

# **Tutorial: Deterministic Parallel Programming in Haskell**

Andres Löh  
Well-Typed LLP

Hal6

7 October 2011

# Before we start ...

Get the bundle for this tutorial:

```
http://www.well-typed.com/Hal6/Hal6.zip
```

Contains:

- ▶ these slides
- ▶ code examples
- ▶ exercises

# 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.

# Why 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.

# Why 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 . . . ?

# Why 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?

# Why 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 (possibly with stm).

# Why so many approaches?

- ▶ Parallelism is “hot”.
- ▶ Parallelising programs (even explicitly) is not trivial.

# Why so many approaches?

- ▶ Parallelism is “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 is “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).

# This tutorial

Given the lack of time, we have to limit ourselves, and will focus on **strategies**.

# This tutorial

Given the lack of time, we have to limit ourselves, and will focus on **strategies**.

However, the other approaches are interesting as well.

# This tutorial

Given the lack of time, we have to limit ourselves, and will focus on **strategies**.

However, the other approaches are interesting as well.

Many of the general concepts and problems are inherent to parallelism as a whole, so quite a bit of knowledge can be transferred to other approaches.

# 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

```
r0      :: Strategy a      -- evaluation:
rseq    :: Strategy a      -- none
rdeepseq :: NFData a => Strategy a -- WHNF
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.

# Excursion: Weak Head Normal Form

## Weak head normal form (WHNF)

An term is said to be in **weak head normal form** if the head position (i.e., the outermost syntactic construct) cannot be reduced.

# Excursion: Weak Head Normal Form

## Weak head normal form (WHNF)

An term is said to be in **weak head normal form** if the head position (i.e., the outermost syntactic construct) cannot be reduced.

This means in practice:

- ▶ literals are in weak head normal form,
- ▶ lambda expressions (and partially applied functions) are in weak head normal form,
- ▶ constructor applications are in weak head normal form.

# WHNF in Haskell

Reducing to WHNF is the “smallest” amount of reduction necessary:

- ▶ for a function before we can meaningfully apply it in any way,
- ▶ for a datatype member to make a case-distinction based on the top-level constructor.

# 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 ()
```

## Demo and exercise

Try to distinguish the values from the previous slides using suitable strategies.

# Running example

## 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

## 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

## 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 --make -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 --make -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.
- ls Create an event log.

# Speedup!

$$\text{speedup } n = \text{sequentialTime} / \text{parallelTime } n$$

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):

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

```
partest2 n =
  runEval $ do
    r1 ← rpar $ maxCP 1      h
    r2 ← rpar $ maxCP (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).

# 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

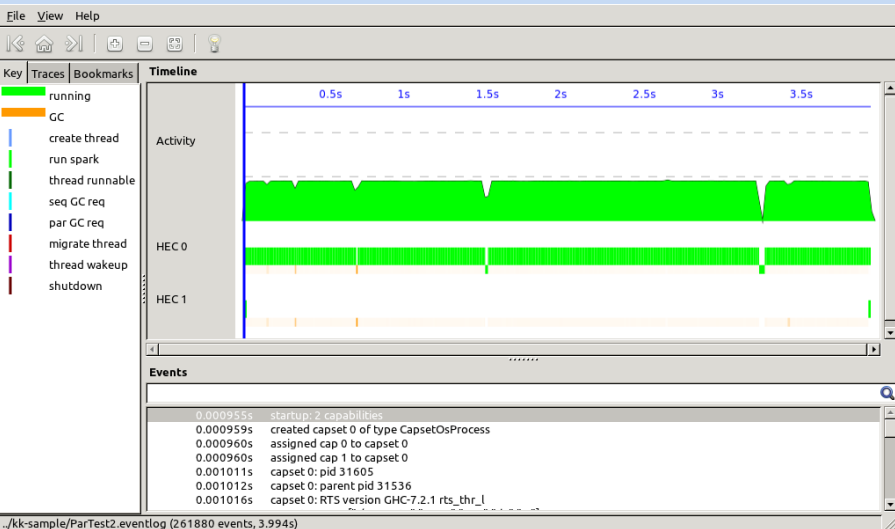
ThreadScope is a graphical debugging tool that visualizes event logs that can be generated from Haskell program runs:

- ▶ 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
```

Make sure you get `threadscope-0.2.0` or later.



## So what happened?

We are calling

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

but

```
maxCP 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 $ maxCP 1      h
r2 ← rpar $ maxCP (h + 1) n
```

but

```
maxCP 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 $ maxCP 1      h  
    r2 ← rpar 'dot' rdeepseq $ maxCP (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 $ maxCP 1      h  
    r2 ← rpar 'dot' rdeepseq $ maxCP (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` combinator composes strategies:

```
dot :: Strategy a → Strategy a → Strategy a  
s1 'dot' s2 = s1 ◦ runEval ◦ s2
```

## Normal form

Unlike `rseq`, the `rdeepseq` combinator evaluates a term completely, to **normal form**.

Whereas `rseq` is primitive (or built on top of the internal primitive `pseq`), `rdeepseq` is not.

```
class NFData a where
```

```
  rnf :: a → ()
```

## Normal form

Unlike `rseq`, the `rdeepseq` combinator evaluates a term completely, to **normal form**.

Whereas `rseq` is primitive (or built on top of the internal primitive `pseq`), `rdeepseq` is not.

```
class NFData a where
```

```
  rnf :: a → ()
```

```
rdeepseq :: NFData a ⇒ Strategy a
```

```
rdeepseq x = do
```

```
  () ← rseq (rnf x)
```

```
  return x
```

There are `NFData` instances for all standard datatypes.

# Exercise

## Exercise

Try to define your own `NFData` instances for a few simple datatypes, such as pairs and lists and binary trees.

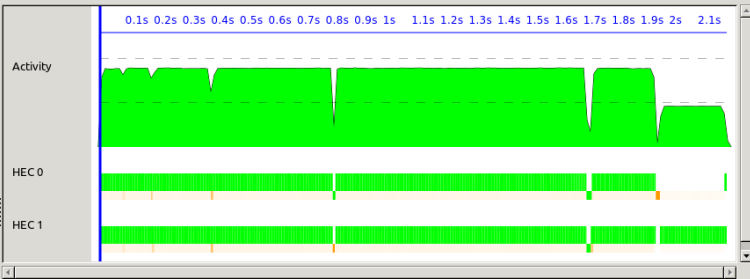
**Hint:** you can reuse the `Eval` monad (even though the standard definitions do not).



Key Traces Bookmarks

- █ running
- █ GC
- | create thread
- | run spark
- | thread runnable
- | seq GC req
- | par GC req
- | migrate thread
- | thread wakeup
- | shutdown

Timeline



Events

```

0.000826s startup: 2 capabilities
0.000830s created capset 0 of type CapsetOsProcess
0.000830s assigned cap 0 to capset 0
0.000831s assigned cap 1 to capset 0
0.000872s capset 0: pid 31677
0.000873s capset 0: parent pid 31536
0.000877s capset 0: RTS version GHC-7.2.1 rts_thr_l

```

# 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!

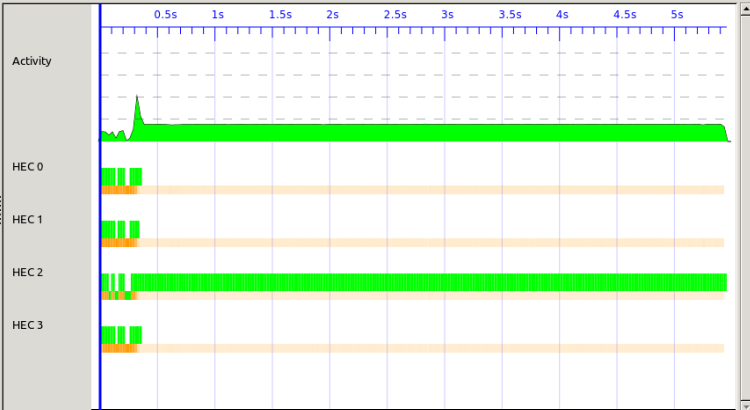
File View Help



Key Traces Bookmarks

Timeline

- running
- GC
- create thread
- run spark
- thread runnable
- seq GC req
- par GC req
- migrate thread
- thread wakeup
- shutdown



/home/andres/well-typed/HaL/kk-sample/ParTest4.eventlog (628722 events, 5.456s)

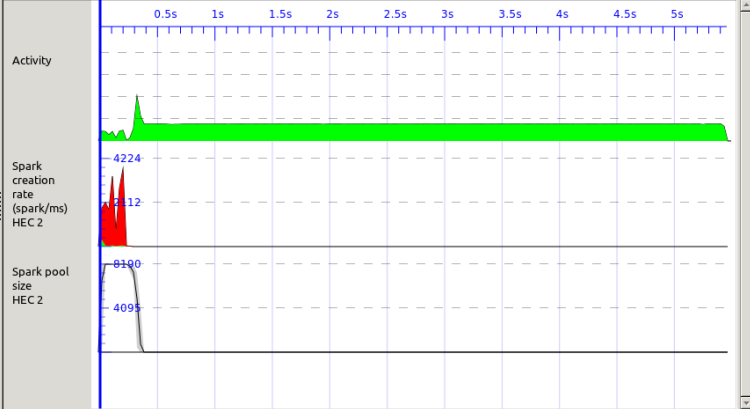
File View Help



Key Traces Bookmarks

Timeline

- Activity Profile
- HEC Traces
- Spark Creation 
  - HEC 0
  - HEC 1
  - HEC 2
  - HEC 3
- Spark Conversion 
  - HEC 0
  - HEC 1
  - HEC 2
  - HEC 3
- Spark Pool 
  - HEC 0
  - HEC 1
  - HEC 2
  - HEC 3



/home/andres/well-typed/HaL/kk-sample/ParTest4.eventlog (628722 events, 5.456s)

# 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)
```

# Summary

- ▶ 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.

# Summary

- ▶ 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.

## Exercise: n-queens

### Problem

Try to place  $n$  queens on a chessboard of size  $n$  such that all queens are safe from each other.

This is a constraint satisfaction problem that is reasonably simple to solve in Haskell using a naive brute-force approach.

# Sequential solution in Haskell

(This solution is adapted from Simon Marlow's parallel programming benchmark.)

A solution is a list of row numbers, one per column:

```
type PartialSolution = [Int]
type Solution = PartialSolution
type BoardSize = Int
```

# Sequential solution in Haskell

(This solution is adapted from Simon Marlow's parallel programming benchmark.)

A solution is a list of row numbers, one per column:

```
type PartialSolution = [Int]
type Solution = PartialSolution
type BoardSize = Int
```

We iteratively try to add `n` queens to the initially empty solution:

```
queens :: BoardSize → [Solution]
queens n = iterate (concatMap (addQueen n)) [[]] !! n
```

Ultimately, we're interested in the number of solutions for a given size.

## Sequential solution in Haskell (contd.)

We try to extend a given partial solution by trying to place a queen in the next column in all safe rows:

```
addQueen :: BoardSize → PartialSolution → [PartialSolution]
addQueen n s = [x : s | x ← [1..n], safe x s 1]
```

## Sequential solution in Haskell (contd.)

We try to extend a given partial solution by trying to place a queen in the next column in all safe rows:

```
addQueen :: BoardSize → PartialSolution → [PartialSolution]
addQueen n s = [x : s | x ← [1..n], safe x s 1]
```

We still have to define when a position is safe:

```
safe :: Int → PartialSolution → Int → Bool
safe x []      n = True
safe x (c : y) n =
  x ≠ c && x ≠ c + n && x ≠ c - n && safe x y (n + 1)
```

## Where to continue from here

Check out Simon Marlow's tutorials:

```
https://github.com/simonmar/par-tutorial.git
```

Contains code and 70-page CEFP summer school lecture notes.

```
http://community.haskell.org/~simonmar/slides/CUFP.pdf
```

The recent CUFP talk on the `Par` monad.

Also make sure to check out Kevin Hammond et al.'s forthcoming book on parallel programming in Haskell.