Cloud Haskell

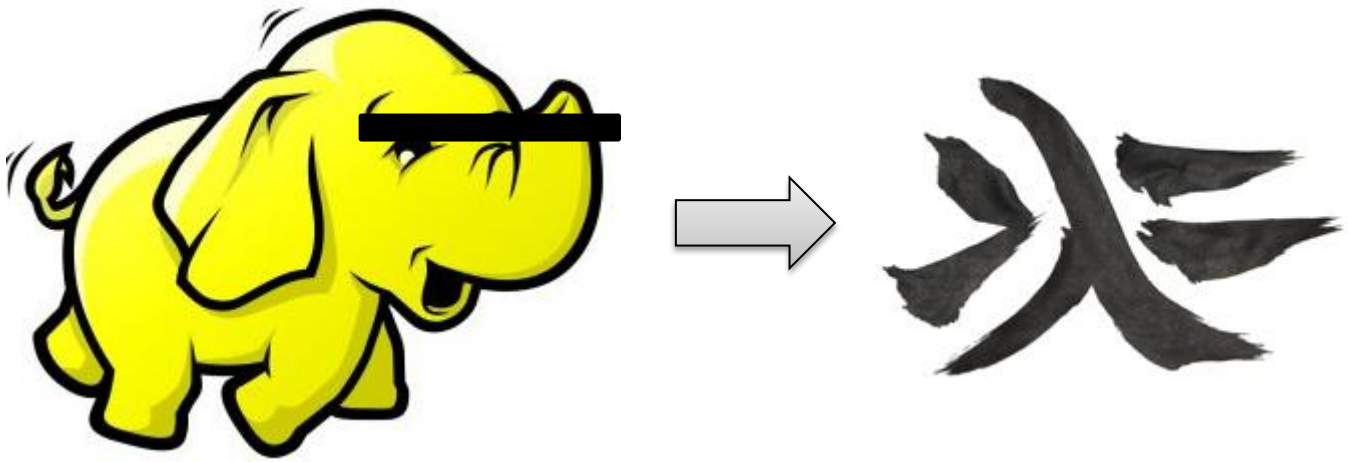
田中英行
tanaka.hideyuki@gmail.com
第0回スタートHaskell LT

時はまさにクラウド時代…

- 猫も杓子もクラウド
  - チューニングするより台数増やす
- 猫も杓子もHadoop/MapReduce
- しかしJava…
  - Map Reduceなのに！
- クラウド時代のHaskellのために！

概要

- クラウド上でHaskellを動かすためのライブラリ
- 並列・分散環境でどのようにHaskellを動かすか？

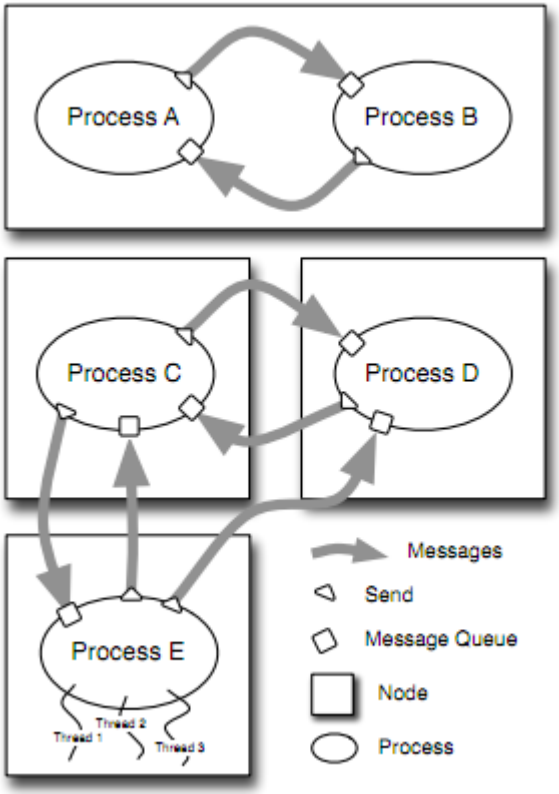


Cloud Haskellとは？

- Erlang風のインターフェースをHaskellに
  - 前は Erlang in Haskell と呼ばれていた
- メッセージパッシングなインターフェース
- 分散環境でのfault-tolerantなタスク管理
- 現在はプロセスレイヤのみ
  - http://www.cl.cam.ac.uk/~jee36/remote.pdf
  - https://github.com/jepst/CloudHaskell
- タスクレイヤはまだ未実装
  - Skywriting in Haskell

プロセスのイメージ

- 一つのノードに複数プロセス
- プロセス間でメッセージ
  - ノード超えてもOK
- プロセスは複数のスレッドからなる場合もある
  - Erlangとは違う



プロセス

- Cloud HaskellのプロセスはHaskellのスレッドとして表現される
- Haskellのスレッドはグリーンスレッド
  - 非常に軽量
- 単にスレッドを使っておけばErlangの軽量プロセスをエミュレートできる

書き方のイメージ(1)

```
ping() → receive
           {pong, Partner} →
             Partner ! {ping, self()}
         end,
         ping().
```

```
ping :: ProcessM ()
ping = do Pong partner ← expect
          self ← getSelfPid
          send partner (Ping self)
          ping
```

書き方のイメージ(2)

- 複数の種類のメッセージ受け取る
  - ちょっとかっこ悪い…

```
math() →
    receive
        {add, Pid, Num1, Num2} →
            Pid ! Num1 +Num2;
        {divide, Pid, Num1, Num2} when Num2 ≠0 →
            Pid ! Num1 /Num2;
        {divide, Pid, _, _} →
            Pid ! div_by_zero
    end,
    math().
```

```
-- omitted: Serializable instances for Add, Divide, and
     DivByZero types
math :: ProcessM ()
math =
  receiveWait
    [ match   (λ(Add pid num1 num2) →
                 send pid (num1 +num2)),
      matchIf (λ(Divide _ _ num2) →num2 ≠0)
              (λ(Divide pid num1 num2) →
                 send pid (num1 /num2)),
      match   (λ(Divide pid _ _) →
                 send pid DivByZero) ]
      ≫ math
```

課題

- ノード間クロージャ転送
  - これができないと直交的にノード間転送プログラミングできない
  - Haskellではクロージャはファーストクラスなので
- クロージャはシリアライズ可能ではない
  - どうするか？

クロージャはなぜ シリアライズできないのか？

- 例えばリストの等値性
  - 部分型もそれを満たす必要がある

```
instance Eq a ⇒ Eq [a] where
  (x:xs) == (y:ys) = x == y && xs == ys
```

- 方やクロージャ
  - 部分型が型に現れていない？？？

```
instance Serializable (types of the free variables of
  an a→b) ⇒ Serializable (a→b) where ...
```

どう解決するか

- 処理系がすべての値のシリアライズをサポート
  - ありうる選択肢
- でも・・・
  - すべての値がシリアライズできるわけではない
  - 例えばロック変数型とか
- そういうアプローチとってる処理系もある
  - Erlangとか

Cloud Haskellでは

- Staticという概念を導入
  - 環境を含まないクロージャという意味
- Staticは、関数ポインタと同等
  - シリアライズにはポインター つ送ればOK

StaticのTyping Rule

- Staticは型システムを拡張することによってstaticに判定可能
  - コンパイラに手を入れる必要がある

$$\Gamma \quad ::= \quad \overline{x :_\delta \sigma}$$

$$\delta \quad ::= \quad \mathsf{S} \mid \mathsf{D}$$

$$\Gamma\downarrow \; = \{x :_\mathsf{s} \sigma \mid x :_\mathsf{s} \sigma \in \Gamma\}$$

$$\frac{\Gamma\downarrow \vdash e : \tau}{\Gamma \vdash \mathsf{static}\ e : \mathsf{Static}\ \tau} \qquad \text{(Static intro)}$$

$$\frac{\Gamma \vdash e : \mathsf{Static}\ \tau}{\Gamma \vdash \mathsf{unstatic}\ e : \tau} \qquad \text{(Static elim)}$$

Figure 3. Typing rules for Static

クロージャの送り方

- クロージャ＝Static＋Environment

```
data Closure a where    -- Wrong
  MkClosure :: Static (env → a) → env → Closure a
```

- こういう定義は？
  - envをシリアライズできない
  - だめぽ(´･_･`)

問題もう一つ

- じゃあこれは？

```
data Closure a where    -- Still wrong
  MkClosure :: Serializable env ⇒
               Static (env → a) → env → Closure a
  deriving (Typeable)
```

  - シリアライズはできるけど
  - デシリアライズはどうするの？
  - デシリアライズ時にenvの型解らない…

  - やっぱりだめぽ(´･_･`)

解決法

- 予めシリアライズしておけばいいじゃない

```
data Closure a where   -- Right
  MkClosure :: Static (ByteString → a) →
               ByteString → Closure a
```

  - これでOK。

あとは

- とりあえずErlang風のインターフェースができました
- アプリ作りました
- ベンチ取りました

サンプルアプリ

```
module Main where
-- omitted: module imports
data CounterMessage = CounterQuery ProcessId
                    | CounterShutdown
                    | CounterIncrement
                    deriving (Typeable)
-- omitted: Serializable instance of CounterMessage

counterLoop :: Int → ProcessM ()
counterLoop val
  = do val' ← receiveWait [match counterCommand]
       counterLoop val'
  where
    counterCommand (CounterQuery pid)
      = do send pid val
           return val
    counterCommand CounterIncrement = return (val +1)
    counterCommand CounterShutdown = terminate

$( remotable ['counterLoop] )

increment :: ProcessId → ProcessM ()
increment cpid = send cpid CounterIncrement

shutdown :: ProcessId → ProcessM ()
shutdown cpid = send cpid CounterShutdown

query :: ProcessId → ProcessM Int
query counterpid =
  do mypid ← getSelfPid
     send counterpid (CounterQuery mypid)
     expect

go "MASTER" =
  do aNode ←liftM (head . flip
       findPeerByRole "WORKER") getPeers
     cpid ← spawn aNode ($(mkClosure 'counterLoop) 0)
     increment cpid
     increment cpid
     newVal ←query cpid
     say (show newVal)  -- prints out 2
     shutdown cpid

go "WORKER" =
  receiveWait []

main = runRemote (Just "config")
       [Main.__remoteTable] go
```

パフォーマンス

- K-meansを実装、Hadoopと比較

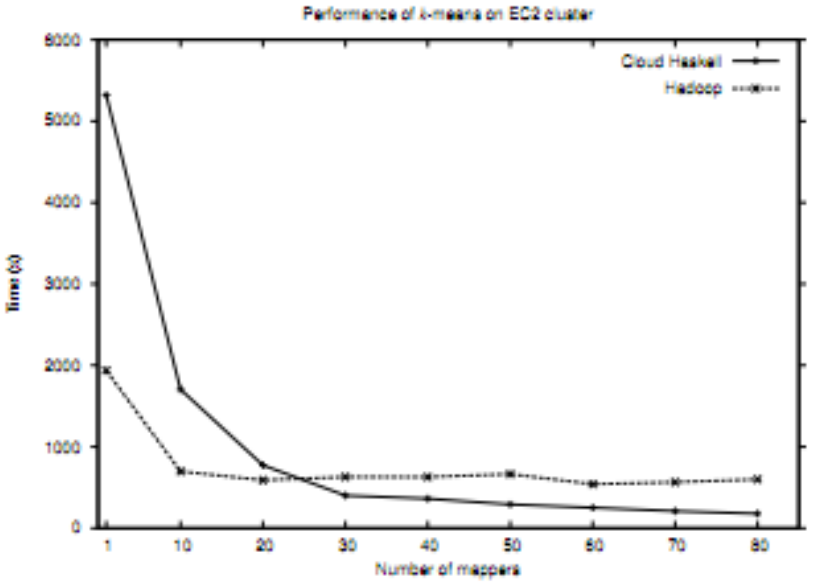


Figure 4. The run-time of the k-means algorithm, implemented under Cloud Haskell and Hadoop. The input data was one million 100-dimensional data points.

まとめ

- 分散環境でHaskell動かす
- とりあえずクロージャが転送できるように技術的課題を解決
- プロセスレイヤを実装
- これから更に上の層が充実するはず！