# Haskell distributed parallel Haskell

Patrick Maier, Rob Stewart, **Phil Trinder**, Majed Al Saeed

{P.Maier,R.Stewart,P.W.Trinder}@hw.ac.uk

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Haskell Implementors Workshop, 14 Sep 2012

# HdpH — What is it and Why?

**HdpH = Haskell distributed parallel Haskell** is

- a parallel Haskell (language extension)
- for distributed memory
- implemented entirely in Haskell (+ GHC extensions).

**What is HdpH going to be used for?**

- The HPC-GAP project aims to scale parallel symbolic computation to high-performance computers, e.g. to HECToR, the UK's supercomputer with currently 90,000 cores.
- Concretely, HdpH will coordinate thousands of instances of the GAP computer algebra system.

**Requirements on HdpH**

- Dynamic work distribution
- Locality control
- Fault tolerance

# HdpH — Key Features

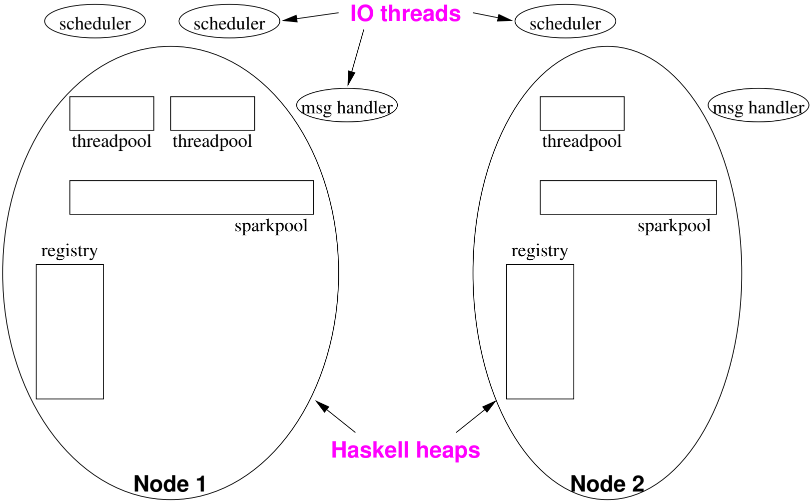
- Monadic language for uniform shared- and distributed-memory parallelism
    - Extends the Par monad [Marlow et al, Haskell 2011]

- Polymorphic serialisable closures
    - To build polymorphic strategies and skeletons [Marlow et al, Haskell 2010]
    - Based on Cloud Haskell ideas [Epstein et al, Haskell 2011]
        - BUT: Closures are truly polymorphic (no `Typeable` constraint).
        - AND: Function closures behave like functions.
        - AND: Cheap closure construction and elimination due to dual representation.

- On-demand work distribution
    - Distributed random work stealing a la GUM [Trinder et al, PLDI 1996]

- Emerging support for fault tolerance
    - Fault tolerant versions of polymorphic skeletons
        - BUT: Fault tolerance rules out determinism ...

# HdpH System Architecture



- Per core: one threadpool (concurrent deque) and scheduler
- Per node: one sparkpool (concurrent deque) and message handler
- Per node: one registry (concurrent map) for global references

# HdpH Primitives

## Shared-memory types and primitives

| | |
|---|---|
| `Par a` | parallel computation monad (returning type a) |
| `IVar a` | write-once buffer (of type a) |

| | |
|---|---|
| `eval :: a -> Par a` | forcing evaluation |
| `fork :: Par () -> Par ()` | thread creation |
| `new :: Par (IVar a)` | communication |
| `put :: IVar a -> a -> Par ()` | and |
| `get :: IVar a -> Par a` | synchronisation |

## Distributed-memory types and primitives

| | |
|---|---|
| `Closure a` | serialisable explicit closure (of type a) |
| `GIVar a` | serialisable global reference to IVar (of type a) |

| | |
|---|---|
| `spark  :: Closure(Par ()) -> Par ()` | spark creation |
| `pushTo :: Closure(Par ()) -> NodeId -> Par ()` | and placement |
| `glob :: IVar (Closure a) -> Par (GIVar (Closure a))` | remote |
| `rput :: GIVar (Closure a) -> Closure a -> Par ()` | communication |

# Computing with Polymorphic Closures

**Example: Function closure application**

```
apC :: Closure (a -> b) -> Closure a -> Closure b
apC clo_f clo_x = $(mkClosure [| unClosure clo_f $ unClosure clo_x |])
```

- Truly polymorphic function closure operations.
  - No `Typeable` constraint.
- Polymorphic operations on function closures are cheap.
  - Dual closure representation and lazy evaluation avoid unnecessary serialisation.
  - Dual representation mandates safe closure construction via Template Haskell.

# Computing with Polymorphic Closures

**Example: Function closure application**

```
apC :: Closure (a -> b) -> Closure a -> Closure b
apC clo_f clo_x = $(mkClosure [| unClosure clo_f $ unClosure clo_x |])
```

- Truly polymorphic function closure operations.
  - No `Typeable` constraint.
- Polymorphic operations on function closures are cheap.
  - Dual closure representation and lazy evaluation avoid unnecessary serialisation.
  - Dual representation mandates safe closure construction via Template Haskell.

**Actual implementation (for want of GHC-supported Static)**

```
apC :: Closure (a -> b) -> Closure a -> Closure b
apC clo_f clo_x = $(mkClosure [| apC_abs (clo_f, clo_x) |])

-- manually constructed toplevel closure abstraction
apC_abs (clo_f, clo_x) = unClosure clo_f $ unClosure clo_x
```

# High-level Abstractions — Strategies and Skeletons

## Strategies for the Par monad

```
type Strategy a = a -> Par a

using :: a -> Strategy a -> Par a
x ‘using‘ strat = strat x

-- strategy combinator for lists (of Closures)
parList :: Closure (Strategy (Closure a)) -> Strategy [Closure a]
```

## Algorithmic Skeletons built on Strategies

```
parMap :: Closure (Strategy (Closure b))
       -> Closure (a -> b)
       -> [Closure a]
       -> Par [Closure b]
parMap clo_strat clo_f clo_xs =
  map f clo_xs ‘using‘ parList clo_strat
    where f = apC clo_f
```

# Scaling to 2000 Cores — Speedup

**Problem:** `sum (map totient [1 .. 160k or 240k])`

- Simple data-parallel problem with irregular parallelism.

**Architecture:** HECToR (1 to 64 nodes, 32 cores each)

**Two-level coordination strategy (controlling locality):**

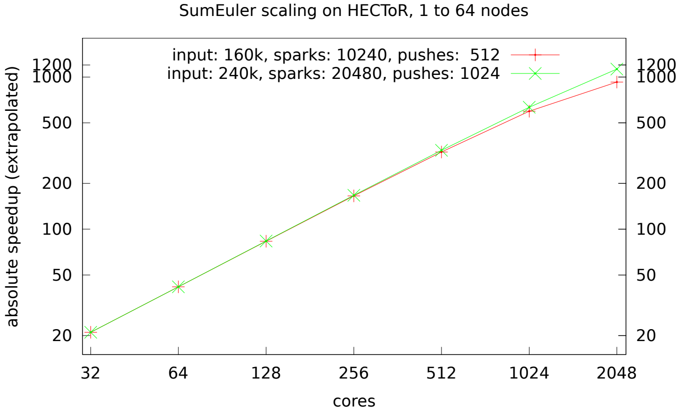
- Main node divides input and explicitly *pushes* large tasks to all nodes,
  - deliberately over-subscribing nodes.
- Each large task further sub-divides its input and *sparks* small tasks
  - to be distributed on-demand across all cores of current node, or
  - to be fished away by idle nodes.

# Scaling to 2000 Cores — Speedup

**Problem:** `sum (map totient [1 .. 160k or 240k])`

- Simple data-parallel problem with irregular parallelism.

**Architecture:** HECToR (1 to 64 nodes, 32 cores each)



# Fault Tolerant Workpool — Cost of Recovery

**Problem:** `sum (map liouville [1 .. 300M])`

**Architecture:** PC cluster (10 nodes, 1 of which fails).

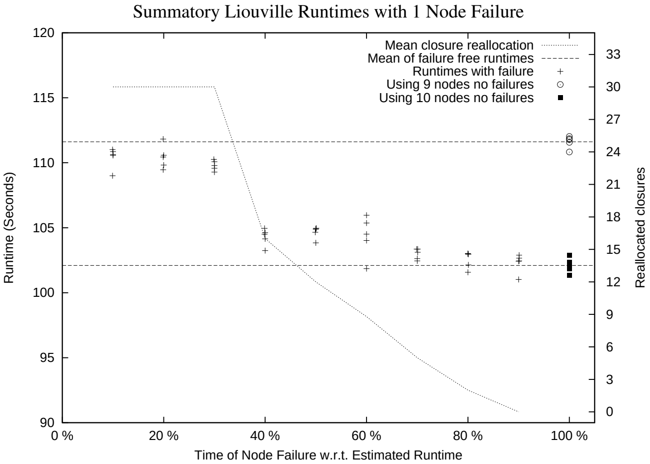
**Fault tolerant coordination strategy:**

- Fault tolerant work pool monitors worker nodes, and
- automatically reallocates tasks residing on failed nodes.

# Fault Tolerant Workpool — Cost of Recovery

**Problem:** `sum (map liouville [1 .. 300M])`

**Architecture:** PC cluster (10 nodes, 1 of which fails).



# Thanks for Listening

**Ongoing Work**

- Tighter integration of fault tolerance and work distribution.
- Refined locality control.
- Profiling tools.

**Public HdpH source repository:**

- `https://github.com/PatrickMaier/HdpH`

**References:**

- P. Maier, P. W. Trinder. *Implementing a High-level Distributed-Memory parallel Haskell in Haskell*, In Proc. IFL 2011, Springer. To appear. `www.macs.hw.ac.uk/~pm175/papers/Maier_Trinder_IFL2011_XT.pdf`
- R. Stewart, P. W. Trinder, P. Maier. *Supervised Workpools for Reliable Parallel Computing*, In Draft Proc. TFP 2012. `www.macs.hw.ac.uk/~rs46/papers/tfp2012/TFP2012_Robert_Stewart.pdf`