Transparent Fault Tolerance for Scalable Functional Computation

Rob Stewart [1]   Patrick Maier [2]   Phil Trinder [2]

26th July 2016

[1]Heriot-Watt University Edinburgh

[2]University of Glasgow

Motivation

Tolerating faults with irregular parallelism

> ***The success of future HPC architectures will depend on the ability to provide reliability and availability at scale.*** — *Understanding Failures in Petascale Computers. B Schroeder and G Gibson. Journal of Physics: Conference Series, 78, 2007.*

- 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:

```
Pid = spawn(NodeB, fun() -> foo() end)
link(Pid)
```

- Be notified of failure:

```
monitor(process, spawn(NodeB, fun() -> foo() end)).
```

- Influence on other languages:

```
-- Akka
spawnLinkRemote[MyActor](host, port)

-- CloudHaskell
spawnLink :: NodeId → Closure (Process ()) → 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

Fault tolerant load balancing

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

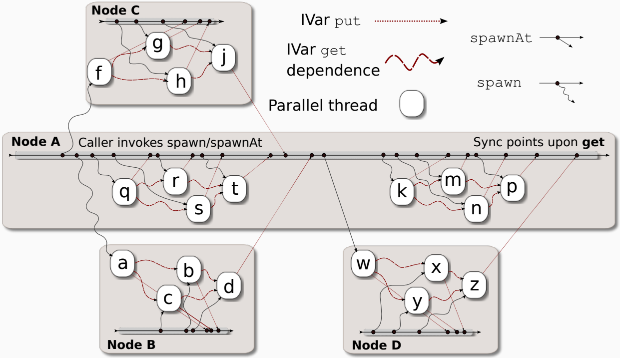
HdpH-RS

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

An extension of the *HdpH* DSL:

The HdpH DSLs for Scalable Reliable Computation. P Maier, R Stewart and P Trinder, ACM SIGPLAN Haskell Symposium, 2014. Göteborg, Sweden.

Distributed fork join parallelism



HdpH-RS API

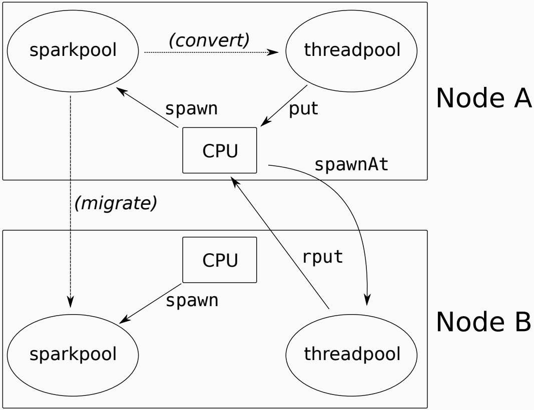
```
data Par a  -- monadic parallel computation of type 'a'
runParIO :: RTSConf → Par a → IO (Maybe a)

-- * 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 (*spawn*)

threads cannot migrate (*spawnAt*)

sparks get converted to *threads* for execution

HdpH-RS scheduling



HdpH-RS example

```
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         -- compute one result
              → Par (Closure a))
  → Closure (Closure a           -- compute two results (associate)
              → Closure a
              → Par (Closure a))
  → Closure a                    -- initial value
  → Par (Closure a)
```

HdpH-RS fault tolerance semantics

HdpH-RS syntax for states

States R, S, T	$::= S \mid T$	parallel composition
	$\mid \langle M \rangle_p$	thread on node p, executing M
	$\mid \langle\langle M \rangle\rangle_p$	spark on node p, to execute M
	$\mid i\{M\}_p$	full IVar i on node p, holding M
	$\mid i\{\langle M \rangle_q\}_p$	empty IVar i on node p, supervising thread $\langle M \rangle_q$
	$\mid i\{\langle\langle M \rangle\rangle_Q\}_p$	empty IVar i on node p, supervising spark $\langle\langle M \rangle\rangle_q$
	$\mid i\{\perp\}_p$	zombie IVar i on node p
	$\mid \mathrm{dead}_p$	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:

- $i\{\langle M \rangle_q\}_p$ **supervised threads**
- $i\{\langle\langle M \rangle\rangle_Q\}_p$ **supervised sparks**

Creating tasks

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

$$
\langle \mathcal{E}[\texttt{spawn}\, M] \rangle_p \longrightarrow \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle\!\langle M \gg\!= \texttt{rput}\, i \rangle\!\rangle_{\{p\}}\}_p \mid \langle\!\langle M \gg\!= \texttt{rput}\, i \rangle\!\rangle_p), \qquad \text{(spawn)}
$$

$$
\langle \mathcal{E}[\texttt{spawnAt}\, q\, M] \rangle_p \longrightarrow \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle M \gg\!= \texttt{rput}\, i \rangle_q\}_p \mid \langle M \gg\!= \texttt{rput}\, i \rangle_q), \qquad \text{(spawnAt)}
$$

Scheduling

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

$$
\begin{array}{rcll}
\langle\!\langle M \rangle\!\rangle_{p_1} \mid i\{\langle\!\langle M \rangle\!\rangle_P\}_q & \longrightarrow & \langle\!\langle M \rangle\!\rangle_{p_2} \mid i\{\langle\!\langle M \rangle\!\rangle_P\}_q, & \text{if } p_1, p_2 \in P \qquad \text{(migrate)} \\
\langle\!\langle M \rangle\!\rangle_p \mid i\{\langle\!\langle M \rangle\!\rangle_{P_1}\}_q & \longrightarrow & \langle\!\langle M \rangle\!\rangle_p \mid i\{\langle\!\langle M \rangle\!\rangle_{P_2}\}_q, & \text{if } p \in P_1 \cap P_2 \qquad \text{(track)} \\
\langle\!\langle M \rangle\!\rangle_p & \longrightarrow & \langle M \rangle_p & \qquad \text{(convert)}
\end{array}
$$

Communicating results

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

$$
\begin{array}{rcll}
\langle \mathcal{E}[\mathtt{rput}\; i\; M] \rangle_p \mid i\{\langle N \rangle_p\}_q & \longrightarrow & \langle \mathcal{E}[\mathtt{return}\; ()] \rangle_p \mid i\{M\}_q & \text{(rput\_empty\_thread)} \\
\langle \mathcal{E}[\mathtt{rput}\; i\; M] \rangle_p \mid i\{\langle\!\langle N \rangle\!\rangle_Q\}_q & \longrightarrow & \langle \mathcal{E}[\mathtt{return}\; ()] \rangle_p \mid i\{M\}_q & \text{(rput\_empty\_spark)} \\
\langle \mathcal{E}[\mathtt{rput}\; i\; M] \rangle_p \mid i\{N\}_q & \longrightarrow & \langle \mathcal{E}[\mathtt{return}\; ()] \rangle_p \mid i\{N\}_q, & \text{(rput\_full)} \\
\langle \mathcal{E}[\mathtt{rput}\; i\; M] \rangle_p \mid i\{\bot\}_q & \longrightarrow & \langle \mathcal{E}[\mathtt{return}\; ()] \rangle_p \mid i\{\bot\}_q & \text{(rput\_zombie)} \\
\langle \mathcal{E}[\mathrm{get}\; i] \rangle_p \mid i\{M\}_p & \longrightarrow & \langle \mathcal{E}[\mathtt{return}\; M] \rangle_p \mid i\{M\}_p, & \text{(get)}
\end{array}
$$

Failure

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

$$\begin{array}{rcll}
\text{dead}_p \mid \langle\!\langle M \rangle\!\rangle_p & \longrightarrow & \text{dead}_p & (\text{kill\_spark}) \\
\text{dead}_p \mid \langle M \rangle_p & \longrightarrow & \text{dead}_p & (\text{kill\_thread}) \\
\text{dead}_p \mid i\{?\}_p & \longrightarrow & \text{dead}_p \mid i\{\bot\}_p & (\text{kill\_ivar})
\end{array}$$

Recovery

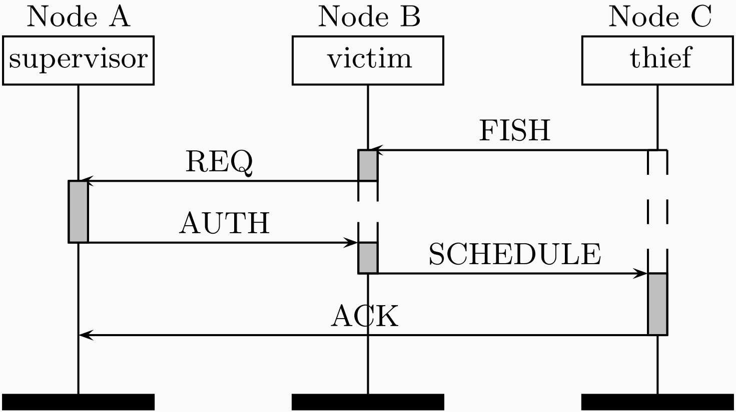
$$
\begin{array}{llll}
\text{States } R, S, T & ::= & S \mid T & \text{parallel composition} \\
 & \mid & \langle M \rangle_p & \text{thread on node } p\text{, executing } M \\
 & \mid & \langle\!\langle M \rangle\!\rangle_p & \text{spark on node } p\text{, to execute } M \\
 & \mid & i\{M\}_p & \text{full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

$$
i\{\langle M \rangle_q\}_p \mid \mathrm{dead}_q \longrightarrow i\{\langle M \rangle_p\}_p \mid \langle M \rangle_p \mid \mathrm{dead}_q, \;\; \text{if } p \neq q \qquad \text{(recover\_thread)}
$$

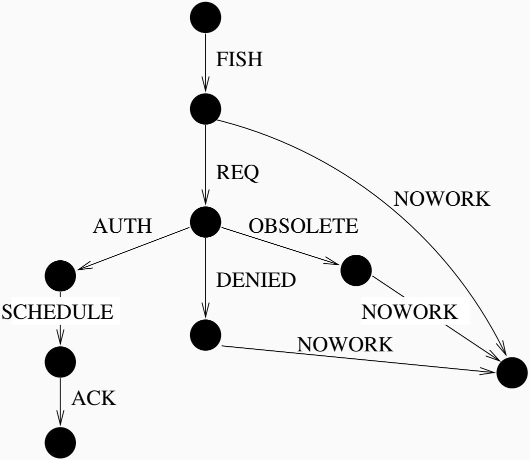
$$
i\{\langle\!\langle M \rangle\!\rangle_Q\}_p \mid \mathrm{dead}_q \longrightarrow i\{\langle\!\langle M \rangle\!\rangle_{\{p\}}\}_p \mid \langle\!\langle M \rangle\!\rangle_p \mid \mathrm{dead}_q, \;\; \text{if } p \neq q \text{ and } q \in Q \quad \text{(recover\_spark)}
$$

Fault tolerant load balancing

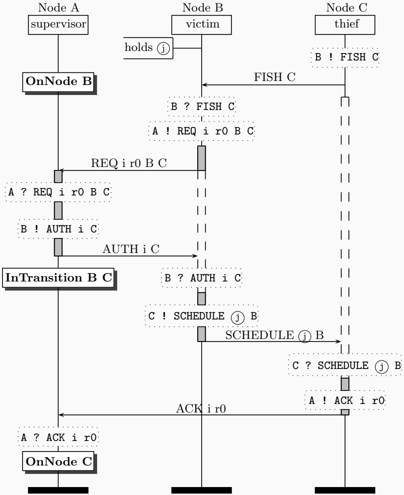
Successful work stealing



Supervised work stealing



Correspondence with language semantics



$$i\{\langle\!\langle M\rangle\!\rangle_{\{B\}}\}_A \quad | \quad \langle\!\langle M\rangle\!\rangle_B$$

$$\Big\downarrow \; (\mathit{track})$$

$$i\{\langle\!\langle M\rangle\!\rangle_{\{B,C\}}\}_A \quad | \quad \langle\!\langle M\rangle\!\rangle_B$$

$$\Big\downarrow \; (\mathit{migrate})$$

$$i\{\langle\!\langle M\rangle\!\rangle_{\{B,C\}}\}_A \quad | \quad \langle\!\langle M\rangle\!\rangle_C$$

$$\Big\downarrow \; (\mathit{track})$$

$$i\{\langle\!\langle M\rangle\!\rangle_{\{C\}}\}_A \quad | \quad \langle\!\langle M\rangle\!\rangle_C$$

Is the scheduling algorithm robust?

- Non-determinism in faulty systems
- Causal ordering not consistent with wall clock times
- Communication delays
    - node availabilty 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.

Modelling communication

```
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;
```

Modelling the scheduling algorithm

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**

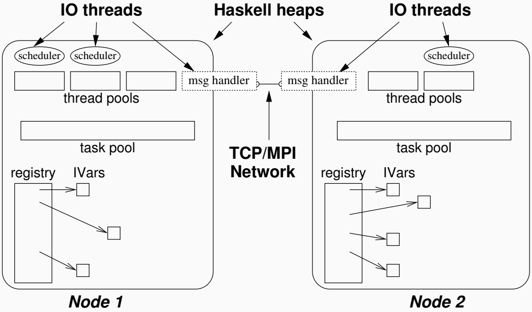
```
#define ivar_full  ( supervisor.ivar == 1 )

#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:

LTL Formula	Depth	States	Transitions	Memory
$\Box\,(ivar\_empty\ U\ any\_result\_sent)$	124	3.7m	7.4m	83.8Mb
$\Diamond\,\Box\, ivar\_full$	124	8.2m	22.4m	84.7Mb

HdpH-RS implementation

HdpH-RS architecture

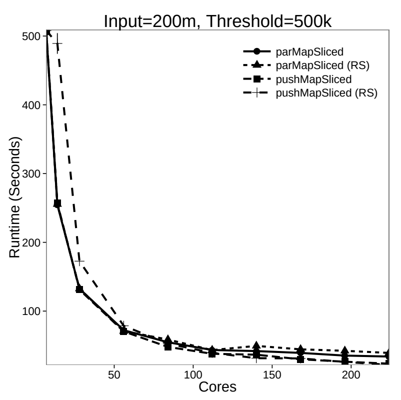


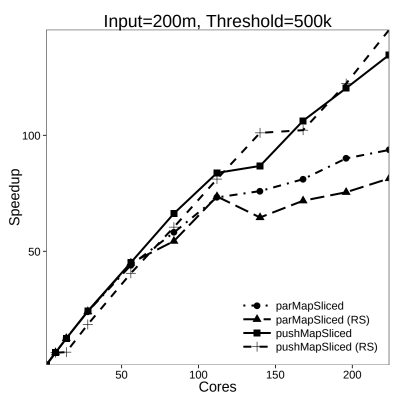
- Threads may migrate within node
- Sparks may migrate between nodes
- Shares TCP transport backend with CloudHaskell
    - rely on failure detection of TCP protocol
- Haskell message handling matches verified Promela model

Evaluation

HdpH-RS fault-free overheads

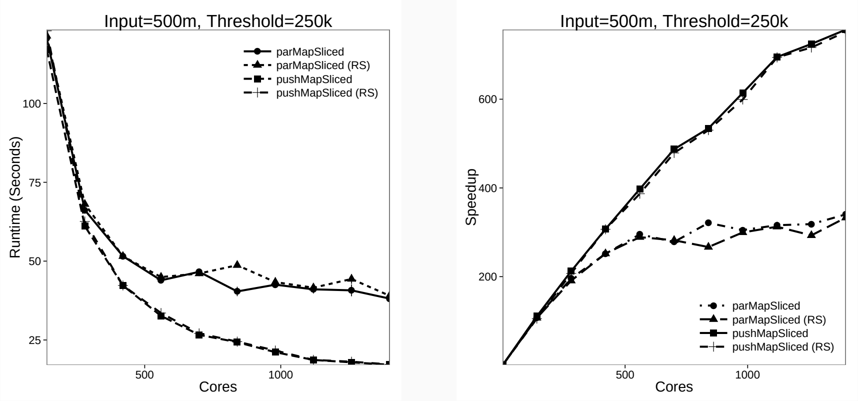
Commodity cluster running Summatory Liouville



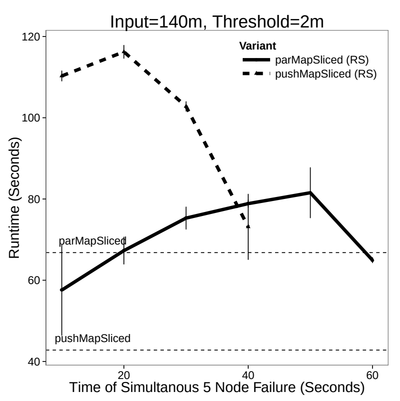


HdpH-RS fault-free overheads

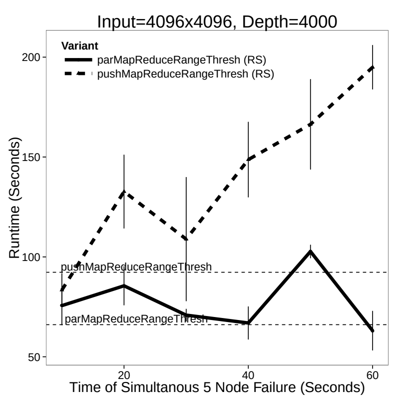
HPC cluster running Summatory Liouville



HdpH-RS recovery



Summatory Liouville



Mandelbrot

Surviving chaos monkey

Benchmark	Skeleton	Failed Nodes (seconds)	Recovery		Runtime (seconds)	Unit Test
			Sparks	Threads		
Summatory Liouville $\lambda = 50000000$ *chunk=100000* *tasks=500* *X=-7608*	`parMapSliced`	-			56.6	pass
	`parMapSliced` (RS)	[32,37,44,46,48,50,52,57]	16		85.1	pass
		[18,27,41]	6		61.6	pass
		[19,30,39,41,54,59,59]	14		76.2	pass
		[8,11]	4		62.8	pass
		[8,9,24,28,32,34,40,57]	16		132.7	pass
	`pushMapSliced`	-			58.3	pass
	`pushMapSliced` (RS)	[3,8,8,12,22,26,26,29,55]		268	287.1	pass
		[1]		53	63.3	pass
		[10,59]		41	68.5	pass
		[13,15,18,51]		106	125.0	pass
		[13,24,42,51]		80	105.9	pass

4 other Chaos Monkey benchmarks in:

Transparent Fault Tolerance for Scalable Functional Computation. R Stewart, P Maier and P Trinder, Journal of Functional Programming, 2015, Cambridge Press.

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.*

High-performance computer algebra: A Hecke algebra case study. P Maier et al. Euro-Par 2014 parallel processing - 20th international conference, Porto, Portugal, August 25-29, 2014. proceedings. LNCS, vol. 8632. Springer.

HdpH-RS applicability

Not suitable for:

- Traditional HPC workloads with regular parallelism
  - little need for dynamic load balancing
  - need highly optimised floating point capabalities
- 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

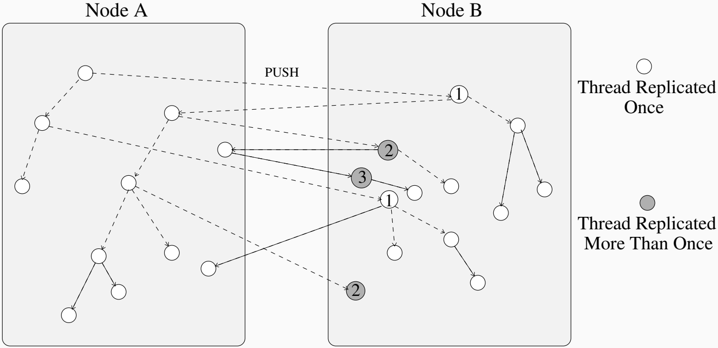
- Re-execution
    - **Hadoop:**
        - map task outputs stored locally, redundant re-execution
    - **HdpH-RS:**
        - results are immediately transmitted once computed

Compared with Erlang

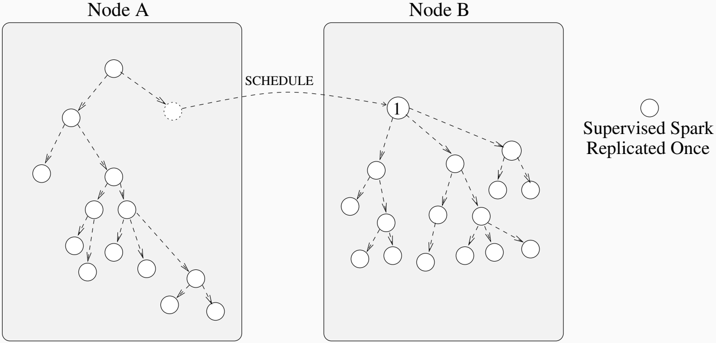
	Load balancing	Fault tolerance	Distributed memory
Erlang	✗	(✓)	✓
CloudHaskell	✗	(✓)	✓
HdpH	✓	✗	✓
HdpH-RS	✓	✓	✓

- 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



Divide and conquer fault tolerance



Lazy scheduling + divide and conquer parallelism

means **less needless replication**

Conclusion

Summary

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

`https://github.com/robstewart57/phd-thesis/blob/master/spin_model/hdph_scheduler.pml`

- HdpH

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

References

Presentation based on:

Transparent Fault Tolerance for Scalable Functional Computation. R Stewart, P Maier and P Trinder, Journal of Functional Programming, 2015, Cambridge Press.

HdpH DSLs overview (including topology aware scheduling):

The HdpH DSLs for Scalable Reliable Computation. P Maier, R Stewart and P Trinder, ACM SIGPLAN Haskell Symposium, 2014. Göteborg, Sweden.

Full HdpH-RS description:

Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell. R Stewart, PhD thesis, Heriot-Watt University, 2013.