JFP **26**, e5, 42 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S095679681600006X

Transparent fault tolerance for scalable functional computation

ROBERT STEWART
Mathematical & Computer Sciences, Heriot-Watt University, Edinburgh, UK
(*e-mail:* R.Stewart@hw.ac.uk)

PATRICK MAIER and PHIL TRINDER
School of Computing Science, Glasgow, UK
(*e-mail:* {Patrick.Maier,Phil.Trinder}@glasgow.ac.uk)

Abstract

Reliability is set to become a major concern on emergent large-scale architectures. While there are many parallel languages, and indeed many parallel functional languages, very few address reliability. The notable exception is the widely emulated Erlang distributed actor model that provides explicit supervision and recovery of actors with isolated state. We investigate scalable *transparent* fault tolerant functional computation with *automatic* supervision and recovery of tasks. We do so by developing *HdpH-RS*, a variant of the Haskell distributed parallel Haskell (HdpH) DSL with Reliable Scheduling. Extending the distributed work stealing protocol of HdpH for task supervision and recovery is challenging. To eliminate elusive concurrency bugs, we validate the HdpH-RS work stealing protocol using the SPIN model checker. HdpH-RS differs from the actor model in that its principal entities are tasks, i.e. independent stateless computations, rather than isolated stateful actors. Thanks to statelessness, fault recovery can be performed automatically and entirely hidden in the HdpH-RS runtime system. Statelessness is also key for proving a crucial property of the semantics of HdpH-RS: fault recovery does not change the result of the program, akin to deterministic parallelism. HdpH-RS provides a simple distributed fork/join-style programming model, with minimal exposure of fault tolerance at the language level, and a library of higher level abstractions such as algorithmic skeletons. In fact, the HdpH-RS DSL is exactly the same as the HdpH DSL, hence users can opt in or out of fault tolerant execution without any refactoring. Computations in HdpH-RS are always as reliable as the root node, no matter how many nodes and cores are actually used. We benchmark HdpH-RS on conventional clusters and an High Performance Computing platform: all benchmarks survive Chaos Monkey random fault injection; the system scales well e.g. up to 1,400 cores on the High Performance Computing; reliability and recovery overheads are consistently low even at scale.

1 Introduction

We know from both data centres and High Performance Computing (HPC) that faults become frequent in large-scale architectures, e.g. around 10^5 cores (Barroso *et al.*, 2013). With the exponential growth in the number of cores, many expect servers with these numbers of cores to become commonplace in the near future. Hence, the massively parallel computations on such architectures must be able to tolerate faults.

For example, there is already intense research to improve the fault tolerance of HPC software (Cappello, 2009), and cloud frameworks like Google MapReduce (Dean & Ghemawat, 2008) and Hadoop (White, 2012) provide transparent fault tolerance for their restricted data retrieval computations.

There are many parallel functional languages e.g. (Scholz, 2003; Marlow *et al.*, 2009), and many scale over distributed memory architectures to support massively parallel computations e.g. (Loogen *et al.*, 2005; Aljabri *et al.*, 2014). There are, however, very few fault tolerant parallel functional languages. The notable exception is the Erlang distributed actor model (Armstrong, 2010) that is widely emulated, e.g. by Cloud Haskell (Epstein *et al.*, 2011) and the Akka library for Scala (Gupta, 2012). Erlang style fault tolerance provides mechanisms for the explicit supervision and recovery of stateful processes, or actors, and hence faults are observable.

We investigate exploiting the statelessness of functional languages to provide scalable *transparent* fault tolerance, in contrast to stateful actor models. If a copy of the closure representing a stateless computation is preserved, it can be recovered by simply re-evaluating the closure. Conveniently functional languages like Haskell clearly distinguish stateless and stateful computations. There remain significant challenges to scaling parallel functional programs, e.g. they commonly employ complex data and control structures, and dynamically create tasks of irregular sizes.

The basis of our investigation is a variant of the Haskell Distributed Parallel Haskell DSL (HdpH), HdpH-RS that extends HdpH with *reliable scheduling* (RS). Both HdpH and HdpH-RS are designed for scalable distributed memory parallelism, and employ work stealing to load balance dynamically generated tasks of irregular sizes.

The paper makes the following research contributions.

Language design for transparent fault tolerance. The HdpH-RS implementations of the HdpH `spawn` and `spawnAt` primitives create and supervise idempotent tasks guaranteeing that they terminate providing the spawning location survives. Idempotent tasks have no side effects whose repetition can be observed, e.g. tasks that are stateless or perform idempotent database updates (Ramalingam & Vaswani, 2013). By the transitivity of spawning, the program terminates if the root node, where the program starts, survives[1]. For ease of programming, HdpH-RS exploits the higher level HdpH programming abstractions, most significantly some nine parallel skeletons that encapsulate common parallel programming patterns (Stewart & Maier, 2013). Logically, the programmer can switch from normal to fault tolerant execution simply by selecting the HdpH-RS implementation, i.e. without refactoring application code.[2]

HdpH-RS is designed to manage dynamic, irregular, and idempotent task parallelism on large-scale architectures like HPCs. It copes well with complex algorithms,

[1] Erlang style automatic restarting of the root node could easily be provided, and would simply rerun the entire computation, typically up to a fixed failure frequency, e.g. 5 restarts/hour.

[2] Section 6.4 outlines some minor pragmatic issues with using fault tolerance.

coordination patterns, and data structures. There are many real-world applications with these characteristics, for example many combinatorial problems or computational algebra problems e.g. (Maier *et al.*, 2014a) (Section 3).

An operational semantics for HdpH-RS that provides a concise and unambiguous model of scheduling in the absence and presence of failure. Provided that all tasks are idempotent, we show that the semantics makes failure unobservable: a program always computes the same result, no matter how many failed tasks had to be recovered (Sections 3.4 and 3.5).

The design and implementation of transparent fault tolerance via a distributed scheduler that implements the HdpH-RS spawn primitives, fault detection, and recovery. The reliable scheduler performs work stealing while tolerating the random loss of any or all nodes other than the root node. It replicates supervised tasks, tracks their location, and reinstates them if the task may have been lost. In the current implementation, any tasks created by a lost task will also be recreated during the re-evaluation. The HdpH-RS implementation is publicly available (Stewart & Maier, 2013) (Sections 4 and 6).

A validation of the fault tolerant distributed scheduler with the SPIN model checker. The work stealing scheduling algorithm is abstracted in to a Promela model and is formally verified with the SPIN model checker (Holzmann, 2004). The model represents all failure combinations of non-root nodes that may occur in real architectures. The key reliability property shows that the variable representing a supervised task is eventually full despite node failures. The property is proven by an exhaustive search of 8.2 million states of the model's state space at a reachable depth of 124 transitions (Section 5).

An evaluation of the HdpH-RS transparent fault tolerance performance using four benchmarks on both conventional (Beowulf (Meredith *et al.*, 2003)) clusters and the HECToR HPC (Edinburgh Parallel Computing Center (EPCC), 2008). All benchmarks execute without observable faults in the presence of Chaos Monkey (Hoff, 2010) random fault injection. We show that HdpH-RS scales well, e.g. achieving a speedup of 752 with explicit task placement and 333 with lazy work stealing when executing Summatory Liouville on 1400 HECToR cores. Supervision overheads are consistently low even at scale and recovery overheads are similarly low in the presence of frequent failure when lazy on-demand work stealing is used (Section 7). The dataset supporting this evaluation is available from an open access archive (Stewart *et al.*, 2015).

Novelty. This paper is the first comprehensive presentation of transparent fault tolerance in HdpH-RS, covering the design, implementation, semantics, validation, and evaluation. HdpH-RS was conceived as a key component of the SymGridPar2 scalable computational algebra framework. The HdpH-RS design is presented as part of the SymGridPar2 design in Maier *et al.* (2014b), together with some very

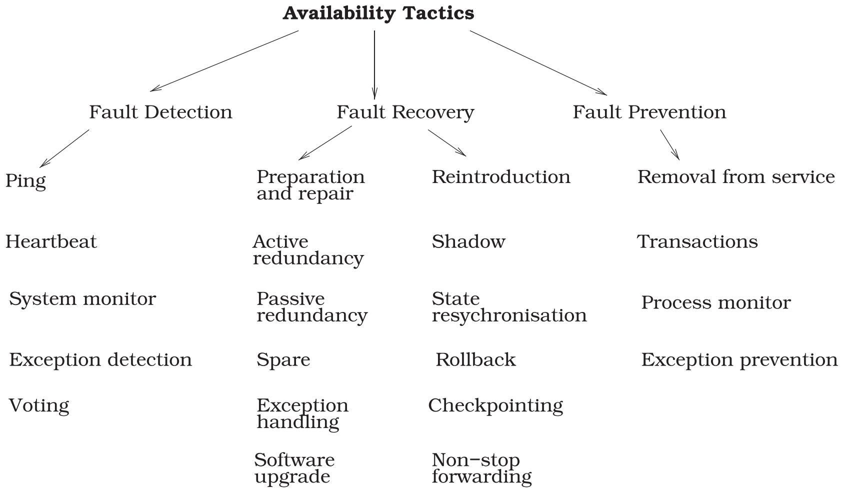


Fig. 1. Availability tactics from Scott & Kazman (2009).

preliminary performance measurements. HdpH-RS is described alongside HdpH in Maier *et al.* (2014c), but with only a few paragraphs dedicated to describing each of the implementation, validation, and evaluation.

2 Related work

2.1 Faults and reliability

A fault is a characteristic of hardware or software that can lead to a system error. An error can lead to an erroneous system state giving a system behaviour that is unexpected by system users. Faults may be due to a range of factors, e.g. incorrect software, or the capacity issues like insufficient memory or persistent storage.

Tactics for fault tolerance detection, recovery, and prevention are shown in Figure 1. Monitors are components that can monitor many parts of a system, such as processors, nodes, and network congestion. They may use heartbeat or ping-pong protocols to monitor remote components in distributed systems. In message passing distributed systems, timestamps can be used to detect or re-order incorrect event sequences. Most existing work in fault tolerance for HPC systems is based on *checkpointing* or *rollback recovery*. Checkpointing methods (Chandy & Lamport, 1985) are based on periodically saving a global or semi-global state to stable storage. Global checkpointing does not scale, and though asynchronous checkpointing approaches have potential to scale on larger systems, they encounter difficult challenges such as rollback propagation, domino effects, and loss of integrity through incorrect rollback in dependency graphs (Elnozahy *et al.*, 2002). *Algorithmic level* fault tolerance is an alternative high-level fault tolerance approach. Examples include leader election algorithms, consensus through voting and quorums, or simply ignore failures by smoothing over missing results with probabilistic accuracy bounds.

2.2 Reliability of large-scale systems

The success of future *HPC architectures* will depend on the ability to provide reliability and availability at scale (Schroeder & Gibson, 2007). As HPC systems continue to increase in scale, their mean time between failure decreases. The current practise for fault tolerance in HPC systems is to use very reliable hardware, e.g. processors and interconnects, and to use checkpointing and rollback. With the increasing error rates and increasing aggregate memory exceeding IO capabilities, checkpointing is fast becoming unusable (Litvinova *et al.*, 2010).

Parallel functional computations are typically very different from most HPC computations that perform relatively regular iterations over arrays of floating points. In contrast, HdpH and other functional computations employ complex data and control structures, use arrays and floating points sparingly, and dynamically create tasks of irregular sizes. Rather than checkpointing, HdpH-RS uses supervision and re-evaluation for fault tolerance.

Big Data frameworks like Google MapReduce (Dean & Ghemawat, 2008), and Hadoop (White, 2012) operate at scale on commodity hardware and hence provide fault tolerance. They provide automatic and largely transparent fault tolerance. Queries are idempotent, the data is replicated in a distributed file store, e.g. by the Hadoop HDFS (White, 2012), the failure of a sub query is detected and it is recomputed. Such frameworks perform a restricted form of data retrieval computation. HdpH-RS, and other parallel functional languages, are more commonly used for large compute-bound tasks, i.e. Big Computation and not Big Data.

HdpH-RS differs from Hadoop in a number of ways. Where a Hadoop master node supervises the health of all slave nodes, supervision is distributed in HdpH-RS, and all HdpH-RS nodes are capable of detecting remote node failures. In Hadoop, *failure detection and task replication is centralised*. The output of map tasks are stored to disk locally in Hadoop. In the presence of failure, completed *map tasks are redundantly re-scheduled*, due to the loss of their results. This is in contrast to HdpH-RS, where the resulting values of evaluating task expressions is transmitted with `rput` as soon as they are calculated. *Failure detection latency* in Hadoop is 10 minutes by default in order to tolerate non-responsiveness and network congestion (Dinu & Ng, 2011). In contrast, the failure detection latency in HdpH-RS is a maximum of 5 seconds by default, and can be modified by the user.

2.3 Fault tolerant distributed languages

Erlang is a distributed actor based functional language that is increasingly popular for developing reliable scalable systems, many with soft real-time requirements. The Erlang approach to failures relies on actors having isolated state, and hence they can ‘let it crash’ and rely on another process to correct the error (Armstrong, 2010). One Erlang process can monitor another Erlang process, which is notified if the monitored process dies. The Erlang reliability model is widely emulated, e.g. by and the Akka library for Scala (Gupta, 2012) and by Cloud Haskell (Epstein *et al.*, 2011).

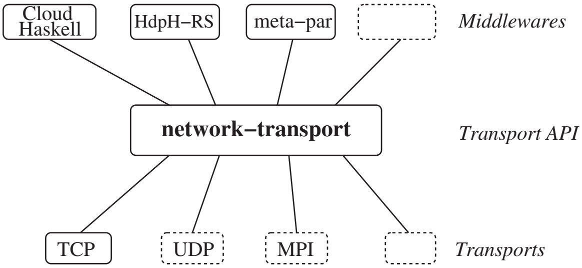


Fig. 2. Distributed Haskell software layers.

The Cloud Haskell and HdpH-RS designs and implementations are closely related. Both languages are implemented entirely in Haskell with GHC extensions and inherit the language features of Haskell, including purity, types, and monads, as well as the multi-paradigm concurrency models in Haskell. Both provide mechanisms for serialising function closures, enabling higher order functions to be used in distributed computing environments. Both are de-coupled into multiple layers, separating the process layer, transport layer, and transport implementations. The software architecture, illustrated in Figure 2, is designed to encourage additional middlewares other than Cloud Haskell for distributed computing, and for alternative network layer implementations other than TCP. Cloud Haskell and HdpH-RS share the TCP network layer.

HdpH-RS reliability differs from Erlang-style reliability in a number of ways, and for concreteness we compare with distributed Erlang. HdpH-RS provides dynamic load management where an idle node may steal a task. In contrast, distributed Erlang spawns a process to a named node (host), and thereafter it cannot migrate. This makes Erlang less suitable than HdpH-RS for computations with irregular parallelism. HdpH-RS recovery in effect automates Erlang OTP supervision behaviours and hence user does not need to handle failures explicitly as in Erlang. Where HdpH-RS is designed to recover only idempotent tasks transparently, i.e. without making faults observable, the recovery of a stateful process in Erlang may make the faults observable. Such a model can provide better availability as nodes (Erlang VMs) can be monitored and restarted from the operating system. In contrast to HdpH-RS and Cloud Haskell, Erlang is dynamically typed, and hence programming errors may be detected only at runtime.

3 Design

3.1 HdpH-RS language

HdpH and HdpH-RS are shallowly embedded Haskell DSLs that support high-level explicit and semi-explicit parallelism. HdpH-RS is *scalable* as it distributes computations across a network of multi-core nodes. It is *portable*, being implemented entirely in Haskell (with GHC extensions) rather than relying on bespoke low-level runtime systems like Glasgow parallel Haskell (Trinder *et al.*, 1996) or Eden (Loogen *et al.*, 2005).

HdpH-RS provides *high-level semi-explicit parallelism* with implicit and explicit task placement and dynamic load management. Implicit placement frees the programmer from controlling work distribution and load management. Instead, idle nodes steal work from busy nodes automatically, thereby maximising utilisation when there is enough work to be stolen at the expense of deterministic execution. HdpH-RS provides low-cost automatic *fault tolerance* using Erlang-style supervision and recovery of location-invariant computations.

HdpH-RS extends the `Par` monad DSL (Marlow *et al.*, 2011) for shared-memory parallelism to distributed memory, and Listing 1 lists the HdpH API. Like (Marlow *et al.*, 2011), HdpH focuses on task parallelism. In distributed memory, this requires serialisation of `Par` computations and results so they can be sent over the network. The *Par* type constructor is a monad for encapsulating a parallel computation. To communicate the results of computation (and to block waiting for their availability), threads employ `IVars` as futures (Halstead Jr., 1985), which are essentially mutable variables that are writable exactly once.

The `spawn` and `spawnAt` primitives immediately return a future of type `IVar (Closure t)`. The actual result can be read by calling `get`, blocking until the result is available. Note that a future is not serialisable, hence cannot be captured by explicit closures. As a result, the future can only be read on the hosting node, i.e. the node it was created on. Details of explicit closures in HdpH-RS are in Maier *et al.* (2014b).

The example in Listing 2 illustrates the use of the HdpH primitives to sum the *Liouville function* (Borwein *et al.*, 2008) from 1 to *n* in parallel. The code shows how to construct a list of `tasks` with the `mkClosure` macro, how to generate parallelism by spawning the `tasks`, how to retrieve the `results` closures, and how to unwrap them and return the final sum. Note how the actual parallel computation, the function `liouville`, applies the `toClosure` primitive to create a result closure containing a fully evaluated `Integer`.

3.2 Algorithmic skeletons

HdpH-RS provides algorithmic skeletons (Cole, 1988) that abstract over the DSL primitives to provide high-level fault tolerant parallelism. Algorithmic skeletons abstract communication, interaction, and commonly used patterns of parallel computation e.g. divide-and-conquer, map/reduce, parallel map, and parallel buffer variants. Programmers that use these skeletons do not create tasks with `spawn`, `spawnAt`, or call `get` on `IVars` directly. Instead, the skeleton implementations handle task creation and placement and synchronisation using `IVars`.

There are two versions of each algorithmic skeleton, one that uses eager scheduling and another that uses lazy scheduling. Lazy skeletons are implemented with combinations of `spawn` and `get`, and rely on the work stealing scheduler to load balance tasks. Eager skeletons are implemented with combinations of `spawnAt` and `get`, and push tasks to nodes as soon as they are created. Node selection with `spawnAt` varies between skeletons. Parallel map skeletons e.g. `pushMap` and `pushMapSliced` select nodes in a round-robin fashion. Nested skeletons e.g. `pushDivideAndConquer`

```
data Par a  -- monadic parallel computation of type 'a'
eval :: a → Par a  -- strict evaluation

data Node   -- explicit location (shared-memory node)
allNodes :: Par [Node]  -- list of all nodes; head is current node

data Closure a  -- explicit closure of type 'a'
unClosure :: Closure a → a
toClosure :: (Binary a) ⇒ a → Closure a
mkClosure   -- Template Haskell closure conversion macro

-- * 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 use)
```

Listing 1. HdpH-RS Primitives.

```
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)
```

Listing 2. Summatory Liouville with HdpH-RS.

(Stewart, 2013b) use random node selection. We explore the performance implications of fault tolerance and lazy/eager scheduling in Section 7.

Two versions of an HdpH-RS parallel map skeleton are shown in Listing 3. Both divide the input list into a number of slices and evaluate each slice in parallel. For example, dividing the list $[e_1, \ldots, e_5]$ into three slices yields a list $[[e_1, e_4], [e_2, e_5], [e_3]]$. The skeletons create three parallel tasks, distributed lazily with `parMapSliced` and eagerly with `pushMapSliced`. The two skeletons actually differ only in task scheduling; task creation and gathering of results, including slicing and unslicing of lists of closures, is uniformly handled by the underlying parametric skeleton `pMapSliced`. The full list of HdpH-RS skeletons are in Stewart (2013b) and their implementations are online (Stewart & Maier, 2013).

3.3 Applicability

HdpH-RS is designed to manage dynamic, irregular, and idempotent task parallelism on large-scale architectures like large clusters or HPCs. Section 7 demonstrates that HdpH-RS delivers good performance for applications with complex algorithms, coordination patterns, and data structures. As HdpH-RS retains backup closures of supervised tasks, its performance is predicated on a small closure footprint:

```
parMapSliced, pushMapSliced  -- lazy and eager slicing parallel maps
  :: (Binary b)          -- result type serialisable (required by toClosure)
  ⇒ Int                  -- number of tasks
  → Closure (a → b)      -- function closure
  → [Closure a]          -- input list
  → Par [Closure b]      -- output list
parMapSliced  = pMapSliced $ mapM spawn
pushMapSliced = pMapSliced $ λtasks → do
                  nodes ← allNodes
                  zipWithM spawnAt (cycle nodes) tasks

pMapSliced  -- slicing parallel map, parametric in task scheduling
  :: (Binary b)
  ⇒ (∀t. [Task t] → Par [Future t])  -- task scheduling parameter
  → Int                              -- number of tasks
  → Closure (a → b)                  -- function closure
  → [Closure a]                      -- input list
  → Par [Closure b]                  -- output list
pMapSliced scheduleTasks n cf cxs = do
  let tasks = [$(mkClosure [| fmap toClosure $ mapM (λcy →
                 fmap toClosure $ eval (unClosure cf $ unClosure cy)) cys |])
              | cys ← slice n cxs]
  ivars    ← scheduleTasks tasks
  results ← mapM get ivars
  return $ unslice $ map unClosure results

slice :: Int → [a] → [[a]]
slice n = transpose ∘ chunk n where
            chunk n [] = []
            chunk n xs = ys : chunk n zs where (ys,zs) = splitAt n xs

unslice :: [[a]] → [a]
unslice = concat ∘ transpose
```

Listing 3. Two versions of a HdpH-RS Skeleton.

either there are few closures, or they are small, or terminate quickly. Thus, HdpH-RS offers a trade-off between fault tolerance and memory use. There are many real-world applications with these characteristics, for example many combinatorial problems or computational algebra problems e.g. Maier *et al.* (2014a).

On the other hand, HdpH-RS is not appropriate for certain classes of application. In particular, we do not target traditional HPC workloads, i.e. regular computations over vectors of floating points. These have little need for the dynamic load management and rich data structures provided by HdpH-RS, and need highly optimised floating point and vector capabilities. Similarly, for good performance task execution time must greatly outweigh communication time, which is largely determined by the size of the closure transmitted. Hence, Big Data workloads with large memory footprints are not well suited. Finally, HdpH-RS would not facilitate the development of applications that rely on reliable distributed data structures, like replicated distributed hash tables, as these are not yet provided.

3.4 Operational semantics

This section presents an operational semantics for HdpH-RS in the style of (Marlow *et al.*, 2011), focusing on fault recovery. Figure 3 introduces the syntax of terms and

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

$$\begin{array}{lll} \text{Values} & V ::= () \mid i \mid p \mid x M_1 \dots M_n \mid \lambda x.M \mid \mathbf{fix}\, M \\ & \quad \mid\ M \texttt{>>=} N \mid \texttt{return}\, M \mid \texttt{eval}\, M \mid \texttt{allNodes} \mid \texttt{spawn}\, M \mid \texttt{spawnAt}\, p\, M \mid \texttt{get}\, i \mid \texttt{rput}\, i\, M \end{array}$$

$$\text{Terms} \quad L, M, N ::= V \mid M N \mid (\texttt{>>=}) \mid \texttt{return} \mid \texttt{eval} \mid \texttt{spawn} \mid \texttt{spawnAt} \mid \texttt{get} \mid \texttt{rput}$$

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

$$\text{Evaluation contexts} \quad \mathscr{E} ::= [\cdot] \mid \mathscr{E} \texttt{>>=} M$$

Fig. 3. Syntax of HdpH-RS terms, values, and states.

values. IVars are represented as names which can be bound by name restriction ν; the set of names occuring free in X (where X may be of any of the syntactic categories in Figure 3) is denoted by $fn(X)$.

The term language of Figure 3 is essentially the same as the embedded DSL presented in Section 3.1, except that it ignores explicit closures, i.e. assumes that all terms are implicitly serialisable. We assume that terms that may be serialised have no free IVars, more precisely, we restrict the last argument of `spawn`, `spawnAt`, and `rput` to terms M such that $fn(M) = \emptyset$. This is justified as in the language design (Section 3.1) these arguments are explicit closures that cannot capture free IVars.

For the purposes of the DSL semantics, the host language is a standard lambda calculus with fixed points and some data constructors for nodes, integers, and lists (omitted to save space). We assume a big-step operational semantics for the host language, and write $M \Downarrow V$ to mean that there is a derivation proving that term M evaluates to value V. The definition of the host language semantics is entirely standard and omitted (Peyton Jones, 2002). Note that the syntax of values in Figure 3 implies that the DSL primitives are strict in arguments of type `Node` and `IVar`.

The operational semantics is a small-step reduction semantics $\longrightarrow$ on the *states* defined in Figure 3. A state is built from *atomic states* by parallel composition and name restriction. Each atomic state has a location, a node indicated by the subscript p. The special atomic state dead_p signals that p has died, modelling node failure detection.

An atomic state of the form $\langle M \rangle_p$ or $\langle\!\langle M \rangle\!\rangle_p$, where M is a computation of type `Par ()`, denotes a *thread* or *spark*, respectively. A thread is a currently executing task and is tied to its current node p; a spark is a task that does not yet execute

$$
\begin{array}{cc}
S \mid T \equiv T \mid S & \nu i.\nu j.S \equiv \nu j.\nu i.S \\
R \mid (S \mid T) \equiv (R \mid S) \mid T & \nu i.(S \mid T) \equiv (\nu i.S) \mid T, \quad i \notin fn(T)
\end{array}
$$

$$
\frac{S \longrightarrow T}{R \mid S \longrightarrow R \mid T} \qquad \frac{S \longrightarrow T}{\nu i.S \longrightarrow \nu i.T} \qquad \frac{S \equiv S' \longrightarrow T' \equiv T}{S \longrightarrow T}
$$

Fig. 4. Structural congruence and structural transitions.

$$
\begin{array}{rll}
\langle \mathscr{E}[M] \rangle_p \longrightarrow \langle \mathscr{E}[V] \rangle_p, & \text{if } M \Downarrow V \text{ and } M \not\equiv V & \text{(normalize)} \\
\langle \mathscr{E}[\mathtt{return}\, N \mathbin{\mathtt{>>=}} M] \rangle_p \longrightarrow \langle \mathscr{E}[M\,N] \rangle_p & & \text{(bind)} \\
\langle \mathscr{E}[\mathtt{eval}\, M] \rangle_p \longrightarrow \langle \mathscr{E}[\mathtt{return}\, V] \rangle_p, & \text{if } M \Downarrow V & \text{(eval)} \\
\langle \mathscr{E}[\mathtt{allNodes}] \rangle_p \longrightarrow \langle \mathscr{E}[\mathtt{return}\, M] \rangle_p, & \text{where } M \text{ is an list of all nodes, starting with } p & \text{(allNodes)} \\
\langle \mathscr{E}[\mathtt{spawn}\, M] \rangle_p \longrightarrow \nu i.(\langle \mathscr{E}[\mathtt{return}\, i] \rangle_p \mid i\{\langle\!\langle M \mathbin{\mathtt{>>=}} \mathtt{rput}\, i \rangle\!\rangle_{\{p\}}\}_p \mid \langle\!\langle M \mathbin{\mathtt{>>=}} \mathtt{rput}\, i \rangle\!\rangle_p), & \text{where } i \notin fn(\mathscr{E}) & \\
& & \text{(spawn)} \\
\langle \mathscr{E}[\mathtt{spawnAt}\, q\, M] \rangle_p \longrightarrow \nu i.(\langle \mathscr{E}[\mathtt{return}\, i] \rangle_p \mid i\{\langle M \mathbin{\mathtt{>>=}} \mathtt{rput}\, i \rangle_q\}_p \mid \langle M \mathbin{\mathtt{>>=}} \mathtt{rput}\, i \rangle_q), & \text{where } i \notin fn(\mathscr{E}) & \\
& & \text{(spawnAt)} \\
\langle \mathscr{E}[\mathtt{rput}\, i\, M] \rangle_p \mid i\{\langle N \rangle_p\}_q \longrightarrow \langle \mathscr{E}[\mathtt{return}\,\mathtt{()}] \rangle_p \mid i\{M\}_q & & \text{(rput\_empty\_thread)} \\
\langle \mathscr{E}[\mathtt{rput}\, i\, M] \rangle_p \mid i\{\langle\!\langle N \rangle\!\rangle_Q\}_q \longrightarrow \langle \mathscr{E}[\mathtt{return}\,\mathtt{()}] \rangle_p \mid i\{M\}_q & & \text{(rput\_empty\_spark)} \\
\langle \mathscr{E}[\mathtt{rput}\, i\, M] \rangle_p \mid i\{N\}_q \longrightarrow \langle \mathscr{E}[\mathtt{return}\,\mathtt{()}] \rangle_p \mid i\{N\}_q, & \text{where } N \neq \bot & \text{(rput\_full)} \\
\langle \mathscr{E}[\mathtt{rput}\, i\, M] \rangle_p \mid i\{\bot\}_q \longrightarrow \langle \mathscr{E}[\mathtt{return}\,\mathtt{()}] \rangle_p \mid i\{\bot\}_q & & \text{(rput\_zombie)} \\
\langle \mathscr{E}[\mathtt{get}\, i] \rangle_p \mid i\{M\}_p \longrightarrow \langle \mathscr{E}[\mathtt{return}\, M] \rangle_p \mid i\{M\}_p, & \text{where } M \neq \bot & \text{(get)} \\
\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 & \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 & \text{(track)} \\
\langle\!\langle M \rangle\!\rangle_p \longrightarrow \langle M \rangle_p & & \text{(convert)} \\
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 & \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 & \text{(recover\_spark)}
\end{array}
$$

$$
\begin{array}{rlrl}
 & & \mathrm{dead}_p \mid \langle\!\langle M \rangle\!\rangle_p \longrightarrow \mathrm{dead}_p & \text{(kill\_spark)} \\
\langle \mathtt{return}\,\mathtt{()} \rangle_p \longrightarrow & \text{(gc\_thread)} & \mathrm{dead}_p \mid \langle M \rangle_p \longrightarrow \mathrm{dead}_p & \text{(kill\_thread)} \\
\nu i.i\{?\}_p \longrightarrow & \text{(gc\_ivar)} & \mathrm{dead}_p \mid i\{?\}_p \longrightarrow \mathrm{dead}_p \mid i\{\bot\}_p & \text{(kill\_ivar)} \\
 & & \longrightarrow \mathrm{dead}_p & \text{(dead)}
\end{array}
$$

Fig. 5. Small-step semantics of HdpH-RS.

and may migrate from p to any other node q. An atomic state of the form $i\{?\}_p$ denotes an *IVar* named i; the place holder "?" signals that we don't care whether i is full, empty, or a zombie. To enable fault recovery, empty IVars $i\{\langle M \rangle_q\}_p$ and $j\{\langle\!\langle N \rangle\!\rangle_Q\}_p$ supervise the thread $\langle M \rangle_q$ resp. spark $\langle\!\langle N \rangle\!\rangle_q$ that is supposed to fill them, maintaining knowledge of their location. In case of a non-migratable thread that knowledge is the exact node q where it was placed by the scheduler. In case of a spark, however, the superving IVar j may not know the actual node due to migration, hence the spark is annotated with a set of nodes Q over-approximating its true location. Figure 4 asserts the usual structural congruence properties of parallel composition and name restriction, and the usual structural transitions propagating reduction under parallel composition and name restriction.

Figure 5 presents the transition rules for HdpH-RS. Most of these rules execute a thread, relying on an *evaluation context* $\mathscr{E}$ to select the first action of the thread's monadic computation. Rules that are similar to those in Marlow *et al.* (2011) are not explained in detail.

The first three rules are standard for monadic DSLs; note how `eval` is just a strict `return`. The rule (allNodes) exposes a list of all the system's nodes, in no particular order except that the current node is the head of the list. The rules (spawn) and (spawnAt) define the work distribution primitives. The primitive `spawn` creates an IVar i on the current node p and wraps its argument M, followed by a write to i, into a spark initially residing on p; it also stores the spark in the empty IVar, as backup for fault recovery. In contrast, `spawnAt` wraps M into a thread, which is placed on node q (and backed up in the empty IVar for fault recovery). The side condition on both rules ensures that the name i is *fresh*, i.e. does not occur free in the current thread. The (rput_*) and (get) rules for IVars are similar to those in Marlow *et al.* (2011) except that IVars in HdpH-RS can only be read on the node they reside on. They can however be written from any node, and writes can be raced[3]; the first write wins and overwrites the backup thread/spark, subsequent writes have no effect. Rules (migrate), (track) and (convert) govern the fault tolerant scheduling of sparks. A spark may freely migrate from node p_1 to p_2, subject to both locations being tracked by set P. The tracking set may change via rule (track) in arbitrary ways, provided the current location of the spark remains a member, modelling the supervising IVar's changing and uncertain knowledge about the location of a supervised spark; Figure 10 in Section 5.4 shows how rules (migrate) and (track) abstract the actual behaviour of the HdpH-RS work stealing algorithm. Migrating sparks cannot be executed directly; instead, rule (convert) must turn them into threads that can execute but not migrate. Finally, the (gc_*) rules eliminate garbage, i.e. terminated threads and inaccessible IVars. Note that to become garbage, IVars must be unreachable, and sparks must be converted and executed to termination; hence, the semantics does not support speculative parallelism.

The remaining rules deal with faults. The four rules at the bottom right of Figure 5 define the *fault model* of HdpH-RS. A node p may die any time, signalled by the spontaneous production of dead_p, and non-deterministically its sparks and threads may disappear and its IVars may turn into zombies. IVars cannot just disappear, or else writes to IVars on dead nodes would get stuck instead of behaving like no-ops. However, some of p's sparks and threads may survive and continue to execute. In this way, the semantics models partial faults and pessimistic notification of faults. Node failure is permanent as no transition consumes dead_p.

Finally, the (recover_thread) and (recover_spark) rules model the recovery of tasks that have been lost due to faults. A thread supervised by IVar i on p and executing on dead node $q \neq p$ is replicated on p, after which i updates the tracking location to p (which effectively rules out further supervision as there is no point supervising a thread on the same node). A spark supervised by IVar i on p and known to reside on some node in the tracking set Q is replicated on p if any node $q \in Q$ is dead; afterwards i continues to supervise, now tracking $\{p\}$, the location of the replica spark. Due to the inherent uncertainty of tracking, sparks may be replicated even when actually residing on healthy nodes.

[3] Since the DSL in Section 3.1 does not expose `rput`, races only occur as a result of task replication by the fault tolerant HdpH-RS scheduler.

The transition rules are illustrated by executions with and without faults in (Stewart, 2013b). Section 5.4 relates the operational semantics to the actual fault tolerant work stealing algorithm of HdpH-RS.

3.5 Transparent fault recovery

We aim to show that, under reasonable assumptions, fault recovery is transparent, i.e. semantically unobservable in the sense that it cannot change the result of reductions. We need to formally define a number of concepts, leading to the *result* of a reduction, namely a normal form of its initial state.

We note that in any reduction, rule (dead) permutes with every rule to the left. That is, in any given reduction it is irrelevant how early a failing node dies; what matters is the subsequent reaction, killing and/or recovering threads and sparks. Thus, for the purpose of investigating normal forms, we ban rule (dead) and instead start reduction from states of the form $S \mid \mathrm{dead}_{p_1} \mid \ldots \mid \mathrm{dead}_{p_n}$, insisting that all failing nodes die right at the start of the reduction. For the remainder of this section, let $F = \{p_1, \ldots, p_n\}$ be a set of failing nodes. We decorate the reduction relation $\longrightarrow$ with F and define $S \longrightarrow_F T$ iff $S \mid \mathrm{dead}_{p_1} \mid \ldots \mid \mathrm{dead}_{p_n} \longrightarrow T \mid \mathrm{dead}_{p_1} \mid \ldots \mid \mathrm{dead}_{p_n}$. We call $\longrightarrow_\emptyset$ reductions *failure-free*; obviously, these reductions cannot use any of the rules (kill_*), (recover_*) and (rput_zombie).

A *main thread* is a thread of the form $\langle M \rangle_p$ with $fn(M) = \emptyset$, and a *root thread* is a thread of the form $\langle M \text{ >>= } \mathtt{rput}\, i \rangle_p$ with $fn(M) = \emptyset$. That is, a main thread has no free IVars, and a root thread has exactly one free IVar, which it only accesses to write its final result. A main thread corresponds to a program's initial thread (e.g. the thread executing the `main` action in a Haskell program) whereas root threads correspond to tasks created by `spawn` or `spawnAt`.

We call a state S *well formed* iff there is a state S_0 consisting of a single main or root thread such that $S_0 \longrightarrow_F^* S$, where $\longrightarrow_F^*$ denotes the reflexive-transitive closure of $\longrightarrow_F$. We observe that reductions starting from well-formed states cannot get stuck except when embedding the host language, namely term M diverging in rules (normalise) and (eval), or when executing the final `rput` of the root thread. In particular, well-formedness guarantees that all other `rput`s find their target IVars, that all `get`s find their source IVars, and that these source IVars reside on the same nodes as the threads executing `get`.

Given a state S, a thread $\langle N \rangle_p$ is *reachable* from S iff there is a state T such that $S \longrightarrow_F^* \nu i_1 \ldots \nu i_n.(T \mid \langle N \rangle_p)$. Thread $\langle N \rangle_p$ is a *normal form* of S, denoted $S \downarrow_F \langle N \rangle_p$, iff $S \longrightarrow_F^* \langle N \rangle_p$ and $\langle N \rangle_p$ is irreducible. If $F = \emptyset$, we call $\langle N \rangle_p$ a *failure-free* normal form. Note that reaching normal form entails $p \notin F$, otherwise $\langle N \rangle_p$ would be reducible by rule (kill_thread). Furthermore, reaching normal form entails that all parallel computation has ceased because threads and sparks have been garbage collected or killed (on failed nodes). Finally note that S may have any number of normal forms.

We observe that the existence of normal forms allows compositional splicing of reductions. More precisely, given a state S and a root thread $\langle M \text{ >>= } \mathtt{rput}\, i_k \rangle_p$ such that $S \longrightarrow_F^* \nu i_1 \ldots \nu i_n.(T \mid \langle M \text{ >>= } \mathtt{rput}\, i_k \rangle_p)$ and $\langle M \text{ >>= } \mathtt{rput}\, i_k \rangle_p \downarrow_F \langle N \rangle_p$, there is a reduction $S \longrightarrow_F^* \nu i_1 \ldots \nu i_n.(T \mid \langle N \rangle_p)$. Note that $fn(\langle N \rangle_p) = fn(\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i_k \rangle_p) = \{i_k\}$ due to the fact that $fn(M) = \emptyset$ and normalisation cannot create free IVars.

We have defined normal forms of arbitrary states S, yet we will mostly be interested in the normal forms of main and root threads, as the latter correspond to tasks spawned and potentially replicated. Ignoring potential divergence in rules (normalise) and (eval), reductions starting from main and root threads do not get stuck — see the above observation on well-formedness — hence, existence (though not uniqueness) of normal forms is guaranteed. In particular, a normal form of a main thread $\langle M \rangle_p$ must be of the form $\langle \texttt{return}\, N \rangle_p$ with $fn(N) = \emptyset$. Likewise, a normal form of a root thread $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p$ must be of the form $\langle \texttt{rput}\, i\, N \rangle_p$ with $fn(N) = \emptyset$. Note that the name of the result IVar has no bearing on the normal forms of a root thread as $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p \downarrow_F \langle \texttt{rput}\, i\, N \rangle_p$ if and only if $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, j \rangle_p \downarrow_F \langle \texttt{rput}\, j\, N \rangle_p$ for all IVars i and j. However, HdpH-RS is a location-aware DSL, so moving a thread from node p to q may substantially alter its normal forms.

We aim to transform reductions with failures into failure-free reductions, preserving normal forms. This isn't possible in general; it does require some restriction on the use of location information. We call a thread $\langle M \rangle_p$ *location invariant* iff it does not matter where it executes; that is, $\langle M \rangle_p \downarrow_F \langle N \rangle_p$ if and only if $\langle M \rangle_q \downarrow_F \langle N \rangle_q$ for all sets of nodes F and all nodes p and q. We call $\langle M \rangle_p$ *transitively location invariant* iff $\langle M \rangle_p$ is location invariant and all root threads $\langle N \mathbin{\texttt{>>=}} \texttt{rput}\, j \rangle_q$ reachable from $\langle M \rangle_p$ are location invariant.

Now we can state the claim of the main theorem below. The normal forms of a transitively location invariant main thread coincide with its failure-free normal forms. That is, the effects of failure and recovery cannot be observed in the results of reductions (provided the main thread executes on a non-failing node).

Theorem 1 *Let F be a set of failing nodes and $\langle M \rangle_p$ a transitively location invariant main thread with $p \notin F$. Then for all normal forms $\langle N \rangle_p$, we have $\langle M \rangle_p \downarrow_F \langle N \rangle_p$ if and only if $\langle M \rangle_p \downarrow_\emptyset \langle N \rangle_p$.*

Proof sketch. The reverse implication is trivial as the failure-free reductions are a subset of all possible reductions.

Proving the forward implication is done by splicing together reductions normalising the root threads arising from calls to `spawn` and `spawnAt` in the body of M. The root thread reductions in question are failure-free according to Lemma 2 below. Hence, an argument similar to the step case in the proof of Lemma 2 yields the conclusion that there is a (spliced) failure-free reduction from $\langle M \rangle_p$ to $\langle N \rangle_p$.

Lemma 2 *Let F be a set of failing nodes, and let $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p$ be a transitively location invariant root thread with $p \notin F$. Then $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p \downarrow_F \langle \texttt{rput}\, i\, N \rangle_p$ implies $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p \downarrow_\emptyset \langle \texttt{rput}\, i\, N \rangle_p$.*

Proof sketch. We prove the claim by induction on the number of calls to `spawn` and `spawnAt` occurring during the normalisation of $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i \rangle_p$.

- Base case. Suppose the reduction from $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i\rangle_p$ to $\langle \texttt{rput}\, i\, N\rangle_p$ involves no calls to `spawn` and `spawnAt`. Then all states along the reduction are single threads of the form $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i\rangle_p$, so none of the rules (recover_*) and (kill_*) apply. Hence, the reduction is already failure-free.
- Step case. Suppose the reduction from $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i\rangle_p$ to $\langle \texttt{rput}\, i\, N\rangle_p$ is already failure-free up to a state S of the form $S = \nu i_1 \ldots \nu i_n.(T \mid \langle \texttt{spawn}\, M' \mathbin{\texttt{>>=}} M''\rangle_p)$. Rule (spawn) will create a new IVar i_{n+1} and spark $\langle\!\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle\!\rangle_p$ on non-failing node p. The spark may migrate, get converted to a root thread, get killed, be recovered, migrate, get killed, and so on. Eventually though, the IVar i_{n+1} will be filled by some thread $\langle \texttt{rput}\, i_{n+1}\, N'\rangle_q$, which implies that $\langle \texttt{rput}\, i_{n+1}\, N'\rangle_q$ is a normal form of the root thread $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle_q$. We may assume that q is not a failing node, for otherwise we could use location invariance to transplant the normalisation of $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle_q$ to a another non-failing node. Thus, we have $q \notin F$ and $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle_q \downarrow_F \langle \texttt{rput}\, i_{n+1}\, N'\rangle_q$, from which we get $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle_q \downarrow_\emptyset \langle \texttt{rput}\, i_{n+1}\, N'\rangle_q$ by induction hypothesis. Therefore, we can extend the above failure-free reduction $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i\rangle_p \longrightarrow_\emptyset^* S$ from state S to a state S' of the form $S' = \nu i_1 \ldots \nu i_{n+1}.(T \mid i_{n+1}\{N'\}_p \mid \langle M''\rangle_p)$ by applying rule (spawn), followed by (track) and (migrate) to move the spark from p to q, followed by (convert) and the spliced-in failure-free reduction of $\langle M' \mathbin{\texttt{>>=}} \texttt{rput}\, i_{n+1}\rangle_q$ to normal form, followed by rule (rput_empty_spark).

  The argument for extending failure-free reductions over calls to `spawnAt` is similar if a little simpler because tasks created by `spawnAt` do not migrate and are recovered at most once. The claim follows by repeating the construction, eventually yielding a failure-free reduction from $\langle M \mathbin{\texttt{>>=}} \texttt{rput}\, i\rangle_p$ to $\langle \texttt{rput}\, i\, N\rangle_p$.

Observations. First, our definition of (transitive) location invariance is in terms of reductions, i.e. rests on the operational semantics. However, there are many cases where location invariance is guaranteed statically. For instance, skeletons like `pushMapSliced` (Listing 3) use location information only for scheduling decisions; in particular, locations are never written to IVars and never compared. Thus, the resulting skeletons are clearly location invariant. We leave a more thorough static characterisation of location invariance, e.g. in the form of a type system, for future work.

Secondly, for the purpose of presenting a simple semantics, we have ignored all observable effects apart from locations, and location invariance took care of reconciling the effects with task replication. A DSL with more realistic effects (e.g. tasks performing IO) would have to take more care. On top of location invariance, effects would need to be *idempotent*, i.e. invariant under replication, in order to guarantee the unobservability of failure and recovery.

Finally, HdpH-RS is a non-deterministic DSL as decisions taken by the non-deterministic scheduler may become observable, e.g. in case migrating tasks query their current location. However, the sub-language that restricts HdpH-RS task distribution to `spawnAt` only is deterministic, due to entirely deterministic scheduling. Whether there are more interesting deterministic sub-languages, in the face of truly

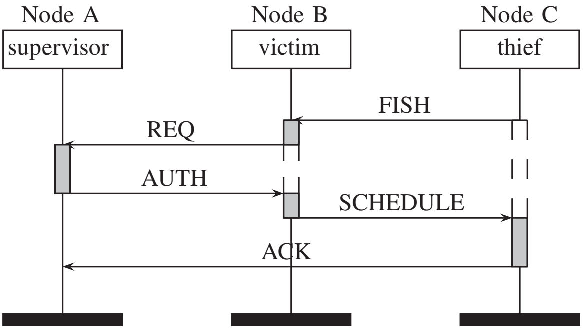


Fig. 6. Fault tolerant fishing protocol in HdpH-RS.

non-deterministic scheduling, is an interesting and timely (Kuper *et al.*, 2014) open question.

4 Reliable work stealing protocol

4.1 *Work stealing protocol*

The HdpH-RS fault tolerant work stealing protocol adds resilience to sparks and threads and involves a victim, a thief, and a supervisor. A supervisor is the node where a supervised spark was created, and is responsible for guaranteeing the execution of that spark. A thief is a node with few sparks to execute. A victim is a node that may hold sparks and is targeted by a thief. The HdpH-RS runtime system messages serve two purposes: to schedule sparks from heavily loaded nodes to idle nodes, and to allow supervisors to track the location of sparks as they migrate between nodes.

The message handlers that implement the fishing protocol are in Section 4.4. A successful work stealing attempt is illustrated in Figure 6. A thief node C targets a victim node B by sending a `FISH` message. The victim requests a scheduling authorisation from the supervisor with `REQ`. The supervisor grants authorisation with `AUTH`, and a spark is scheduled from the victim to the thief in a `SCHEDULE` message. When the thief receives the `SCHEDULE`, it sends an `ACK` to the supervisor.

The HdpH-RS runtime system messages are described in Table 1. The *Message* column is the message type, the *From* and *To* columns distinguish a supervisor node (S), a thief node (T), and a victim node (V). The *Description* column shows the purpose of the message. The use of each message are described in the scheduling algorithms in Section 4.4.

4.2 *Task locality*

For the supervisor to determine whether a spark is lost when a remote node has failed, the migration of a spark needs to be tracked. This is made possible by the runtime system messages `REQ` and `ACK`. An example of task migration tracking is in shown in Figure 7. Node A is supervising spark 0. The messages `REQ` and `ACK` are

Table 1. *HdpH-RS messages*

Message	From	To	Description
`FISH` *thief*	T	V	Fishing request from a thief.
`SCHEDULE` *spark victim*	V	T	Victim schedules a spark to a thief.
`NOWORK`	V	T	Response to `FISH`: victim informs thief that it either does not hold a spark, or was not authorised to schedule a spark.
`REQ` *ref seq victim thief*	V	S	Victim requests authorisation to send spark to thief.
`DENIED` *thief*	S	V	Supervisor denies a scheduling request with respect to `REQ`.
`AUTH` *thief*	S	V	Supervisor authorises a scheduling request with respect to `REQ`.
`OBSOLETE` *thief*	S	V	In response to `REQ`: the task waiting to be scheduled by victim is an obsolete task copy. Victim reacts to `OBSOLETE` by discarding task and sending `NOWORK` to thief.
`ACK` *ref seq thief*	T	S	Thief sends an `ACK` to the supervisor of a spark it has received.
`DEADNODE` *node*	Any node to itself		A message from a node to itself reporting failure of a remote node.

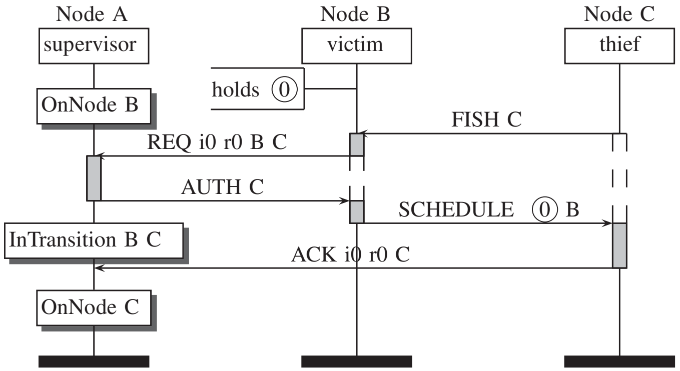


Fig. 7. Migration tracking with message passing.

received by the supervising node A to keep track of a spark's location. Sparks and threads can therefore be identified by their corresponding `IVar`.

The `IVar` data structure in Hdph-RS plays an important role for fault tolerance, shown in Listing 4. Not only are they used to implement futures, they also store supervision information about the corresponding task. An empty `IVar` keeps track of the `get` calls from other threads in `blockedThreads`, and supervision state in `supervisedState`. The `SupervisedTaskState` keeps a local copy of the corresponding task in `task`, a task replica count in `replicaNum`, and the known task location in `location` — either `OnNode node` or `InTransition [node]`.

```
type IVar a = IORef (IVarContent a)
data IVarContent a = Full a
                   | Empty { blockedThreads  :: [a → Thread]
                           , supervisedState :: SupervisedTaskState }

data SupervisedTaskState = SupervisedTaskState
  { task :: Closure (Par ())
  , replicaNum :: Int
  , location :: TaskLocation }

data TaskLocation = OnNode NodeId
                  | InTransition [NodeId]
```

Listing 4. Holding Supervision State in Empty IVars.

The message REQ is used to request authorisation to schedule the spark to another node. If the supervisor knows that it is in the sparkpool of a node (i.e. OnNode thief), then it will authorise the fishing request with AUTH. If the supervisor believes it is in-flight between two nodes, then it will deny the request with DENIED. Examples of unsuccessful work stealing attempts are given in Stewart (2013b).

4.3 Duplicate sparks

Location tracking for a task switches between two states in the corresponding empty IVar (Listing 4), either OnNode if the supervisor receives an ACK or InTransition if the supervisor authorises a migration in response to a REQ. To ensure the safety of sparks, the scheduler makes pessimistic assumptions that tasks have been lost when a node fails. If a supervisor is certain that a spark was on the failed node, then it is replicated. If a supervisor believes a spark to be in-flight either towards or away from the failed node during a fishing operation, again the spark is replicated. The consequence is that the scheduler may create duplicates.

Duplicates of the same spark can co-exist in a distributed environment with one constraint. Older obsolete spark replicas are not permitted to migrate through work stealing, as multiple migrating copies with the same reference may potentially lead to inconsistent location tracking. However, they *are* permitted to transmit results to IVars using rput. Thanks to idempotence, this scenario is indistinguishable from the one where the obsolete replica has been lost. This possibility is illustrated in Figure 8. A spark is created with spawn on node A and then fished by node B. B fails during a fishing exchange between B and C, and A receives a DEADNODE notification of the failure of B. At this point, supervising node A has not yet received an ACK from C, and pessimistically replicates the spark locally. In this instance, the original spark survived the failure of node B and there are now two copies of the spark. This is a partial execution, so neither nodes C or D send an ACK in this message sequence to return the spark supervision state to OnNode on node A.

4.4 Fault tolerant scheduling algorithm

The message handlers that implement the fault tolerant algorithm are in Algorithm 1. Each node is proactive in their search for work with fishing, triggering a sequence

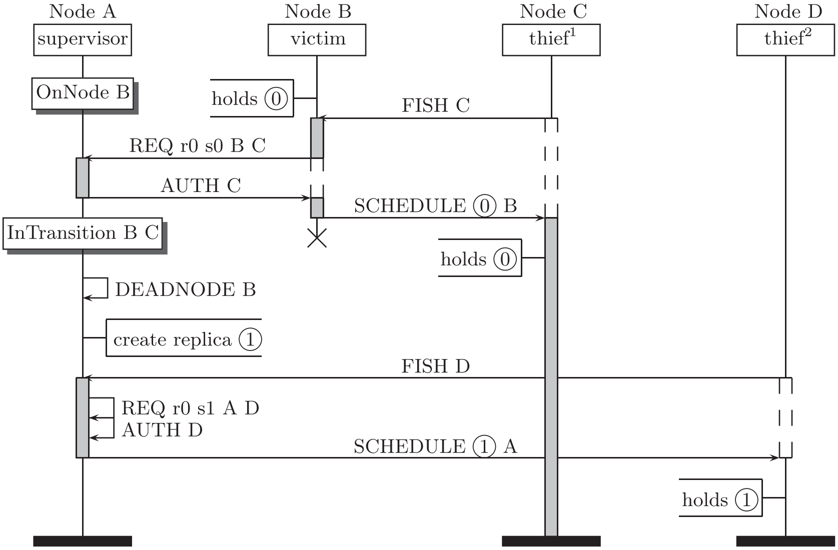


Fig. 8. Pessimistic scheduling that leads to spark duplication.

of messages between the victim, thieving, and supervising nodes. The `FISH` handler (line 1) shows how a node that receives a `FISH` has been targeted by a thief. The victim checks that this node has not already been targeted by another thief (line 2) and is waiting for authorisation. If it is, then a `NOWORK` reply is sent to the thief. Otherwise, if the victim has an available spark, then it is reserved and the victim sends a `REQ` to the supervisor. If the sparkpool is empty, a `NOWORK` is sent to the thief.

The `REQ` handler is shown on line 11. If the location of the reserved spark is known precisely by the supervisor, the request is granted with `AUTH`. Otherwise, if the task is believed to be in transition between two nodes, the request is rejected with `DENIED`. If the spark is obsolete, then the victim is instructed to discard it with an `OBSOLETE` message.

A victim's `AUTH` handler is shown on line 21. It sends the reserved spark to a thief in a `SCHEDULE` message on line 22. The handler for scheduled sparks is shown on line 23. A thief adds the spark to its own sparkpool (line 24), and sends an acknowledgement of its arrival to its supervisor (line 25). However, if a victim is informed that the reserved spark is an obsolete copy with `OBSOLETE` from the supervisor, it discards the reserved spark and sends a `NOWORK` message to the thief. The pseudo code for the `OBSOLETE` and `DENIED` message handlers is in Stewart (2013b).

The supervisor handler for `ACK` messages is shown on line 26. It updates the migration tracking for this spark to `OnNode`, a state that will allow another thief to steal from the new host of the spark.

The handler for `DEADNODE` messages is on line 28. There are four checks performed by every node when a remote node fails. First, it checks if it is waiting for a fishing

Algorithm 1 Fault Tolerant Algorithm Message Handlers

```
    Assumption: Thief (fisher) is looking for work.
 1: function HANDLE(FISH thief)
 2:     if not waiting for auth then          ▷ is there an outstanding authorisation request
 3:         if sparkpool not empty then
 4:             spark ← pop sparkpool
 5:             reserve spark
 6:             send spark.supervisor (REQ spark.ref spark.replica myNode thief)
 7:         else
 8:             send thief NOWORK
 9:     else
10:         send thief NOWORK

    Assumption: A schedule request is sent from a victim to this supervising node.
11: function HANDLE(REQ ref replica victim thief)
12:     if replicaOfref == replica then       ▷ remote task is most recent copy
13:         location ← locationOf ref
14:         if location == OnNode then        ▷ task location known
15:             update location (InTransition victim thief)
16:             send victim (AUTH thief)      ▷ authorise the request
17:         else if location == InTransition then
18:             send victim (DENIED thief)    ▷ deny the request
19:     else
20:         send victim (OBSOLETE thief)      ▷ task is old copy, order victim to discard

    Assumption: Location state on supervisor was OnNode.
21: function HANDLE(AUTH ref thief)
22:     send thief (SCHEDULE reserved_spark)  ▷ send thief the spark

    Assumption: A Victim was authorised to send this node a spark in a SCHEDULE.
23: function HANDLE(SCHEDULE spark)
24:     insert spark sparkpool                ▷ add spark to sparkpool
25:     send spark.supervisor (ACK spark.ref spark.replica myNode)

    Assumption: Thief receives a spark.
26: function HANDLE(ACK ref thief)
27:     update (locationOf ref) (OnNode thief)  ▷ set spark location to OnNode

    Assumption: A remote node has died.
28: function HANDLE(DEADNODE deadNode)
29:     if fishVictim == deadNode then
30:         resume fishing
31:     if thiefOfGuardedSpark == deadNode then
32:         insert reserved_spark sparkpool   ▷ return reserved spark to sparkpool
33:     for all s ∈ sparks on deadNode do
34:         insert s sparkpool                ▷ replicate potentially lost spark
35:     for all t ∈ threads on deadNode do
36:         insert t threadpool               ▷ replicate potentially lost thread: convert & execute locally
```

reply from the dead node (line 29). Second, whether the dead node is the thief of the spark it has requested authorisation for (line 31). Third, it identifies the sparks are at-risk due to the remote node failure (line 33). Fourth, it identifies the threads are at-risk due to the remote node failure (line 35). All at-risk sparks are replicated and added to the local sparkpool. These duplicates can be fished again for load balancing (line 34). All at-risk threads are replicated and are converted and executed locally (line 36).

5 Validating reliable work stealing

5.1 Desirable scheduling properties with non-deterministic systems

The SPIN model checker is used to ensure that the HdpH-RS scheduling algorithm honours the small-step semantics (Section 3.4), supervising sparks in the absence and presence of faults. Due to the many sources of non-determinism in faulty systems, it is easy to make mistakes in correctness arguments for fault tolerant distributed systems. Hence, distributed faulty systems are natural candidates for model checking (John *et al.*, 2013), and specifically because of two properties of HdpH-RS:

1. **Asynchronous message passing.** Causal ordering (Lamport, 1978) of asynchronous distributed scheduling events is not consistent with wall-clock times. HdpH-RS message passing between nodes is asynchronous and non-blocking, instead writing to channel buffers. Because of communication delays, knowledge of remote node availability could be outdated (Xu & Lau, 1997).
2. **Work stealing.** To recover tasks in the presence of failure, a supervisor must be able to detect node failure and must always know the location of its tasks. The asynchronous message passing for stealing work complicates location tracking. The protocol for reliably relocating tasks between nodes in the presence of failure is intricate (Section 4), and model checking the protocol increases confidence in the design.

The HdpH-RS reliable scheduling algorithm is abstracted into a Promela model. The supervisor and worker abstractions are in Appendix A, and the full model is available online (Stewart, 2013a). Promela (Holzmann, 2004) is a meta-language for building verification models, and the language features are intended to facilitate the construction of high-level abstractions of distributed systems. Temporal logic (Prior, 1957) provides a formalism for describing the occurrence of event in time that is suitable for reasoning about concurrent programs (Pnueli, 1977). The SPIN model checker is used to verify key RS properties of the algorithm design from Section 4.4, expressed in linear temporal logic (LTL). The properties are as follows:

1. *The IVar is empty until a result is sent*: Evaluating a task involves transmitting a value to the supervisor, the host of the `IVar` associated with the task. This property verifies that the `IVar` cannot be full until one of the nodes has transmitted a value to the supervisor.
2. *The IVar is eventually always full*: The `IVar` will eventually be filled by either a remaining worker, or the supervisor. This is despite the failure of any or all worker nodes.

The LTL properties guarantee spark evaluation, i.e. the associated future (`IVar`) on the supervising node is eventually filled. A counter property is used to ensure the Promela abstraction does model potential failure of any or all of the worker nodes. It checks that the model potentially kills one or more mortal workers, and SPIN is used to find counter-example executions when one of the workers terminates.

5.2 HdpH-RS abstraction

Promela programs consist of processes, message channels, and variables. Processes are global objects that represent the concurrent nodes in HdpH-RS. The HdpH-RS scheduler has been simplified to its core supervision behaviours that ensure task survival. The model considers tasks scheduled with `spawn` in HdpH-RS — as the location of threads scheduled with `spawnAt` is always known and would not elicit race conditions on location tracking `REQ` and `ACK` messages.

Scope. The Promela model is an abstraction that encapsulates behaviours necessary for guaranteeing the evaluation of sparks. There are six characteristics of the HdpH-RS scheduler in the Promela model:

1. **One supervisor** that initially puts a spark into its local sparkpool. It also creates spark replicas when necessary (item 6). The supervisor does not die.
2. **Three workers** that attempt to steal work from the supervisor and each other. Failure of these nodes is modelled by terminating the Promela process for each node. The workers can die.
3. **Computation** of a spark may happen at any time by any node that holds a copy of the spark. This simulates the execution of the spark, which would eventually invoke an `rput` call to fill the `IVar` on the supervisor. It is modelled by sending a `RESULT` message to the supervisor.
4. **Failure** of a worker node means that future messages to it are lost. The (dead) transition rule is modelled with a non-deterministic suicidal choice any of the three worker nodes can make. This choice results in a node asynchronously broadcasting its death to the remaining healthy nodes and then terminating. Failure detection is modelled by healthy nodes receiving `DEADNODE` messages.
5. **Asynchronicity** of both message passing and failure detection is modelled in Promela using buffered channels. Buffered channels model the buffered FIFO TCP connections in HdpH-RS.
6. **Replication** is used by the supervisor to ensure the safety of a potentially lost spark in the presence of node failure. The model includes spark replication from Section 4.4, honouring the (recover_spark) small-step transition rule in Section 3.4. Replication numbers are used to tag spark replicas in order to identify obsolete spark copies. Obsolete replica migration could potentially invalidate location records for a spark. Therefore, victims are asked to discard obsolete sparks.

Termination of the scheduling algorithm is enforced in the model by aging the spark through transitions of the model. The age of the spark is zero at the initial system state. Each time it is scheduled to a node, its age is incremented. Moreover, each time it must be replicated by the supervisor its age is again incremented. When the age of the spark reaches 100, it is executed immediately. This models the HdpH-RS assumption that a scheduler will eventually execute the spark in a sparkpool and send a result to an empty `IVar` with an `rput` call.

Some aspects of the HdpH-RS scheduler design and implementation are abstracted away in the Promela model, because they are not part of the fault tolerance mechanism. The model involves only *one* `IVar` and one spark, which may manifest into multiple replicas — one active and the rest obsolete. Multiple `IVars` are not modelled, nor are threads created with `spawnAt`.

5.3 Scheduling model

Nodes interact with message passing: `!` to send and `?` to receive. The channels in the Promela model of HdpH-RS are asynchronous, so that messages can be sent to a channel buffer, rather than being blocked waiting on a synchronised participatory receiver. This reflects the data buffers in the underlying TCP sockets and the asynchronous HdpH-RS model.

Supervisor node. The supervisor is modelled as an *active* `proctype`, so is instantiated in the initial system state. The supervisor executes a repetitive control flow that receives work stealing messages from worker nodes and authorisation messages from the supervisor, shown in Listing A1. It creates the initial spark copy and then initiates the three workers. The underlying automaton is a message handling loop from `SUPERVISOR_RECEIVE`. The exception is when the spark has migrated 100 times the supervisor sends a `RESULT` message to itself to force termination. The label `SUPERVISOR_RECEIVE` is re-visited after the non-deterministic message handling choice, and is only escaped if a `RESULT` message has been received. In this case, the `IVar` becomes full and the supervisor terminates.

Worker nodes. Each worker executes a repetitive control flow that receives work stealing message from worker nodes and authorisation messages from the supervisor, shown in Listing A2. The underlying automaton is a message handling loop from `WORKER_RECEIVE`.

The exception is when the spark has migrated 100 times a `RESULT` message is sent to the supervisor. Otherwise, the control flow takes one of three non-deterministic choices: the node may die; it may send a `RESULT` message to the supervisor if it holds a replica; it may receive a work stealing message from a work or scheduling request response from the supervisor. The `WORKER_RECEIVE` label is re-visited after the non-deterministic message handling choice, and is only escaped if it has died or the `IVar` on the supervisor is full. In either case the worker terminates.

The HdpH-RS transport layer sends `DEADNODE` messages to the scheduler message handler on each node when a connection with a remote node is lost. Failure is modelled by a non-deterministic choice that worker nodes can make whilst waiting for messages to arrive. A node can choose to die, which triggers the sending of a `DEADNODE` message to all remaining alive nodes. Sending this message to all four nodes is not executed atomically in the Promela model, reflecting the different failure detection latencies on each HdpH-RS node. SPIN is therefore able to search

```
typedef Sparkpool
 { int spark_count; /* #sparks in pool */
   int spark; }  /* highest #replica */

typedef Spark
 { int highestReplica=0;
   Location location;
   int age=0; }

mtype = { ONNODE , INTRANSITION };
typedef Location
 { int from;
   int to;
   mtype context=ONNODE;
   int at=3; }
```

```
typedef SupervisorNode
 { Sparkpool sparkpool;
   bool waitingSchedAuth=false;
   bool resultSent=false;
   bit ivar=0; }

typedef WorkerNode
 { Sparkpool sparkpool;
   int waitingFishReplyFrom;
   bool waitingSchedAuth=false;
   bool resultSent=false;
   bool dead=false; }
```

Fig. 9. State abstraction in Promela model.

through state transitions whereby a node failure is only partially detected across all nodes.

5.3.1 Node state

The supervisor and worker nodes each have a local sparkpool. The state of the supervisor, each worker node, and a sparkpool is in Figure 9. The sparkpool either holds a spark (where `spark_count` >1), or it is empty. When it holds a spark, its replication number `spark` is used to send messages to the supervisor: to request scheduling authorisation in a `REQ` message, and confirm receipt of the spark with an `ACK` message.

The spark's location is stored in the `Location` typedef. The `from`, `to`, `at`, and `context` fields are modified by the supervisor's `REQ` and `ACK` message handlers. The most recently allocated replica number is held in `highestReplica` and is initially set to 0. The age of the spark on line 4 is initially set to 0, and is incremented when the spark is scheduled to another node or when it is replicated.

In the initial system state, it adds the spark to its sparkpool, which may be fished away by a worker node. To minimise the size of the state machine, the supervisor does not try to steal the spark or subsequent replicas once they are fished away. The `IVar` is represented by a bit on line 5, 0 for empty and 1 for full. Lastly, a `waitingSchedAuth` (line 3) is used to reject incoming `REQ` messages, whilst it waits for an `AUTH` from itself if it is targeted by a thief.

When a thieving node proactively sends a fish to another node, the channel index identifier is stored in `waitingFishReplyFrom` for the victim i.e. 0, 1 or 2 if targeting a worker node, or 3 if targeting the supervisor. The value of `waitingFishReplyFrom` is reset to −1 when a `SCHEDULE` is received, or `NOWORK` message is received allowing the node to resume fishing. When a victim has sent a `REQ` to the supervisor, the `waitingSchedAuth` boolean on line 4 is used to reject subsequent fishing attempts from other thieves until it receives a `AUTH` or `NOWORK`. To avoid infinite work stealing attempts between a thief and a victim, thief keeps track of which nodes it has failed to steal a spark from and does not repeat the fishing request to these nodes (Stewart, 2013b).

Table 2. *Model checking results*

LTL Formula	Counter property	Depth	States	Transitions	Memory
$\Box\ all\_workers\_alive$	Yes	11	5	5	0.2 Mb
$\Box\ (ivar\_empty\ U\ any\_result\_sent)$	No	124	3.7 m	7.4 m	83.8 Mb
$\Diamond\Box\ ivar\_full$	No	124	8.2 m	22.4 m	84.7 Mb

5.3.2 Spark location tracking

When the task tracker records on the supervisor is `ONNODE`, then it can be sure that the node identified with `spark.location.at` is holding the spark. If the node fails at this point, then the spark should be recreated as it has certainly been lost. However, when a spark is in transition between two nodes i.e. `INTRANSITION`, the supervisor cannot be sure of the location of the spark; it is on either of the nodes identified by `spark.location.from` or `spark.location.to`. To overcome this uncertainty, the model faithfully reflects the HdpH-RS pessimistic duplication strategy when a `DEADNODE` is received. This potentially generates replicas that concurrently exist in the model and they are handled using replica counts (Section 4.3).

5.4 Correspondence with operational semantics

An example of location tracking with the `migrate_spark` rule (Section 3.4) is shown in Figure 10. It is a more detailed version of the fault tolerant fishing protocol from Figure 7. The Promela message passing syntax in the message sequence diagram is an abstraction of `send` and `receive` Haskell function calls in the HdpH-RS implementation. Node B is the victim, and node C is the thief. The states for the `IVar` and the spark are $i\{\langle\langle M\rangle\rangle_{\{B\}}\}_A$ and $\langle\langle M\rangle\rangle_B$, showing that the supervisor A holds `IVar` i and that its corresponding spark is on node B. The (track) rule is fired before sending `AUTH` to the victim. The (migrate) rule is fired when the thief receives the `SCHEDULE` message containing the spark. The (track) rule is fired when the supervisor receives an `ACK` message from the thief. Once this message sequence is executed, the new states for the `IVar` and the spark are $i\{\langle\langle M\rangle\rangle_{\{C\}}\}_A$ and $\langle\langle M\rangle\rangle_C$.

5.5 Model checking results

LTL is used to reason about causal and temporal relations of the HdpH-RS scheduler properties. The propositional symbols used in the verification of scheduling Promela model are shown in Listing 5. The results of model checking the three LTL properties are in Table 2. Taking the $\Diamond\Box\ ivar\_full$ property as an example, the results can be interpreted as follows. A reachable depth of 124 is found by SPIN for the model. The reachable state space is 8.2 million. A total of 22.4 million transitions were explored in the search. Actual memory usage for states was 84.7 Mb.

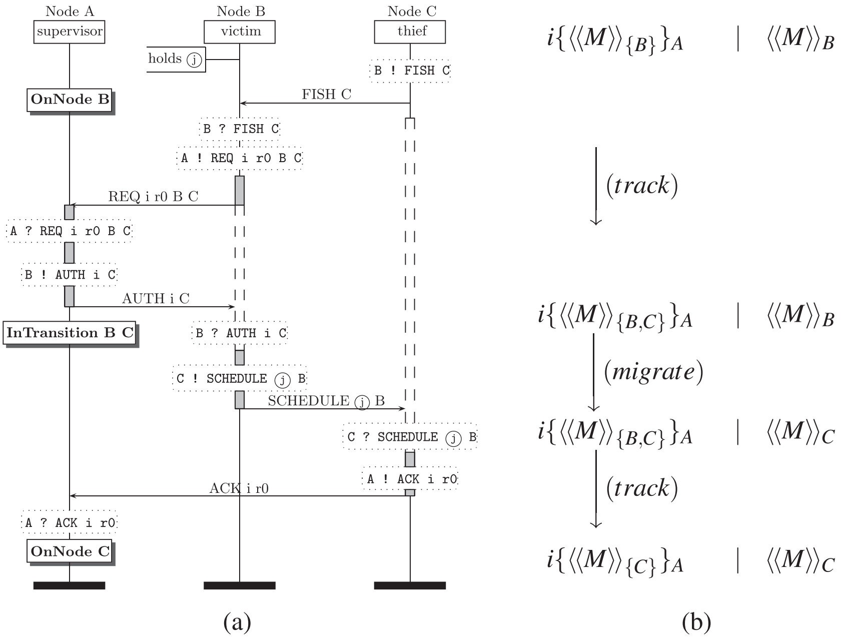

Node A
supervisor
Node B
victim
Node C
thief
holds ⓙ
B ! FISH C
FISH C
OnNode B
B ? FISH C
A ! REQ i r0 B C
REQ i r0 B C
A ? REQ i r0 B C
B ! AUTH i C
AUTH i C
InTransition B C
B ? AUTH i C
C ! SCHEDULE ⓙ B
SCHEDULE ⓙ B
C ? SCHEDULE ⓙ B
A ! ACK i r0
ACK i r0
A ? ACK i r0
OnNode C
(a)
$i\{\langle\langle M\rangle\rangle_{\{B\}}\}_A \mid \langle\langle M\rangle\rangle_B$
$(track)$
$i\{\langle\langle M\rangle\rangle_{\{B,C\}}\}_A \mid \langle\langle M\rangle\rangle_B$
$(migrate)$
$i\{\langle\langle M\rangle\rangle_{\{B,C\}}\}_A \mid \langle\langle M\rangle\rangle_C$
$(track)$
$i\{\langle\langle M\rangle\rangle_{\{C\}}\}_A \mid \langle\langle M\rangle\rangle_C$
(b)

Fig. 10. Location tracking with *(migrate)* and *(track)* rules. (a) Message sequence. (b) State transitions.

```
/* IVar on the supervisor node is full */
#define ivar_full  ( supervisor.ivar == 1 )

/* IVar on the supervisor node is empty */
#define ivar_empty ( supervisor.ivar == 0 )

/* No worker nodes have failed */
#define all_workers_alive ( !worker[0].dead && !worker[1].dead && !worker[2].dead )

/* One or more nodes have transmitted a value to supervisor to fill IVar */
#define any_result_sent
    (  supervisor.resultSent || worker[0].resultSent
     || worker[1].resultSent || worker[2].resultSent )
```

Listing 5. Propositional Symbols used in LTL Formulae of Fault Tolerant Properties.

The IVar is empty until a result is sent. To check that the model is faithful to the fact that an IVar is empty at least until its corresponding task has been evaluated, the $\Box$ $(ivar\_empty\ U\ any\_result\_sent)$ formula is verified. SPIN searches for two violating system states. First, where ivar_empty is false before any_result_sent is true. Second, a cycle of identical system states is identified while any_result_sent remains false. This is due to the nature of the *strong until* connective stipulating that the symbol any_result_sent must eventually be true in some future state. SPIN cannot find a violating system state after exhaustively searching 3.7 million reachable states up to a depth of 124.

The IVar is eventually always full. The key property for fault tolerance is that the IVar on the supervisor must eventually always be full. This would indicate that either a worker node has sent a RESULT message, or the supervisor has written to the IVar locally. To verify, SPIN searches for a system state cycle when ivar_full remains false i.e. supervisor.ivar==0.

This LTL property checks for the fatal scenario when the supervisor does not replicate the spark in the model, when instead it must ensure the existence of at least one replica. This would happen if the HdpH-RS fault tolerant fishing protocol algorithm (Section 4) did not correctly track spark location. This may cause a supervisor to ignore DEADNODE message when it instead should replicate a spark, which would mean no RESULT message would be sent to the supervisor—a cause of deadlock in the HdpH-RS implementation. To verify SPIN searches for states where the spark_count value is 0 for all nodes, and the supervisor does not create a replica in any future state. SPIN cannot find a violating system state after exhaustively searching 8.2 million reachable states up to a depth of 124.

Validating the possibility of worker node(s) fail. To check that worker nodes are able to fail in the model, a verification attempt is made on the $\Box$ *all_workers_alive* LTL formula. To check that the model has the potential to kill mortal workers, SPIN searches for a counter-example system state with any of the worker[0].dead, worker[1].dead, or worker[2].dead fields set to true. SPIN trivially identifies a counter example after searching 5 system states by executing the choice to kill a node.

6 Implementation

6.1 HdpH-RS architecture

The HdpH-RS architecture, shown in Figure 11, is closely based on the HdpH architecture (Maier & Trinder, 2012), and both share some components with Cloud Haskell as outlined in Section 2. The architecture supports semi-explicit parallelism with work stealing, message passing, and the remote writing to IVars. The reliable scheduler is an implementation of the design from Section 4. Inter-node communication is abstracted into a *communication layer*, that provides startup and shutdown functionality, node IDs, and peer-to-peer message delivery between nodes. The communication layer detects and reports faults to the scheduler. Each node runs several thread schedulers, typically one per core. Each scheduler owns a dedicated *threadpool* that may be accessed by other schedulers for stealing work. Each node runs a message handler, which shares access to the *sparkpool* with the schedulers. In extending HdpH, one module is added for the fault tolerant strategies, and 14 modules are modified. The fault detection, fault recovery, and task supervision functionality amounts to approximately 2,500 lines of Haskell code in HdpH-RS, and an increase of approximately 50% over HdpH.

6.2 Recovering supervised sparks and threads

When a supervisor receives a DEADNODE message indicating a node failure (Section 4.4), it may replicate tasks if their survival is at risk. This is decided by the DEADNODE

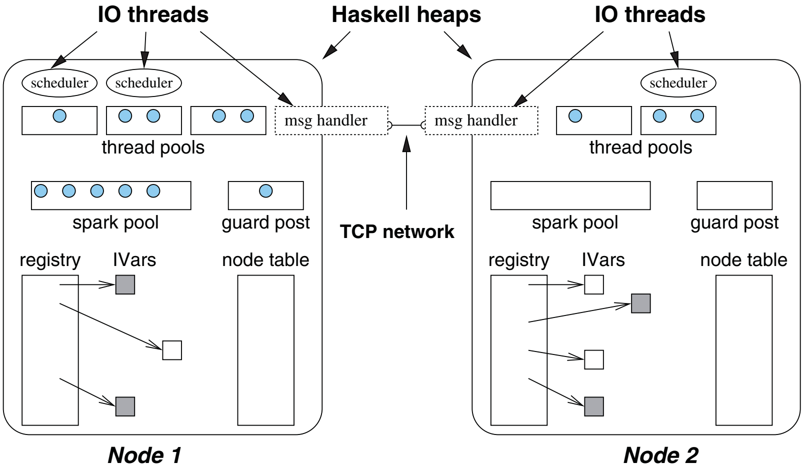


Fig. 11. HdpH-RS system architecture.

handler in Algorithm 1 of Section 4. It uses replicateSpark and replicateThread in Listing 6. Both return a Maybe type, due to a potential race condition whereby another local scheduler or node writes a value to the IVar during the local recovery operation. If the IVar becomes full, then a Nothing value is returned indicating a full IVar and no recovery action needs taking.

```
data SupervisedSpark m = SupervisedSpark
  {clo :: Closure (ParM m ()) , gref :: GRef , replicaNum :: Int }
replicateSpark  :: IVar a → IO (Maybe (SupervisedSpark m))
replicateThread :: IVar a → IO (Maybe (Closure (Par ()))
```

Listing 6. Replicating Sparks & Threads in Presence of Failure.

The replicateSpark and replicateThread functions both take an IVar as an argument. The DEADNODE handler has determined the safety of the corresponding task to be at risk as a consequence of node failure. Globalised IVar references are used to match remote tasks with locally hosted futures. The DEADNODE handler increments the replication number replicaNum in the IVar. A new task is created from a copy of the lost task (stored in SupervisedSpark as clo), and is scheduled according the recover_spark and recover_thread transition rules in the operational semantics (Section mr). If a spark is being recovered, a SupervisedSpark is returned and added to the local sparkpool. If a thread is being recovered, a Closure (Par ()) is returned, unpacked with unClosure, and added to a local threadpool.

6.3 Fault detecting communications layer

Any fault in an MPI setting will typically bring down the entire MPI communicator, making this an unsuitable backend for fault. Instead, HdpH-RS uses a TCP-based transport layer, and nodes discover other nodes using UDP. The distributed virtual machine software stack is shown in Figure 12.

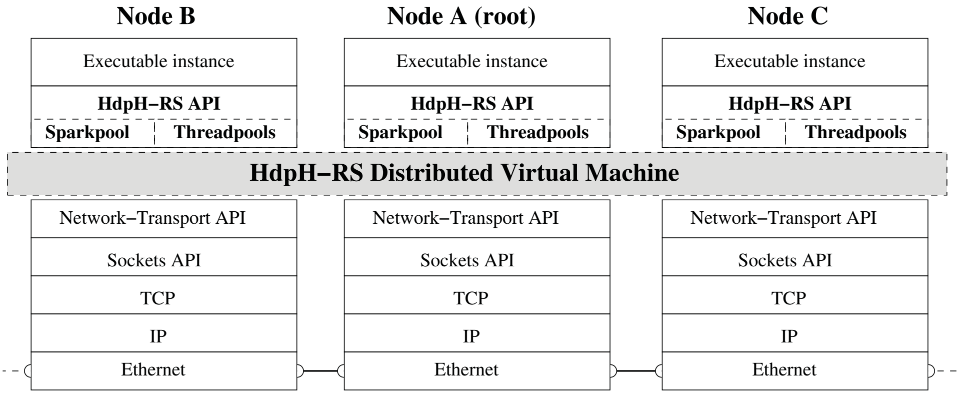


Fig. 12. HdpH-RS distributed virtual machine software stack.

TCP is an idle protocol, so if neither side sends any data once a connection has been established, then no packets are sent over the connection (Cleary, 2009). The act of receiving data is completely passive in TCP, and an application that only reads from a socket cannot detect a dropped connection.

Two options are available for detecting failure with TCP. The first is to transmit keep-alive messages over a connection. The second is to assume the worst, and implement timers for receiving messages on connections. If no packets are received within the timeout period, a connection may be regarded as lost. As architectures scale to thousands of nodes, error propagation through work stealing message transmissions cannot be relied upon. The HdpH-RS keep-alive messages is a guarantee of periodic traffic between nodes independent of work stealing messages, enabling nodes to discover failure by discovering lost connections.

Whilst the transmission of work stealing messages will probably trigger timely TCP failures for smaller architectures, there is a high failure detection latency in larger networks. This has an important implication for performance in HdpH-RS. Take an example where node A creates a supervised spark $spark_1$ and `IVar` i_1 with the execution of `spawn`. Node B fishes $spark_1$, and later suffers a power outage. Node A may not receive a TCP `FIN` message from B due to the sudden failure. Node A does not send any work stealing messages to B, but is waiting for the value of evaluating $spark_1$ to be written to i_1. To ensure a more reliable failure detection of B, node A needs some other message transmission mechanism than just work stealing.

The keep-alive implementation in HdpH-RS is simple. As TCP failures are detected on send attempts, the keep-alive is silently ignored on the receiving end. After N seconds, a node broadcasts a `HEARTBEAT` to all other nodes. When a node receives a `HEARTBEAT` message, it is silently ignored. For small architectures, heartbeats are unlikely to be the trigger that detects failure. On large architectures, work stealing messages between any two nodes are less likely to be transmitted within the keep-alive frequency, so the keep-alive messages are an important mechanism for failure detection. If a node detects the failure of another node, that remote node is removed from the list of peers returned by future calls to `allNodes`.

Table 3. *HdpH-RS benchmarks*

Benchmark	Skeleton	Code origin	Regularity	Code size (lines)
Sum Euler	Chunked parallel maps	HdpH	Some	2
Summatory Liouville	Sliced parallel map	GUM	Little	30
N-Queens	Divide-and-conquer	monad-par	Little	11
Mandelbrot	MapReduce	monad-par	Very little	12

The main drawback to this failure detection strategy is the dependency on connection oriented protocols like TCP. There are two main weaknesses. First, the failure detection strategy of using connection-oriented transmission attempts would not work for connectionless protocols like UDP (Postel, 1980). Second, the design assumes a fully connected network. Every node has a persistent connection with every other node. The scalability limitations of TCP connections are well known (Ha *et al.*, 2008).

6.4 HdpH and HdpH-RS pragmatics

In the design and semantics presented in Section 3, the fault tolerant spawn primitives have identical names to the HdpH primitives, making it trivial to switch between normal and fault tolerant execution. However, in the current HdpH-RS implementation, the fault tolerant primitives have different names, i.e. `supervisedSpawn` and `supervisedSpawnAt`, as do the HdpH-RS skeletons, e.g. `parMapSlicedFT` and `pushMapSlicedFT`. However, both primitives and skeletons preserve the same types, so switching between normal and fault tolerant versions of a program is a trivial alpha conversion. A more elegant solution would be to use the same name for the spawning primitives, and hence have identical skeleton implementations. Better still would be to integrate the HdpH and HdpH-RS implementations, then programmers could switch between reliable and normal schedulers with a runtime flag.

7 Evaluation

7.1 Measurement platform

Benchmarks. The runtime performance of HdpH-RS is measured using the four benchmarks in Table 3. Existing Haskell parallel implementations are ported to HdpH-RS from Glasgow parallel Haskell (Hammond *et al.*, 2007) and `monad-par` (Marlow & Newton, 2013) and parallelised using fault tolerant and non-fault tolerant skeletons. An open access dataset (Stewart *et al.*, 2015) accompanies this paper, which includes the SPIN model (Section 5), scripts for collecting and plotting results and the Haskell source code for the following benchmarks.

Sum Euler is a symbolic benchmark that sums Euler's totient function ϕ over long lists of integers. Sum Euler is an irregular data parallel problem where the irregularity stems from computing ϕ on smaller or larger numbers.

The Liouville function $\lambda(n)$ is the completely multiplicative function where $\lambda(p) = -1$ for each prime p. Summatory Liouville $L(n)$ denotes the sum of the values of the Liouville function $\lambda(n)$ up to n, where $L(n) := \sum_{k=1}^{n} \lambda(k)$.

The Mandelbrot set is the set of points on the complex plane that remains bounded to the set when an iterative function is applied to that. It consists of all points defined by the complex elements c for which the recursive formula $z_{n+1} = z_n^2 + c$ does not approach infinity when $z_0 = 0$ and n approach infinity. A depth parameter is used to control the cost of sequential Mandelbrot computations. A higher depth gives more detail and subtlety in the final image representation of the Mandelbrot set (Boije & Johansson, 2009).

The N-Queens problem computes how many ways n queens can be put on an $n \times n$ chessboard so that no two queens attack each other (Rivin *et al.*, 1994). The implementation uses divide-and-conquer parallelism with an explicit threshold. An exhaustive search algorithm is used.

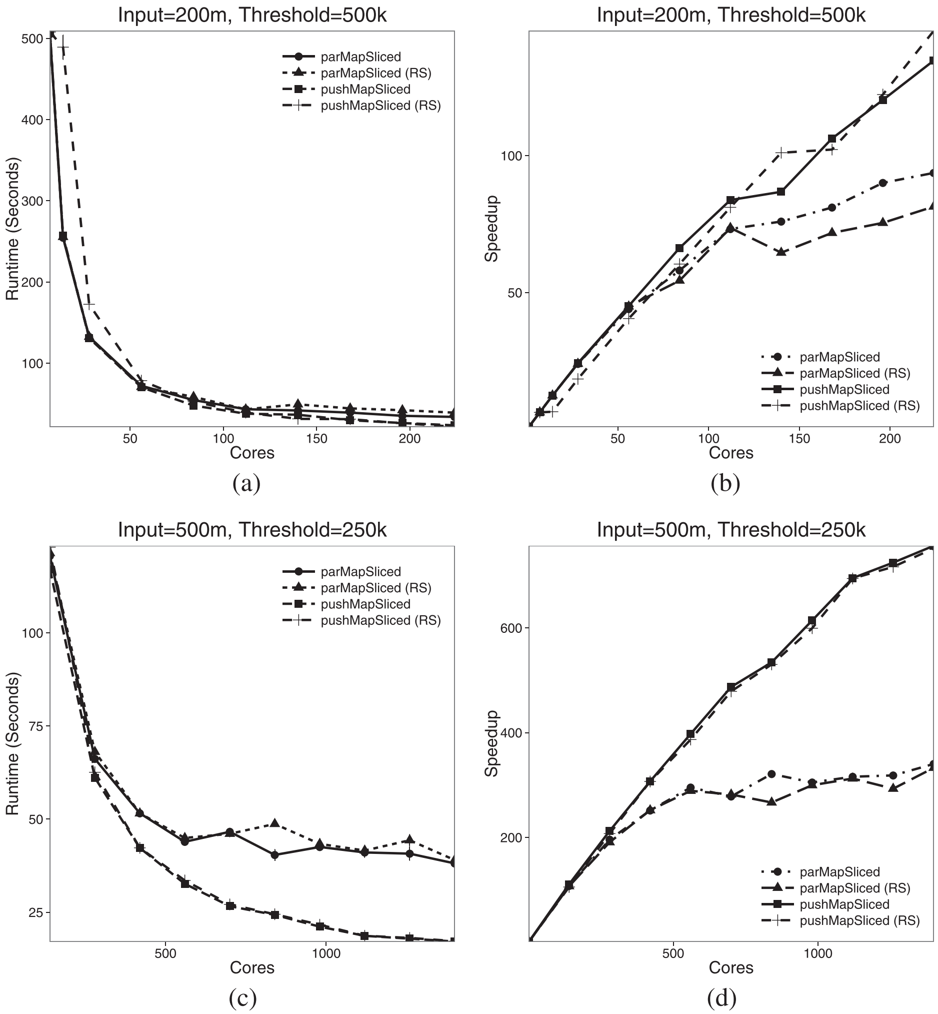
Hardware platforms and configurations. The HdpH-RS benchmarks are measured on two platforms. The first is a Beowulf cluster and is used to measure supervision overheads, and recovery latency in the presence of simultaneous and random failure. Each Beowulf node comprises two Intel Xeon E5504 quad-core CPUs at 2 GHz, sharing 12 Gb of RAM. Nodes are connected via Gigabit Ethernet and run Linux CentOS 5.7 *x*86_64. Benchmarks are run on up to 32 HdpH-RS nodes, one per Beowulf node, scaling up to 256 cores.

The second is HECToR, a national UK HPC service. The HECToR computed hardware is contained in 30 cabinets and comprises 704 compute blades, each containing four compute nodes running Compute Node Linux. Each node has two 16 core AMD Opteron 2.3 GHz Interlagos processors split into four NUMA regions, with 16 Gb memory. Benchmarks are run on up to 200 HdpH-RS nodes, one per NUMA region, scaling up to 1,400 cores. Peer discovery with UDP is not supported on HECToR, so the HdpH-RS fault detecting TCP-based transport layer cannot be used. The MPI-based HdpH transport layer has been retrofitted in to HdpH-RS for the purposes of assessing the scalability of the supervised work stealing in HdpH-RS on HECToR in the absence of faults.

HdpH-RS nodes on both the Beowulf and HECToR have eight cores, seven of which are used by the HdpH-RS node. This is common practice to limit performance variation on shared-memory nodes (Harris *et al.*, 2005; Maier & Trinder, 2012). For every data point, the mean of five runs are reported along with standard error.

7.2 Performance with no failure

Figure 13 shows runtime and speedup graphs for Summatory Liouville, though Figures 13(a) and (c) omit runtimes on one core for readability. Table 4 summarises the relative speedups obtained in strong scaling for the benchmarks with and without fault tolerance. Strong scaling uses the same input on one core as on many cores, and relative speedup compares with the same, potentially parallel, program running on a single core. A complete set of performance results is available in Stewart (2013b).



(a)

(b)

(c)

(d)

Fig. 13. Summatory Liouville on Beowulf & HECToR. (a) Beowulf runtime. (b) Beowulf speedup. (c) HECToR runtime. (d) HECToR speedup.

The speedup measurements compare eager and lazy, and normal and fault tolerant skeleton implementations of the benchmarks. On the Beowulf Sum Euler to 250k is computed with `parMapSliced` skeletons and a threshold of 1k. Summatory Liouville is computed with `parMapSliced` skeletons to 200 m and a threshold of 500k on the Beowulf, and to 500 m with a threshold of 250k on HECToR. Mandelbrot is computed `mapReduce` skeletons with a dimension of 4096 × 4096 and a threshold of 4 and a depth of 4,000 on the Beowulf, and threshold 4, depth 8,000 on HECToR.

The results show that the overheads of fault tolerance are low in the absence of faults as the maximum speedups for the fault tolerant and normal versions are very similar. Eager scheduling consistently out performs lazy work distribution for these programs that have limited irregularity. The parallel efficiency with eager scheduling is approximately 50%, e.g. a speedup of 752 on 1,400 cores, far lower than for regular

Table 4. *HdpH-RS speedup summary (strong scaling)*

	244 Beowulf cores				1,400 HECToR cores			
	Lazy		Eager		Lazy		Eager	
	N-FT	FT	N-FT	FT	N-FT	FT	N-FT	FT
N-Queens	4	5	1	2	...	...	...	...
Sum Euler	85	73	113	114	...	...	...	...
Summatory Liouville	94	81	135	146	340	333	757	752
Mandelbrot	60	57	57	57	50	56	91	93

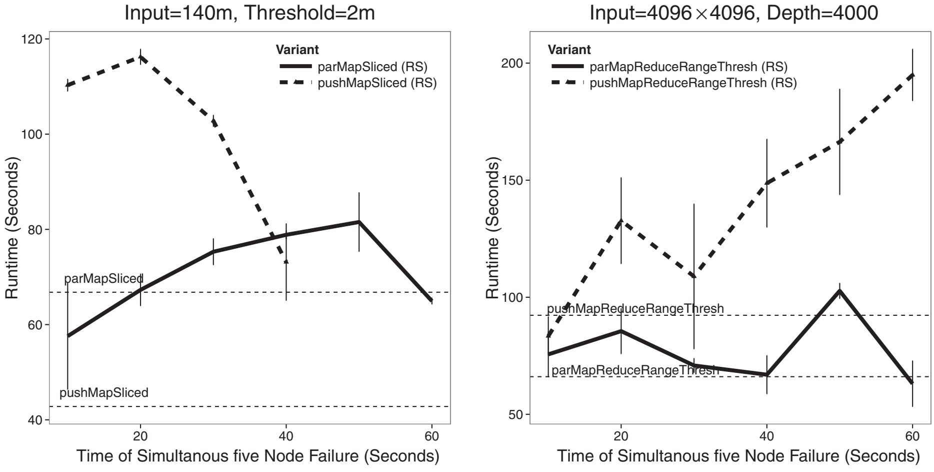
HPC parallelism, but entirely acceptable for irregular symbolic computation. The exception is N Queens, which doesn't scale beyond eight nodes, and we attribute this to a high communication to computation ratio.

7.3 Performance with recovery

The HdpH-RS scheduler is designed to survive simultaneous failures. Recovery from node failure is distributed and asynchronous, the failure of a node will eventually be detected by all healthy nodes. A node replicates tasks corresponding to the supervised futures it hosts, in accordance with the `DEADNODE` message handler in Algorithm 1 from Section 4.4.

The experiment inputs in this section are set so that failure-free runtime is a little more than 60 s on 20 Beowulf nodes. Five nodes are scheduled to die from 10 s to 60 s, at 10 s intervals, from the start of execution. The runtimes are compared to failure-free runs using the equivalent non-fault tolerant skeleton to assess recovery times. Two benchmarks are used to measure recovery overheads, Summatory Liouville for task parallelism and Mandelbrot for divide-and-conquer parallelism. All Summatory Lioville tasks are generated by the root node, and tasks are not recursively decomposed. Executing Mandelbrot generates divide-and-conquer supervision pattern, i.e. generates futures across multiple nodes. The recovery costs of simultaneous failure are in Figure 14. The annotated horizontal dashed lines show mean of five runtimes in the absence of failure.

Summatory Liouville. The recovery costs of Summatory Liouville are shown in Figure 14(a). 70 tasks are generated. The mean runtime with `parMapSliced` is 66.8 s, and 42.8 s with `pushMapSliced`. When eager scheduling is used, the recovery overheads are more substantial early on i.e. at 10 s, 20 s, and 30 s. These overheads compared with fault-free execution with `pushMapSliced` with RS are 158%, 172%, and 140% respectively. As more tasks are evaluated, the recovery costs reduce. There are no measurements taken at 50 s and 60 s as the mean failure-free runtime with `pushMapSliced` is 43 s, so killing the nodes at these times have no effect. Using lazy work stealing with `parMapSliced` with RS, the mean runtime is shorter than failure-free execution with `parMapSliced` by 14%, likely due to the small tasks



(a) (b)

Fig. 14. Recovery costs of simultaneous failure of five Beowulf nodes. (a) Summatory Liouville. (b) Mandelbrot.

resulting from the inputs to the program – executing them on the root node is cheaper than transmitting them for remote execution. As the delay until failure is increased to 20, 30, 40, and 50 s, the recovery overheads are 2%, 13%, 18%, and 22%. Most futures on the root node are full towards the 60 second runtime, so few or no sparks need replicating which accounts for the runtimes with failure at 60 s matching runtimes with no failure.

Mandelbrot. The recovery costs for Mandelbrot are shown in Figure 14(b). The inputs for Mandelbrot are $X = 4,096$, $Y = 4,096$, $threshold = 4$, and $depth = 4,000$. 1,023 tasks are generated. The mean runtime with parMapReduce is 66 s, and 92 s with pushMapReduce. The recovery overheads for the lazy parMapReduce skeleton with RS are low, even as the number of generated supervised futures increases. The recovery overheads for the eager pushMapReduceRangeThresh skeleton with RS increases as more threads are replicated needlessly, e.g. memoization (Michie, 1968) is not employed. When the five node failure occurs at 20, 30, 40, 50, and 60 s, the recovery costs increase, with recovery overheads of 44%, 18%, 59%, 80%, and 110% respectively.

7.4 *Random fault injection with Chaos Monkey*

Random failure can be introduced to HdpH-RS programs, analogous to Netflix's Chaos Monkey (Hoff, 2010). This mechanism is used to ensure that HdpH-RS returns a result in chaotic and unreliable environments. The Chaos Monkey implementation for HdpH-RS is described in (Stewart, 2013b). A unit testing suite implemented with the HUnit test framework (Herington, 2006–2013) is used on the four HdpH-RS benchmarks running on 10 Beowulf nodes to check that results computed by

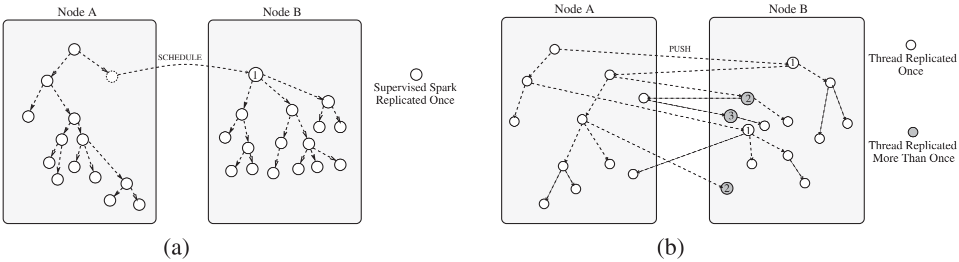


Fig. 15. Recovering lazy & eager scheduling. (a) Supervision with lazy scheduling. (b) Supervision with eager scheduling.

HdpH-RS in the presence of random failure are correct. The test-suite passes 100% of the unit tests.

7.4.1 Chaos Monkey results

The results of chaos Monkey unit testing is shown in Table B1 in Appendix B. It shows a list of integers indicating failures. Each value in the list indicates the time in seconds when each node failed. It also shows the sum of tasks that were replicated to recover from these failure(s). For the lazy scheduling skeletons, only sparks are generated and thus recovered. For the eager scheduling skeletons, only threads are generated and recovered.

For all benchmarks, lazy scheduling reduces recovery costs in the presence of failure. For example, the N-Queens benchmark generates 65, 234 tasks. There is an execution with eight node failures between 3 s and 48 s, resulting in 40, 696 tasks being replicated and eagerly distributed. This results in a runtime of 650 s, compared to the mean failure-free runtime with eager scheduling of 15 s. This is probably due to multiple replications of tasks, due to losses of tasks higher in the supervision tree. In contrast, N-Queens with lazy scheduling and eight node failures results in only eight tasks being replicated as sparks. The runtime is 52 s, compared to a failure-free runtime with lazy scheduling of 28 s — a much lower overhead than recovery with eager scheduling. A similar recovery pattern for lazy versus eager scheduling is shown for Sum Euler, Summatory Liouville, and Mandelbrot. For example, Mandelbrot executions with six and eight node failures incur 419 and 686 task replications (of the total 1023 tasks) respectively. Two Mandelbrot executions with eight node failures incur zero and six task replications respectively.

Recovery overheads with eager scheduling. The Chaos Monkey results shows that eager scheduling incurs high levels of task replication and high runtime recovery costs. It is very likely that many threads are unnecessarily re-scheduled, as HdpH-RS does not currently cache completed task results. For the simple example of lazy scheduling shown in Figure 15(a), far fewer tasks are exposed to the risk of remote node failure as nodes only obtain a task when they are idle. The reason for the high level of rescheduling with eager skeletons is illustrated in Figure 15(b), namely more tasks are vulnerable to node failure. Moreover, the supervision tree will be regenerated at each level, replicating children further down the task graph multiple times, even if tasks at the leaves have completed.

8 Conclusion

We investigate an alternative to Erlang style explicit supervision and recovery of stateful distributed actors, namely the transparent supervision and recovery of idempotent tasks. We introduce the HdpH-RS DSL which transparently adds RS to HdpH. That is, the distributed fork/join-style API of HdpH carries over to HdpH-RS unchanged, including libraries of high-level skeleton abstractions like `parMapSliced` (Section 3.1). We provide an operational semantics for HdpH-RS, modelling task migration due to work stealing and task replication due to fault recovery (Section 3.4). Fault recovery is semantically unobservable, i.e. does not change the result of computations provided all tasks are idempotent.

We present the fault tolerant distributed work stealing protocol of HdpH-RS in detail. Under the protocol a task's creator becomes its supervisor: tracking the task's migration due to work stealing, and replicating the task if it may have been lost due to node failure (Sections 4, 6). We use model checking to validate that the protocol honours the operational semantics (Section 5).

Performance evaluation shows that all benchmarks survive Chaos Monkey fault injection; and that fault tolerance overheads are consistently low in the absence of faults. HdpH-RS well scales on both conventional clusters and HPC architectures. With tasks of similar sizes, eager scheduling scales better than lazy work stealing, but when failures occur the overheads of recovery are much lower with lazy work stealing as there are fewer tasks to recover (Section 7).

The current fault tolerance model in HdpH-RS has a number of limitations that could be addressed in future work. It would be intriguing to thoroughly explore the determinism properties of transparent fault tolerance in languages like HdpH-RS. The implementation could be improved in various ways: by integrating HdpH-RS with the topology aware HdpH implementation (Section 6.4); by extending the communication layer to allow nodes to join a computation; and by avoiding the scalability limitation imposed by instantiating a fully connected graph of nodes during execution as in SD Erlang (Chechina *et al.*). The current implementation re-evaluates any tasks created by a lost task, and this may be very expensive e.g. as we have seen for divide-and-conquer programs. Some form of memoization could avoid this re-evaluation. HdpH-RS could be made a more complete reliable language by providing support for reliable distributed data structures, e.g. a replicated distributed hash table or Erlang's MNesia (Mattsson *et al.*, 1999). Such a language would support reliable big computation over reliable distributed data structures.

Acknowledgments

The work was funded by EPSRC grants HPC-GAP (EP/G05553X), AJITPar (EP/L000687/1) and Rathlin (EP/K009931/1), and EU grant RELEASE (FP7-ICT 287510). The authors thank Blair Archibald and the anonymous referees for helpful feedback.

References

Aljabri, M., Loidl, H.-W. & Trinder, P W. (2014) The design and implementation of GUMSMP: A multilevel parallel Haskell implementation. In Proceedings of Implementation and Application of Functional Languages (IFL'13). New York, NY: ACM, pp. 37–48.

Armstrong, J. (2010) Erlang. *Commun. ACM* **53**(9), 68–75.

Barroso, L. A., Clidaras, J. & Hölzle, U. (2013) *The Datacenter as a Computer*, 2nd ed. Morgan & Claypool.

Boije, J. & Johansson, L. 2009 (December) *Distributed Mandelbrot Calculations*. Tech. rept. TH Royal Institute of Technology.

Borwein, P. B., Ferguson, R. & Mossinghoff, M. J. (2008) Sign changes in sums of the Liouville function. *Math. Comput.* **77**(263), 1681–1694.

Cappello, F. (2009) Fault tolerance in Petascale/ Exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA* **23**(3), 212–226.

Chandy, K. M. & Lamport, L. (1985) Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**(1), 63–75.

Chechina, N., Li, H., Ghaffari, A., Thompson, S. & Trinder, P. (2016) Improving network scalability of Erlang. *J. Parallel Distrib. Comput.* **90–91**, 22–34.

Cleary, S. 2009 (May) *Detection of Half-Open (Dropped) Connections*. Technical Report Microsoft.

`http://blog.stephencleary.com/2009/05/detection-of-half-open-dropped.html`.

Cole, M. I. (1988) *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD Thesis, Computer Science Department, University of Edinburgh.

Dean, J. & Ghemawat, S. (2008) MapReduce: Simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113.

Dinu, F. & Ng, T. S. E. (2011) Hadoop's overload tolerant design exacerbates failure detection and recovery. In Proceedings of 6th International Workshop on Networking Meets Databases, NETDB 2011. Athens, Greece. June.

Edinburgh Parallel Computing Center (EPCC). (2008) *HECToR National UK Super Computing Resource, Edinburgh*. `https://www.hector.ac.uk`.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M. & Johnson, D. B. (2002) A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* **34**(3), 375–408.

Epstein, J., Black, A. P. & Jones, S. L. P. (2011) Towards Haskell in the cloud. In Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011, pp. 118–129.

Gupta, M. (2012) *Akka Essentials*. Packt Publishing Ltd.

Ha, S., Rhee, I. & Xu, L. (2008) CUBIC: A new TCP-friendly high-speed TCP variant. *Oper. Syst. Rev.* **42**(5), 64–74.

Halstead Jr., R. H. (1985) Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* **7**(4), 501–538.

Hammond, K., Zain, A. Al, Cooperman, G., Petcu, D. & Trinder, P. W. (2007) SymGrid: A framework for symbolic computation on the grid. In Proceedings of 13th International Euro-Par Conference, Rennes, France, August 28–31, 2007, pp. 457–466.

Harris, T., Marlow, S. & Jones, S. L. P. (2005) Haskell on a shared-memory multiprocessor. In Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005, pp. 49–61.

Herington, D. (2006–2013) *Haskell Library: `hunit` Package. A Unit Testing Framework for Haskell*. `http://hackage.haskell.org/package/HUnit`.

Hoff, T. (2010 December) *Netflix: Continually Test by Failing Servers with Chaos Monkey.* `http://highscalability.com`.

Holzmann, G. J. (2004) *The SPIN Model Checker - Primer and Reference Manual.* Addison-Wesley.

John, A., Konnov, I., Schmid, U., Veith, H. & Widder, J. (2013) Towards modeling and model checking fault-tolerant distributed algorithms. In Model Checking Software – Proceedings of 20th International Symposium, SPIN 2013, Stony Brook, NY, July, 2013, pp. 209–226.

Kuper, L., Turon, A., Krishnaswami, N. R. & Newton, R. R. (2014) Freeze after writing: Quasi-deterministic parallel programming with LVars and handlers. In Proceedings of POPL 2014, San Diego, ACM, pp. 257–270.

Lamport, L. (1978) Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565.

Litvinova, A., Engelmann, C. & Scott, S. L. 2010 (February 16–18) A Proactive fault tolerance framework for high-performance computing. In Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2010, Innsbruck, Austria.

Loogen, R., Ortega-Mallén, Y. & Peña-Marí, R. (2005) Parallel functional programming in Eden. *J. Funct. Program.* **15**(3), 431–475.

Maier, P., Livesey, D., Loidl, H.-W. & Trinder, P. (2014a) High-performance computer algebra: A Hecke algebra case study. In *Proceedings of Euro-par 2014 Parallel Processing - 20th International Conference,* Porto, Portugal, August 25–29, 2014, Silva, F. M. A., de Castro Dutra, I. & Costa, V. S. (eds), Lecture Notes in Computer Science, vol. 8632. Springer, pp. 19–35.

Maier, P., Stewart, R. J. & Trinder, P. W. (2014b) Reliable scalable symbolic computation: The design of SymGridPar2. *Comput. Lang. Syst. Struct.* **40**(1), 19–35.

Maier, P., Stewart, R. J. & Trinder, P. (2014c) The HdpH DSLs for scalable reliable computation. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Gothenburg, Sweden, September 4–5, 2014. ACM, pp. 65–76.

Maier, P. & Trinder, P. (2012) Implementing a high-level distributed-memory parallel Haskell in Haskell. In *Implementation and Application of Functional Languages, 23rd International Symposium 2011,* Lawrence, KS, USA, October 3-5, 2011. Revised Selected Papers. Lecture Notes in Computer Science, vol. 7257. Springer, pp. 35–50.

Marlow, S., Jones, S. L. P. & Singh, S. (2009) Runtime support for multicore Haskell. In Proceedings of ICFP, Edinburgh, Scotland, pp. 65–78.

Marlow, S. & Newton, R. (2013) *Source code for `monad-par` library.* `https://github.com/simonmar/monad-par`.

Marlow, S., Newton, R. & Jones, S. L. P. (2011) A monad for deterministic parallelism. In Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011, pp. 71–82.

Mattsson, H., Nilsson, H. & Wikstrm, C. (1999) Mnesia - a distributed robust DBMS for telecommunications applications. In Proceedings of PADL, San Antonio, Texas, USA, pp. 152–163.

Meredith, M., Carrigan, T., Brockman, J., Cloninger, T., Privoznik, J., & Williams, J. (2003) Exploring Beowulf clusters. *J. Comput. Sci. Colleges* **18**(4), 268–284.

Michie, D. (1968) “Memo” functions and machine learning. *Nature* **218**(5136), 19–22.

Peyton Jones, S. (2002) Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, Marktoberdorf Summer School, pp. 47–96.

Pnueli, A. (1977) The temporal logic of programs. In Proceedings of 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, 31 October–1 November 1977. IEEE Computer Society, pp. 46–57.

Postel, J. 1980 (August) *User Datagram Protocol*. RFC 768 Standard. `http://www.ietf.org/rfc/rfc768.txt`.

Prior, A. N. (1957) *Time and Modality*. Oxford University Press.

Ramalingam, G. & Vaswani, K. (2013) Fault tolerance via idempotence. In Proceedings of POPL, Rome, Italy, pp. 249–262.

Rivin, I., Vardi, I. & Zimmerman, P. (1994) The N-queens problem. *Am. Math. Mon.* **101**(7), 629–639.

Scholz, S.-B. (2003) Single assignment C: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.* **13**(6), 1005–1059.

Schroeder, B. & Gibson, G. A. (2007) Understanding failures in Petascale computers. *J. Phys.: Conf. Ser.* **78**, 012022 (11pp) `http://stacks.iop.org/1742-6596/78/012022`.

Scott, J. & Kazman, R. (2009) *Realizing and Refining Architectural Tactics: Availability*. Technical Report. Carnegie Mellon University, Software Engineering Institute.

Stewart, R. (2013a December) *Promela Abstraction of the HdpH-RS Scheduler*. `https://github.com/robstewart57/phd-thesis/blob/master/spin_model/hdph_scheduler.pml`.

Stewart, R. (2013b November) *Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell*. PhD Thesis, Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland.

Stewart, R. & Maier, P. (2013) *HdpH-RS source code*. `https://github.com/robstewart57/hdph-rs`.

Stewart, R., Maier, P. & Trinder, P. (2015 June) *Open access dataset for "Transparent Fault Tolerance for Scalable Functional Computation"*. `http://dx.doi.org/10.5525/gla.researchdata.189`.

Trinder, P. W., Hammond, K., Jr., J. S. Mattson, Partridge, A. S. & Jones, S. L. P. (1996) GUM: A portable parallel implementation of Haskell. In Proceedings of ACM Programming Language Design and Implementation (PLDI'96), Philadephia, Pennsylvania, May, pp. 79–88.

White, T. (2012) *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated)*. O'Reilly.

Xu, C. & Lau, F. C. (1997) *Load Balancing in Parallel Computers: Theory and Practice*. Norwell, MA: Kluwer Academic Publishers.

Appendix A. Promela abstraction of the verified HdpH-RS Scheduler

```
active proctype Supervisor() {
  int thiefID, victimID, deadNodeID, seq, authorizedSeq, deniedSeq;
  supervisor.sparkpool.spark_count = 1;
  run Worker(0); run Worker(1); run Worker(2);

SUPERVISOR_RECEIVE:
  if  :: (supervisor.sparkpool.spark_count > 0 && spark.age > maxLife) →
         atomic { supervisor.resultSent = 1; supervisor.ivar = 1;
                  goto EVALUATION_COMPLETE; }
      :: else →
         if :: (supervisor.sparkpool.spark_count > 0) →
              atomic { supervisor.resultSent = 1; supervisor.ivar = 1;
                       goto EVALUATION_COMPLETE; }
            :: chans[3] ? DENIED(thiefID, deniedSeq,null) →
              supervisor.waitingSchedAuth = false;
              chans[thiefID] ! NOWORK(3, null, null) ;
            :: chans[3] ? FISH(thiefID, null,null) → /* React to FISH request */
              if :: (supervisor.sparkpool.spark_count > 0 && !supervisor.waitingSchedAuth) →
                      supervisor.waitingSchedAuth = true;
                      chans[3] ! REQ(3, thiefID, supervisor.sparkpool.spark);
                 ::  else → chans[thiefID] ! NOWORK(3, null,null) ; /*We don't have the spark */
              fi;
            :: chans[3] ? AUTH(thiefID, authorizedSeq, null) →
              d_step { supervisor.waitingSchedAuth = false;
                       supervisor.sparkpool.spark_count--; }
              chans[thiefID] ! SCHEDULE(3, supervisor.sparkpool.spark ,null);
            :: chans[3] ? REQ(victimID, thiefID, seq) →
              if :: seq == spark.highestReplica →
                   if :: spark.context == ONNODE && ! worker[thiefID].dead→
                          d_step { spark.context  = INTRANSITION;
                                   spark.location.from = victimID ;
                                   spark.location.to = thiefID ; }
                          chans[victimID] ! AUTH(thiefID, seq, null);
                      :: else → chans[victimID] ! DENIED(thiefID, seq, null);
                   fi
                 :: else → chans[victimID] ! OBSOLETE(thiefID, null, null);
              fi
            :: chans[3] ? ACK(thiefID, seq, null) →
              if :: seq == spark.highestReplica →
                     d_step { spark.context = ONNODE;
                              spark.location.at = thiefID ; }
                 :: else → skip ;
              fi
            :: atomic { chans[3] ? RESULT(null, null, null);
                        supervisor.ivar = 1; goto EVALUATION_COMPLETE; }
            :: chans[3] ? DEADNODE(deadNodeID, null, null) →
              bool should_replicate;
              d_step { should_replicate = false;
                       if :: spark.context == ONNODE \
                        && spark.location.at == deadNodeID → should_replicate=true;
                          :: spark.context  == INTRANSITION \
                        && (spark.location.from == deadNodeID \
                       || spark.location.to == deadNodeID) → should_replicate=true;
                          :: else → skip;
                       fi;
                if :: should_replicate →
                       spark.age++; supervisor.sparkpool.spark_count++; spark.highestReplica++;
                       supervisor.sparkpool.spark = spark.highestReplica ; spark.location.at=3 ;
                       spark.context = ONNODE;
                   :: else → skip;
                fi;
              }
  fi;    fi;

  if :: (supervisor.ivar == 0) → goto SUPERVISOR_RECEIVE;
     :: else → skip;
  fi;
EVALUATION_COMPLETE: }
```

Fig. A1. Repetitive control flow options for supervisor.

```
proctype Worker(int me) {
  int thiefID, victimID, deadNodeID, seq, authorisedSeq, deniedSeq;

WORKER_RECEIVE:
  if :: (worker[me].sparkpool.spark_count > 0 && spark.age > maxLife) →
       atomic { worker[me].resultSent = true;
                chans[3]  ! RESULT(null,null,null);
                goto END; }

     :: else →
       if ::  skip →  /* die */
            worker[me].dead = true;
            report_death(me);
            goto END;
          :: (worker[me].sparkpool.spark_count > 0) →
              chans[3] ! RESULT(null,null,null);
          :: (worker[me].sparkpool.spark_count == 0
              && (worker[me].waitingFishReplyFrom == -1) && spark.age < (maxLife+1)) →
              int victim;
              d_step {
              if
              :: (0!=me)&&!worker[0].dead&&(worker[me].lastTried-0) → victim=0;
              :: (1!=me)&&!worker[1].dead&&(worker[me].lastTried-1) → victim=1;
              :: (2!=me)&&!worker[2].dead&&(worker[me].lastTried-2) → victim=2;
              :: skip →  = 3; /* supervisor */
              fi;
              worker[me].lastTried=victim;
              worker[me].waitingFishReplyFrom = victim;
            };
            chans[chosenVictimID] ! FISH(me, null, null) ;
          :: chans[me] ? NOWORK(victimID, null, null) →
              worker[me].waitingFishReplyFrom = -1;  /* can fish again */
          :: chans[me] ? FISH(thiefID, null, null) → /* React to FISH request */
            if :: (worker[me].sparkpool.spark_count > 0 && ! worker[me].waitingSchedAuth) →
                   worker[me].waitingSchedAuth = true;
                   chans[3] ! REQ(me, thiefID, worker[me].sparkpool.spark);
               :: else → chans[thiefID] ! NOWORK(me, null, null) ;
            fi
          :: chans[me] ? AUTH(thiefID, authorisedSeq, null) →
              d_step { worker[me].waitingSchedAuth = false;
                       worker[me].sparkpool.spark_count--;
                       worker[me].waitingFishReplyFrom = -1; }
            chans[thiefID] ! SCHEDULE(me, worker[me].sparkpool.spark, null);
          :: chans[me] ? DENIED(thiefID, deniedSeq, null) →
              worker[me].waitingSchedAuth = false;
              chans[thiefID] ! NOWORK(me, null, null) ;
          :: chans[me] ? OBSOLETE(thiefID, null, null) →
            d_step { worker[me].waitingSchedAuth = false;
                     worker[me].sparkpool.spark_count--;
                     worker[me].waitingFishReplyFrom = -1; }
            chans[thiefID] ! NOWORK(me, null, null) ;
          :: chans[me] ? SCHEDULE(victimID, seq, null) →
              d_step { worker[me].sparkpool.spark_count++;
                       worker[me].sparkpool.spark = seq ;
                       spark.age++; }
            chans[3] ! ACK(me, seq, null) ; /* Send ACK To supervisor */
          :: chans[me] ? DEADNODE(deadNodeID, null, null) →
            d_step { if :: worker[me].waitingFishReplyFrom > deadNodeID →
                             worker[me].waitingFishReplyFrom = -1 ;
                        :: else → skip ;
                     fi ; };
  fi;   fi ;

  if :: (supervisor.ivar == 1) → goto END;
     :: else → goto WORKER_RECEIVE;
  fi;
END: }
```

Fig. A2. Repetitive control flow options for a worker.

Appendix B. HdpH-RS Chaos Monkey results

Table B1. *Fault tolerance unit testing: Chaos Monkey runtimes*

Benchmark	Skeleton	Failed nodes	Recovery		Runtime	Unit test
		(seconds)	Sparks	threads	(seconds)	
Sum Euler *lower=0* *upper=100000* *chunk=100* *tasks=1001* *X=3039650754*	parMapChunked	-			126.1	pass
	parMapChunked (RS)	[6,30,39,49,50]	10		181.1	pass
		[5,11,18,27,28,33,44,60]	16		410.2	pass
		[31,36,49]	6		139.7	pass
		[37,48,59]	6		139.5	pass
		[1,17,24,27,43,44,47,48,48]	17		768.2	pass
	pushMapChunked	-			131.6	pass
	pushMapChunked (RS)	[4,34,36,37,48,49,58]		661	753.7	pass
		[2,6,11,15,17,24,32,37,41]		915	1179.7	pass
		[2,37,39,45,49]		481	564.0	pass
		[4,7,23,32,34,46,54,60]		760	978.1	pass
		[35,38,41,43,46,51]		548	634.3	pass
Summatory Liouville λ = 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
Queens 14×14 *board* *threshold=5* *tasks=65234* *X=365596*	parDnC	-			28.1	pass
	parDnC (RS)	[3,8,9,10,17,45,49,51,57]	8		52.1	pass
		[1,30,32,33,48,50]	5		49.4	pass
		[8,15]	2		53.3	pass
		[20,40,56]	2		49.9	pass
		[]	0		52.8	pass
	pushDnC	-			15.4	pass
	pushDnC (RS)	[14,33]		5095	57.1	pass
		[3,15,15,23,24,28,32,48]		40696	649.5	pass
		[5,8,26,41,42,42,59]		36305	354.9	pass
		[0,5,8,10,14,28,31,51,54]		32629	276.9	pass
		[31,31,58,60]		113	47.8	pass
Mandelbrot *x=4048* *y=4048* *depth=256* *threshold=4* tasks=1023 *X=449545051*	parMapReduce	-			23.2	pass
	parMapReduce (RS)	[28,30,36,44,49,54,56,56]	0		29.1	pass
		[]	0		27.8	pass
		[7,24,25,25,44,53,54,59]	6		32.6	pass
		[17,30]	0		55.4	pass
		[0,14]	2		33.7	pass
	pushMapReduce	-			366.3	pass
	pushMapReduce (RS)	[9,24,34,34,52,59]		419	205.3	pass
		[7,8,11,22,32,35,44,46]		686	395.9	pass
		[27,49]		2	371.8	pass
		[]		0	380.4	pass
		[9,33,50,50,52,53]		84	216.1	pass