Beautiful Parallelism

Harnessing Multicores with Haskell

Don Stewart | dons@galois.com

|galois|

Haskell for Multicores

What the functional programming community doing:

Lots of fine grained parallelism via annotations
Fast and cheap threads
Statelessness by default
Even cheaper thread *sparks (*lazy futures*)*
Communication via full/empty *MVars*
Or software transactional memory
Nested data parallelism

Open source, with a big community and lots of libraries

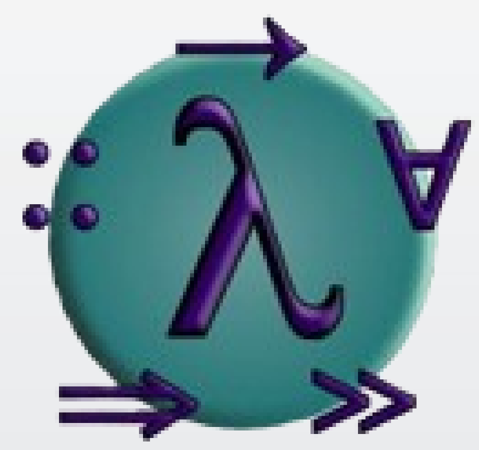
haskell.org

2008 Galois, Inc. All rights reserved.

The Product of the Haskell Community

This talk made possible by:

Simon Peyton Jones
Satnam Singh
Manuel Chakravarty
Gabriele Keller
Roman Leschinkskiy
Simon Marlow
Tim Harris
Phil Trinder
Kevin Hammond
Martin Sulzmann



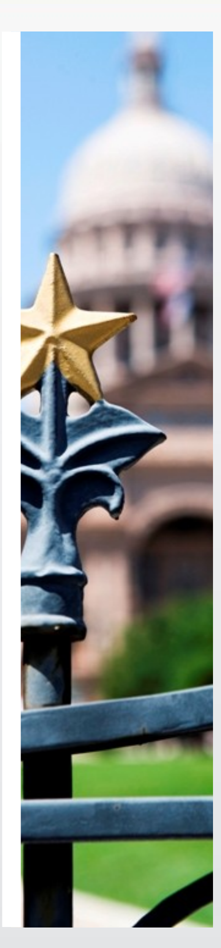
Read their papers or visit haskell.org for the full story!

2008 Galois, Inc. All rights reserved.

Galois and Haskell

Compiler and language engineering
Formal methods
High assurance systems
High performance cryptography
Parallel languages for FPGAs
Domain-specific languages

And using Haskell since inception



2008 Galois, Inc. All rights reserved.

What flavor is Haskell

- A functional programming language
- High level language: not tied to any execution model
- Statically typed (with type inference)
- An aggressive optimizing compiler
- (Parallel) garbage collection
- **Pure**: side-effects are disallowed by default

The last feature goes a long way towards making parallelism tractable

2008 Galois, Inc. All rights reserved.

A quick taste of the syntax

```
main = print (take 1000 primes)

primes = sieve [2..]
  where
    sieve (p:xs) =
      p : sieve [ x | x <- xs, x `mod` p > 0]
```

2008 Galois, Inc. All rights reserved.

A quick taste (with explicit types)

```
main :: IO ()
main = print (take 1000 primes)

primes :: [Int]
primes = sieve [2..]
  where
    sieve :: [Int] -> [Int]
    sieve (p:xs) =
      p : sieve [ x | x <- xs, x `mod` p > 0]
```

2008 Galois, Inc. All rights reserved.

Compiling Haskell programs

```
$ ghc -O2 --make A.hs
[1 of 1] Compiling Main             ( A.hs, A.o )
Linking A …

$ ./A
[2,3,5,7,11,13,17,19,23, … 7883,7901,7907,7919]
```

2008 Galois, Inc. All rights reserved.

Parallel Haskell

Parallel Haskell programs:

- Semi-explicit parallelism via sparks
- Explicit parallelism with threads and synchronization
- Nested data parallelism

Language runtime arranges for Haskell threads to be scheduled on to OS threads

2008 Galois, Inc. All rights reserved.

Compiling Parallel Haskell programs

Add the –threaded flag for parallel programs

```
$ ghc -O2 --make -threaded Foo.hs
[1 of 1] Compiling Main             ( Foo.hs, Foo.o )
Linking Foo …
```

Specify at runtime how many real (OS) threads to map Haskell's logical threads to:

```
$ ./A +RTS -N8
```

In this talk “thread” means Haskell's cheap logical threads, not those 8 OS threads

2008 Galois, Inc. All rights reserved.

Parallel Haskell: Technique 1

Lightweight parallel annotations

2008 Galois, Inc. All rights reserved.

Semi-explicit parallelism

Lack of side effects makes parallelism easy, right?

```
f x y = (x * y) + (y ^ 2)
```

- We could just evaluate **every** sub-expression in parallel
- It is always safe to speculate on pure code

Creates far too many small work tasks to execute

So in Haskell, the strategy is to give user has control over which expressions are sensible to run in parallel

2008 Galois, Inc. All rights reserved.

import Control.Parallel

Two functions for controlling which expressions to run:

```
par    :: a → b → b
pseq :: a → b → b
```

par says

- “*might be a good idea to evaluate the first argument 'a' in parallel with 'b'*”
- Creates a *spark* which might be run in a Haskell thread, if one becomes available
- Very cheap speculative parallelism
- A “lazy future”.

2008 Galois, Inc. All rights reserved.

Import Control.Parallel

The expression:

```
a `par` b
```

- Creates a spark for 'a'
- Runtime sees chance to convert spark into a thread
- And thus run it in parallel, on another core
- 'b' is returned
- No restrictions on what you can annotate

2008 Galois, Inc. All rights reserved.

import Control.Parallel

And the second function, pseq:

pseq :: a → b → b

Says

- “*evaluate 'a' in the* **current** *thread, then return b*”
- Ensures work is run in the right thread

2008 Galois, Inc. All rights reserved.

Putting it together

Together we can parallelise expressions correctly:

```
f `par` e `pseq` f + e
```

- One spark created for 'f'
- 'f' spark converted to a thread and executed
- 'e' evaluated in current thread in parallel with 'f'
- Result combined

```
$ time ./B +RTS -N2
100000020000000
0.84s user 0.02s system 190% cpu 0.478 total
```

2008 Galois, Inc. All rights reserved.

Parallel Strategies

From these two building blocks, a range of strategies for common parallelisation tasks are built up:

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

Default parallelisation strategies driven by the shape of the data, saves reimplementing common strategies time and again.

More data structures should come with parallelised APIs!

Very flexible, light programming model: just annotate your code speculating on what is worthwhile to evaluate.

2008 Galois, Inc. All rights reserved.

Summary: thread sparks

Very cheap to annotate programs with `par` and `pseq`

Lessons:

- Fine-grained parallelism
- Sparks need to be cheap
- Work-strealing runtime underneath
- Relies on purity: no side effects to get in the way
- Takes practice to learn where `par` is beneficial
- A good tool to have in the kit

2008 Galois, Inc. All rights reserved.

Parallel Haskell: Technique 2

Explicit threads,
MVars
&
Software Transactional Memory

2008 Galois, Inc. All rights reserved.

Explicit concurrency: threads

For stateful or imperative programs, we need explicit threads, not speculative sparks.

To create a real Haskell thread:

forkIO :: IO () → IO ThreadId

- The “IO” means “can have side effects”
- Takes a block of code to run, and executes it in a new Haskell thread

Note: all the parallelism and concurrency abstractions done via library **functions**, not new syntax.

2008 Galois, Inc. All rights reserved.

IO: Caging the effects monster

In Haskell, side effecting code is tagged statically, via its type.

```
getChar :: IO Char
putChar :: Char → IO ()
```

Such side-effecting code can only interact with other side effecting code. It can't mess with pure code. Checked statically.

Imperative is “off” by default in Haskell :-)

Haskellers control effects by trapping them in the IO box

2008 Galois, Inc. All rights reserved.

Explicit concurrency: communication

Explicit threads need explicit communication, via shared synchronized state.

- State manipulation is a side effect.
- So forkIO runs “IO” actions.

Synchronization achieved via *MVars* or *STM*

2008 Galois, Inc. All rights reserved.

Synchronizing threads with MVars

A box containing some value:

```
data MVar a
```

Operations:

```
newMVar  :: a → IO (MVar a)
takeMVar :: MVar a → IO a
putMVar  :: MVar a → a → IO ()
```

2008 Galois, Inc. All rights reserved.

MVar semantics

```
takeMVar :: MVar a → IO a
```

Take contents of MVar when full, else thread blocks.

```
putMVar  :: MVar a → a → IO ()
```

Put value into MVar if empty, else block.

Simple, efficient way to pass results between threads.

2008 Galois, Inc. All rights reserved.

Explicit parallelism with threads

```
do box <- newEmptyMVar
   forkIO (f `pseq` putMVar box f)
   e `pseq` return ()
   f <- takeMVar box
   print (e + f)
```

2008 Galois, Inc. All rights reserved.

Parallelism with explicit threads

```
$ ghc -threaded -O2 --make C.hs
[1 of 1] Compiling Main             ( C.hs, C.o )
Linking C …

$ ./C +RTS -N2
100000020000000
0.86s user 0.02s system 182% cpu 0.485 total
```

2008 Galois, Inc. All rights reserved.

Deadlock free synchronization

MVar programs can deadlock, if one thread is waiting for a value from another, that will never appear.

Haskell let's us write lock-free synchronization via *software transactional memory*

Higher level than MVars, much safer, composable, but a bit slower.

Continuing theme: multiple levels of resolution

2008 Galois, Inc. All rights reserved.

Software transactional memory

- STM added to Haskell in 2005 (MVars in 1995, from Id).
- Used in real systems (ones I work on)
- A composable, safe synchronization abstraction

- *An optimisitic model*
  - Transactions run inside atomic blocks assuming no conflicts
  - System checks consistency at the end of the transaction
  - Retry if conflicts
  - Requires control of side effects (handled in the type system)

2008 Galois, Inc. All rights reserved.

Software transactional memory

```
data STM a
atomically :: STM a → IO a
retry      :: STM a
orElse     :: STM a → STM a → STM a
```

- We use 'STM a' to build up *atomic blocks.*
- Transaction code can **only** run inside atomic blocks
  - checked statically
- Inside atomic blocks it appears as if no other threads are running
- However, the system uses *logs and rollback to handle conflicts*
- 'orElse' lets us compose atomic blocks into larger pieces

2008 Galois, Inc. All rights reserved.

Transaction variables

TVars replace MVars, and are used inside (logically) atomic blocks

```
data TVar a
newTVar     :: a → STM (TVar a)
readTVar    :: TVar a → STM a
writeTVar   :: TVar a → a → STM ()
```

Similar operations to MVars, but implemented by logging and rollback when there are conflicts, so no deadlocks!

Type system ensures only side effects in transactions are reads and writes to TVars

2008 Galois, Inc. All rights reserved.

Atomic bank transfers

```
transfer :: TVar Int -> TVar Int -> Int -> IO ()

transfer from to amount =
  atomically $ do
    balance <- readTVar from
    if balance < amount
      then retry
      else do
        writeTVar from (balance - amount)
        tobalance <- readTVar to
        writeTVar to (tobalance + amount
```

2008 Galois, Inc. All rights reserved.

Software transactional memory

- Deadlock-free synchronization variables
- More overhead though: log and rollback
  - Shows up under heavy contention
- We gain compositionality of blocks, and deadlock avoidance
- Great productivity/correctness/flexibility gain

2008 Galois, Inc. All rights reserved.

Parallel Haskell: Technique 3

Nested Data Parallelism

2008 Galois, Inc. All rights reserved.

Data Parallel Haskell

We can write a *lot* of parallel programs with the last two techniques, but:

- par/seq are very light, but granularity is hard
- forkIO/MVar/STM are more precise, but more complex
- Trade offs between abstraction and precision

The third way to parallel Haskell programs:

- **nested** data parallelism

2008 Galois, Inc. All rights reserved.

Data Parallel Haskell

Simple idea:

Do the same thing in parallel
to every element of a large collection

If your program can be expressed this way, then,

- No explicit threads or communication
- Clear cost model (unlike `par`)
- Good locality, easy partitioning

2008 Galois, Inc. All rights reserved.

import Data.Array.Parallel

Sum the squares of a parallel vector of floats.

```
sumsq :: [: Float :] → Float
sumsq a = sumP [: x*x | x ← a :]
```

Similar functions for map, zip, append, filter, length etc.

- Break array into N chunks (for N cores)
- Run a sequential loop to apply 'f' to each chunk element
- Run that loop on each core
- Combine the results

Clean support for flat data parallelism.

2008 Galois, Inc. All rights reserved.

Cons of flat data parallelism

While simple, the downside is that a single parallel loop drives the whole program.

Not very compositional.

No rich data structures, just flat things.

So how about *nested* data parallelism?

2008 Galois, Inc. All rights reserved.

Nested Data Parallelism

Simple idea:

Do the same thing in parallel
to every element of a large collection
and
Each thing you do may in turn be a nested parallel computation

2008 Galois, Inc. All rights reserved.

Nested Data Parallelism

If your program can be expressed this way, then,

- No explicit threads or communication
- Clear cost model (unlike `par`)
- Good locality, easy partitioning

Breakthrough:

Flattening: a compiler transformation to systematically transform any nested data parallel program into a flat one

2008 Galois, Inc. All rights reserved.

import Data.Array.Parallel

Nested data-parallel programming:.

```
type Vector = [: Float  :]
type Matrix = [: Vector :]

matMul :: Matrix → Vector → Vector
matMul m v = [: vecMul r v | r ← m :]
```

Data parallel functions (vecMul) inside data parallel functions

2008 Galois, Inc. All rights reserved.

Summary of DPH

Nested data-parallel programming:.

- Arbitrary nesting and arbitrary flattening of data structures into flat data parallel array code
- Clear performance model
- Available as a library
- Alpha release in GHC 6.10
- Project to map DPH onto GPUs
- Go and try it out!

2008 Galois, Inc. All rights reserved.

Haskell for mainstream multicores

What you've seen:

- Semi-explicit parallelism via sparks
- Explicit parallelism with threads, MVars and STM
- Nested data parallelism

All in a widely used open source compiler

- Safe, productive, parallel
- Large community around the language
- Used in industry, research, open source
- And emerging use for mainstream multicores

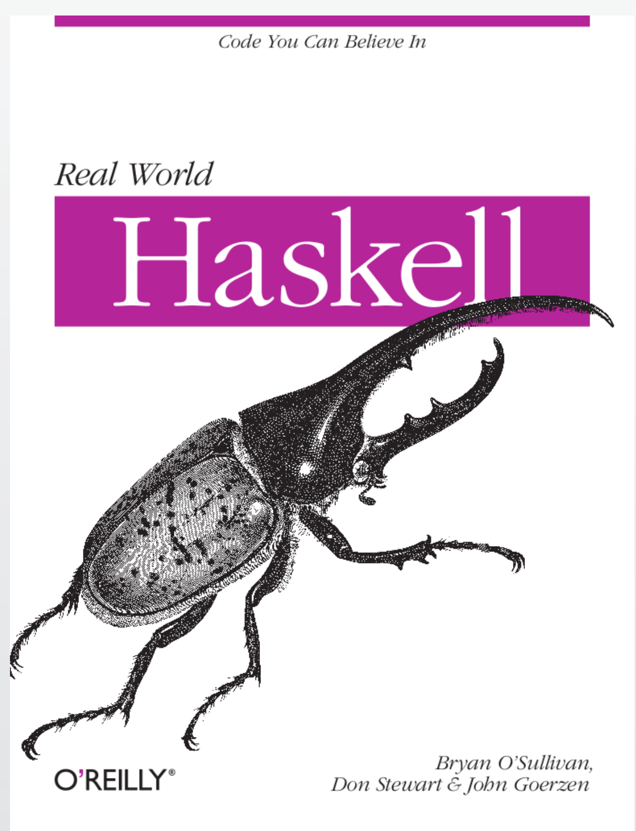
2008 Galois, Inc. All rights reserved.

More about Haskell!

haskell.org
book.realworldhaskell.org

Thanks to:

Simon Peyton Jones & Satnam Singh:
“A Tutorial on Parallel and Concurrent Programming in Haskell”



2008 Galois, Inc. All rights reserved.

Thanks!

Galois

Compiler and language engineering

Formal methods

High assurance systems

High performance cryptography

Parallel languages for FPGAs

Domain-specific languages

Don Stewart

dons@galois.com

2008 Galois, Inc. All rights reserved.