount)
          modifyTVar to   (λx → x + amount)
          return $ putStrLn $ "ok: " ++ show amount
```

Note that `return` is used on something of type `IO ()` here.

Example (contd.)

```
randomTransfer :: [TVar Integer] → IO ()
randomTransfer xs = do
  let maxIndex = length xs − 1
  from   ← randomRIO (0, maxIndex)
  to     ← randomRIO (0, maxIndex)
  amount ← randomRIO (− 100, 100)
  log    ← atomically $ transfer (xs !! from) (xs !! to) amount
  log  -- execute the logging action after the transaction
```

Retrying or combining transactions

```
retry    :: STM a
orElse :: STM a → STM a → STM a
```

Retrying or combining transactions

```
retry    :: STM a
orElse :: STM a → STM a → STM a
```

Example:

```
transfer :: Num a ⇒ TVar a → TVar a → a → STM (IO ())
transfer from to amount = do
    current ← readTVar from
    when (current < amount) retry
    modifyTVar from (λx → x − amount)
    modifyTVar to     (λx → x + amount)
    return $ putStrLn $ "ok:   " ++ show amount
```

Retrying or combining transactions

```
retry    :: STM a
orElse :: STM a → STM a → STM a
```

Example:

```
transfer :: Num a ⇒ TVar a → TVar a → a → STM (IO ())
transfer from to amount = do
  current ← readTVar from
  when (current < amount) retry
  modifyTVar from (λx → x − amount)
  modifyTVar to     (λx → x + amount)
  return $ putStrLn $ "ok:  " ++ show amount
```

A `retry` does not actually rerun the transaction unless some of the inputs have changed.

An `orElse` tries the second computation only if the first retries.