Multicore Haskell Now!

Don Stewart | Reflections | Projections | UIUC | Oct 2009

The Grand Challenge

- Making effective use of multicore hardware is **the** challenge for programming languages now
- Hardware is getting increasingly complicated:
  - Nested memory hierarchies
  - Hybrid processors: GPU + CPU, Cell, FPGA...
  - Massive compute power sitting mostly idle
- Will require a number of new programming models to program commodity machines effectively

Haskell is ...

- A purely functional language
- Strongly statically typed
- 20 years old
- Open source
- Compiled and interpreted
- Used in research, open source and industry

http://haskell.org
http://haskell.org/platform
http://hackage.haskell.org

- Built for parallel programming

Haskell and Parallelism: Why?

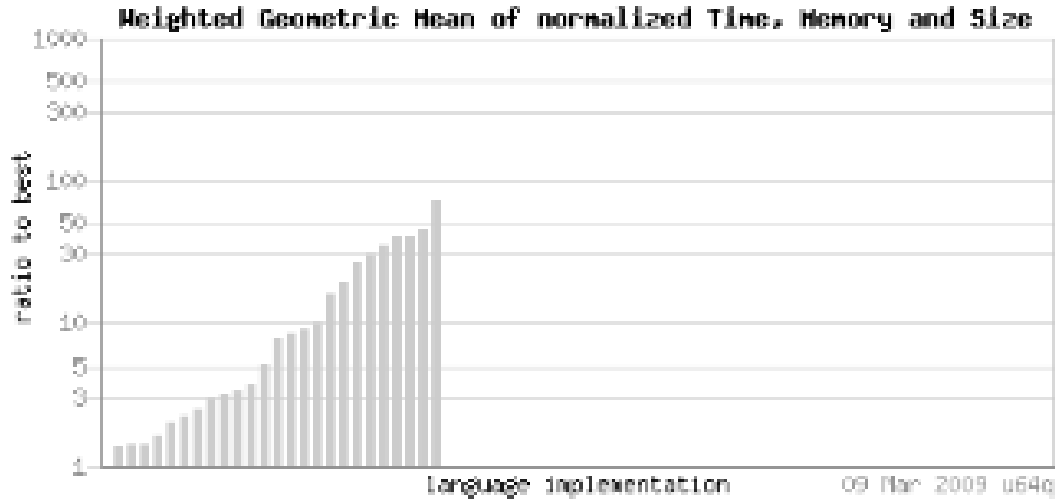
- Language reasons:
  - Purity, laziness and types mean you can find more parallelism in your code
  - No specified execution order
  - Speculation and parallelism safe.
- Purity provides inherently more parallelism
- High level: more productivity than say, C++

Haskell and Parallelism

- Statically typed and heavily optimized: more performance than, say, Python or Erlang.
- Custom multicore runtime: high performance threads a primary concern – thanks Simon Marlow!
- Mature: 20 year code base, long term industrial use, massive library system
- Demonstrated performance

"For every complex problem there is an answer that is clear, simple, and wrong." H. L. Mencken

What fun! Can you manipulate the multipliers and weights to make your favourite language the best programming language in



/	language	GM	missing
1.0	C++ GNU g++	1.41	
1.4	ATS	1.93	3
1.4	C GNU gcc	1.97	
1.4	Haskell GHC	2.00	
1.6	Java 6 -server	2.30	
2.0	Fortran Intel	2.83	6
2.2	OCaml	3.11	4
2.5	C# Mono	3.47	1
3.0	Scala	4.17	
3.2	Lisp SBCL	4.55	2
3.3	Clean	4.65	5
3.7	Pascal Free Pascal	5.23	2
5.1	Ada 2005 GNAT	7.25	4
7.9	Erlang HiPE	11.07	1
8.5	F# Mono	12.03	6
9.2	Java 6 -Xint	12.95	
10	Scheme PLT	14.40	

data for largest N

calculate reset

multipliers	
Time secs	1
Memory KB	0
Source size B	0

benchmark	weight
binary-trees	1
chameneos-redux	1
fannkuch	1
fasta	1
k-nucleotide	1
mandelbrot	1

The Goal

- **Parallelism**: exploit parallel computing hardware to improve performance
- **Concurrency**: logically independent tasks as a structuring technique

- Improve performance of programs by using multiple cores at the same time
- Improve performance by hiding latency for IO-heavy programs

Getting started with multicore

- Background + Refresh
- Toolchain
- GHC runtime architecture
- The Kit
  - **Sparks and parallel strategies**
  - **Threads, messages and shared memory**
  - **Transactional memory**
  - **Data parallelism**
- Debugging and profiling
- Garbage collection

Source for this talk

- Slides and source on the blog, along with links to papers for further reading
  - Google “multicore haskell now”
- or
  - Visit http://donsbot.wordpress.com

Syntax refresh

```
main = print (take 1000 primes)

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

Syntax refresh

```
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]
```

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]
```

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

IO is kept separate

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 (default sequentialisation and side effects) off by default :-)

Haskellers control effects by trapping them in the IO box

The Toolchain

Toolchain

- GHC 6.10.x or 6.12.x
- Haskell Platform 2009.2.0.2
  - http://haskell.org/platform
- Dual core x86-64 laptop running Linux

- GHC 6.12 is even better (out next week!)
  - Sparks cheaper
  - GC parallelism tuned

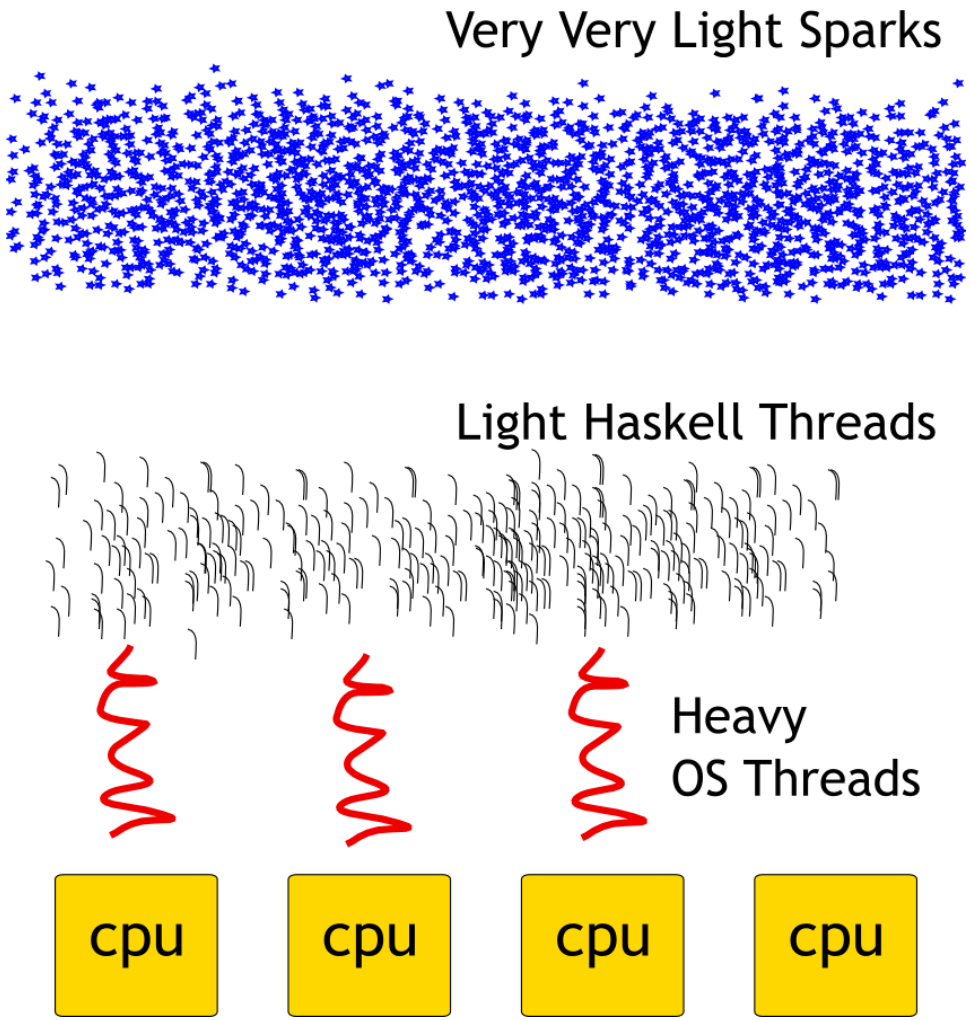
Haskell Platform

The GHC Runtime

The GHC Runtime

- Multiple virtual cpus
  - Each virtual cpu has a pool of OS threads
  - CPU local spark pools for additional work
- Lightweight Haskell threads map onto OS threads: many to one.
- Automatic thread migration and load balancing
- Parallel, generational GC
- Transactional memory and Mvars

Concurrency Hierarchy



Runtime Settings

Standard flags when compiling and running parallel programs

- Compile with
  - -threaded -O2
- Run with
  - +RTS -N2
  - +RTS -N4
  - ...
  - +RTS -N64
  - ...

1. Implicit Parallelism: Sparks and Strategies

The `par` combinator

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 parallel tasks to execute

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

Semi-implicit parallelism

- Haskell gives us “parallel annotations”.
- Annotations on code to that hint when parallelism is useful
  - Very cheap post-hoc/ad-hoc parallelism
- Deterministic multicore programming **without** :
  - Threads
  - Locks
  - Communication
- Often good speedups with very little effort

Provided by: the parallel library

http://hackage.haskell.org/packages/parallel

```
$ ghc-pkg list parallel
/usr/lib/ghc-6.10.4/./package.conf:
    parallel-1.1.0.1
```

```
import Control.Parallel
```

```
$ cabal unpack paralllel
```

Ships with the Haskell Platform.

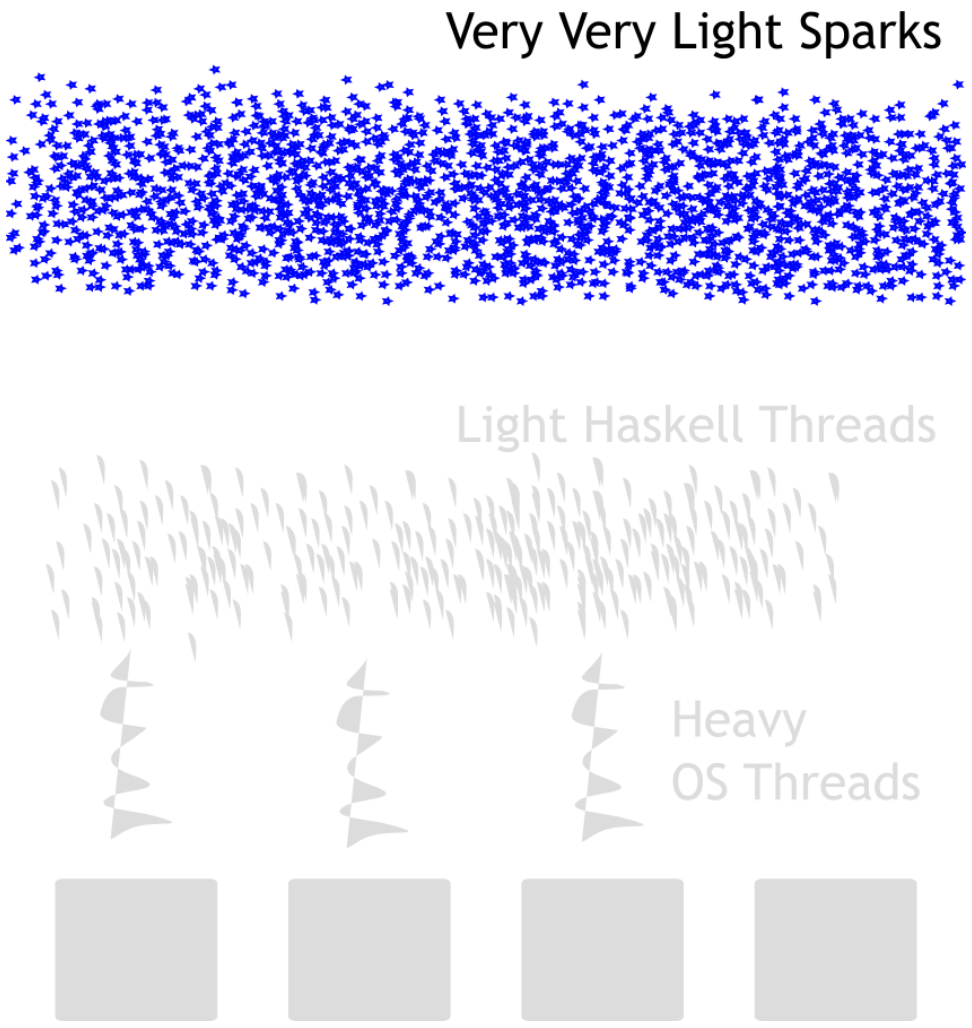
The `par` combinator

All parallelism built up from the `par` combinator:

```
a `par` b
```

- Creates a spark for 'a'
- Runtime sees chance to convert spark into a thread
- Which in turn may get run in parallel, on another core
- 'b' is returned
- **No restrictions on what you can annotate**

Sparks



What `par` guarantees

- `par` doesn't guarantee a new Haskell thread
- It “hints” that it would be good to evaluate the argument in parallel
- The runtime is free to decide to push a spark down
  - Depending on workload
  - Depending on cost of the value
- This allows `par` to be very cheap
- So we can use it almost anywhere
- To overapproximate the parallelism in our code

The `pseq` combinator

We also need a way to say “do it in this thread first”

And the second function, pseq:

$$pseq :: a \rightarrow b \rightarrow b$$

Says not to create a spark, instead:

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

Putting it together

Together we can parallelise expressions:

```
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'

Simple sparks

02.hs

```
$ ghc-6.11.20090228 02.hs --make -threaded -O2
$ time ./02
14050061177528798985431426062445115699363840000008189
./02  2.00s user 0.01s system 99% cpu 2.015 total

$ time ./02 +RTS -N2
14050061177528798985431426062445115699363840000008189
./02 +RTS -N2  2.14s user 0.03s system 140% cpu 1.542 total
```

Cautions

- Don't “accidentally parallelize”:
  - f `par` f + e – – depends on eval order of (+)
- `pseq` lets us methodically prevent accidents
- Main thread works on 'f' causing spark to *fizzle*
- Need roughly the same amount of work in each thread
- ghc 6.12: use ThreadScope to determine this

Reading runtime output

- Add the -sstderr flag to the program:
  - ./02 +RTS -N2 -sstderr
- And we get:

7,904 bytes maximum residency (1 sample(s))

2 MB total memory in use (0 MB lost due to fragmentation)

Generation 0: 2052 collections, 0 parallel, 0.19s, 0.18s elapsed

Generation 1: 1 collections, 0 parallel, 0.00s, 0.00s elapsed

Parallel GC work balance: nan (0 / 0, ideal 2)

SPARKS: 2 (2 converted, 0 pruned)

%GC time 7.9% (10.8% elapsed)

Productivity 92.1% of total user, **144.6% of total elapsed**

ThreadScope output

- ThreadScope just released (in beta), but it already helps us think about spark code. Try it out!

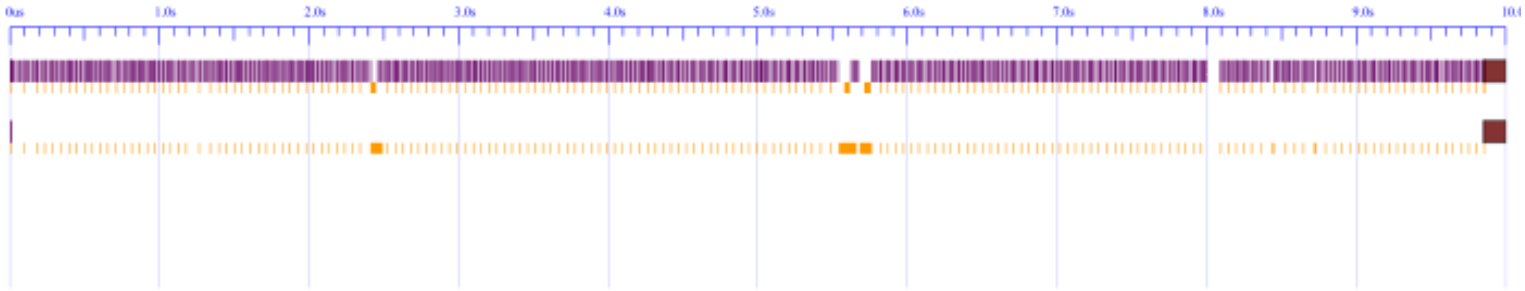


Figure 1. No parallelization of `f 'par' (f + e)`

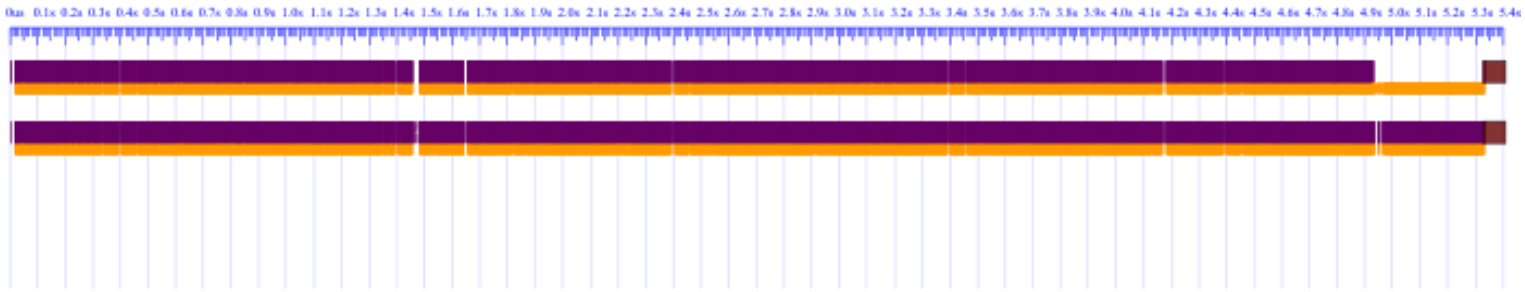


Figure 2. A lucky parallelization of `f 'par' (e + f)`

Finding more parallelism

```
parfib :: Int -> Int
parfib 0 = 0
parfib 1 = 1
parfib n = n1 `par` (n2 `pseq` n1 + n2)
 where
    n1 = parfib (n-1)
    n2 = parfib (n-2)
```

Increasing the parallelism : 03.hs

- Push the sparks down from the top level, into the recursion
- Parfib!! 03.hs
- Single core:

- $ time ./03 43

  parfib 43 = 433494437

  ./03 43 +RTS 22.42s user 0.05s system **97% cpu 23.087** total

Increasing the parallelism : 03.hs

- Push the sparks down from the top level, into the recursion
- Parfib!! 03.hs

- $ time ./03 43 +RTS **-N2**

  parfib 43 = 433494437

  ./03 43 +RTS -N2 27.21s user 0.27s system **136% cpu 20.072** total
- Only a little faster... what went wrong?

Check what the runtime says

- ./03 43 +RTS -N2 -sstderr 24.74s user 0.40s system 120% cpu 20.806 total

  ...

  SPARKS: 701,498,971 (**116 converted**, 447,756,454 pruned)

  ...

- Seems like an awful lot of sparks

- N.B. Sparks stats available only in >= ghc 6.11

Still not using all the hardware

- Key trick:
  - Push sparks into recursion
  - But then have cutoff for when the costs are too high.
- `par` is cheap (and getting cheaper!), but not free.

Sparks with cutoffs

```
parfib :: Int -> Int -> Int
parfib n t
    | n <= t       = nfib n – – cutoff triggers
    | otherwise = n1 `par` n2 `pseq` n1 + n2
 where n1 = parfib (n-1) t
           n2 = parfib (n-2) t

-- sequential version of the code
nfib :: Int -> Int
nfib 0 = 0
nfib 1 = 1
nfib n = nfib (n-2) + nfib (n-1)
```

Not too fine grained: 04.hs

- Use **thresholds** for sparking in the recursion

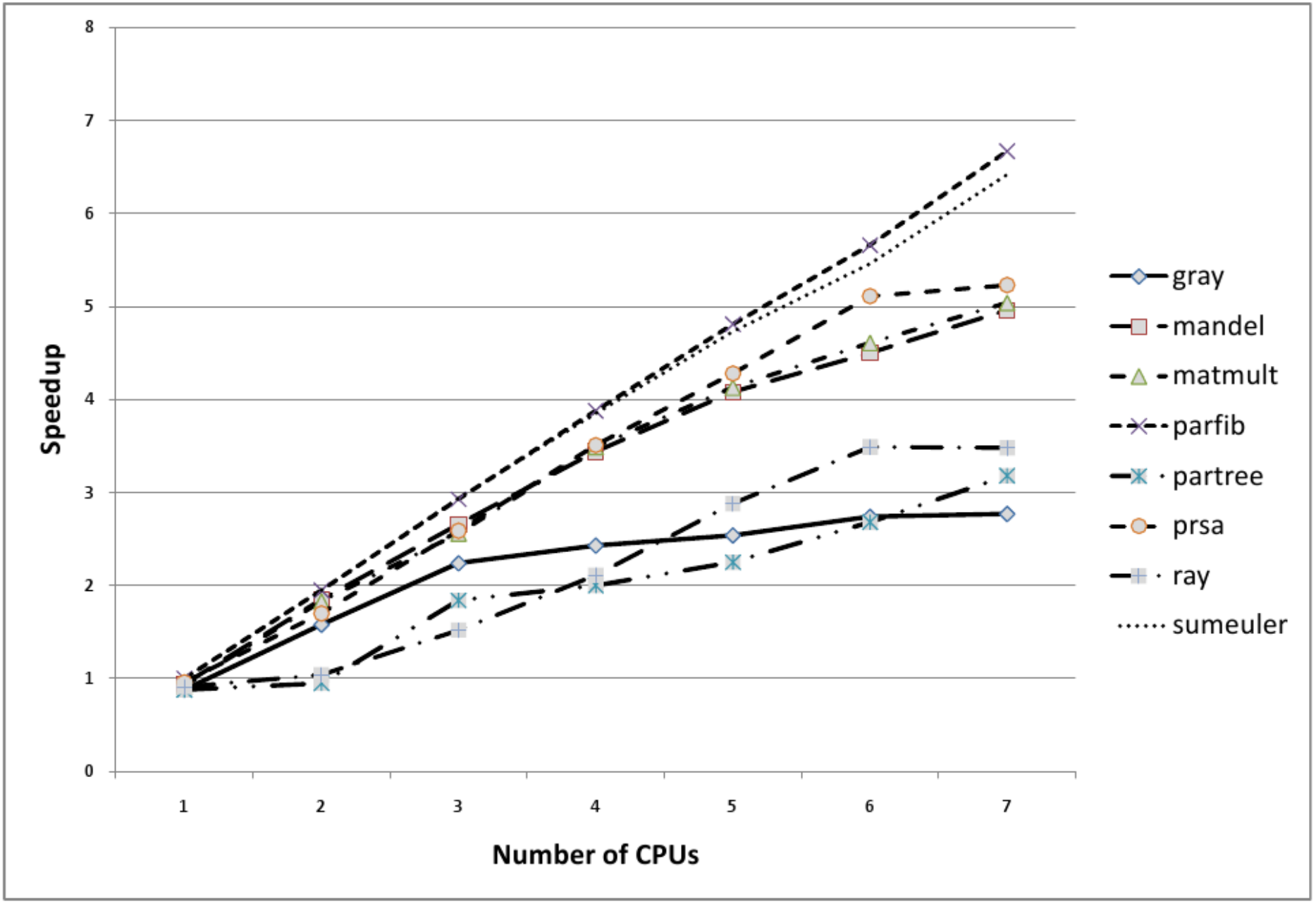
- $ time ./04 43 11 +RTS -N2

  parfib 43 = 433494437

  ./04 43 17 +RTS -N2 -sstderr 8.05s user 0.03s system **190% cpu 4.239 total**

Garbage collection

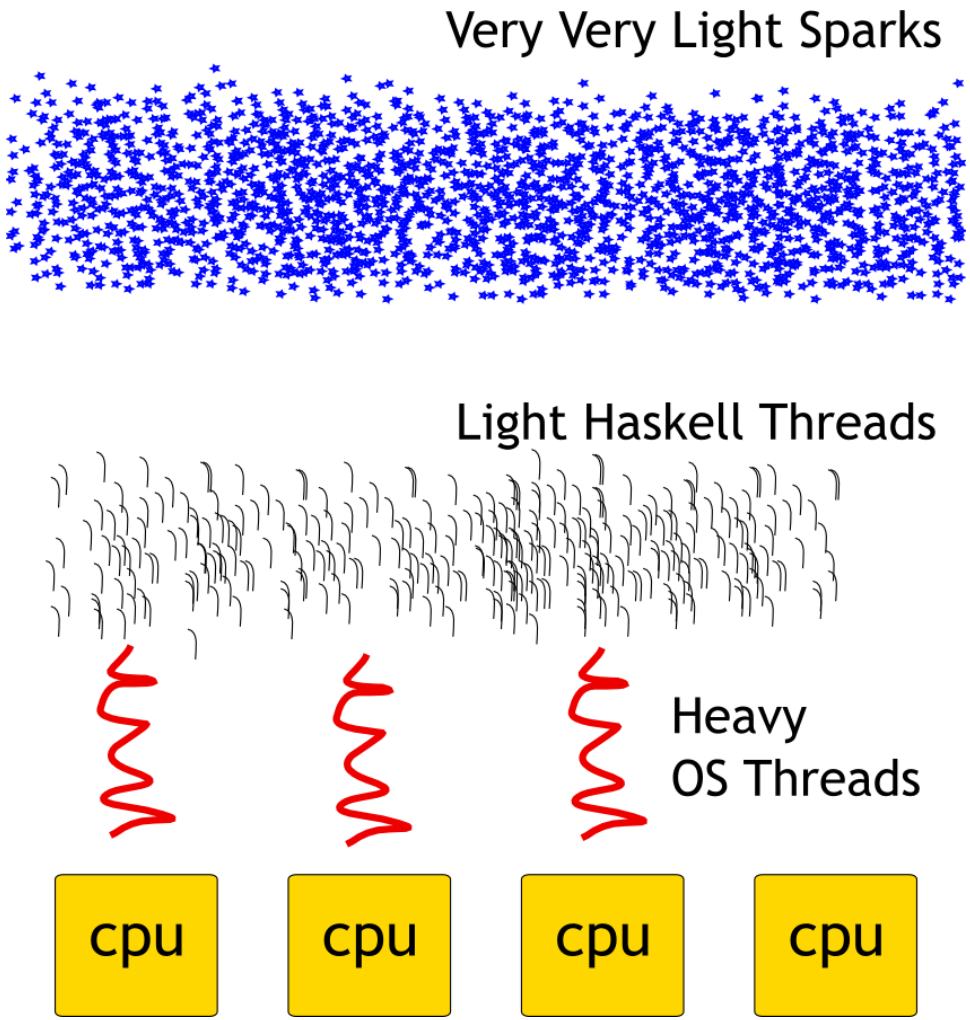
- The GHC garbage collector is a parallel stop-the-world collector
- Stopping-the-world means running no threads
- You don't want to do that very often
- Check your GC stats (-sstderr) and bring the GC percent down by increasing the default allocation (-H400M or -A400M)
- Stay tuned for per-CPU garbage collectors


Speedup
Number of CPUs
0
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
gray
mandel
matmult
parfib
partree
prsa
ray
sumeuler

Sparks Programming Model

- Deterministic:
  - Same results with parallel and sequential programs
  - No races, no errors
  - Good for reasoning: erase the `par` and get the original program
- Cheap: sprinkle par as you like, then measure and refine
- Measurement much easier with Threadscope
- Strategies: high level combinators for common patterns

Thread model



Spark queues

- How does it work?
  - -N4 gives us 4 heavy OS threads
  - Runtime muliplexes many Haskell threads
  - Generated with forkIO or par
  - ~One OS thread (“worker thread”) per cpu
  - Worker threads may migrate
  - Each cpu has a spark pool. `par` adds your thunk to the current cpu's list of work
  - Idle worker threads turn a spark into a Haskell thread
  - Haskell threads keeps stealing sparks from others

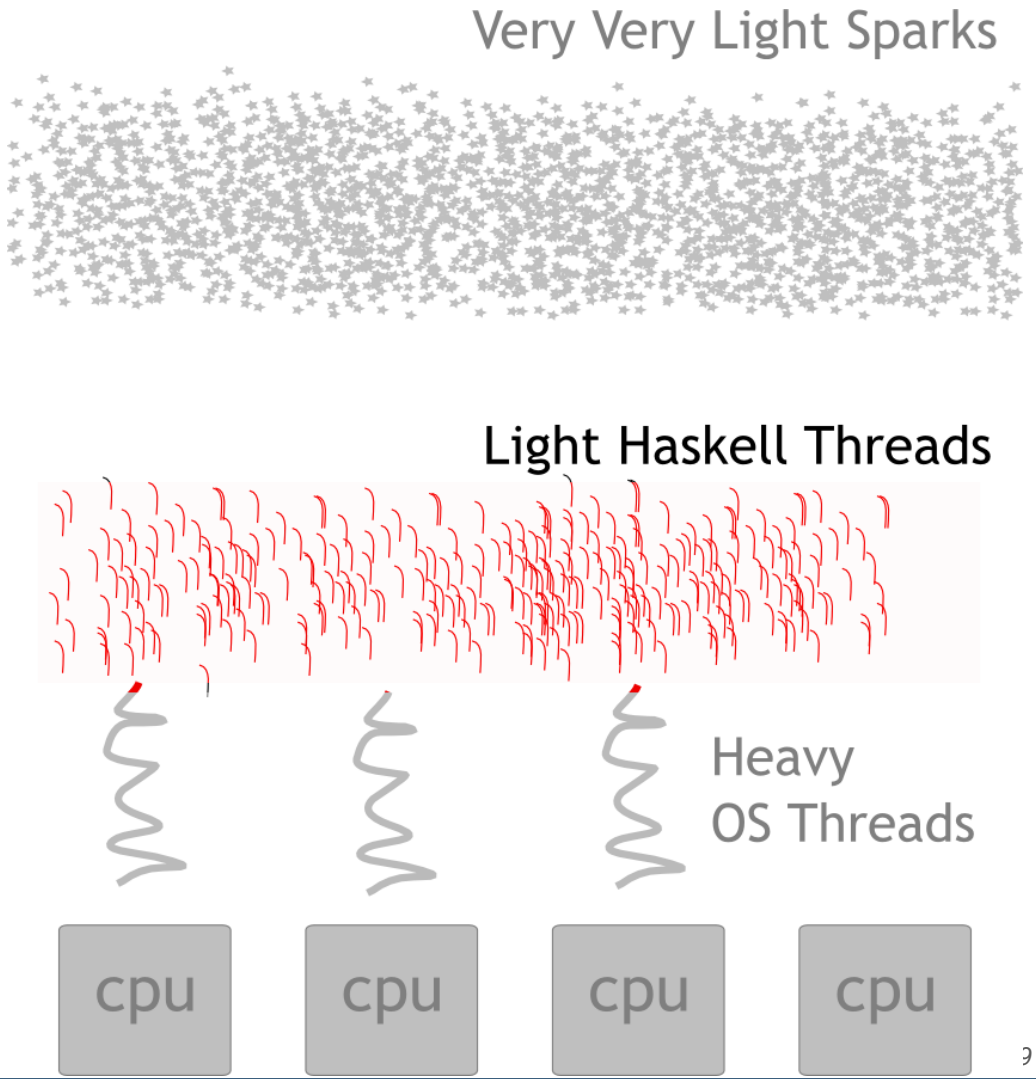
Sparks and Strategies: Summary

Cheap to annotate programs with `par` and `pseq`

- Fine-grained parallelism
- Sparks need to be cheap
- Work-strealing thread pool in 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

2. Explicit Parallelism: Threads and Shared Memory

Explict Haskell Threads



Explicit concurrency with threads

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

forkIO :: IO () → IO ThreadId

Takes a block of code to run, and executes it in a new Haskell thread

Concurrent programming with threads: 07.hs

```
import Control.Concurrent
import System.Directory

main = do
    forkIO (writeFile "xyz" "thread was here")
    v ← doesFileExist "xyz"
    print v
```

Non-determinism – welcome to concurrent programming

(Unlike spark programming)

Programming model

- Threads are preemptively scheduled
- Non-deterministic scheduling: random interleaving
- When the main thread terminates, all threads terminate (“daemonic threads”)
- Threads may be preempted when they allocate memory
- Communicate via messages or shared memory

Asynchronous Exceptions: 08.hs 09.hs

- We need to communicate with threads somehow.
- One simple way is via asynchronous messages.
  - import Control.Exception
- Just throw messages at each other, catching them and handling them as you see fit.
- throwTo and catch/handle
- Good technique to know
- Good for writing fault tolerant code

Shared Memory: MVars

- We need to communicate between threads
- We need threads to wait on results
- In pure code, values are immutable, so safe to share
- However, with threads, we use shared, mutable synchronizing variables to communicate

Synchronization achieved via MVars or STM

Shared Memory: MVars

- import Control.Concurrent.MVar

- MVars are boxes. They are either full or empty
  - putMVar :: MVar a → a → IO ()
  - takeMVar :: MVar a → IO a

- “put” on a full MVar causes the thread to sleep until the MVar is empty
- “take” on an empty MVar blocks until it is full.
- The runtime will wake you up when you're needed

Putting things in their boxes

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

Forking tasks and communicating: 10.hs

- Here we create explicit Haskell threads, and set up shared memory for them to communicate
- Lower level than using sparks. More control

```
$ time ./10 +RTS -N2 -stderr
933262154439441526 81...
./10 +RTS -N2 -stderr  2.32s user 0.06s system
    146% cpu 1.627 total
```

Hiding IO Latency

- When you have some expensive IO action, fork a thread for the work
- And return to the user for more work
- Works well for hiding disk and network latency
- Transparently scales: just add more cores and the Haskell threads will go there.
- Handle network connections in thousands of threads concurrently

Shared Memory: Chans: 14.hs

- Chans: good for unbounded numbers of shared messages
- Send and receive messages of a pipe-like structure
- Can be converted to a lazy list, representing all future messages!

Channels

```
main = do
    ch ← newChan
    forkIO (worker ch)
    xs ← getChanContents ch – – convert future msgs to list
    mapM_ print xs              – – lazily print as msgs arrive

worker ch = forever $ do
    v ← readFile "/proc/loadavg"
    writeChan ch v              – – send msg back to receiver
    threadDelay (10^5)
```

Transactional Memory

MVars can deadlock

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

Haskell lets 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

Software Transactional Memory

- Each atomic block appears to **work in complete isolation**
- Runtime publishes modifications to shared variables to all threads, or,
- Restarts the transaction that suffered contention
- You have the illusion you're the only thread

STM

- STM added to Haskell in 2005 (MVars in 1995, from Id).
- Used in a number of real, concurrent systems
- 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)

The stm package

- http://hackage.haskell.org/packages/stm

- $ ghc-pkg list stm
  /usr/lib/ghc-6.10.4/./package.conf:
      stm-2.1.1.2

- import Control.Concurrent.STM
- $ cabal unpack stm
- In the Haskell Platform

STM

```
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
- Inside atomic blocks it appears as if no other threads are running (notion of **isolation**)
- However, the system uses *logs and rollback to handle conflicts*
- 'orElse' lets us compose atomic blocks into larger pieces

Transaction variables

TVars are the variables the runtime watches for contention:

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

Actions always succeed: implemented by logging and rollback when there are conflicts, so no deadlocks!

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
```

Safety

- For it to be possible to roll back transactions, atomic blocks can't have visible side effects
- Enforced by the type system
  - In the STM monad, you can guarantee atomic safety
- atomically :: STM a → IO a
- No way to do IO in a transaction...
  - Only pure code
  - Exceptions
  - Non termination
  - Transactional effects

retry: where the magic is

- How does the runtime know when to wake up an atomic section?
- It blocks the thread until something changes in one of the in-scope transaction variables
- Automatically waits until we can make progress!

OrElse: trying alternatives

- Don't always just want to retry forever
- Sometimes we need to try something else
  - orElse :: STM a → STM a → STM a
- Compose two atomic sections into one
- If the first fails, try the second.

Treating the world as a transaction

- You can actually run IO actions from STM
  - GHC.Conc.unsafeIOToSTM :: IO a → STM a
- If you can fulfil the proof obligations...
- Useful for say, lifting transactional database actions into transactions in Haskell.
- Mostly we'll try to return a value to the IO monad from the transaction and run that

Summary of benefits

- STM composes easily!
- Just looks like imperative code
- Even when there are atomic sections involved

- **No deadlocks.**
- Lock safe code when composed is still lock safe

- Progress: keep your transactions short

Data Parallelism: Briefly

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

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

Parallel Arrays

- Adds parallel array syntax:
  - [: e :]
  - Along with many parallel “combinators”
    - mapP, filterP, zipP, foldP, …
  - Very high level approach
- Parallel comprehensions
  - Actually have parallel semantics
- DPH is oriented towards large array programming

Import Data.Array.Parallel

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

dotp :: [:Float:] -> [:Float:] -> Float
dotp v w = sumP (zipWithP (*) v w)
```

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

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?

Nested Data Parallelism

Simple idea:

Do the same thing in parallel

to every element of a large collection

plus

Each thing you do may in turn be a nested parallel computation

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

Import Data.Array.Parallel

Nested data-parallel programming, via the vectoriser:

```
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

The vectorizer

- GHC gets significantly smarter
  - Implements a vectorizer
  - Flattens nested data, changing representations, automatically
  - Project to add a GPU backend well advanced
    - (see the “accelerate” library)
- See:
  - “Running Haskell Array Computations on a GPU” (video)

Small example: vect.hs

- Uses the dph libraries
  - dph-prim-par
  - dph

```
sumSq :: Int → Int
sumSq n = sumP (mapP (*) (enumFromToP 1 n))
```

Requires -fvectorize

Example: sumsq

- $ ghc -O2 -threaded --make vect.hs -package dph-par -package dph-prim-par-0.3

- $ time ./vect 100000000 +RTS -N2

  N = 100000000: 2585/4813 2585/4813 2585/4813

  ./vect 100000000 +RTS -N2 2.81s user 2.22s system **178%** cpu 2.814 tota

Notes

- Still in “technology preview”
- Significantly better in GHC 6.12
  - More programs actually speedup
- Latest status at:
  - http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

In conclusion...

Multicore Haskell Now

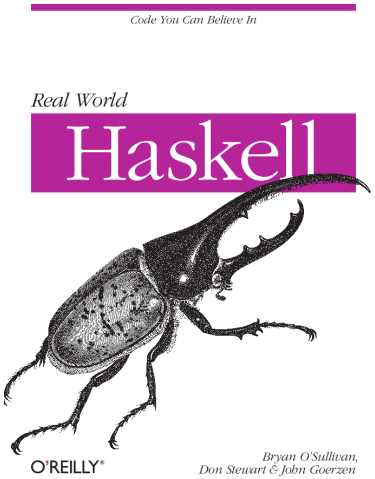
- Sophisticated, fast runtime
- Sparks and parallel strategies
- Explicit threads
- Messages and MVars for shared memory
- Transactional memory
- Data parallel arrays
- All in GHC 6.10, even better in GHC 6.12
- http://hackage.haskell.org/platform

Thanks

This talk made possible by:

Simon Peyton Jones
Satnam Singh
Manuel Chakravarty
Gabriele Keller
Roman Leschinkskiy
Bryan O'Sullivan

Simon Marlow
Tim Harris
Phil Trinder
Kevin Hammond
Martin Sulzmann
John Goerzen



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

|galois|

Galois is Hiring!

Research and tech transition company

Just over 10 years old, built with Haskell

Build systems with

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

Send resumes to jobs2009@galois.com

Go Program Your Multicore!