Transparent Fault Tolerance for Scalable Functional Computation

Rob Stewart ¹  Patrick Maier ²  Phil Trinder ²
26th July 2016

¹Heriot-Watt University Edinburgh
²University of Glasgow
Motivation

- As HPC & Cloud architectures grow, failure rates increase.
- Non traditional HPC workloads: irregular parallel workloads.
- How do we scale languages whilst tolerating faults?
Language approaches
Fault tolerance with explicit task placement

Erlang ’let it crash’ philosophy:

- Live together, die together:

  \[ \text{Pid} = \text{spawn}(\text{NodeB}, \text{fun}() \rightarrow \text{foo}() \text{ end}) \]
  \[ \text{link}(\text{Pid}) \]

- Be notified of failure:

  \[ \text{monitor(\text{process}, \text{spawn}(\text{NodeB}, \text{fun}() \rightarrow \text{foo}() \text{ end)})}. \]

- Influence on other languages:

  \[ \text{-- Akka} \]
  \[ \text{spawnLinkRemote[MyActor](host, port)} \]

  \[ \text{-- CloudHaskell} \]
  \[ \text{spawnLink :: NodeId \rightarrow Closure (Process () \rightarrow Process ProcessId} \]
Limitations of eager work placement

- Only *explicit* task placement
  - *irregular parallelism.*
  - Explicit placement cannot fix scheduling accidents

- Only lazy scheduling
  - nodes initially idle until saturation
  - load balancing *communication protocols* cause delays

- *Solution* is to use *both lazy and eager scheduling*
  - *push* big tasks early on
  - *load balance* smaller tasks to fix scheduling accidents
Problem 1: irregular parallelism

- Explicit "spawn at" not suitable for irregular workloads

Solution!

- Employ lazy scheduling and load balancing

Problem 2: fault tolerance

- How do know what to recover?
- What tasks were lost when the a node disappears?
HdpH-RS: a fault tolerant
distributed parallel DSL
Context

**HdpH-RS**

- **H**: implemented in Haskell
- **d**: distributed at scale
- **pH**: task parallel Haskell DSL
- **RS**: reliable scheduling

An extension of the *HdpH* DSL:

Distributed fork join parallelism

Node A: Caller invokes spawn/spawnAt
Node B: Parallel thread
Node C: IVar get
Node D: Sync points upon get

Dependency:
- IVar put
- SpawnAt
- Spawn

Sync points upon get:
- k
- m
- n
- p

Parallel thread: q

Nodes:
- Node A
- Node B
- Node C
- Node D
data Par a  -- monadic parallel computation of type 'a'
runParIO :: RTSConf → Par a → IO (Maybe a)

-- * task distribution

:type Task a = Closure (Par (Closure a))
spawn    :: Task a → Par (Future a)  -- lazy
spawnAt :: Node → Task a → Par (Future a)  -- eager

-- * communication of results via futures

data IVar a  -- write-once buffer of type 'a'
type Future a = IVar (Closure a)
get   :: Future a → Par (Closure a)  -- local read
rput :: Future a → Closure a → Par ()  -- global write (internal)

sparks can migrate \((\text{spawn})\)
threads cannot migrate \((\text{spawnAt})\)

sparks get converted to threads for execution
HdpH-RS scheduling

Node A

sparkpool

(spawn)

CPU

(threadpool)

(Node A)

(Node B)

sparkpool

(spawn)

CPU

(threadpool)

(Node B)

Node B

(spawnAt)

rput

(migrate)

(convert)
parSumLiouville :: Integer → Par Integer
parSumLiouville n = do
  let tasks = [$(mkClosure [ | liouville k | ]) | k ← [1..n]]
futures ← mapM spawn tasks
results ← mapM get futures
return $ sum $ map unClosure results

liouville :: Integer → Par (Closure Integer)
liouville k = eval $ toClosure $ (-1)^(length $ primeFactors k)
Fault tolerant algorithmic skeletons

parMapSliced, pushMapSliced -- slicing parallel maps

:: (Binary b) -- result type serialisable
⇒ Int -- number of tasks
→ Closure (a → b) -- function closure
→ [Closure a] -- input list
→ Par [Closure b] -- output list

parMapReduceRangeThresh -- map/reduce with lazy scheduling

:: Closure Int -- threshold
→ Closure InclusiveRange -- range over which to calculate
→ Closure (Closure Int
→ Par (Closure a))
→ Closure (Closure a
→ Closure a
→ Par (Closure a))
→ Closure a -- compute two results (associate)
→ Par (Closure a) -- initial value
HdpH-RS fault tolerance semantics
HdpH-RS syntax for states

States $R, S, T ::= S \mid T$ parallel composition
| $⟨M⟩_p$ thread on node $p$, executing $M$
| $⟨⟨M⟩⟩_p$ spark on node $p$, to execute $M$
| $i{⟨M⟩}_p$ full IVar $i$ on node $p$, holding $M$
| $i{⟨⟨M⟩⟩}_p$ empty IVar $i$ on node $p$, supervising thread $⟨M⟩_q$
| $i{⟨⟨M⟩⟩}_Q_p$ empty IVar $i$ on node $p$, supervising spark $⟨⟨M⟩⟩_Q$
| $i{⊥}_p$ zombie IVar $i$ on node $p$
| dead$_p$ notification that node $p$ is dead

Meta-variables $i, j$ names of IVars
$p, q$ nodes
$P, Q$ sets of nodes
$x, y$ term variables

The key to tracking and recovery:
- $i{⟨⟨M⟩⟩}_p$ supervised threads
- $i{⟨⟨M⟩⟩}_Q_p$ supervised sparks
Creating tasks

States $R, S, T ::= S \mid T$ parallel composition

| $\langle M \rangle_p$ thread on node $p$, executing $M$
| $\langle\langle M \rangle\rangle_p$ spark on node $p$, to execute $M$
| $i\{M\}_p$ full IVar $i$ on node $p$, holding $M$
| $i\{\langle M \rangle_q\}_p$ empty IVar $i$ on node $p$, supervising thread $\langle M \rangle_q$
| $i\{\langle\langle M \rangle\rangle_Q\}_p$ empty IVar $i$ on node $p$, supervising spark $\langle\langle M \rangle\rangle_q$
| $i\{\bot\}_p$ zombie IVar $i$ on node $p$
| $\text{dead}_p$ notification that node $p$ is dead

\[
\langle E[\text{spawn } M] \rangle_p \rightarrow \nu i. (\langle E[\text{return } i] \rangle_p \mid i\{\langle M \rangle = \text{rput } i\}_p \mid \langle M \rangle = \text{rput } i)_p),
\]

(spawn)

\[
\langle E[\text{spawnAt } q M] \rangle_p \rightarrow \nu i. (\langle E[\text{return } i] \rangle_p \mid i\{\langle M \rangle = \text{rput } i\}_q \mid \langle M \rangle = \text{rput } i)_q),
\]

(spawnAt)
Scheduling

States $R, S, T ::= S \mid T$ parallel composition
\begin{itemize}
  \item $\langle M \rangle_p$ thread on node $p$, executing $M$
  \item $\langle\langle M \rangle \rangle_p$ spark on node $p$, to execute $M$
  \item $i\{M\}_p$ full IVar $i$ on node $p$, holding $M$
  \item $i\{\langle M \rangle_q\}_p$ empty IVar $i$ on node $p$, supervising thread $\langle M \rangle_q$
  \item $i\{\langle\langle M \rangle \rangle_Q\}_p$ empty IVar $i$ on node $p$, supervising spark $\langle\langle M \rangle \rangle_q$
  \item $i\{\bot\}_p$ zombie IVar $i$ on node $p$
  \item $\text{dead}_p$ notification that node $p$ is dead
\end{itemize}

\[
\langle\langle M \rangle \rangle_p \mid i\{\langle\langle M \rangle \rangle_P\}_q \rightarrow \langle\langle M \rangle \rangle_p \mid i\{\langle\langle M \rangle \rangle_P\}_q, \text{ if } p_1, p_2 \in P \quad (\text{migrate})
\]
\[
\langle M \rangle_p \mid i\{\langle M \rangle_{P_1}\}_q \rightarrow \langle M \rangle_p \mid i\{\langle M \rangle_{P_2}\}_q, \text{ if } p \in P_1 \cap P_2 \quad (\text{track})
\]
\[
\langle M \rangle_p \rightarrow \langle M \rangle_p
\quad (\text{convert})
States $R, S, T ::= S \mid T$ parallel composition
\begin{align*}
| \langle M \rangle_p & \quad \text{thread on node } p, \text{ executing } M \\
| \langle\langle M \rangle \rangle_p & \quad \text{spark on node } p, \text{ to execute } M \\
i \{M\}_p & \quad \text{full IVar } i \text{ on node } p, \text{ holding } M \\
i \{\langle M \rangle_q\}_p & \quad \text{empty IVar } i \text{ on node } p, \text{ supervising thread } \langle M \rangle_q \\
i \{\langle\langle M \rangle \rangle_Q\}_p & \quad \text{empty IVar } i \text{ on node } p, \text{ supervising spark } \langle\langle M \rangle \rangle_Q \\
i \{\bot\}_p & \quad \text{zombie IVar } i \text{ on node } p \\
\text{dead}_p & \quad \text{notification that node } p \text{ is dead}
\end{align*}

\begin{align*}
\langle \mathcal{E}[rput \; i \; M] \rangle_p \mid i \{\langle N \rangle \}_q & \longrightarrow \langle \mathcal{E}[\text{return ()}] \rangle_p \mid i \{M\}_q & \hspace{1cm} \text{(rput\_empty\_thread)} \\
\langle \mathcal{E}[rput \; i \; M] \rangle_p \mid i \{\langle\langle N \rangle \rangle \}_q & \longrightarrow \langle \mathcal{E}[\text{return ()}] \rangle_p \mid i \{M\}_q & \hspace{1cm} \text{(rput\_empty\_spark)} \\
\langle \mathcal{E}[rput \; i \; M] \rangle_p \mid i \{N\}_q & \longrightarrow \langle \mathcal{E}[\text{return ()}] \rangle_p \mid i \{N\}_q, & \hspace{1cm} \text{(rput\_full)} \\
\langle \mathcal{E}[rput \; i \; M] \rangle_p \mid i \{\bot\}_q & \longrightarrow \langle \mathcal{E}[\text{return ()}] \rangle_p \mid i \{\bot\}_q & \hspace{1cm} \text{(rput\_zombie)} \\
\langle \mathcal{E}[\text{get } i] \rangle_p \mid i \{M\}_p & \longrightarrow \langle \mathcal{E}[\text{return } M] \rangle_p \mid i \{M\}_p, & \hspace{1cm} \text{(get)}
\end{align*}
Failure

States  \( R, S, T ::= S \mid T \)  
\[ \text{parallel composition} \]
  \[ | \langle M \rangle_p \]  thread on node \( p \), executing \( M \)
  \[ | \langle\langle M \rangle\rangle_q \]  spark on node \( p \), to execute \( M \)
  \[ | i\{ M \}_p \]  full IVar \( i \) on node \( p \), holding \( M \)
  \[ | i\{\langle M \rangle_q \}_p \]  empty IVar \( i \) on node \( p \), supervising thread \( \langle M \rangle_q \)
  \[ | i\{\langle\langle M \rangle\rangle_q \}_p \]  empty IVar \( i \) on node \( p \), supervising spark \( \langle\langle M \rangle\rangle_q \)
  \[ | i\{\bot\}_p \]  zombie IVar \( i \) on node \( p \)
  \[ | \text{dead}_p \]  notification that node \( p \) is dead

\[
\begin{align*}
\text{dead}_p | \langle M \rangle_p & \rightarrow \text{dead}_p & \text{(kill\_spark)} \\
\text{dead}_p | \langle M \rangle_p & \rightarrow \text{dead}_p & \text{(kill\_thread)} \\
\text{dead}_p | i\{?\}_p & \rightarrow \text{dead}_p | i\{\bot\}_p & \text{(kill\_ivar)}
\end{align*}
\]
Recovery

States $R, S, T ::= S | T$ parallel composition
| $\langle M \rangle_p$ thread on node $p$, executing $M$
| $\langle\langle M \rangle\rangle_p$ spark on node $p$, to execute $M$
| $i\{M\}_p$ full IVar $i$ on node $p$, holding $M$
| $i\{\langle M \rangle_q\}_p$ empty IVar $i$ on node $p$, supervising thread $\langle M \rangle_q$
| $i\{\langle\langle M \rangle\rangle Q\}_p$ empty IVar $i$ on node $p$, supervising spark $\langle\langle M \rangle\rangle_q$
| $i\{\bot\}_p$ zombie IVar $i$ on node $p$
| dead$_p$ notification that node $p$ is dead

\[
i\{\langle M \rangle_q\}_p | \text{dead}_q \rightarrow i\{\langle M \rangle_p\}_p | \langle M \rangle_p | \text{dead}_q, \text{ if } p \neq q \quad \text{(recover_thread)}
\]
\[
i\{\langle\langle M \rangle\rangle Q\}_p | \text{dead}_q \rightarrow i\{\langle\langle M \rangle\rangle\}_p | \langle\langle M \rangle\rangle_p | \text{dead}_q, \text{ if } p \neq q \text{ and } q \in Q \quad \text{(recover_spark)}
\]
Fault tolerant load balancing
Successful work stealing

Node A
supervisor

Node B
victim

Node C
thief

FISH
REQ
AUTH
SCHEDULE
ACK
Supervised work stealing
Correspondence with language semantics

\[ i\{\langle M\rangle\{B\}\}\_A \quad | \quad \langle M\rangle\_B \]

\[ i\{\langle M\rangle\{B,C\}\}\_A \quad | \quad \langle M\rangle\_B \]  
\[ (track) \]

\[ i\{\langle M\rangle\{B,C\}\}_{migrate} \quad | \quad \langle M\rangle\_B \]

\[ i\{\langle M\rangle\{B,C\}\}_{track} \quad | \quad \langle M\rangle\_C \]

\[ i\{\langle M\rangle\{C\}\}_{track} \quad | \quad \langle M\rangle\_C \]
Is the scheduling algorithm robust?

- Non-determinism in faulty systems
- Causal ordering not consistent with wall clock times
- Communication delays
  - node availability info could be outdated
  - asynchronous scheduling messages complicates tracking

*Model checking* increases confidence in scheduling algorithm.
Model checking the scheduler
Abstracting HdpH-RS scheduler to a Promela model

- 1 spark, 1 supervisor.
- 3 workers, they can all die with (*dead*) transition rule.
- A worker holding a task copy can send result to supervisor.
- Messages to a dead node are lost.
- Supervisor will *eventually* receive DEADNODE messages.
- Buffered channels model asynchronous message passing.
- Tasks replicated by supervisor with (*recover_spark*) rule.
active proctype Supervisor() {
    int thiefID, victimID, deadNodeID, seq, authorizedSeq, deniedSeq;

    SUPERVISOR_RECEIVE:
        /* evaluate task once spark age exceeds 100 */
        if :: (supervisor.sparkpool.spark_count > 0 && spark.age > maxLife) →
            supervisor ! RESULT(null, null, null);
        :: else →
            if :: (supervisor.sparkpool.spark_count > 0) →
                supervisor ! RESULT(null, null, null);
            :: supervisor ? FISH(thiefID, null, null) → ...
            :: supervisor ? REQ(victimID, thiefID, seq) → ...
            :: supervisor ? AUTH(thiefID, authorizedSeq, null) → ...
            :: supervisor ? ACK(thiefID, seq, null) → ...
            :: supervisor ? DENIED(thiefID, deniedSeq, null) → ...
            :: supervisor ? DEADNODE(deadNodeID, null, null) → ...
            :: supervisor ? RESULT(null, null, null) →
                supervisor.ivar = 1;
                goto EVALUATION_COMPLETE;
            fi;
        fi;
    goto SUPERVISOR_RECEIVE;
Example: worker response to a FISH message:

workers[me] ? FISH(thiefID, null, null) →
  if /* worker has spark and not waiting for scheduling authorisation */
    :: (worker[me].sparkpool.spark_count > 0
    && ! worker[me].waitingSchedAuth) →
      worker[me].waitingSchedAuth = true;
      supervisor ! REQ(me, thiefID, worker[me].sparkpool.spark);
  /* worker doesn’t have the spark */
  :: else → workers[thiefID] ! NOWORK(me, null, null) ;
fi
Two intended properties

1. The IVar is empty until a result is sent
2. IVar eventually gets filled

#define ivar_full ( supervisor.ivar == 1 )
#define ivar_empty ( supervisor.ivar == 0 )
#define any_result_sent
    ( supervisor.resultSent || worker[0].resultSent
    || worker[1].resultSent || worker[2].resultSent )

No counter examples, exhaustively checked with SPIN:

<table>
<thead>
<tr>
<th>LTL Formula</th>
<th>Depth</th>
<th>States</th>
<th>Transitions</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ (ivar_empty U any_result_sent)</td>
<td>124</td>
<td>3.7m</td>
<td>7.4m</td>
<td>83.8Mb</td>
</tr>
<tr>
<td>◇ □ ivar_full</td>
<td>124</td>
<td>8.2m</td>
<td>22.4m</td>
<td>84.7Mb</td>
</tr>
</tbody>
</table>
HdpH-RS implementation
Threads may migrate within node
- Sparks may migrate between nodes
- Shares TCP transport backend with CloudHaskell
  - rely on failure detection of TCP protocol
- Haskell message handling matches verified Promela model
Evaluation
HdpH-RS fault-free overheads

Commodity cluster running Summatory Liouville

Input=200m, Threshold=500k

Run time (Seconds)

Cores

Input=200m, Threshold=500k

Speedup

Cores
HdpH-RS fault-free overheads

**HPC cluster** running Summatory Liouville

![Runtime vs Cores](image1)

Input=500m, Threshold=250k

- parMapSliced
- parMapSliced (RS)
- pushMapSliced
- pushMapSliced (RS)

![Speedup vs Cores](image2)

Input=500m, Threshold=250k

- parMapSliced
- parMapSliced (RS)
- pushMapSliced
- pushMapSliced (RS)
Surviving chaos monkey

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Skeleton</th>
<th>Failed Nodes (seconds)</th>
<th>Recovery (seconds)</th>
<th>Runtime (seconds)</th>
<th>Unit Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>parMapSliced</td>
<td>-</td>
<td>[32,37,44,46,48,50,52,57]</td>
<td>16</td>
<td>56.6</td>
<td>pass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[18,27,41]</td>
<td>6</td>
<td>85.1</td>
<td>pass</td>
</tr>
<tr>
<td>parMapSliced (RS)</td>
<td>[19,30,39,41,54,59,59]</td>
<td>14</td>
<td>85.1</td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[8,11]</td>
<td>4</td>
<td>76.2</td>
<td>pass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[8,9,24,28,32,34,40,57]</td>
<td>16</td>
<td>62.8</td>
<td>pass</td>
</tr>
<tr>
<td>pushMapSliced</td>
<td>-</td>
<td>[3,8,8,12,22,26,26,29,55]</td>
<td>268</td>
<td>132.7</td>
<td>pass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1]</td>
<td>53</td>
<td>63.3</td>
<td>pass</td>
</tr>
<tr>
<td>pushMapSliced (RS)</td>
<td>[10,59]</td>
<td>41</td>
<td>62.8</td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[13,15,18,51]</td>
<td>106</td>
<td>125.0</td>
<td>pass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[13,24,42,51]</td>
<td>80</td>
<td>105.9</td>
<td>pass</td>
</tr>
</tbody>
</table>

4 other Chaos Monkey benchmarks in:

Comparison with other approaches
HdpH-RS applicability

Fault tolerance versus memory use trade off:

- HdpH-RS retains duplicate closures
- Performance predicated on small closure footprint
  - few closures
  - small in size
  - terminate quickly
- Many applications areas with these characteristics, e.g.

Not suitable for:

- Traditional HPC workloads with regular parallelism
  - little need for dynamic load balancing
  - need highly optimised floating point capabilities
- Task execution time must outweigh communication
- Closures with big memory footprint not well suited
  - *i.e.* HdpH-RS not for Big Data applications
Compared with Hadoop

- **Applicability**
  - **Hadoop** big data
  - **HdpH-RS** big computation

- **Failure detection**
  - **Hadoop** centralised, takes minutes
  - **HdpH-RS** decentralised, takes seconds

- **Re-execution**
  - **Hadoop:**
    - map task outputs stored locally, redundant re-execution
  - **HdpH-RS:**
    - results are immediately transmitted once computed
## Compared with Erlang

<table>
<thead>
<tr>
<th></th>
<th>Load balancing</th>
<th>Fault tolerance</th>
<th>Distributed memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>✗</td>
<td>(✔)</td>
<td>✔</td>
</tr>
<tr>
<td>CloudHaskell</td>
<td>✗</td>
<td>(✔)</td>
<td>✔</td>
</tr>
<tr>
<td>HdpH</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>HdpH-RS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Erlang processes cannot migrate
  - less suitable for irregular parallelism
- Erlang is dynamically typed
  - *programming errors* only detected at runtime
- Fault tolerance
  - **Erlang**
    - fault tolerance explicit with link and monitor
    - programmatic recovery
    - automatic with supervision behaviours
  - **HdpH-RS**
    - fault tolerance automatic
Divide and conquer fault tolerance

Node A

Node B

Thread Replicated
More Than Once

Thread Replicated
Once

PUSH
Lazy scheduling + divide and conquer parallelism means less needless replication
Conclusion
The challenge:

- Failure rates as HPC architectures grow.
- Load balancing for irregular parallelism.
- Need to support fault tolerant load balancing
- Intricate details of asynchronous non-determinism.

The HdpH-RS approach:

- Language semantics + exhaustive model checking.
- Increases confidence in the design.

HdpH-RS evaluation:

- Low supervision overheads.
- Survives random fault injection.
Software

- **HdpH-RS**

  https://github.com/robstewart57/hdph-rs

- **Promela model**


- **HdpH**

  https://github.com/PatrickMaier/HdpH
Presentation based on:


HdpH DSLs overview (including topology aware scheduling):


Full HdpH-RS description: