Multi-core with less pain

Deterministic Parallel Programming with Haskell

Duncan Coutts

December 2012, Tech Mesh

Well-Typed
The Haskell Consultants

Briefly about myself...

Background in FP in academia and open source

Co-founded Well-Typed

Well-Typed

- Haskell consultancy
- Support, planning, development, training
- Help a wide range of clients: startups to multinationals

Parallelism & Concurrency

Parallelism $\neq$ Concurrency

The goal

What’s the point?

The goal

What's the point?

Making programs run faster.

The goal

What's the point?

Making programs run faster.

Parallelism is all about making programs run faster by using more hardware (like multiple CPU or GPU cores)

Parallelism is hard

Lots of reasons why parallelism is hard

- have to understand our programs better
- need to know which bits take most time to execute
- need to know the dependencies within the program
- parallel work granularity vs overheads
- threads, shared variables, locks
- non-deterministic execution

Some of these reasons are intrinsic, some depend on the programming model we choose

Concurrency

Concurrency is about ways of structuring programs into independent activities that can communicate and synchronise with each other

- e.g. threads, shared variables and locks
- e.g. lightweight processes and message passing
- typically reacting to events from the outside world
- inherently non-deterministic

Concurrency is also hard!

Lots of reasons why concurrency is hard

- deadlocks
- data races
- non-deterministic behaviour
- testing possible interleavings

Concurrency makes parallelism even harder!

Many of the difficulties with parallelism are really difficulties with concurrency

- threads, shared variables, locks
- non-deterministic execution
- deadlocks
- data races

Taking a sequential program and making it concurrent makes it

- more complicated
- harder to read, understand, test & maintain

It makes programmers grumpy!

Let’s not knock concurrency too much

Concurrency does have its place.

For some problems using concurrency **simplifies** programs.

The server example

Servers handling conversations with multiple clients:

Using a separate thread of control for each client we can (mostly) just think about the interaction with that single client.

The alternative is dealing with the interactions with all clients simultaneously, e.g. using a complex state machine.

Concurrency & parallelism

Parallelism

making programs run faster by using more hardware
like multiple CPU or GPU cores

Concurrency

ways of structuring programs into independent activities that can communicate and synchronise with each other

These are orthogonal ideas

- one is about performance of running programs;
- the other is about the structure of programs.

Concurrency $\neq$ parallelism

Why make this distinction between parallelism and concurrency?

Concurrency $\neq$ parallelism

Why make this distinction between parallelism and concurrency?

So we can think about what we're really after.

If the goal is parallelism, then concurrency can be a distraction.

Well-Typed

Concurrency $\neq$ parallelism

Why make this distinction between parallelism and concurrency?

So we can think about what we're really after.

If the goal is parallelism, then concurrency can be a distraction.

All combinations of concurrency and parallelism make sense:

	No Concurrency	Concurrency
No Parallelism	most programs!	OS processes running on a single core
Parallelism	. . .	OS processes running on multiple cores

Parallelism and concurrency support in Haskell

Haskell supports both parallelism and concurrency

Relatively traditional approach to concurrency

- threads and shared mutable variables
- actors and other abstractions as libraries

Many different parallel styles

- expression style
- data flow style
- data-parallel style

Each implemented as a library (with some RTS support)

Different styles suitable for different kinds of problem

Well-Typed

Brief aside about Haskell concurrency...

Concurrency in Haskell

Haskell has excellent concurrency support

- uses IO monad
- lightweight threads
- nicer locking/synchronisation primitive (MVar)
- composable concurrency with STM
- traditional style blocking file/network I/O

Lightweight threads

Haskell threads are **very cheap**

- 10s of 1000s is no problem

Threads scheduled across multiple cores

Blocking works as you would hope

- blocking on I/O only blocks individual Haskell thread, not whole OS thread
- “safe” foreign calls only block individual Haskell thread (RTS uses a pool of OS threads)

Well-Typed

I/O done right

About the “Threads vs Events” debate...

Well-Typed

I/O done right

About the “Threads vs Events” debate...

We can have our cake and eat it

- performance of event-based I/O
- programming model of traditional blocking I/O
  - no Node.js-style callback madness
  - don’t even need .NET style async/futures
- makes use of all cores

Parallelism without concurrency

Deterministic parallelism

Parallelism without concurrency is often called **deterministic** or **pure** parallelism.

Means we write a program

- that is not explicitly concurrent;
- then execute it in parallel.

Deterministic parallelism

Parallelism without concurrency is often called **deterministic** or **pure** parallelism.

Means we write a program

- that is not explicitly concurrent;
- then execute it in parallel.

Example: SQL queries

A query itself expresses no concurrency.
Queries are deterministic, given the state of the database.

The execution engine is free to use multiple cores to do the work for a query.

Deterministic parallelism

Deterministic in the usual sense

- always gives the same answer, given the same inputs
- like an ordinary sequential program

And if it's deterministic then...

- does not depend on the scheduling or number of cores
- no data races
- no deadlocks

Sounds nice right?

Pure parallelism in expressions

Suppose `f x` and `g y` are expensive calculations.

In `f x + g y` we have an opportunity to evaluate the two parts in parallel.

Pure parallelism in expressions

Suppose `f x` and `g y` are expensive calculations.

In $\mathsf{f\ x} + \mathsf{g\ y}$ we have an opportunity to evaluate the two parts in parallel.

Note that $\mathsf{f\ x} + \mathsf{g\ y}$ does **not express any concurrency**.
It's just a pure calculation!

The **mechanism** for evaluating the expression has the possibility to use parallelism to get the results sooner.

We can push this idea a long way.

Side effects and safety

In a **pure** functional programming language like Haskell, evaluating `f x + g y` has no side effects.

So it is **always** safe to evaluate both parts in parallel, we will always get the same answer.

Data parallelism

Closely related to the idea of parallelism within expressions is **data parallelism**.

Data parallelism is all about doing the same operation to a large number of data items. The operation on each item of data is **independent** so they can all be done in **parallel**.

The typical examples are bulk operations on large vectors and arrays.

A quick look behind the scenes...

Implementation

Behind the nice veneer of pure parallelism we need an evaluation mechanism.

- At some level it must use OS threads.
- It must guarantee the deterministic properties.
- A good quality implementation is vital for performance and correctness.

Fortunately we have the GHC multi-core runtime system (RTS)

GHC's multi-core runtime system

GHC has a very good runtime system.

- provides lightweight Haskell threads
  (for concurrency support)
- uses one OS thread per core
- lightweight threads scheduled across multiple cores
- well-tuned generational GC
  - per-core young GC generation
  - old GC generation is shared
  - parallel GC for old GC generation

Evaluating in parallel

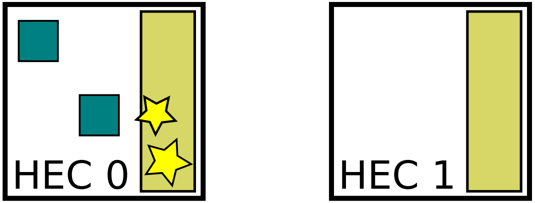
Remember the example `f x + g y`

The RTS has special support for evaluating individual expressions in parallel.

We can take an unevaluated expression and ‘**spark**’ it off. This makes it available to be evaluated on another core.

For example, we could spark off `f x` on another core, and let `g y` be evaluated as normal.

Spark evaluation system

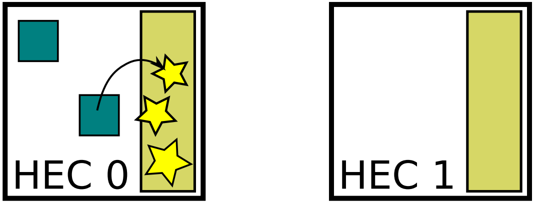


- per-core task queue

Terminology:

- a task is called a ‘spark’
- a task queue is called a ‘spark pool’

Spark evaluation system

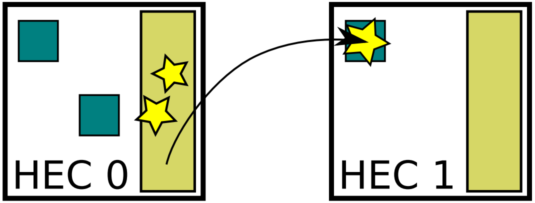


- per-core task queue
- tasks created using par primitive function

Terminology:

- a task is called a ‘spark’
- a task queue is called a ‘spark pool’

Spark evaluation system



- per-core task queue
- tasks created using par primitive function
- tasks run on any available core

Terminology:

- a task is called a ‘spark’
- a task queue is called a ‘spark pool’
- sparks get ‘converted’, meaning evaluated

Well-Typed

Spark evaluation system

The spark evaluation system has quite low overheads:

- the spark pool is a lock-free work stealing queue
- each spark is just a pointer
- evaluation is just calling a function pointer
- no thread startup costs

Low overheads lets us take advantage of more fine grained parallelism.

Spark evaluation system

The spark evaluation system has quite low overheads:

- the spark pool is a lock-free work stealing queue
- each spark is just a pointer
- evaluation is just calling a function pointer
- no thread startup costs

Low overheads lets us take advantage of more fine grained parallelism.

But it's still not free: parallel work granularity is still important.

Well-Typed

The programmers view of expression style parallelism

Deciding what to evaluate in parallel

We said it is always safe to evaluate both parts of $\text{f x} + \text{g y}$ in parallel.

Unfortunately we have no guarantee this will make it run faster.

Deciding what to evaluate in parallel

We said it is always safe to evaluate both parts of $f\ x + g\ y$ in parallel.

Unfortunately we have no guarantee this will make it run faster.

It depends on the granularity: if the amount of work done in parallel overcomes the extra overhead of managing the parallel evaluation.

Deciding what to evaluate in parallel

We said it is always safe to evaluate both parts of `f x + g y` in parallel.

Unfortunately we have no guarantee this will make it run faster.

It depends on the granularity: if the amount of work done in parallel overcomes the extra overhead of managing the parallel evaluation.

Conclusion

Fully automatic parallelism will probably remain a dream.

The programmer has to decide what is worth running in parallel.

Well-Typed

Specifying what to evaluate in parallel

The low level primitive function is called par

- implemented in the RTS by making sparks

It has a slightly strange looking type

$$\text{par} :: a \rightarrow b \rightarrow b$$

Operationally it means

- when the result is needed
- start evaluating the first argument in parallel
- evaluate and return the second argument

Well-Typed

Using par

Using the low level par primitive, we would rewrite $f\ x + g\ y$ as

```
let x' = f x
    y' = g y
in par x' (pseq y' (x' + y'))
```

It turns out we also need a primitive pseq to evaluate sequentially (but the combination of the two is enough).

```
pseq :: a → b → b
```

- evaluate the first argument
- then evaluate and return the second argument

Well-Typed

Parallel evaluation strategies

The par and pseq primitives are very low level, and rather tricky to use.

Haskell provides a library of higher level combinators **parallel strategies**. A strategy describes how to evaluate things, possibly in parallel.

```
type Strategy a
using :: a → Strategy a → a
```

There are a few basic strategies

```
r0 :: Strategy a        -- none
rseq :: Strategy a      -- evaluate sequentially
rpar :: Strategy a      -- evaluate in parallel
```

Well-Typed

Parallel evaluation strategies

Strategies can be composed together to make custom strategies.

For example, a strategy on lists

```
parList :: Strategy a → Strategy [a]
```

- given a strategy for the list elements,
- evaluate all elements in parallel,
- using the list element strategy.

We would use this if we had a list of complex structures where there was further opportunities for parallelism within the elements. In simple cases we would just use `parList rseq`.

Strategies can help with granularity

It is very common that the structure of our data doesn't give a good granularity of parallel work.

We can use or write strategies that split or coalesce work into better sized chunks.

For example:

```
parListChunk :: Int → Strategy a → Strategy [a]
```

- takes chunks of N elements at a time
- each chunk is evaluated in parallel
- within the chunk they're evaluated serially

So it increases granularity by a factor of N.

Strategies can help with granularity

Example from a real program

```
reports `using` parListChunk 10 rseq
```

- one line change to the program
- scaled near-perfectly on 4 cores

So we can get excellent results, but it’s often still tricky.

Parallel algorithm skeletons

Strategies try to completely separate the parallel evaluation from the algorithm. That works well for data structures (like lists, trees, arrays etc) but doesn't work everywhere.

Sometimes we have to mix the parallel evaluation in with the algorithm.

We can still use general algorithm skeletons, like divide and conquer or map-reduce.

A map-reduce parallel skeleton

```
mapReduce :: Int →              -- threshold
             (Int, Int) →       -- bounds
             Strategy a →       -- strategy
             (Int → a) →        -- map
             ([a] → a) →        -- reduce
             a
```

This version is for functions on integer ranges

- recursively subdivide range until we hit the threshold
- for each range chunk, map function over range
- for each range chunk, reduce result using given strategy
- reduce all intermediate results

Having the threshold is important, or we would usually end up with far too small parallel granularity.

Profiling tools

Parallelism is still hard

Even with all these nice techniques, getting real speed ups can still be hard.

There are many pitfalls

- exposing too little parallelism, so cores stay idle
- exposing too much parallelism
- too small chunks of work, swamped by overheads
- too large chunks of work, creating work imbalance
- speculative parallelism that doesn't pay off

Sparks have a few more

- might spark an already-evaluated expression
- spark pool might be full

We need to profile to work out the cause.

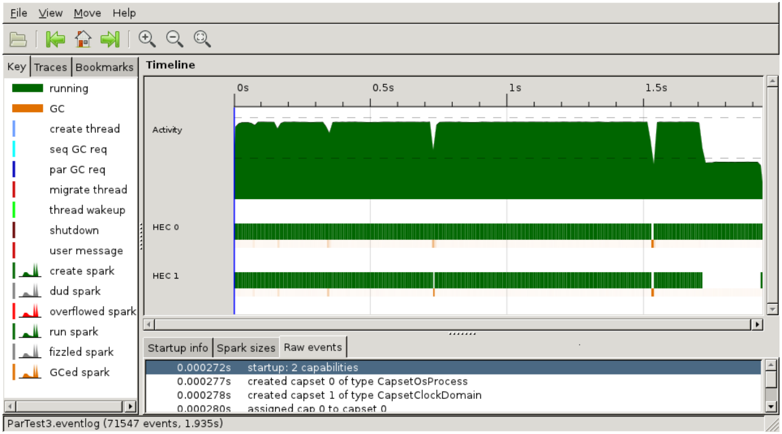
Well-Typed

ThreadScope and event tracing

GHC's RTS can log runtime events to a file

- very low profiling overhead

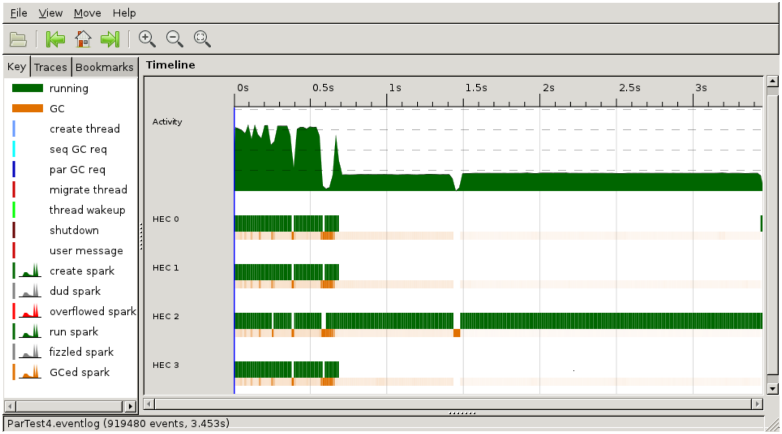
ThreadScope is a post-mortem eventlog viewer



```
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)

ThreadScope and event tracing



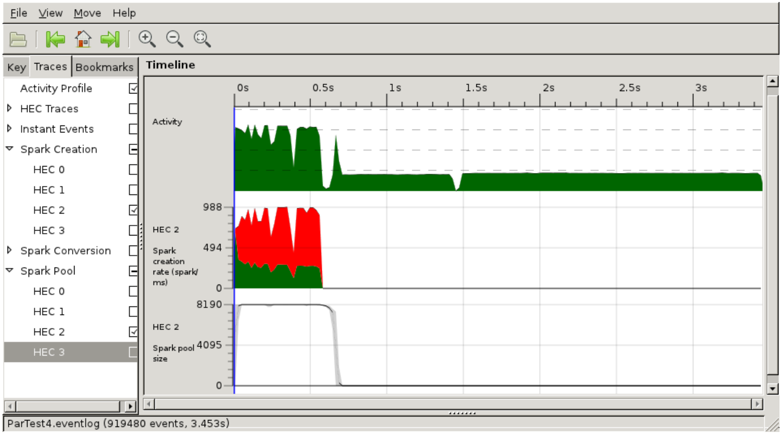
ParTest4.eventlog (919480 events, 3.453s)

ThreadScope shows us

- Overall utilisation across all cores
- Activity on each core
- Garbage collection

Well-Typed

ThreadScope and event tracing



Also some spark-related graphs:

- Sparks created and executed
- Size of spark pool
- Histrogram of spark evaluation times (i.e. parallel granularity)

Data parallelism with Repa

Introducing Repa

A library for data-parallelism in Haskell:

- high-level parallelism
- mostly-automatic
- for algorithms that can be described in terms of operations on arrays

Notable features

- implemented as a library
- based on dense multi-dimensional arrays
- offers “delayed” arrays
- makes use of advanced type system features

Introducing Repa

A library for data-parallelism in Haskell:

- high-level parallelism
- mostly-automatic
- for algorithms that can be described in terms of operations on arrays

Notable features

- implemented as a library
- based on dense multi-dimensional arrays
- offers “delayed” arrays
- makes use of advanced type system features

Demo `http://www.youtube.com/watch?v=UGN0GxGEDsY`

Well-Typed

Repa’s arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa’s array type looks as follows:

```
data family Array r sh e    -- abstract
```

Repa’s arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa’s array type looks as follows:

```
data family Array r sh e   -- abstract
```

- there are **three** type arguments;

Repa’s arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa’s array type looks as follows:

```
data family Array r sh $\boxed{e}$   -- abstract
```

- there are **three** type arguments;
- the final is the element type;

Repa's arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;

Repa's arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;
- the second the **shape**.

Repa's arrays

Arrays are the key data type in Repa. It relies heavily on types to keep track of important information about each array.

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;
- the second the **shape**.

But what are **representation** and **shape**?

Array shapes

Repa can represent dense multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index.**

Array shapes

Repa can represent dense multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index**.

```
type DIM1
type DIM2
 ...
```

So `DIM2` is (roughly) the type of pairs of integers.

Array representations

Repa distinguishes two fundamentally different states an array can be in:

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;
- a **delayed** array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;
- a **delayed** array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Let's look at the "why" and the delayed representation in a moment.

Array representations

The standard **manifest** representation is denoted by a type argument `U` (for unboxed).

For example, making a manifest array from a list

```
fromListUnboxed
   :: (Shape sh, Unbox a) ⇒ sh → [a] → Array U sh a
```

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
```

Operations on arrays

$$\begin{aligned}\mathrm{map} :: {} & (\mathrm{Shape\ sh}, \mathrm{Repr\ r\ a}) \Rightarrow \\ & (\mathrm{a} \to \mathrm{b}) \to \mathrm{Array\ r\ sh\ a} \to \mathrm{Array\ D\ sh\ b}\end{aligned}$$

This function returns a **delayed** array (D).

Why delayed arrays?

We want to describe our array algorithms by using combinations of standard array bulk operators

- nicer style than writing monolithic custom array code
- but also essential for the automatic parallelism

But if we end up writing code like this

```
(map f ∘ map g) arr
```

Then we are making a full intermediate copy for every traversal (like `map`).

Performing **fusion** becomes essential for performance – so important that we'd like to make it **explicit** in the type system.

The delayed arrays are what enables automatic fusion in Repa.

Delayed arrays

Delayed arrays are internally represented simply as **functions**:

```
data instance Array D sh e = ADelayed sh (sh → e)
```

- Delayed arrays aren't really arrays at all.
- Operating on an array does not create a new array.
- Performing another operation on a delayed array just performs function composition.
- If we want to have a manifest array again, we have to **explicitly force** the array.

Creating delayed arrays

From a function:

```
fromFunction :: sh → (sh → a) → Array D sh a
```

Directly maps to `ADelayed`.

The implementation of map

```
map :: (Shape sh, Repr r a)
     ⇒ (a → b) → Array r sh a → Array D sh b
map f arr = case delay arr of
                ADelayed sh g → ADelayed sh (f ∘ g)
```

Well-Typed

The implementation of map

```
map  :: (Shape sh, Repr r a)
     ⇒ (a → b) → Array r sh a → Array D sh b
map f arr = case delay arr of
                ADelayed sh g → ADelayed sh (f ∘ g)
```

Many other functions are only slightly more complicated:

- think about pointwise multiplication `(*.)`,
- or the more general `zipWith`.

Forcing delayed arrays

Sequentially:

```
computeS :: (Fill r1 r2 sh e) ⇒
            Array r1 sh e → Array r2 sh e
```

Forcing delayed arrays

Sequentially:

$$\text{computeS} :: (\text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{Array r2 sh e}$$

In parallel:

$$\text{computeP} :: (\text{Monad m}, \text{Repr r2 e}, \text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{m (Array r2 sh e)}$$

This is the only place where we specify parallelism.

Well-Typed

Forcing delayed arrays

Sequentially:

```
computeS :: (Fill r1 r2 sh e) ⇒
            Array r1 sh e → Array r2 sh e
```

In parallel:

```
computeP :: (Monad m, Repr r2 e, Fill r1 r2 sh e) ⇒
            Array r1 sh e → m (Array r2 sh e)
```

This is the only place where we specify parallelism.

Key idea

Describe the array we want to compute (using delayed arrays).

Compute the array in parallel.

“Automatic” parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don’t have to concern the user.

So Repa deals with the granularity problem for us (mostly).

Repa Summary

- Describe algorithm in terms of arrays
- The true magic of Repa is in the `computeP` -like functions, where parallelism is automatically handled.
- Haskell's type system is used in various ways:
  - Adapt the representation of arrays based on it's type.
  - Keep track of the shape of an array, to make fusion explicit.
  - Keep track of the state of an array.
- A large part of Repa's implementation is actually quite understandable.

Summary

A range of parallel styles

The ones we looked at

- expression style
- data parallel style
- and yes, concurrent

These are now fairly mature technologies

Others worth mentioning

- data flow style
- nested data parallel
- GPU

Practical experience

We ran a 2-year project with MSR to see how real users manage with parallel Haskell

- mostly scientific applications, simulations
- one group working on highly concurrent web servers
- mostly not existing Haskell experts

No significant technical problems

- we helped people learn Haskell
- developed a couple missing libraries
- extended the parallel profiling tools

Practical experience

Los Alamos National Laboratory

- high energy physics simulation
- existing mature single-threaded C/C++ version
- parallel Haskell version 2x slower on one core
  but scaled near perfectly on 8 cores
- Haskell version became the reference implementation
  C version ‘adjusted’ to match Haskell version
- also distributed versions: Haskell/MPI and Cloud Haskell

Happy programmers!

That’s it!

Thanks!

Questions?

Well-Typed

Repa example

Example: 1-D Poisson solver

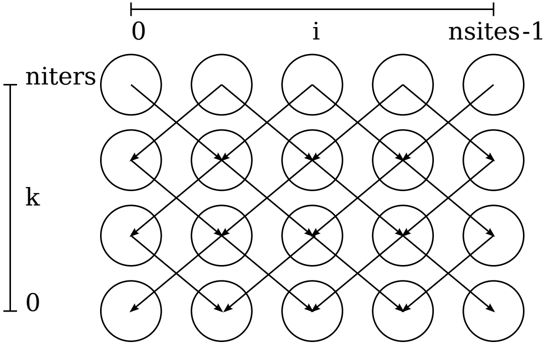
Specification as code

```
phi k i | k ≡ 0                = 0
        | i < 0 ∨ i ⩾ sites   = 0
        | otherwise            =
            (phi (k − 1) (i − 1) + phi (k − 1) (i + 1)) / 2
              + h / 2 ∗ rho i

rho i   | i ≡ sites ‘div‘ 2 = 1
        | otherwise         = 0

h = 0.1    -- lattice spacing

n = 10     -- number of sites
```

Well-Typed

Example: 1-D Poisson solver – contd.

Data dependencies



- whole row could be calculated in parallel
- other parallel splits not so easy and will duplicate work

Example: 1-D Poisson solver – contd.

Serial array version of the inner loop

```
phiIteration :: UArray Int Double → UArray Int Double
phiIteration phik1 =
    array (0, n + 1) [(i, phi i) | i ← [0 . . n + 1]]
  where
    phi i | i ≡ 0 ∨ i ≡ n + 1 = 0
    phi i = (phik1 ! (i − 1) + phik1 ! (i + 1)) / 2
              + h / 2 ∗ rho i
```

- uses immutable arrays
- new array defined in terms of the old array
- we extend the array each end by one to simplify boundary condition

Example: 1-D Poisson solver – contd.

Parallel array version of the inner loop

```
phiIteration :: Array U DIM1 Double → Array U DIM1 Double
phiIteration phik1 = computeP (fromFunction (extent phik1) phi)
  where
    phi (Z :. i) | i ≡ 0 ∨ i ≡ n + 1 = 0
    phi (Z :. i) = (phik1 ! (i − 1) + phik1 ! (i + 1)) / 2
                   + h / 2 ∗ rho i
```

- define the new array as a delayed array
- compute it in parallel

More performance tricks

A few tricks gets us close to C speed

- Unsafe indexing
- Handelling edges separately

Comparison with C OpenMP version

Cores	OpenMP		Repa	
	time	speedup	time	speedup
1	22.0s	1×	25.3s	1×
4	6.9s	3.2×	11.4s	2.2×
8	5.3s	4.2×	8.4s	3.0×

Larger Repa example: Matrix multiplication

Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-example` package which contains more than just this example.

Start with the types

We want something like this:

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
```

- We inherit the `Monad` constraint from the use of a parallel compute function.
- We work with two-dimensional arrays, it's an additional prerequisite that the dimensions match.

Well-Typed

Strategy

We get two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`:

- we expect `w1` and `h2` to be equal,
- the resulting matrix will have shape `Z :. h1 :. w2`,
- we have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays,
- for each of these pairs, we have to take the sum of the products,
- and these results determine the values of the result matrix.

Well-Typed

Strategy

We get two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`:

- we expect `w1` and `h2` to be equal,
- the resulting matrix will have shape `Z :. h1 :. w2`,
- we have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays,
- for each of these pairs, we have to take the sum of the products,
- and these results determine the values of the result matrix.

Some observations:

- the result is given by a **function**,
- we need a way to **slice** rows or columns out of a matrix,

Starting top-down

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction   (Z :. h1 :. w2)
                             (λ(Z :. r   :. c   ) → ...)
```

Well-Typed

Slicing

A quite useful function offered by Repa is `backpermute`:

```
backpermute :: (Shape sh1, Shape sh2, Repr r e) ⇒
               sh2 →                   -- new shape
               (sh2 → sh1) →           -- map new index to old index
               Array r sh1 e → Array D sh2 e
```

- We compute a delayed array simply by saying how each index can be computed in terms of an old index.
- This is trivial to implement in terms of `fromFunction`.

Slicing – contd.

We can use backpermute to slice rows and columns.

```
sliceCol   :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol  c a =
   let (Z :. h :. w) = extent a
   in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceRow r  a =
   let (Z :. h :. w) = extent a
   in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

Slicing – contd.

We can use backpermute to slice rows and columns.

```
sliceCol   :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol   c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

```
>>> computeUnboxedS (sliceCol 3 example)
AUnboxed (Z :. 2) (fromList [4, 9])
```

Note that sliceCol and sliceRow do not actually create a new array unless we force it!

Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

```
slice :: (Slice sl, Shape (SliceShape sl), Shape (FullShape sl),
          Repr r e) ⇒
         Array r (FullShape sl) e → sl → Array D (SliceShape sl) e
```

A member of class `Slice`:

- looks similar to a member of class `Shape`,
- but describes **two** shapes at once, the orginal and the sliced.

Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

```
slice :: (Slice sl, Shape (SliceShape sl), Shape (FullShape sl),
          Repr r e) ⇒
         Array r (FullShape sl) e → sl → Array D (SliceShape sl) e
```

A member of class `Slice`:

- looks similar to a member of class `Shape`,
- but describes **two** shapes at once, the orginal and the sliced.

```
sliceCol, sliceRow :: Repr r e ⇒
                      Int → Array r DIM2 e → Array D DIM1 e
sliceCol c  a = slice a (Z :. All :. c  )
sliceRow r a = slice a (Z :. r    :. All)
```

Well-Typed

Putting everything together

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction (Z :. h1 :. w2)
                (λ(Z :. r :. c) →
                   sumAllS (sliceRow r m1 ∗. sliceCol c m2)
                ))
```

That's all. Note that we compute no intermediate arrays.

Well-Typed