Cloud Haskell

Duncan Coutts

28th February 2012, FiTA

About us

Well-Typed LLP

- founded in spring 2008
- Haskell consultancy
- support, planning, development, training
- help a wide range of clients: startups to multinationals

Cloud Haskell

What's it all about?

- slogan might be “Erlang for Haskell” (as a library)
- distributed programming in Haskell
- a programming model + an implementation

Cloud Haskell

What's the point?

- to let you program a cluster as a whole
- or a data centre
- or a bunch of VMs rented from Amazon / Azure / ...
  (hence the cheesy "Cloud Haskell" marketing terminology)

Other people's good ideas

Papers

- Jeff Epstein, Andrew Black and Simon Peyton Jones, *Towards Haskell in the Cloud*, Haskell Symposium 2011
- Jeff Epstein, *Functional programming for the data centre*, MPhil thesis, 2011

Distributed programming?

Is that like client / server?

Distributed programming?

Is that like client / server?

What's so special? I can do that using the network library.

Distributed programming?

Is that like client / server?

What's so special? I can do that using the network library.

Key idea

Program the cluster as a whole, not individual nodes

Local vs global perspective

Traditional approach gives you a local node perspective

- you're in a box, reacting to messages from your peers
- encourages a homogeneous style, all nodes do the same (or write different programs for different tasks, e.g. master/worker)

Erlang and Cloud Haskell give you a global perspective

- you can see all nodes,
- run one program with easy access to them all
- you fire off lots of processes, placing them on nodes
- doesn't need to be homogeneous, it's easy to have different processes on different nodes

Different use cases

Traditional local approach

- client / server
- loose coupling
- programs and nodes can be written and operated by different organisations
- requires well-defined protocols

Global approach

- "I have 50–5000 machines I want to run this program on"
- requires control over all nodes
- internal protocol doesn't matter

Programming model

- explicit concurrency
- lightweight processes
- no state shared between processes
- asynchronous message passing

Some people call this the “actor model”

Existing instances

This isn't a new idea

Erlang, 1986

"Erlang is a general-purpose programming language and runtime environment. Erlang has built-in support for concurrency, distribution and fault tolerance."

Eden, 1996

"Eden extends Haskell with a small set of syntactic constructs for explicit process specification and creation."

Glasgow distributed Haskell (GdH), 1999

Minimal superset of parallel and concurrent Haskell, providing explicitly placed I/O threads making use of the individual resources of each machine

Implementations

Seems to need a new RTS (or major surgery to an existing one)

Erlang

"Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance."

Eden

"The changes of the GHC concern the back end of the compiler and mainly the runtime system (RTS). The necessary modifications have been designed as orthogonal additions to the existing implementation. The implementation re-uses simplified kernel parts of the parallel functional RTS GUM, the implementation of GpH (Trinder et al., 1996)."

Implementations

What's the problem with adding it directly into the RTS?

- it's a lot of work!
- ongoing maintenance burden
- yet more complexity
- RTS is monolithic
- everyone has to trust the RTS

Implementations

What's the problem with adding it directly into the RTS?

- it's a lot of work!
- ongoing maintenance burden
- yet more complexity
- RTS is monolithic
- everyone has to trust the RTS

Counterpoint

But we did it for SMP parallelism...

Implementations

Can we avoid adding distribution into the RTS?

Implementations

Can we avoid adding distribution into the RTS?

What distributed parallelism features are usually in the RTS?

- networking
- serialisation and transfer of arbitrary data types
  - including functions!
- lightweight processes and scheduler
- starting processes on remote nodes

Implementations

Can we avoid adding distribution into the RTS?

What distributed parallelism features are usually in the RTS?

- networking
- serialisation and transfer of arbitrary data types
    - including functions!
- lightweight processes and scheduler
- starting processes on remote nodes

Question

What is stopping us putting all this in a library?

The breakthrough

Question

What is stopping us putting all this in a library?

Jeff, Andrew and Simon have an answer and a solution

The breakthrough

Question

What is stopping us putting all this in a library?

Jeff, Andrew and Simon have an answer and a solution

Their answer

- It's the function serialisation!

The breakthrough

Question

What is stopping us putting all this in a library?

Jeff, Andrew and Simon have an answer and a solution

Their answer

- It's the function serialisation!

Their solution

- **not** "just put function serialisation in the RTS"
- add a minimal language extension
  - labels for 'static' functions without free variables
  - function closures can then be serialised within the language

The Cloud Haskell design

Basic approach

- must be implementable as a library
  - e.g. no distributed `MVar` as in GdH
- should have a clear cost model (explicit communication)
- if in doubt, do it the way Erlang does it

(Other distributed middleware designs are also possible)

The Cloud Haskell design

Highlights

- lightweight processes as a `Process` monad
  - piggybacks on GHC's lightweight threads
- processes addressed by abstract `ProcessId`
  - can only send if you know the `ProcessId`
  - cannot create `ProcessId` from nothing
  - can pass `ProcessId`s around
- can send and receive messages of any type
- can spawn new processes on any node

The Cloud Haskell design

The core API

```
instance Monad Process
instance MonadIO Process

data ProcessId
data NodeId

class (Typeable a, Binary a) ⇒ Serializable a

send   :: Serializable a ⇒ ProcessId → a → Process ()
expect :: Serializable a ⇒ Process a

spawn :: NodeId → Closure (Process ()) → Process ProcessId

getSelfPid  :: Process ProcessId
getSelfNode :: Process NodeId
```

Error handling style

Errors are everywhere in distributed programming

Cloud Haskell steals Erlang's solution

- let processes fail
  - communication loss counts as failure
- notify interested processes
  - often they just fail too (linked processes)
  - common pattern is to monitor & restart

```
linkProcess    :: ProcessId → Process ()
monitorProcess :: ProcessId → ProcessId → MonitorAction →
                  Process ()
```

Like Erlang, but not too much

Typed FP people can't resist adding typed channels

$$
\begin{aligned}
&\text{newChan} :: \text{Serializable a} \Rightarrow \text{Process (SendPort a, ReceivePort a)}\\
&\text{sendChan} \quad :: \text{Serializable a} \Rightarrow \text{SendPort a} \rightarrow \text{a} \rightarrow \text{Process ()}\\
&\text{receiveChan} :: \text{Serializable a} \Rightarrow \text{ReceivePort a} \rightarrow \text{Process a}
\end{aligned}
$$

- more type safety
- two ways of doing everything
- unclear if one approach will dominate

The existing prototype implementation

- written by Jeff Epstein for his MPhil thesis
- it works!
- code is on hackage
- uses TCP/IP
- k-means example benchmarked with Amazon EC2
- static function language extension not yet implemented (prototype uses Template Haskell magic)

Ping pong example

```
newtype Ping = Ping ProcessId deriving (Binary, Typeable)

ping :: Process ()
ping = do self ← getSelfPid
          Ping partner ← expect
          send partner (Ping self)
          say "ping!"
          ping

initialProcess _ = do nid   ← getSelfNode
                      ping1 ← spawn nid ping__closure
                      ping2 ← spawn nid ping__closure
                      send ping1 (Ping ping2)

$(remotable ['ping])   -- Template Haskell magic

main = remoteInit (Just "config") [__remoteCallMetaData]
                  initialProcess
```

What we’ve been up to

Advantages of the library approach

Not building everything into the RTS has other advantages

- much greater flexibility
- easier to experiment
- can use different implementations in different environments

Library author has control over

- serialisation approach
- network data transport layer
- network configuration and performance parameters
- communication network topology
- peer node discovery or creation

Example use cases

Distributed parallelism covers diverse use cases

- traditional clusters
  - ethernet + IP (TCP or UDP)
  - local peer discovery
  - or config via cluster job scheduler
- HPC clusters
  - infiniband and other exotic non-IP networks
  - latency is critical
- supercomputer-scale parallelism
  - ‘interesting’ scaling issues with 10k+ nodes
- VMs in the cloud
  - nodes can be created on demand
- large multi-core servers
  - shared memory or pipes

Erlang anecdote

HPC expert looked into using Erlang for HPC clusters
(think 1000's of nodes, infiniband networking)

Conclusions:

- not suitable for HPC
- interpreter too slow
- networking layer too inflexible
  - UDP/IP networking built into the RTS
  - minimal control over network tuning parameters
  - implementation not designed to scale to 1000's of nodes

Lesson

Practical negative consequences of building it all into the RTS

Exploiting the library advantage

Goal

Exploit the advantages of the library approach to better handle the diverse range of use cases

Axes of variation between use cases

- network data transport layer (hardware and protocol)
  - IP
  - exotic non-IP HPC networks (infiniband and faster)
  - shared memory or local pipes
- network tuning parameters
  - totally different between IP, HPC, shared memory
  - makes a big performance difference (especially for HPC)
  - ideal tuning depends on application and network environment

Exploiting the library advantage

Yet more axes of variation between use cases

- How to start your executable on each machine
  - remote login via ssh
  - cloud service API
  - cluster job scheduler
- How to configure each node
  - via ssh from master node
  - config files, env vars, string and glue
  - config distributed via cluster job scheduler
- How to find initial peers or all peers
  - discover dynamically on LAN
  - known from config
  - cluster job scheduler
  - peers created in new VMs
  - in large clusters, each node may only know a few peers

A new Cloud Haskell implementation

We are working on a new implementation

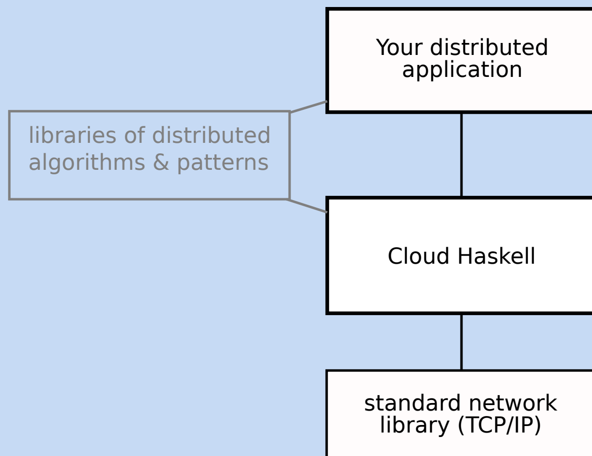
- keeping the same user-facing API
  (except peer node discovery)
- new networking layer(s)
- new internals
- goals are flexibility, robustness & performance

A new Cloud Haskell implementation

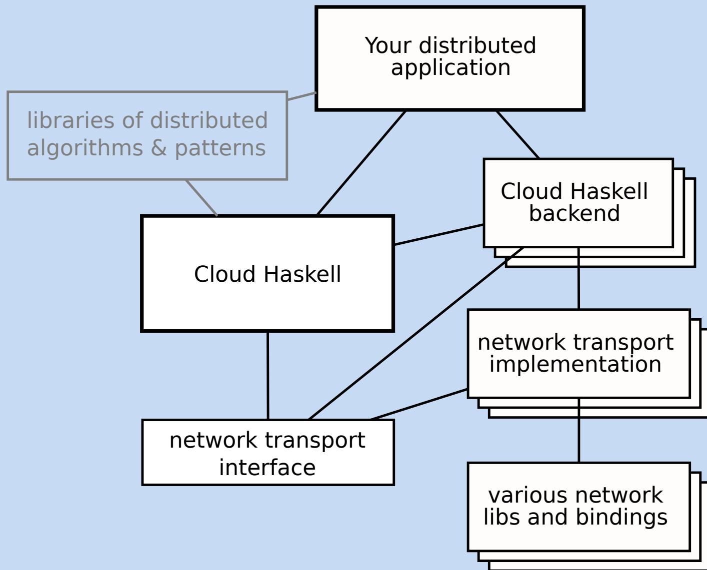
Key differences vs the existing implementation

- swappable network transport layer
- multiple backends to handle
  - selection of transport implementation
  - initialisation
  - configuration
  - peer discovery / creation

Existing prototype design



New internal design



Application initialisation

Initialisation sequence looks something like

```
import Control.Distributed.Process
import Control.Distributed.Process.TCP

init :: (...) → Process () → IO ()
init config initialProcess = do
  transport  ← mkTcpTransport config
  localnode ← newLocalNode transport
  runProcess localnode initialProcess
```

- initialise a transport, with some transport-specific config
- initialise the local Cloud Haskell node
- run the initial process
- various ways to slice and dice this

Speculation

Potential to push this technology a long way

- try other parallel models
  - apply experience from GdH & Eden
  - deterministic parallel e.g. DPH or Par monad
  - map/reduce
  - distributed transactions
- backends to conveniently target various environments
  - cloud services
  - ‘SSH clusters’
  - conventional multi-core
- escape hatch if we cannot make shared GC heaps scale
- future “cloud on a chip” hardware
  (shared memory without cache coherency)

If we have time...

Network transport interface

```
data Transport = Transport {
  newConnection :: IO TargetEnd,
  deserialize   :: ByteString → Maybe Address
}
data Address = Address {
  connect   :: IO SourceEnd,
  serialize :: ByteString
}
data SourceEnd = SourceEnd {
  send    :: [ByteString] → IO ()
}
newtype TargetEnd = TargetEnd {
  receive :: IO [ByteString],
  address :: Address
}
```

The configuration problem

Networking experts tell us

- network hardware & protocol parameters can be crucial for performance (e.g. TCP/IP socket options)
- sets of parameters differ between protocols (IP, infiniband, shared memory)
- the ideal tuning for the parameters varies
  - with application
  - between different connections within an application
  - with network environment (hardware and topology)

The configuration problem

Networking experts tell us

- network hardware & protocol parameters can be crucial for performance (e.g. TCP/IP socket options)
- sets of parameters differ between protocols (IP, infiniband, shared memory)
- the ideal tuning for the parameters varies
  - with application
  - between different connections within an application
  - with network environment (hardware and topology)

Question

How do we get the ideal tuning **and** have code be reusable with multiple network backends?

The configuration problem

```
data Transport = Transport {
  newConnection :: Hints → IO TargetEnd,
  ...
}
```

```
mkTcpTransport :: TCPConfig → IO Transport

data TCPConfig = TCPConfig {
  socketConfiguration :: Hints → IP.Address → IP.Address
                                → IP.SocketOptions
}
```

- config remains backend specific and app-global
- per-connection parameters can depend on per-connection hints
- somewhat CSS-like separation

Thanks!

Questions?