An Introduction to (Deterministic) Parallelism in Haskell

Munich Lambda Meetup

Andres Löh

17 February 2014 —

Well-Typed
The Haskell Consultants

Overview

- Parallelism and Concurrency
- A first example
- Haskell and Effects
- Writing parallel programs
- (N Queens)
- Conclusions

About me

- PhD (Utrecht University) 2004 on “Generic Haskell”
- Lecturer at Utrecht University 2007–2010
- Partner at Well-Typed 2010–

About Well-Typed

- Founded 1998.
- Haskell consulting (development, advice, support, training).
- Currently $\sim$7 people working full-time in various places.
- Clients mainly in Europe and USA (most work done remotely).
- Also helped to set up the Industrial Haskell Group.

Parallelism and Concurrency

Parallelism and Concurrency

Parallelism

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

Concurrency

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

Concurrency for Parallelism?

Using concurrency to implement parallelism is quite common, but not necessarily a good idea:

- reasoning about threads is difficult,
- communication between threads,
- exceptions,
- potential deadlocks and race conditions.

Often, code we want to parallelise is pure – it involves no side effects at all. So why introduce them just for parallelism?

Automatic parallelism?

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

Automatic parallelism?

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!

Automatic parallelism?

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.

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

A few deterministic approaches:

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

The Haskell landscape

A few deterministic approaches:

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

A few non-deterministic approaches:

- concurrency primitives (forkIO , . . .),
- dataflow with side effects (`monad-par`),
- asynchronous computations (`async`),
- Cloud Haskell (`distributed-process`).

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`).

Today

Given the lack of time, we have to limit ourselves, and will focus on the `Par` monad.

A first example

Example

```
collatz :: Integer → Integer
collatz n
    | even n = n ‘div‘ 2
    | odd n = 3 ∗ n + 1
collatzSeq :: Integer → [Integer]
collatzSeq = takeWhile (>1) . iterate collatz
collatzSteps :: [Int]
collatzSteps = map (length . collatzSeq) [1 . .]
```

Example

```
collatz :: Integer → Integer
collatz n
    | even n = n ‘div‘ 2
    | odd n = 3 ∗ n + 1
collatzSeq :: Integer → [Integer]
collatzSeq = takeWhile (>1) . iterate collatz
collatzSteps :: [Int]
collatzSteps = map (length . collatzSeq) [1 ..]
```

```
GHCi> collatzSeq 9
[9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2]
GHCi> take 10 collatzSteps
[0, 1, 7, 2, 5, 8, 16, 3, 19, 6]
```

Let’s find the maximum number of steps in a given range.

Parallelisation

```
collatzMax :: Integer → Integer → Int
collatzMax lo hi = maximum (map (length . collatzSeq) [lo . . hi])
```

Parallelisation

```
collatzMax :: Integer → Integer → Int
collatzMax lo hi = maximum (map (length . collatzSeq) [lo . . hi])
```

Binary division:

```
parCollatzMax :: Integer → Integer → Int
parCollatzMax lo hi = runPar $
    do
      r1 ← spawnP (collatzMax lo mi)
      r2 ← spawnP (collatzMax (mi + 1) hi)
      m1 ← get r1
      m2 ← get r2
      return (max m1 m2)
  where
    mi = (lo + hi) ‘div‘ 2
```

Compilation and running

Compile with:

```
$ ghc -O2 -threaded -rtsopts Collatz
```

Run with:

```
$ ./Collatz +RTS -N -s
```

Compared with a plain implementation provides modest speedup.

Flag explanation

Compiler flags:

- `-O2` enables optimisation
- `-threaded` links in the threaded run-time system
- `-rtsopts` allows configuration of run-time system at run time
- `-eventlog` allows eventlog generation for debugging

Run-time system flags:

- `-N` runs on all available cores
- `-s` produces run-time statistics
- `-l` generates an eventlog for debugging

Well-Typed

Haskell and Effects

Effects

Java/C-like

```
int add0 (int x, int y) {
  return x + y;
}
```

Effects

Java/C-like

```
int add0 (int x, int y) {
   return x + y;
}


int add1 (int x, int y) {
   launch_missiles (now);
   return x + y;
}
```

Well-Typed

Effects

Java/C-like

```
int add0 (int x, int y) {
   return x + y;
}


int add1 (int x, int y) {
   launch_missiles (now);
   return x + y;
}
```

Both functions have the same type!

Effects

Haskell

```
add0 :: Int → Int → Int
add0 x y = x + y
add1 :: Int → Int → IO Int
add1 x y = do
  launch_missiles
  return (x + y)
```

Effectful computations are tagged by the type system!

Effects in Haskell's types

We have rather fine-grained control about effects just by looking at the types:

	A	some type, no effect
IO	A	IO, exceptions, random numbers, concurrency, …
Gen	A	random numbers only
ST s	A	mutable variables only
STM	A	software transactional memory log variables only
State s	A	(persistent) state only
Error	A	exceptions only
Signal	A	time-changing value

- All effect types share a common interface (monad; allows sequencing of operations and **do** notation).
- New effect types can be defined. Effects can be combined.

More on **do** notation

```
interaction :: IO String
interaction = do
  putStrLn "Who are you?"
  name ← getLine
  putStrLn ("Hello, " ⧺ name ⧺ "!")
  return name
```

More on **do** notation

```
interaction :: IO String
interaction = do
   putStrLn "Who are you?"
   name ← getLine
   putStrLn ("Hello, " ⧺ name ⧺ "!")
   return name
```

Look at the types:

```
putStrLn :: String → IO ()
getLine  :: IO String
return   :: a → IO a
```

More on **do** notation

```
interaction :: IO String
interaction = do
  putStrLn "Who are you?"
  name ← getLine
  putStrLn ("Hello, " ⧺ name ⧺ "!")
  return name
```

Look at the types:

```
putStrLn :: String → IO ()
getLine  :: IO String
return   :: a → IO a
```

```
(≫=) :: IO a → (a → IO b) → IO b
```

No escape from IO

There's no function of type:

`IO a → a`

Example:

- An `Int` is a constant integer.
- An `IO Int` is an IO action yielding an integer.
- We shouldn't be able to forget about the potential side effects.

No escape from `IO`

There's no function of type:

```
IO a → a
```

Example:

- An `Int` is a constant integer.
- An `IO Int` is an IO action yielding an integer.
- We shouldn't be able to forget about the potential side effects.

```
unsafePerformIO :: IO a → a
```

The Par monad

- very limited interface
- carefully designed to guarantee deterministic results

The Par monad

- very limited interface
- carefully designed to guarantee deterministic results

$$\text{runPar} :: \text{Par a} \to \text{a}$$

- create an annotated parallel computation of type `Par a`
- run it with `runPar`
- obtain a deterministic result of type `a`

Writing parallel programs

Back to our example

```
parCollatzMax :: Integer → Integer → Int
parCollatzMax lo hi = runPar $
    do
      r1 ← spawnP (collatzMax lo mi)
      r2 ← spawnP (collatzMax (mi + 1) hi)
      m1 ← get r1
      m2 ← get r2
      return (max m1 m2)
  where
    mi = (lo + hi) `div` 2
```

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,
- reading a variable blocks until the result is available.

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
```

Let's ignore `NFData` for a moment.

More functions

```
new  :: Par (IVar a)
put  :: NFData a ⇒ IVar a → a → Par ()
get  :: IVar a → Par a
fork :: Par () → Par ()
```

- The functions `spawn` and `spawnP` can be implemented in terms of the functions above.
- Writing twice to an `IVar` is an error.

Why `NFData`?

Haskell is, by default, not strict:

- data is stored in unevaluated form unless demanded;
- storing data in a variable does not normally force it.

Why NFData?

Haskell is, by default, not strict:

- data is stored in unevaluated form unless demanded;
- storing data in a variable does not normally force it.

Here, we want to make sure that the result is fully computed before it is communicated to the consuming computation:

- the `NFData` type class contains functions for fully evaluating terms of a given type;
- it is used in `spawn` and `put` to make sure that results are fully evaluated before they're written to a write-once variable.

Static vs. dynamic partitioning

Static partitioning is bad:

- fixed number of tasks, limited speedup on many cores;
- difficult to balance the load;
- difficult to control granularity.

Let’s create parallel tasks depending on the problem size.

Parallel map

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

Parallel map

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

```
mapM :: (a → Par b) → [a] → Par [b]
mapM f [ ]        = return [ ]
mapM f (x : xs) = do
   r   ← f x
   rs ← mapM f xs
   return (r : rs)
```

Using parMap

Sequential version:

```
collatzMax :: Integer → Integer → Int
collatzMax lo hi = maximum (map (length . collatzSeq) [lo .. hi])
```

Parallel version:

```
parCollatzMax :: Integer → Integer → Int
parCollatzMax lo hi =
   maximum (runPar (parMap (length . collatzSeq) [lo .. hi]))
```

Chunking

Spawning a computation for each list element causes a granularity problem:

- too many too small computations are spawned too fast;
- we still get some speedup, but not as much as we'd like.

Chunking

Spawning a computation for each list element causes a granularity problem:

- too many too small computations are spawned too fast;
- we still get some speedup, but not as much as we'd like.

A common solution is to chunk the list:

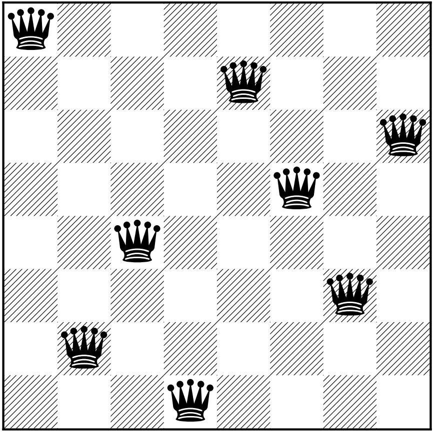
```
type ChunkSize = Int

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

Using chunking

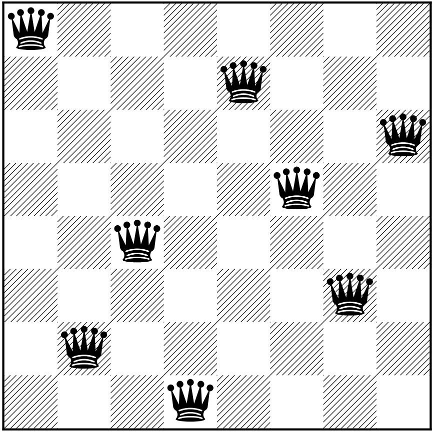
```
parCollatzMax :: ChunkSize → Integer → Integer → Int
parCollatzMax cs lo hi =
   maximum (
      concat (
         runPar (
            parMap
               (map (length . collatzSeq))
               (chunk cs [lo . . hi])
            )
         )
      )
```

N Queens

The N Queens problem



The N Queens problem

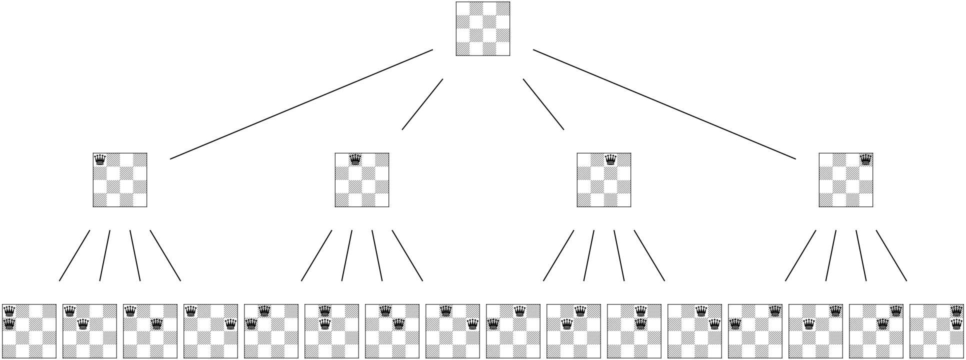


How many solutions for a given board size?

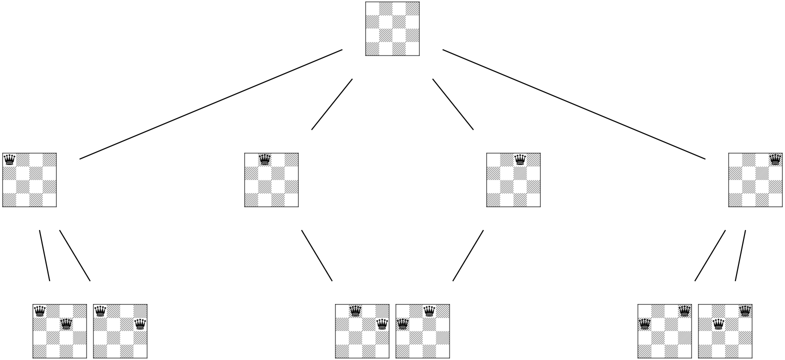
Idea

- Pick queens row by row.
- Generate a tree of all possible choices.
- Remove illegal choices from the tree.
- Traverse the tree, counting the number of valid solutions.

Generate, filter, explore



Generate, filter, explore



Demo

Conclusions

What have we learned

- Annotating a program for parallelisation is (relatively) easy.
- We can build domain-specific abstractions such as `parMap`.
- Deterministic results are guaranteed – no deadlocks, no race conditions.
- We can focus on achieving speedup.

Related approaches

- The `ParIO` monad combines `IO` with `Par` – at the price of determinism.
- The `Async` monad is similar to `IO`, but for concurrent applications.

Learn more

- Time for some exercises now.
- Lots of online material.
- Simon Marlow's book.

Exercises

Exercises

- Try to write twice to a single `IVar`.
- Reproduce the Collatz example.
- Replace the Collatz function by the Fibonacci function – what changes?
- Try to abstract and define a function

```
parMapChunked :: NFData b ⇒
  ChunkSize → (a → b) → [a] → Par [b]
```

- Try to abstract and define a “skeleton” for map-reduce.
- Reproduce the N Queens example.
- From Simon Marlow’s materials: try Sudoku solving, k-means, conference timetable scheduling; all using the `Par` monad.