# The HdpH DSLs for Scalable Reliable Computation

Patrick Maier
University of Glasgow
Patrick.Maier@glasgow.ac.uk

Robert Stewart
Heriot-Watt University
R.Stewart@hw.ac.uk

Phil Trinder
University of Glasgow
Phil.Trinder@glasgow.ac.uk

## Abstract

The statelessness of functional computations facilitates both parallelism and fault recovery. Faults and non-uniform communication topologies are key challenges for emergent large scale parallel architectures. We report on *HdpH* and *HdpH-RS*, a pair of Haskell DSLs designed to address these challenges for irregular task-parallel computations on large distributed-memory architectures. Both DSLs share an API combining explicit task placement with sophisticated work stealing. HdpH focuses on scalability by making placement and stealing topology aware whereas HdpH-RS delivers reliability by means of fault tolerant work stealing.

We present operational semantics for both DSLs and investigate conditions for semantic equivalence of HdpH and HdpH-RS programs, that is, conditions under which topology awareness can be transparently traded for fault tolerance. We detail how the DSL implementations realise topology awareness and fault tolerance. We report an initial evaluation of scalability and fault tolerance on a 256-core cluster and on up to 32K cores of an HPC platform.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***Keywords*** embedded domain specific l anguages; parallelism; topology awareness; fault tolerance

## 1. Introduction

As the manycore revolution drives up the number of cores, the use of compute resources with 100,000+ cores will become commonplace in the near future. On such architectures core failures are predicted to become relatively common [3, 6]. Moreover, at such scales core-to-core communication latency may vary by several orders of magnitude, depending on whether the communicating cores belong to different compute nodes, possibly living in different subnets, or whether they share memory or cache.

Reliability and topology awareness are substantial challenges even for applications that are embarrassingly parallel, or have other simplifying properties like regular static task decomposition, or simple data structures. The challenges are yet greater for applications without these properties. Symbolic computations, for example, often have highly irregular task granularity, dynamic task creation, complex control flows, or complex data structures.

We present *HdpH (Haskell distributed parallel Haskell)* and *HdpH-RS (HdpH + Reliable Scheduling)*, a pair of DSLs designed to address the challenges of reliability and scalability for computations with irregular parallelism. Specifically HdpH allows the programmer to exploit communication topologies, and HdpH-RS provides low cost automatic fault tolerance. The languages were developed for symbolic computation on capability class high-performance computing (HPC) platforms (currently around $10^5$ cores) and on commodity off-the-shelf (COTS) platforms, but have broad application for large-scale irregularly-parallel computations.

We start by outlining related work on parallel languages/runtimes and parallel symbolic computation (Section 2). We present the design of HdpH that is novel in combining the following features (Section 3). It is *scalable*, providing a distributed-memory parallel DSL that manages computations on multiple multicore nodes. It provides *high-level semi-explicit parallelism* with implicit and explicit task placement, and dynamic load management by work stealing. It provides semi-explicit *topology awareness* allowing the programmer to exploit the deep communication topology of large architectures using an abstract distance metric (Section 3.2). HdpH-RS features a *reliable scheduler* to provide low cost automatic *fault tolerance* using Erlang-style supervision and recovery of location-invariant computations (Section 3.4). HdpH-RS currently omits topology awareness, but this is an engineering artifact rather than a fundamental issue. Both HdpH and HdpH-RS provide high-level coordination abstractions via *polymorphic algorithmic skeletons* (Section 7).

The initial HdpH and HdpH-RS designs are reported in [25]. The following are novel research contributions.

**(1)** We present operational semantics for HdpH and HdpH-RS. The semantics combines non-deterministic scheduling of parallelism with topology awareness and fault tolerance. We investigate conditions for semantic equivalence of HdpH and HdpH-RS programs to enable trading topology awareness for fault tolerance (Section 4).

**(2)** We outline the HdpH and HdpH-RS implementations, focusing on how the work stealing schedulers achieve topology awareness and fault tolerance (Section 6), to implement the operational semantics. This includes validating HdpH-RS' sophisticated fault tolerant work stealing protocol by model checking (Section 5).

**(3)** We present an initial evaluation of HdpH and HdpH-RS on a 256-core COTS cluster and on HECToR, a capability class HPC platform with a total of 90,000 cores. We demonstrate the scalability of both HdpH and HdpH-RS. We investigate the fault tolerance of HdpH-RS, and the overheads both in the presence and absence of faults. We sketch a case study using HdpH to coordinate 1000 instances of the GAP computer algebra system [10] on HECToR to solve problems in algebraic representation theory (Section 8).

## 2. Related Work

***Parallel sublanguages.*** Most production languages have multiple parallel or distributed sub-languages that may be built-in to

| | Built-In | | | DSLs | | | |
|---|---|---|---|---|---|---|---|
| Property \ Language | GpH GUM | Eden | GHC | `Par` monad | Cloud Haskell | Meta Par | HdpH (RS) |
| Scalable - dist. mem. | + | + | − | − | + | + | + |
| Fault Tolerance | − | − | − | − | + | − | + |
| Polymorphic Closures | + | + | + | + | − | − | + |
| Pure (non-monad) API | + | + | + | − | − | − | − |
| Determinism | (+) | (+) | (+) | + | − | + | − |
| Auto Load Balancing | + | + | + | + | − | + | + |

**Table 1.** Parallel Haskell comparison.

the language implementation like OpenMP or built-on like MPI libraries. Recently DSL technology is being exploited to build parallel or distributed sub-languages onto existing languages. For example the Akka toolkit [1] that provides Erlang-style distributed actors for Scala and Java can be viewed as a shallowly-embedded DSL.

In addition to standard DSL support capabilities like higher-order functions and meta-programming, parallel/distributed DSLs require to manipulate computations, often expressed as futures or closures. Haskell augments these capabilities with a non-strict semantics that minimises sequentialisation and makes it relatively easy to provide a `Monad` to specify the required parallel or distributed coordination behaviour.

***Parallel Haskells.*** Haskell language extensions like Eden [20], GpH [33] and GHC [26] build in parallelism in the form of elaborate runtime systems (RTS) that support parallelism primitives. To improve maintainability and ease development several recent parallel Haskells are monadic DSLs embedded in Concurrent Haskell [29], e. g. CloudHaskell [8], the `Par` monad [27], Meta-Par [9] and our new languages HdpH and HdpH-RS. Table 1 compares the key features of some important *general purpose* parallel Haskells, excluding more specialised variants like Data Parallel Haskell [7]. Most of the entries in the table are self-explanatory. The determinism properties of these languages are not trivial [16], and here we mean that the language guarantees that parallel evaluation does not introduce observable concurrency, e. g. due to races between parallel threads.[1]

The crucial differences between HdpH/HdpH-RS and other parallel Haskells can be summarised as follows. Both GHC and the `Par` monad provide parallelism only on a single multicore, where HdpH scales onto distributed-memory architectures with many multicore nodes. Meta-Par focuses on exploiting heterogeneous, rather than the relatively homogeneous HPC platforms that HdpH/HdpH-RS target. CloudHaskell replicates Erlang style [2] explicit distribution and is the only other Haskell variant to provide fault tolerance. It is most closely related to HdpH, but provides lower level coordination with explicit task placement and no load management. As CloudHaskell distributes only monomorphic closures it is not possible to construct general coordination abstractions like algorithmic skeletons.

***Topology aware and fault tolerant scheduling*** are novel features of the HdpH/HdpH-RS DSLs. Topology aware work stealing in HotSLAW [28] and load balancing in CHARM++ [17] minimise the cost of task migration but do not expose the topology to the programmer, and hence unlike HdpH cannot guarantee that tasks remain close to each other. While some GRID/cloud middleware like [15] exposes complex topologies, the architectures are very different from HPC and the schedulers typically aim to minimise the cost of inter-process communication rather than migration. Perhaps most closely related is a parallel Haskell [14] that exposes a two-level topology. In contrast HdpH topologies may be arbitrarily deep.

[1] GUM, Eden and GHC guarantee determinism only for pure computations.

Erlang [2] fault tolerance links processes, and supervision trees are commonly constructed where one process supervises others, that may in turn be supervisors. The supervisor is informed of the failure of any supervised process and takes actions like respawning the failed process or killing sibling processes. Unlike HdpH-RS the supervised processes are stateful and hence recovery is observable; moreover recover policies are explicitly stated for each supervisor. Distributed query frameworks like Google MapReduce or Hadoop [34] provide automatic recovery of read-only, and hence idempotent, functions. Unlike HdpH-RS the programming model provided by these frameworks is a restricted to distributed data retrieval. Both Erlang and Hadoop tasks are placed only once, simplifying replication and recovery. In contrast HdpH-RS must recover sparks that may have migrated to a new location since their initial placement.

General purpose fault tolerant work stealing is a relatively unexplored area. Some closely related work is [21] that provides task parallel fault tolerant scheduling of idempotent computations with work stealing. Satin [35] uses a global result table for sharing computation values to limit re-computation in the presence of failure, and Cilk-NOW [4] that checkpoints individual computations to allow available schedulers to resume partially executed computations in the presence of failure.

***Symbolic computation and GAP.*** Symbolic computation is key to both mathematics and computer science, e. g. for cryptography. Computational algebra is an important class of symbolic computation with many complex and expensive computations that would benefit from parallel execution. Besides well-known general-purpose Computational Algebra Systems (CAS) like Maple, there are a number of CAS specialised to particular mathematical domains, e. g. GAP [10] to combinatorial group theory.

***Parallel symbolic computation.*** Some discrete mathematical problems are embarrassingly parallel, and this has been exploited for years even at Internet scale, e. g. the "Great Internet Mersenne Prime Search". Other problems have more complex coordination patterns and both parallel algorithms and parallel CAS implementations have been developed, e. g. `ParGAP`. Many parallel algebraic computations exhibit high degrees of irregularity, with varying numbers and sizes of tasks. Some computations have both multiple levels of irregularity, and enormous (5 orders of magnitude) variation in task sizes [18]. They use complex user-defined data structures and have complex control flows, often exploiting recursion. They make little, if any, use of floating-point operations.

This combination of characteristics means that symbolic computations are not well suited to conventional HPC paradigms with their emphasis on iteration over matrices of floating point numbers, and has motivated the development of domain specific scheduling and management frameworks like *SymGridPar* [18].

***SymGridPar, SymGridPar2 and HdpH.*** The SymGridPar framework [18] is a client/server infrastructure for orchestrating multiple CAS instances into a parallel application. To the user it presents itself as a set of algorithmic skeletons for introducing parallelism, embedded into the user's CAS (the client). The skeletons are implemented in a CAS-independent distributed middleware (the coordination server), which performs load balancing and calls other CAS (the compute servers) via remote procedure call.

SymGridPar2 (SGP2) is a successor to SymGridPar that aims to *scale symbolic computation to architectures with* $10^5$ *cores*. The SGP2 design aims to preserve the user experience of SGP, specifically the high-level *skeleton API*. That is, to the CAS user SGP2 will look like SGP, apart from a few new skeleton parameters for tuning locality control and/or fault tolerance. SGP2 retains the architecture of SGP but provides a scalable fault tolerant coordination server. A key design decision is to realise the coordination server

using the HdpH and HdpH-RS DSLs that are the focus of this paper.

***Faults in large-scale architectures.*** HPC architectures exploit extremely reliable processor and interconnect technologies and current system still exhibit low fault rates. However, fault rates grow rapidly with the number of cores. In consequence fault tolerance for large HPC architectures is a very active research area [6]. We know from warehouse computing that fault rates are likely to be far greater with the much cheaper and more prevalent commodity-off-the-shelf (COTS) architectures [3].

## 3. Language Design

This section presents the designs of HdpH and HdpH-RS, shallowly embedded Haskell DSLs for semi-explicit parallelism on large distributed-memory platforms. The DSLs have the following novel combination of features. They are *scalable*, each providing a parallel DSL for distributing computations across a network of multicore nodes. They are *portable*, being implemented entirely in Haskell (with GHC extensions) rather than relying on bespoke low-level runtime systems like Glasgow parallel Haskell (GpH) [32] or Eden [20]. HdpH and HdpH-RS provide *high-level semi-explicit parallelism* with implicit and explicit task placement and dynamic load management. Implicit placement frees the programmer from coding 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 (Section 3.3). HdpH focuses on semi-explicit *topology awareness* allowing the programmer to exploit the deep communication topology of large architectures using an abstract distance metric (Section 3.2). HdpH-RS provides low cost automatic *fault tolerance* using Erlang-style supervision and recovery of location-invariant computations (Section 3.4). Switching between topology awareness and fault tolerance comes at minimal cost as both DSLs share the same polymorphic API (Section 3.1). Polymorphism is also a key feature of advanced coordination abstractions such as algorithmic skeletons combining explicit and implicit task placement (Section 7).

### 3.1 Primitives

HdpH extends the `Par` monad DSL [27] for shared-memory parallelism to distributed memory, and Figure 1 lists the HdpH API. HdpH exposes locations and distances between locations as abstract types `Node` and `Dist`. The functions `dist` and `equiDist` provide information about nodes and distances as detailed in Section 3.2.

Like [27], HdpH focuses on task parallelism. In distributed memory, this requires serialisation of `Par` computations and results so they can be sent over the network. While the `Binary` typeclass provides serialisation of evaluated values (normal forms), computations (thunks) must be wrapped into *explicit closures*. An explicit closure is a term of type `Closure t`, which wraps a possibly unevaluated value of type `t`. Generalising CloudHaskell's closures [8], the explicit closures of HdpH are fully polymorphic as there is no constraint on the type parameter `t`; this is crucial for building general purpose coordination abstractions like *polymorphic skeletons* (Section 7) with the HdpH primitives.

HdpH provides the following closure primitives: `unClosure` unwraps a `Closure t` and returns its value of type `t`; `toClosure` wraps a normal form of any serialisable type `t`, i. e. any type which an instance of `Binary`, into a `Closure t`. To construct explicit closures by wrapping thunks, including of types that cannot have `Binary` instances like `Par` computations, HdpH offers a Template Haskell macro for *explicit closure conversion*. More precisely, the Template Haskell splice `$(mkClosure [|e|])` constructs a

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

data Node   -- explicit location (shared-memory node)
data Dist   -- distances between locations
dist     :: Node → Node → Dist       -- metric
equiDist :: Dist → Par [(Node, Int)]  -- basis

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

-- Distribution of tasks
type Task a = Closure (Par (Closure a))
spawn   :: Dist → 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 () -- intern. write
```

**Figure 1.** Types and primitives of HdpH and HdpH-RS.

`Closure t` wrapping the unevaluated thunk `e` of type `t`, provided the captured free variables of `e` are serialisable; see [23] for details.

In HdpH, a *task* computing a value of type `t` is an expression of type `Closure (Par (Closure t))`, i. e. a serialisable monadic computation that will deliver a serialisable value of type `t`. HdpH offers two task distribution primitives, `spawn` and `spawnAt`, the scheduling of which is discussed below (Section 3.3). Both primitives immediately return a *future* [12] of type `IVar (Closure t)`. Such an IVar is a write-once buffer expecting the result of the task, which is an explicit closure of type `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. The internal primitive `rput`[2] transparently writes to a remote future, regardless where it is hosted, and silently fails if the future is already full or the host is dead.

The example below illustrates the use of the HdpH primitives[3] to sum the *Liouville function* [5] 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` (the distance argument `1` will be explained later), how to retrieve the `results` closures, and how to unwrap them and return the final sum.

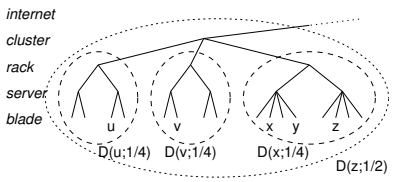
```
parSumLiouville :: Integer → Par Integer
parSumLiouville n = do
  let tasks = [$(mkClosure [|lv k|]) | k ← [1..n]]
  futures ← mapM (spawn 1) tasks
  results ← mapM get futures
  return $ sum $ map unClosure results

lv :: Integer → Par (Closure Integer)
lv k = eval $ toClosure $ (-1)^(length $ primeFactors k)
```

### 3.2 Distance Metric and Equidistant Bases

HdpH takes an abstract view of the network topology, modelling it as a hierarchy, i. e. an unordered tree whose leaves correspond to compute nodes, as in Figure 2 for instance. Every subtree of the hierarchy forms a *virtual cluster*. The interpretation of these

[2] One cannot call `rput` directly; it is used only by scheduler and semantics.

[3] Parallel map skeletons (Section 7) provide a more elegant solution.




| $d$ | u | v | x | y | z |
|---|---|---|---|---|---|
| u | 0 | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ |
| v | $\frac{1}{2}$ | 0 | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ |
| x | $\frac{1}{2}$ | $\frac{1}{2}$ | 0 | $\frac{1}{8}$ | $\frac{1}{4}$ |
| y | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{8}$ | 0 | $\frac{1}{4}$ |
| z | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | 0 |

**Figure 2.** Hierarchy, distance metric and equidistant partition.

virtual clusters is not fixed; e. g. Figure 2 suggests a cluster, possibly connected to others over the Internet, consisting of multiple racks, which in turn house several servers, each containing multiple blades. The hierarchy need not exactly reflect the physical network topology. Rather, it presents a logical arrangement of the network into virtual clusters of manageable size. However, actual latencies should be reasonably compatible, i. e. in general the latency within a virtual cluster should be no higher than the latency between sibling clusters.

Such hierarchies can be represented concisely by a *distance* function $d$ on nodes that is defined by

$$d(p,q) = \begin{cases} 0 & \text{if } p = q \\ 2^{-n} & \text{if } p \neq q \text{ and } n = \text{length of longest} \\ & \text{common path from root to } p \text{ and } q. \end{cases}$$

Figure 2 tables sample distances corresponding to the hierarchy. Mathematically, $d$ defines an *ultrametric space* on the set of nodes. That is, $d$ is non-negative, symmetric, 0 on the diagonal, and satisfies the *strong triangle inequality*: $d(p_1, p_3) \leq \max\{d(p_1, p_2), d(p_2, p_3)\}$ for all nodes $p_1$, $p_2$, $p_3$.

Given a node $p$ and $r \geq 0$, define $D(p; r) = \{q \mid d(p, q) \leq r\}$ to be the *ball* with centre $p$ and radius $r$. Balls correspond to virtual clusters in the hierarchy, see Figure 2 for a few examples. Balls have the following properties, thanks to $d$ being an ultrametric.

1. Every node inside a ball is its centre.
2. Every ball of radius $r = 2^{-n}$ is uniquely partitioned by a set of balls of radius $\frac{1}{2}r$, the centres of which are pairwise spaced distance $r$ apart.

We call the set $\{D(q; \frac{1}{2}r) \mid q \in D(p; r)\}$ the *equidistant partition* of $D(p; r)$. A set $Q$ of nodes is an *equidistant basis* for $D(p; r)$ if $Q$ contains exactly one centre of each ball in the equidistant partition of $D(p; r)$. Due to property 1 equidistant bases are not unique. To illustrate, Figure 2 shows the equidistant partition of $D(z; \frac{1}{2})$, from which we can read off that $\{u, v, x\}$ and $\{u, v, y\}$ are two equidistant bases.

HdpH reifies the metric $d$ as the pure function `dist`, and this implies that all nodes agree on the metric, and that the metric cannot change over time. The primitive `equiDist` takes a radius $r$ and returns a *size-enriched* equidistant basis for $D(p_0; r)$, where $p_0$ is the current node. More precisely, `equiDist` returns a non-empty list `[(`$q_0$`,`$n_0$`),(`$q_1$`,`$n_1$`),...]` such that the $q_i$ form an equidistant basis for $D(p_0; r)$ and $n_i$ is the size of $D(q_i; \frac{1}{2}r)$. By convention, $q_0$ is the current node $p_0$, so the current node can be queried thus:

```
myNode :: Par Node
myNode = do { ((p,_):_) ← equiDist 0; return p }
```

The operator `allNodes :: Par [Node]` for computing a list of all known nodes is also expressible in terms of `equiDist` as a recursive distributed gather operation.

### 3.3 Scheduling

***Lazy, implicit task placement.*** The `spawn` primitive places a task into a local task pool, from where it is scheduled by on-demand work stealing, either locally or onto a remote node looking for work. Crucially, work stealing is non-deterministic, which makes HdpH a non-deterministic DSL because location-awareness, e. g. via calls to `myNode`, may reveal scheduling decisions.

The first argument to `spawn` is the *task radius* $r$ that constrains how far the task can travel from the spawning node $p_0$: it can be scheduled precisely by the nodes in the ball $D(p_0; r)$. The extreme radii deserve special attention: $r = 1$ means the task may be scheduled on any node, and $r = 0$ means the task cannot be scheduled anywhere but $p_0$.

The key features of topology aware on-demand scheduling are as follows, and their implementation is outlined in Section 6.2. No task is ever scheduled beyond its task radius. Tasks with small radii are preferred for local execution. Tasks with big radii are preferably scheduled far away, depending on demand.

***Eager, explicit task placement.*** Scheduling tasks on demand by random work stealing performs well with irregular parallelism. However, it tends to under-utilise large scale architectures at the beginning of the computation. To combat this drawback, HdpH complements `spawn` with `spawnAt`, which places a task on a named node where it is scheduled for execution immediately, taking priority over any implicitly placed tasks. Eager execution implies that the task is meant to perform coordination, e. g. `spawn` further tasks, rather than actual computation.

### 3.4 Fault Tolerance

Crucially each HdpH node's heap is *isolated* from the heaps of other nodes. Hence the failure of one node does not poison computations on other nodes. HdpH-RS provides automatic fault tolerance using Erlang style supervision and recovery of *location-invariant* computations, that is computations that always produce the same effect regardless where they are executed (Section 4). Compared to other languages, fault tolerance in HdpH-RS is relatively sophisticated: for example when Erlang [2] and Hadoop [34] place tasks on remote nodes, these tasks do not move. This simplifies replication and recovery, whereas HdpH-RS provides replication and recovery even when computations migrate between nodes.

***Reliable scheduling.*** Fault tolerance in HdpH-RS is provided by replacing the HdpH scheduler with a *reliable scheduler* that handles failures automatically. The core of the reliable scheduler is an alternative work stealing protocol that enables supervisors to track the locations of tasks. In HdpH-RS, a *supervisor* is a future created by `spawn` or `spawnAt`. As long as it is empty, a supervising future stores a backup copy of the spawned task and monitors the health of all nodes potentially holding the task. The reliable scheduler will recover tasks lost due to node failure by replicating the backups stored in supervising futures elsewhere, ensuring that all futures are eventually filled. The implementation of the HdpH-RS scheduler is sketched in Section 6.3; a complete exposition of the design and implementation can be found in the thesis [30].

## 4. Operational Semantics

This section presents an operational semantics for HdpH and HdpH-RS in the style of [27], focusing on topology aware scheduling and fault recovery. Figure 3 introduces the syntax of terms and values. The language is essentially the same as the embedded DSL presented in Section 3, except that the semantics ignores explicit closures, i. e. assumes that all terms are implicitly serialisable. However, the semantics does restrict the second arguments of `spawn`, `spawnAt` and `rput` to terms $M$ such that $fn(M) = \emptyset$, i. e. terms not containing free (names of) IVars; this is justified because in Section 3 these arguments are explicit closures, which cannot capture free IVars. For simplicity, the semantics also treats the pure function `dist` as a monadic primitive.

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

Values $V ::= \texttt{()} \mid i \mid p \mid r \mid x\ M_1 \ldots M_n \mid \lambda x.M \mid \textbf{fix}\ M$
$\mid\ M\ \texttt{>>=}\ N \mid \texttt{return}\ M \mid \texttt{eval}\ M \mid \texttt{dist}\ p\ q \mid \texttt{equiDist}\ r$
$\mid\ \texttt{spawn}\ r\ M \mid \texttt{spawnAt}\ p\ M \mid \texttt{get}\ i \mid \texttt{rput}\ i\ M$

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

States $R, S, T ::= S \mid T$ parallel composition
$\mid\ \nu i.S$ name restriction
$\mid\ \langle M \rangle_p$ thread on node $p$, executing $M$
$\mid\ \langle\!\langle M \rangle\!\rangle^r_p$ spark on $p$ with radius $r$, to exec $M$
$\mid\ i\{M\}_p$ full IVar $i$ on node $p$, holding $M$
$\mid\ i\{\}_p$ empty IVar $i$ on node $p$
$\mid\ i\{\langle M \rangle_q\}_p$ RS: empty IVar $i$ on $p$, sv'ing thread on $q$
$\mid\ i\{\langle\!\langle M \rangle\!\rangle^r_Q\}_p$ RS: empty IVar $i$ on $p$, sv'ing spark on $Q$
$\mid\ i\{\bot\}_p$ RS: zombie IVar $i$ on node $p$
$\mid\ \text{dead}_p$ RS: notification that node $p$ is dead

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

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

$$S \mid T \equiv T \mid S \qquad \nu i.\nu j.S \equiv \nu j.\nu i.S$$

$$R \mid (S \mid T) \equiv (R \mid S) \mid T \qquad \nu i.(S \mid T) \equiv (\nu i.S) \mid T, \quad i \notin fn(T)$$

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

**Figure 4.** Structural congruence and structural transitions.

For the purposes of the DSL semantics, the host language is a standard lambda calculus with fixed points and some data constructors for nodes, distances, 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 big-step semantics is entirely standard (and omitted). Note that the syntax of values in Figure 3 implies that the DSL primitives are strict in arguments of type `Node`, `Dist` and `IVar`.

### 4.1 Semantics of HdpH

The operational semantics of the HdpH DSL is a small-step reduction semantics $\longrightarrow_d$ indexed by a distance metric $d$. The reduction relation operates 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 indicated by the subscript $p$. An atomic state of the form $\langle M \rangle_p$ or $\langle\!\langle M \rangle\!\rangle^r_p$, where $M$ is a computation of type `Par ()`, denotes a *thread* or *spark*, respectively; sparks differ from threads in that they may migrate within radius $r$ around their current node $p$. 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 empty or full. 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. Most of these rules execute a thread, relying on an *evaluation context* $\mathcal{E}$ to select the first action of the thread's monadic computation. Rules that are similar to those in [27] are not explained in detail.

The first three rules are standard for monadic DSLs; note how `eval` is just a strict `return`. 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 residing on $p$ and bounded by radius $r$. In contrast, `spawnAt` wraps $M$ into a thread, which is placed on node $q$. The side condition on both rules ensures that the name $i$ is *fresh*, i. e. does not occur free in the current thread. The rules for IVars are similar to those in [27] except that IVars in HdpH can only be read on the node they reside on. They can however be written from any node, and writes can be raced;[4] the first write wins, subsequent writes have no effect. The rules (dist) and (equiDist) define the eponymous topology aware primitives. These two rules, and the spark migration rule, are the only ones that actually require the distance metric $d$.

Rules (migrate) and (convert) govern the scheduling of sparks. A spark may migrate from node $p$ to $q$, provided the distance between the two is bounded by the spark's radius $r$. Sparks cannot be executed directly; instead they must be converted into threads that can execute but not migrate. The (gc_*) rules eliminate garbage, i. e. terminated threads and inaccessible IVars. Note that to become garbage, IVars must be filled and sparks must be converted and executed to termination.

We call a thread $\langle M \rangle_p$ *reachable* from a state $S$ iff there is a state $T$ such that $S \longrightarrow^*_d \nu i_1 \ldots \nu i_n.(T \mid \langle M \rangle_p)$, where $\longrightarrow^*_d$ denotes the reflexive-transitive closure of $\longrightarrow_d$. We call state $S$ *well-formed* iff there is a *root thread* $\langle M \rangle_p$ with $fn(M) = \emptyset$ such that $\langle M \rangle_p \longrightarrow^*_d S$. We observe that $\longrightarrow_d$ reductions starting from well-formed states cannot get stuck except when embedding the host language, namely term $M$ diverging in rules (normalize) and (eval). In particular, well-formedness guarantees that all `rput`s find their target IVars, that all `get`s find their source IVars, and that these source IVars are hosted locally.

### 4.2 Fault Tolerant Semantics of HdpH-RS

The operational semantics of HdpH-RS $\xrightarrow{\text{RS}}_d$ is an extension of $\longrightarrow_d$, i. e. it is a small-step reduction relation on states defined by the same rules, with some small adaptions and some additions.

To model supervision, empty IVars $i\{\langle M \rangle_q\}_p$ and $j\{\langle\!\langle N \rangle\!\rangle^r_Q\}_p$ are annotated with the thread $M$ resp. spark $N$ that is supposed to fill them and with some knowledge of the current location of $M$ resp. $N$. In case of non-migratable thread $M$ that knowledge is the node $q$ where $M$ was scheduled by `spawnAt`. In case of spark $N$, however, the supervisor may not know the actual node due to migration, hence $i$ is annotated with a set of nodes $Q$ overapproximating the true location of $N$ (or of the thread that $N$ has been converted to).

To model node failure, we add atomic states $\text{dead}_p$, signalling that node $p$ has died, and $i\{\bot\}_p$, representing a *zombie* IVar $i$, i. e. an effectively dead IVar $i$ on a dead node $p$. The four rules in the top right corner of Figure 6 define the *fault model* of HdpH-RS. A node $p$ may die any time, signalled by the spontaneous production of $\text{dead}_p$, 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 $\text{dead}_p$.

The remaining rules in Figure 6 are the new/adapted transitions for HdpH-RS. Rules (rput_empty_thread) and (rput_empty_spark) fill empty supervising IVars. Rule (rput_zombie) lets remote writes to zombie IVars to fail silently, and (gc_zombie) garbage collects inaccessible zombie IVars. The rules (spawn) and (spawnAt) are identical to the HdpH rules except for remembering the new

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

$$\langle \mathcal{E}[M] \rangle_p \longrightarrow_d \langle \mathcal{E}[V] \rangle_p, \quad \text{if } M \Downarrow V \text{ and } M \not\equiv V \qquad \text{(normalize)}$$

$$\langle \mathcal{E}[\texttt{return}\, N \texttt{ >>= } M] \rangle_p \longrightarrow_d \langle \mathcal{E}[M\ N] \rangle_p \qquad \text{(bind)}$$

$$\langle \mathcal{E}[\texttt{eval}\, M] \rangle_p \longrightarrow_d \langle \mathcal{E}[\texttt{return}\, V] \rangle_p, \quad \text{if } M \Downarrow V \qquad \text{(eval)}$$

$$\langle \mathcal{E}[\texttt{spawn}\, r\, M] \rangle_p \longrightarrow_d \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\}_p \mid \langle\!\langle M \texttt{ >>= rput } i \rangle\!\rangle^r_p), \quad \text{where } i \notin fn(\mathcal{E}) \qquad \text{(spawn)}$$

$$\langle \mathcal{E}[\texttt{spawnAt}\, q\, M] \rangle_p \longrightarrow_d \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\}_p \mid \langle M \texttt{ >>= rput } i \rangle_q), \quad \text{where } i \notin fn(\mathcal{E}) \qquad \text{(spawnAt)}$$

$$\langle \mathcal{E}[\texttt{rput}\, i\, M] \rangle_p \mid i\{\}_q \longrightarrow_d \langle \mathcal{E}[\texttt{return ()}] \rangle_p \mid i\{M\}_q \qquad \text{(rput\_empty)}$$

$$\langle \mathcal{E}[\texttt{rput}\, i\, M] \rangle_p \mid i\{N\}_q \longrightarrow_d \langle \mathcal{E}[\texttt{return ()}] \rangle_p \mid i\{N\}_q \qquad \text{(rput\_full)}$$

$$\langle \mathcal{E}[\texttt{get}\, i] \rangle_p \mid i\{M\}_p \longrightarrow_d \langle \mathcal{E}[\texttt{return}\, M] \rangle_p \mid i\{M\}_p \qquad \text{(get)}$$

$$\langle \mathcal{E}[\texttt{dist}\, q_1\, q_2] \rangle_p \longrightarrow_d \langle \mathcal{E}[\texttt{return}\, d(q_1, q_2)] \rangle_p \qquad \text{(dist)}$$

$$\langle \mathcal{E}[\texttt{equiDist}\, r] \rangle_p \longrightarrow_d \langle \mathcal{E}[\texttt{return}\, M] \rangle_p, \quad \text{where } M \text{ is an equidistant basis for the ball } D(p; r) \qquad \text{(equiDist)}$$

$$\langle\!\langle M \rangle\!\rangle^r_{p_1} \longrightarrow_d \langle\!\langle M \rangle\!\rangle^r_{p_2}, \quad \text{if } d(p_1, p_2) \leq r \qquad \text{(migrate)}$$

$$\langle\!\langle M \rangle\!\rangle^r_p \longrightarrow_d \langle M \rangle_p \qquad \text{(convert)}$$

$$\langle \texttt{return ()} \rangle_p \longrightarrow_d \qquad \text{(gc\_thread)}$$

$$\nu i.i\{M\}_p \longrightarrow_d \qquad \text{(gc\_ivar)}$$

**Figure 5.** Small-step semantics of HdpH.

$$\langle \mathcal{E}[\texttt{rput}\, i\, M] \rangle_p \mid i\{\langle N \rangle_p\}_q \xrightarrow{\text{RS}}_d \langle \mathcal{E}[\texttt{return ()}] \rangle_p \mid i\{M\}_q \qquad \text{(rput\_empty\_thread)}$$

$$\langle \mathcal{E}[\texttt{rput}\, i\, M] \rangle_p \mid i\{\langle\!\langle N \rangle\!\rangle^r_Q\}_q \xrightarrow{\text{RS}}_d \langle \mathcal{E}[\texttt{return ()}] \rangle_p \mid i\{M\}_q \qquad \text{(rput\_empty\_spark)}$$

$$\langle \mathcal{E}[\texttt{rput}\, i\, M] \rangle_p \mid i\{\bot\}_q \xrightarrow{\text{RS}}_d \langle \mathcal{E}[\texttt{return ()}] \rangle_p \mid i\{\bot\}_q \qquad \text{(rput\_zombie)}$$

$$\nu i.i\{\bot\}_q \xrightarrow{\text{RS}}_d \qquad \text{(gc\_zombie)}$$

$$\xrightarrow{\text{RS}}_d \text{dead}_p \qquad \text{(dead)}$$

$$\text{dead}_p \mid \langle M \rangle_p \xrightarrow{\text{RS}}_d \text{dead}_p \qquad \text{(kill\_thread)}$$

$$\text{dead}_p \mid \langle\!\langle M \rangle\!\rangle^r_p \xrightarrow{\text{RS}}_d \text{dead}_p \qquad \text{(kill\_spark)}$$

$$\text{dead}_p \mid i\{?\}_p \xrightarrow{\text{RS}}_d \text{dead}_p \mid i\{\bot\}_p \qquad \text{(kill\_ivar)}$$

$$\langle \mathcal{E}[\texttt{spawn}\, r\, M] \rangle_p \xrightarrow{\text{RS}}_d \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle\!\langle M \texttt{ >>= rput } i \rangle\!\rangle^r_{\{p\}}\}_p \mid \langle\!\langle M \texttt{ >>= rput } i \rangle\!\rangle^r_p), \quad \text{where } i \notin fn(\mathcal{E}) \qquad \text{(spawn}^\dagger\text{)}$$

$$\langle \mathcal{E}[\texttt{spawnAt}\, q\, M] \rangle_p \xrightarrow{\text{RS}}_d \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle M \texttt{ >>= rput } i \rangle_q\}_p \mid \langle M \texttt{ >>= rput } i \rangle_q), \quad \text{where } i \notin fn(\mathcal{E}) \qquad \text{(spawnAt}^\dagger\text{)}$$

$$\langle\!\langle M \rangle\!\rangle^r_{p_1} \mid i\{\langle\!\langle M \rangle\!\rangle^r_P\}_q \xrightarrow{\text{RS}}_d \langle\!\langle M \rangle\!\rangle^r_{p_2} \mid i\{\langle\!\langle M \rangle\!\rangle^r_P\}_q, \quad \text{if } d(p_1, p_2) \leq r \text{ and } p_1, p_2 \in P \qquad \text{(migrate}^\dagger\text{)}$$

$$\langle\!\langle M \rangle\!\rangle^r_p \mid i\{\langle\!\langle M \rangle\!\rangle^r_{P_1}\}_q \xrightarrow{\text{RS}}_d \langle\!\langle M \rangle\!\rangle^r_p \mid i\{\langle\!\langle M \rangle\!\rangle^r_{P_2}\}_q, \quad \text{if } p \in P_1 \cap P_2 \qquad \text{(track)}$$

$$i\{\langle M \rangle_q\}_p \mid \text{dead}_q \xrightarrow{\text{RS}}_d i\{\}_p \mid \langle M \rangle_p \mid \text{dead}_q \qquad \text{(recover\_thread)}$$

$$i\{\langle\!\langle M \rangle\!\rangle^r_Q\}_p \mid \text{dead}_q \xrightarrow{\text{RS}}_d i\{\langle\!\langle M \rangle\!\rangle^r_{\{p\}}\}_p \mid \langle\!\langle M \rangle\!\rangle^r_p \mid \text{dead}_q, \quad \text{if } q \in Q \qquad \text{(recover\_spark)}$$

**Figure 6.** Additional rules for small-step semantics of HdpH-RS; rules marked with † replace eponymous HdpH rules.

spark/thread in the empty IVar $i$. Rule (migrate) works similarly as in HdpH except for ensuring that the supervising IVar $i$ continues to track the migrating spark, i. e. the new location of the spark remains a member of the tracking set $P$. That set may change via rule (track) in arbitrary ways, provided the current location of the supervised spark remains a member, modelling the supervisor's changing and uncertain knowledge about the location of a supervised spark.

The final two 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 $q$ is replicated on $p$, after which $i$ ceases to supervise 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.

### 4.3 Relating Fault Tolerant and Fault Oblivious Semantics

In order to relate the HdpH and HdpH-RS semantics, we need to compare their respective normal forms. In HdpH, thread $\langle N \rangle_p$ is a *normal form* of state $S$, denoted $S \downarrow_d \langle N \rangle_p$, iff $S \longrightarrow_d^* \langle N \rangle_p$ and $\langle N \rangle_p$ is irreducible or $N = \texttt{return ()}$. We have defined normal forms of arbitrary states $S$, yet we will mostly be interested in normal forms of threads $\langle M \rangle_p$ with $fn(M) = \emptyset$, as these threads correspond to tasks spawned and potentially replicated. Note that the restriction $fn(M) = \emptyset$ makes $\langle M \rangle_p$ a root thread guaranteeing well-formedness of normal forms, hence precluding normal forms being stuck (up to divergence).

Before defining HdpH-RS normal forms, we note that in any $\xrightarrow{\text{RS}}_d$ reduction, rule (dead) permutes with every rule to the left. Consequently, we ban rule (dead) and instead start reduction from states of the form $S \mid \text{dead}_P$, where $P = \{p_1, \ldots, p_n\}$ is a set of nodes and $\text{dead}_P$ is short for $\text{dead}_{p_1} \mid \ldots \mid \text{dead}_{p_n}$.

In HdpH-RS, thread $\langle N \rangle_p$ is a *normal form* of state $S$, written $S \downarrow_d^{\text{RS}} \langle N \rangle_p$, iff there is a set $P$ such that $S \mid \text{dead}_P \xrightarrow{\text{RS}}_d^* \langle N \rangle_p \mid \text{dead}_P$ and $\langle N \rangle_p$ is irreducible or $N = \texttt{return ()}$. Moreover, $\langle N \rangle_p$ is a *failure-free normal form* of $S$, denoted $S \downarrow_d^{\text{FF}} \langle N \rangle_p$, if it satisfies the above definition with $P = \emptyset$. It is immediate that reductions leading to a failure-free normal form cannot use any of the rules (kill_*), (recover_*) and (gc_zombie).

We can prove that HdpH and HdpH-RS agree on normal forms in the absence of failures.

**Lemma 1.** *Let $M$ be a term with $fn(M) = \emptyset$. Then for all terms $N$ and nodes $p$, $\langle M \rangle_p \downarrow_d \langle N \rangle_p \Leftrightarrow \langle M \rangle_p \downarrow_d^{FF} \langle N \rangle_p$.*

*Proof sketch.* The bisimilarity between HdpH and HdpH-RS reductions is obvious, except for the rules governing spark migration. For the forward direction a (track) transition must be inserted before and after every (migrate) transition; for the reverse direction it suffices to delete all (track) transitions. □

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. Let $M$ be a term with $fn(M) = \emptyset$. We call $M$ *location-invariant* iff it does not matter where it executes, that is $\langle M \rangle_p \downarrow_d \langle N \rangle_p \Leftrightarrow$

$\langle M\rangle_q \downarrow_d \langle N\rangle_q$, for all terms $N$ and nodes $p$ and $q$. We call $M$ *transitively location-invariant* iff for all nodes $p$ and all root threads $\langle N\rangle_q$ reachable from $\langle M\rangle_p$, $N$ is location-invariant.

Now we can prove that the failure-free normal forms of transitively location-invariant terms are exactly their HdpH-RS normal forms.

**Lemma 2.** *Let $M$ be a term with $fn(M) = \emptyset$. If $M$ is transitively location-invariant then for all terms $N$ and nodes $p$, $\langle M\rangle_p \downarrow_d^{RS} \langle N\rangle_p \Leftrightarrow \langle M\rangle_p \downarrow_d^{FF} \langle N\rangle_p$.*

*Proof sketch.* The reverse direction is trivial. For the forward direction, construct a failure-free reduction from a HdpH-RS reduction by induction on the number of (recover_*) rules. For each replicated spark, the failure-free reduction mimics the migration and execution of the successful replica, i. e. the replica that eventually filled the spark's IVar. For each replicated thread, the failure-free reduction mimics the execution of the successful replica, yet translated to the node $q$ to which the original thread was spawned (ignoring that $q$ is supposed to be dead); thanks to location-invariance this translation does not affect the normal form that is eventually written to the thread's IVar. □

Combining lemmas 1 and 2, we find that, for transitively location-invariant terms at least, HdpH and HdpH-RS agree on the normal forms.

**Theorem 3.** *Let $M$ be a term with $fn(M) = \emptyset$. If $M$ is transitively location-invariant then for all terms $N$ and nodes $p$, $\langle M\rangle_p \downarrow_d \langle N\rangle_p \Leftrightarrow \langle M\rangle_p \downarrow_d^{RS} \langle N\rangle_p$.*

***Observations.*** Firstly, inspecting the proof sketch of Lemma 2, it is obvious that location-invariance is not actually required of all reachable root threads but only of the ones arising from `spawnAt`. Thus the precondition of Theorem 3 could be weakened. In particular, location-invariance could be dropped completely for the sublanguage that restricts task distribution to `spawn` only.

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 semantic equivalence between HdpH and HdpH-RS programs.

Finally, HdpH and HdpH-RS are non-deterministic in general as decisions taken by the non-deterministic scheduler may become observable, e. g. in case migrating tasks call `myNode` (defined in Section 3.2). The sublanguage that restricts task distribution to `spawnAt` only is deterministic, due to entirely deterministic scheduling. Whether there are more interesting deterministic sublanguages, in the face of truly non-deterministic scheduling, is an interesting and timely [16] open question.

## 5. Validating the HdpH-RS Scheduler

Due to the various sources of non-determinism in faulty distributed systems it is easy to make mistakes in their correctness arguments, hence the need for validation by model checking and testing.

Replication is a common fault tolerance technique, for example in the Erlang *supervisor* behaviour [19] and Hadoop [34]. In both Erlang and Hadoop, tasks are placed only once, simplifying replication and recovery. In contrast, the HdpH-RS scheduler must consider spark migration when identifying replication candidates.

***HdpH-RS Promela abstraction.*** The unbounded state space of the HdpH-RS scheduler is abstracted as a finite state Promela model. The Promela abstraction models node failure, and the latencies of work stealing and failure detection in the network abstraction layer (Section 6.1).

The model only validates the supervision of a spark created with `spawn`; threads placed by `spawnAt` cannot migrate and hence are a far simpler to supervise. The model includes one immortal supervising node, three mortal work stealing nodes, one spark and its corresponding initially empty future. The work stealing routines on the supervisor and three thieves are translated to a finite automaton, incorporating the six additional RTS messages needed for reliable work stealing (Section 6.3). If the supervisor detects a node failure that may eliminate the spark from the abstract machine state, it adds a replica to its local sparkpool, honouring the (recover_spark) rule from Section 4. Any node that holds a spark replica may at any time transmit a value into the empty future hosted on the supervisor. The model is described in full in [30], the Promela code is available [31].

***Verification with SPIN.*** A key property of the HdpH-RS design is to guarantee the evaluation of supervised sparks, as recorded by filling the corresponding IVar on the supervising node. We model this by defining *ivar_full* to be a Boolean that becomes `true` when the future hosted on the supervisor is filled. Now we can specify in linear temporal logic that this variable is always eventually true, i. e. $\Diamond\,\Box$ *ivar_full*. The property is true despite arbitrary combinations of node failures, provided that the supervising node itself does not fail. Checking this property increases our confidence that the elaborate HdpH-RS work stealing protocol outlined in Section 6.3 correctly implements the semantics in Section 4, and in particular the rules (migrate) and (track).

The SPIN model checker exhaustively searches the model's state space to validate that the property holds in all states. SPIN explores 22.4 million transitions to a reachable depth of 124 transitions, proving that none of the 8.2 million reachable states violate the property.

***Chaos Monkey testing.*** Besides model checking an abstraction, fault injection [13] was used on a suite of benchmarks (Section 8) to test the resilience of HdpH-RS in the presence of multiple random failures. Tests compare failure-free HdpH runs with HdpH-RS runs in the presence of random failures. All tests pass [30].

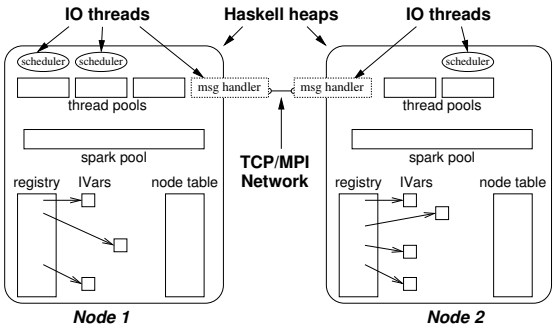



**Figure 7.** Runtime system architecture of HdpH and HdpH-RS.

## 6. Implementation

### 6.1 RTS architecture

Figure 7 depicts the key data structures and threads that make up the shared HdpH and HdpH-RS RTS architecture. As the RTS is implemented in Haskell, the data structures are concurrently mutable maps, tables and queues in the Haskell heap, and the threads are Haskell IO threads. Each node runs several *scheduler* IO threads, typically one per core, and a *message handler* IO thread.

Each scheduler owns a *thread pool*, a concurrent double-ended queue storing *threads*, i. e. computations of type `Par ()`. The back end of the queue is only accessible to the owner, both for enqueueing, e. g. after unblocking, and for retrieving threads. The front end of the queue is accessible to all schedulers for stealing threads, similar to [27]. The message handler also owns a thread pool. However, unlike the schedulers, the message handler never retrieves threads; its threads must be stolen by a scheduler in order to be executed.

HdpH maintains one *spark pool* per node, a distance-indexed set of concurrent double-ended queues for storing *sparks*, i. e. serialisable computations of type `Closure (Par ())`. For the role of the spark pool in scheduling see Section 6.2.

*IVars* are write-once buffers that are either empty or full, where empty IVars may also store a list of blocked threads to support the blocking `get` primitive as in [27], and a record of the thread or spark that is supposed to fill the IVar, to support task replication in HdpH-RS (Section 6.3).

HdpH maintains a *registry* per node, providing globally unique handles to locally hosted IVars, in order to support remote writing via `rput`. The registry is a concurrent map linking handles to their underlying IVars as detailed in [23].

For scalability, HdpH avoids a central table of all nodes. Instead, each node maintains its own *node table*, which is a distance-indexed set of some other nodes that it knows about. At system startup, the node table is initialised so that it holds random equidistant bases $Q_r$, one per distance $r$. The primitive `equiDist` returns exactly these $Q_r$. The node table also records individual nodes $q_r$, one per distance $r$, that have recently scheduled work to this node.

***Two communication backends*** have been developed for HdpH. The first provides MPI-based message passing for HPC architectures. Failures are fatal as MPI aborts on discovering faults.

The second backend targets COTS architectures and uses `network-transport`, a TCP-based network abstraction layer (NAL) designed for distributed Haskells such as CloudHaskell [8]. The NAL provides connection-oriented communication primitives that simplify message passing and fault detection. In particular, the NAL generates connection-lost events which indicate potential remote node failure. The backend reacts to these events and eventually, after failing to re-establish a lost connection over a period of time, propagates `DEADNODE` messages to the HdpH-RS scheduler.

### 6.2 Topology Aware Scheduling

Rule (migrate) in Section 4.1 models topology aware scheduling non-deterministically, constrained by the distance metric. This section details aspects of HdpH's topology aware work stealing algorithm, including its task selection policy. When a node $p_0$ executes the call `spawn r task`, the `task` is converted into a *spark* (which involves creating and registering an IVar for the result) and added to the spark pool queue for distance `r`.

When $p_0$ runs out of work, and its own spark pool is non-empty, it uses the following *local spark selection policy*: Pick a spark with minimal radius; if there are several such sparks, pick the one at the back of the queue, i. e. the youngest or most recently stolen spark. Thus, HdpH prioritises sparks with small radii for local scheduling. As an aside, local scheduling requires to `unClosure` the spark, thereby *converting* it into a thread.

If, on the other hand, $p_0$ runs out of work with its own spark pool empty then it will engage in a distributed work stealing protocol comprising the messages FISH, SCHEDULE and NOWORK. In fact $p_0$ does not wait for its spark pool to drain completely; to hide latency work stealing is initiated as soon as the spark pool hits a *low water mark*.

Figure 8 illustrates the protocol with a successful and an unsuccessful stealing attempt. In both cases, $p_0$ starts by sending a FISH message to a random node nearby, that is a minimal distance away.

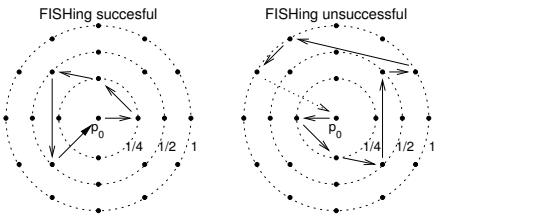



**Figure 8.** HdpH topology aware work stealing protocol.

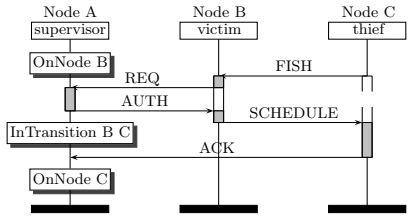



**Figure 9.** Fault tolerant work stealing protocol in HdpH-RS.

When a node $p$ receives a FISH message originating from $p_0$, it tries to find a suitable spark using the following *remote spark selection policy*: Pick a spark with minimal radius from the set of sparks whose radius is greater or equal to $d(p, p_0)$; if there are several such sparks, pick the one at the front of the queue, i. e. the oldest one. Thus for remote scheduling, HdpH prioritises sparks whose radii match the distance to the node requesting work. If remote spark selection is successful, $p$ sends a SCHEDULE message containing the selected spark $M$ and its radius $r$ to $p_0$. On receipt of SCHEDULE from $p$, $p_0$ inserts $M$ into the spark pool queue for distance $r$, and records $p$ in the node table at distance $r$. If remote spark selection is unsuccessful, $p$ forwards the FISH message to a random node $q$ such that $d(q, p_0) = d(p, p_0)$ or $d(q, p_0) > d(p, p_0)$. That is, the FISH message slowly "ripples away" from its originating node, as illustrated in Figure 8; how slowly depends on how often the message is forwarded to a node at the same distance. If there is no work the FISH will eventually be forwarded to a node $q$ such that $d(q, p_0) = 1$ is maximal. To stop the FISH bouncing forever, it will only be forwarded further a fixed number of times, 2 in Figure 8, after which a NOWORK message is sent to $p_0$. Upon receipt of NOWORK, $p_0$ backs off for some milliseconds before initiating another work stealing attempt.

The node table is used to select the targets for a forwarded FISH. Before $p$ forwards a FISH originating from $p_0$ it decides whether to increase the distance $r = d(p, p_0)$ i. e. whether to ripple further out. If not, $p$ forwards to a random node of the equidistant basis $Q_r$ recorded in its node table. If the distance is increased (minimally) to $r' > d(p, p_0)$ then $p$ forwards to the node $q_{r'}$ recorded in its node table as recent source of work at distance $r'$, if it exists, otherwise to a random node of the equidistant basis $Q_{r'}$. Thus, the work stealing protocol combines random searching for work with targeted probing of recent sources, while prioritising stealing from nearby.

### 6.3 Fault Tolerant Scheduling

The fault tolerance implementation in HdpH-RS is threefold. First, the HdpH scheduler is extended to track task locations. Second, the structure of `IVars` are extended to fulfil the role of supervised futures. Third, the scheduler replicates potentially lost tasks when failures are detected.

***Reliable scheduling extension.*** The fault tolerant work stealing protocol is illustrated with a message sequence chart in Figure 9.

The protocol involves the supervisor in spark migration, and six additional RTS messages are used in HdpH-RS for the purpose of supervised work stealing. A thief targets a victim with a FISH message. If the victim has a sufficient number of sparks, then it sends a request to the supervisor as a REQ message for it be scheduled to the thief. The location state recorded by a supervisor for a spark is either `OnNode` or `InTransition`. The supervisor checks that the spark's location is marked as `OnNode`. If it is, an AUTH message is returned to the victim. Otherwise, a DENIED message is returned. When the supervisor and victim is the same node i.e. the spark is on the supervisor, the REQ and AUTH messages by-pass the network layer (Section 6.1). Instead, local function calls are used to determine the response to a FISH message.

Replica counts are used to avoid race conditions when multiple replicas co-exist. Only the spark tagged with the highest replica number may be scheduled elsewhere. The response to a REQ message regarding an older replica is an OBSOLETE message. A node that receives an OBSOLETE reply will discard the spark and send a NOWORK message to the thief.

***Supervised futures.*** The `spawn` and `spawnAt` HdpH-RS primitives create extended versions of IVars to store additional state for fault tolerance. A copy of the task closure is held within the empty IVar, in case replication is later necessary. The location of the corresponding spark or thread, either `OnNode(`$p$`)` or `InTransition(`$p$`,`$q$`)`, is stored in the IVar, together with a replica number counting how often the spark or thread has been replicated. A flag indicating whether to schedule the task lazily or eagerly is also stored in the IVar.

A spark created with `spawn` in HdpH-RS is transmitted as a tuple consisting of the following three components: the task to be evaluated, the task replica number, and a global handle to the IVar that will receive the task's result. The replica number and IVar handle are included in REQ and ACK messages to allow the supervisor to update the location state of the corresponding IVar.

***Task replication.*** Task location state is used in the recovery phase to ensure that lost tasks are replicated. If failure is reported, i.e. a DEADNODE($p$) message is received from the transport layer (Section 6.1), then the state of all empty IVars in the registry is inspected to identify replication candidates. A task is replicated in either of two cases. First, when its location record is `OnNode(`$p$`)`, indicating that it was on the dead node at the point of failure. Second, when its location record is `InTransition(`$p$`,`$q$`)` or `InTransition(`$q$`,`$p$`)`, indicating that the task was in-flight either towards or away from the dead node.

This pessimistic replication strategy may lead to multiple copies of a spark. A migrating spark may survive a node failure, provided it was stolen from the failed node in time. Hence, an obsolete spark may be executed and its result written to the IVar. Assuming idempotence, this scenario is indistinguishable from the one where the obsolete spark has been lost.

The replication of sparks conforms to the (recover_spark) rule in Section 4. If a spark is to be re-scheduled, the replica count in the IVar is incremented. Then a new spark, consisting of the stored task, replica number and IVar handle, is added to the supervisor's spark pool, from where it may be stolen once again. The replication of threads is simpler, and conforms to the (recover_thread) rule. Rescheduling a thread is done by adding the stored task to the thread pool of the supervisor's message handler.

## 7. Algorithmic Skeletons

HdpH skeletons provide high-level coordination abstractions and are implemented using the primitives of Section 3. These abstractions provide topology awareness or fault tolerance depending on whether they are run by the topology aware scheduler of HdpH, or the reliable scheduler of HdpH-RS. HdpH and HdpH-RS provide libraries with around 30 skeletons, including several divide-and-conquer, map/reduce, parallel map, and parallel buffer variants [25, 30]. Figure 10 outlines a selection of skeletons used in the evaluation in Section 8.

```
parMapSliced, pushMapSliced  -- slicing parallel map
  :: Int                -- number of slices
  → Closure (a → b)     -- function closure
  → [Closure a]         -- input list
  → Par [Closure b]     -- output list

parMapReduceRangeThresh, pushMapReduceRangeThresh -- d&c
  :: Closure Int        -- threshold
  → Closure (Int,Int)   -- range to divide/compute over
  → Closure (Closure Int → Par (Closure a)) -- map fun
  → Closure (Closure a → Closure a → Par (Closure a))
  → Closure a           -- initial value for reduction
  → Par (Closure a)     -- mapreduced result

parMapLocal             -- bounded parallel map
  :: Dist               -- bounding radius
  → Closure (a → b)     -- function closure
  → [Closure a]         -- input list
  → Par [Closure b]     -- output list
parMapLocal r f xs = mapM fork xs >>= mapM get where
  fork x = spawn r $(mkClosure
    [|eval $ toClosure (unClosure f $ unClosure x)|])

parMap2Level, parMap2LevelRelaxed  -- 2-level par map
  :: Dist               -- pushing radius
  → Closure (a → b)     -- function closure
  → [Closure a]         -- input list
  → Par [Closure b]     -- output list
parMap2Level r f xs = do
  basis ← equiDist r
  let chunks = chunkWith basis xs
  futures ← mapM spawnChunk chunks
  concat <$> mapM (fmap unClosure ∘ get) futures where
    spawnChunk (q,xs) = spawnAt q $(mkClosure
      [|toClosure <$> parMapLocal (r/2) f xs|])
```

**Figure 10.** Some HdpH skeleton APIs and implementations.

***Topology agnostic skeletons*** make no use of the HdpH distance primitives.[5] Four such skeletons are used in the evaluation of HdpH-RS (Section 8.3). The skeletons `parMapSliced` and `pushMapSliced` divide the input list into a given number of slices and evaluate each slice in parallel. For example, dividing the list $[e_1, ..., e_5]$ into three slices yields a list $[[e_1, e_4], [e_2, e_5], [e_3]]$ and three parallel tasks that are distributed lazily by `parMapSliced` or eagerly in a round-robin fashion by `pushMapSliced`.

Two divide-and-conquer skeletons are used to implement Mandelbrot in Section 8.3, again with both lazy and eager task placement. The skeletons generalise the `parMapReduceRangeThresh` skeleton of the `Par` monad library [27] to distributed memory. The skeletons combine a map over a finite range, which is recursively split until its size falls under a threshold, with a binary reduction of the map results. Task placement relies on work stealing for `parMapReduceRangeThresh`, whereas tasks are eagerly pushed to random nodes with `pushMapReduceRangeThresh`. In HdpH-RS these skeletons create a nested supervision tree that reflects the divide-and-conquer call tree.

***Topology aware skeletons*** exploit the HdpH distance primitives to *control locality* by (1) restricting work stealing to nearby nodes,

[5] The HdpH distance primitives can be used in HdpH-RS but HdpH-RS assumes the topology to be discrete.

e. g. `parMapLocal` (Figure 10) creates tasks bounded by radius `r`, resulting in a lazy distribution of work to nodes at most distance `r` from the caller; and (2) eagerly spreading tasks to distant nodes across the system. For example `parMap2Level` uses a combination of eager and lazy work distribution. After obtaining an equidistant `basis` for radius `r`, it splits the input list into chunks, one per basis node, taking into account the size information present in the basis, and eagerly spawns a *big task* per basis node. This achieves quick distribution of big tasks across the architecture. Eagerly evaluating their big tasks with `parMapLocal`, each basis node becomes a local coordinator: spawning *small tasks* to be evaluated in their vicinity, i. e. at a distance of no more than $\mathtt{r}/2$. Thanks to equidistance of the basis nodes, the bounding radius of $\mathtt{r}/2$ guarantees that small tasks cannot stray too far.

A variant of this two-level skeleton, `parMap2LevelRelaxed`, differs only in relaxing the bound imposed on small tasks from $\mathtt{r}/2$ to `r`. The effect is to allow the stealing of small tasks even between previously isolated local coordinators, which can help mitigate imbalances in task distribution arising from irregular parallelism. Due to the work stealing algorithm's preference for local work (Section 6.3), stealing due to the relaxation is a last resort, and occurs mostly in the final stages of a computation when work is drying up.

All topology aware skeletons provide a semi-explicit interface for tuning of locality via a single distance parameter, without ever exposing locations. This abstract locality control is intended to facilitate performance portability between parallel architectures. By not exposing locations these skeletons are *location-invariant* in the sense of Section 4.3, so their semantics won't change when switching from HdpH to HdpH-RS, although the performance is likely to change. We conjecture that not exposing locations also guarantees that these skeletons hide the effects of non-deterministic scheduling and compute deterministic results.

## 8. Evaluation

***Benchmark platforms.*** HdpH and HdpH-RS are evaluated on HECToR and a COTS Beowulf cluster using the appropriate communication backends. HECToR is the UK's publicly funded HPC platform with a total of 90K cores; it comprises 2816 compute nodes, each with 32 AMD Opteron cores at 2.3GHz sharing 32GB of RAM, divided into 4 NUMA regions. The 256 core Beowulf cluster comprises 32 nodes connected via Gigabit Ethernet; each node has 12GB of memory and 8 Intel Xeon cores at 2GHz.

***Benchmarks applications.*** We evaluate scaling and topology awareness of HdpH on two version of the SumEuler benchmark (Sections 8.1 and 8.2) and on a computational algebra case study (Section 8.4). Scaling and fault tolerance of HdpH-RS is evaluated on the Mandelbrot and Summatory Liouville benchmarks (Section 8.3). The benchmarks typically compare several coordination alternatives like distributing work lazily/eagerly, being topology aware/agnostic, or being fault tolerant/oblivious. The sources of HdpH and HdpH-RS, including benchmark applications, are publicly available [22, 30].

### 8.1 Scaling

We investigate the weak scaling of HdpH from 1 to 1024 HECToR nodes (i. e. from 32 to 32K cores) using the moderately irregular SumEuler benchmark, a data-parallel computation of the sum of Euler's $\varphi$ function over an integer interval. This benchmark relies on GAP to compute $\varphi$, and each HECToR node is populated with 31 GAP instances, coordinated by one HdpH instance. Distributed coordination is performed by the `parMap2Level` and `parMap2LevelRelaxed` skeletons, and the topology is discrete, i. e. the distance between HdpH instances is always 1.

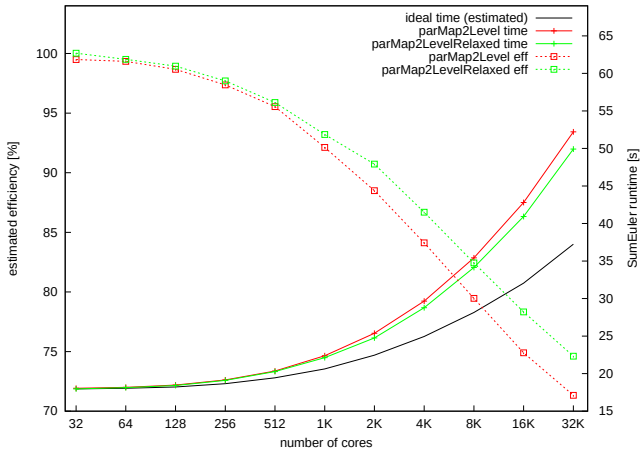



**Figure 11.** SumEuler — weak scaling up to 32K cores.

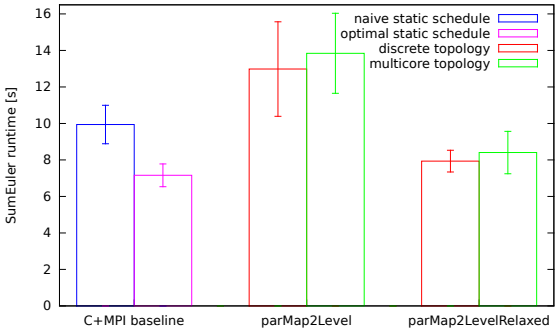



**Figure 12.** SumEuler — impact of topology on 256 cores.

Figure 11 shows weak scaling and efficiency results. The input interval starts at 6.25 million integers on one HECToR node (32 cores), and doubles when doubling the number of cores up to 6.4 billion integers on 1024 nodes (32K cores). Doubling the size of the input interval more than doubles the amount of work as computing $\varphi$ is more expensive on larger numbers, so we estimate a runtime curve for perfect scaling (by sampling and interpolating the runtimes of small tasks). The runtime graphs in Figure 11 show that the two skeletons do not scale perfectly. However, even on 32K cores their runtimes are still within a factor of 1.5 of the ideal.

Efficiency (i. e. speedup divided by number of cores) is estimated by relating the observed runtimes to the (estimated) perfect scaling time. The graphs show that efficiency is steadily declining, yet remains above 70% even on 32K cores. These graphs also show that `parMap2LevelRelaxed` offers a small efficiency advantage over `parMap2Level`.

### 8.2 Topology Awareness

The impact of different topologies and of different modes of task placement on HdpH performance are also investigated with the SumEuler benchmark. Yet, here $\varphi$ is *computed naively in HdpH* rather than relying on GAP. Coordination is again performed by the skeletons `parMap2Level` and `parMap2LevelRelaxed`, both with radius 1. The experiments are performed on the 256 core Beowulf with either the discrete topology, or the standard multicore topology (i. e. distance between cores sharing memory is $\frac{1}{2}$).

Figure 12 shows runtimes, averaged over 11 runs, and 95% confidence intervals. Sequential runtime is $1115 \pm 20$ seconds.

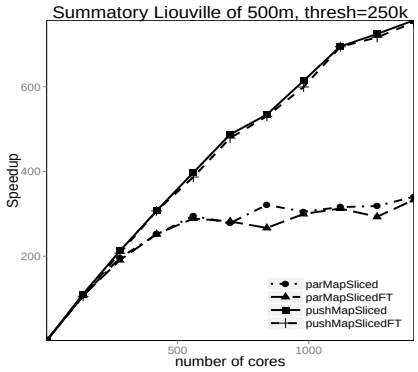



**Figure 13.** Summatory Liouville speedup on HECToR.

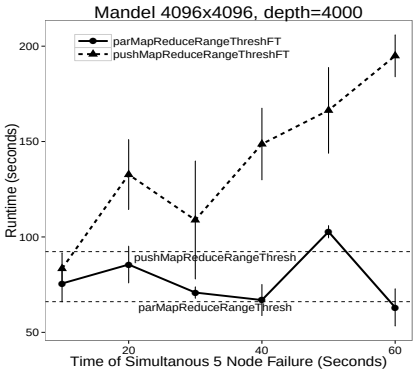



**Figure 14.** Simultaneous node failures (Mandelbrot on Beowulf).

The skeletons divide the input into 1024 tasks, so average task granularity is about 1 second, but varies by 3 orders of magnitude, between 2 seconds and a few milliseconds. We observe that, as in the weak scaling experiment, `parMap2LevelRelaxed` performs best, with speedups of 130 to 140, whereas `parMap2Level` only achieves speedups of 80 to 90. Remarkably, the topology does not matter; the multicore topology appears to perform slightly worse but the overheads stay well within the error margin. We conclude that a 256-core cluster is too small to suffer from locality issues.

Figure 12 also compares the performance of HdpH to a baseline SumEuler benchmark, *implemented natively in C+MPI*. Sequential runtime of the C code is $956 \pm 1$ seconds, about 15% faster than Haskell. A naive static MPI task placement achieves speedups of about 95; the optimal static schedule (found by experiment) yields speedups of about 130. Ultimately, C+MPI with optimal static scheduling is about 10 to 15% faster than HdpH with `parMap2LevelRelaxed`, matching the sequential performance gap. This shows that (1) HdpH introduces minimal overheads, and (2) HdpH work stealing can compete with optimal static scheduling for this benchmark.

### 8.3 Fault Tolerance

A total of five benchmarks are used to measure scalability, supervision overheads, and recovery overheads of HdpH-RS in the thesis [30].

***Scaling and supervision overheads.*** The speedup of the Summatory Liouville program outlined in Section 3.1 is measured on HECToR up to 1400 cores using $[20, 40..200]$ nodes with $n{=}500m$ and a threshold of $250k$. This generates 2000 tasks so that all PEs may be saturated with at least one task up to 1400 cores with ideal scheduling.

Figure 13 compares the performance of the slicing parallel map skeletons `parMapSliced` and `pushMapSliced` (Section 7), both with reliable scheduling enabled (indicated by suffix `FT`) and disabled. Beyond 280 cores, the eager skeletons outperform the lazy ones, reaching peak speedups of around 750 versus 340. More importantly, however, we observe that the `FT` graphs stay close to the graphs of their unreliable cousins, that is the overhead of reliable scheduling is negligible.

***Fault recovery costs.*** The HdpH-RS scheduler is designed to survive both single and simultaneous node failures. The cost of recovering from such failures is assessed with the well-known Mandelbrot benchmark. The coordination is performed by the divide-and-conquer skeletons `par/pushMapReduceRangeThresh` (Section 7), generating 1023 tasks.

For each skeleton the mean duration of five fault-free runs, 66 and 92 seconds respectively, is plotted as a horizontal base line in Figure 14. To assess recovery time, 5 nodes are killed simultaneously after $10, 20, \ldots, 60$ seconds. Each recovery experiment is performed 5 times, and Figure 14 plots the average recovery runtimes (including standard error bars). The recovery overheads for `parMapReduceRangeThresh` are consistently low, and variability limited. Recovery overheads for `pushMapReduceRangeThresh` increase over time, and variability is generally higher. These results highlight a preference towards lazy on-demand scheduling to minimise recovery time.

### 8.4 Representation Theory Case Study

This section briefly reports the performance of HdpH coordinating GAP on a case study [24] from the representation theory of Hecke algebras [11]. Given generators $M_1, \ldots, M_m$, square matrices of polynomials in $\mathbb{Z}[x, x^{-1}]$, the problem is to find a (non-trivial) symmetric matrix $Q$ over $\mathbb{Z}[x, x^{-1}]$ such that the product of $Q$ with each generator is itself symmetric. Depending on the Hecke type $\mathrm{E}_m$ ($m$ = 6, 7, 8), the dimension of the generators and the degrees of the polynomials in $Q$ may vary considerably.

We parallelise the three most time-consuming phases of the algorithm for finding $Q$: (1) solving of homomorphic images over finite fields, (2) solving of interpolation problems over rationals, and (3) final product symmetry check over polynomial matrices. All algebraic computations are done by sequential GAP instances and coordinated by HdpH, as in Section 8.1. Some illustrative results are as follows. For medium-size $\mathrm{E}_7$ representations (23 to 38) we obtain relative speedups of between 40 and 55 using 106 GAP instances on 16 Beowulf nodes (128 cores). For small $\mathrm{E}_8$ representations (11 to 15) we obtain relative speedups of between 116 and 548 using 992 GAP instances on 32 HECToR nodes (1024 cores).

## 9. Discussion

Large commodity manycore architectures will have high failure rates and a non-uniform communication topology between cores. We have outlined the design of a pair of shallowly embedded Haskell DSLs, HdpH and HdpH-RS, to address these challenges for computations with irregular parallelism (Section 3). We have presented operational semantics for both DSLs and established conditions for semantic equivalence (Section 4). We have briefly sketched validation of the sophisticated work stealing protocol of HdpH-RS by model checking and testing (Section 5). We have described the DSL implementations, focusing on how the work stealing schedulers achieve topology awareness and fault tolerance (Section 6). We have provided examples of algorithmic skeletons,

including skeletons for sophisticated topology aware work distribution (Section 7). An initial evaluation using 5 benchmarks on a Beowulf cluster and the HECToR HPC platform shows good weak scaling of HdpH up to 32K cores, and that HdpH-RS has low overheads both in the presence and absence of faults. In a computational algebra case study we obtain speedups of up to 548 coordinating 992 GAP instances on 1024 cores (Section 8).

Although developed for symbolic computation the HdpH DSLs are general purpose, being designed to manage dynamic and irregular task parallelism on large scale hierarchical architectures. They cope well with complex algorithms, coordination patterns, and data structures, but typical numeric HPC workloads are not well suited. The HdpH programming model works particularly well where tasks are stateless. For good performance, task execution time should greatly outweigh communication time, which is largely determined by the size of the closures transmitted, hence Big Data workloads with large memory footprints are also not suitable. As HdpH-RS retains backups of supervised closures, its performance is additionally predicated on a small retained closure footprint. That is, either the number of supervised closures is small, or the closures are small in size (on average). Thus HdpH-RS offers a trade-off between fault tolerance and memory use.

Currently, HdpH and HdpH-RS provide orthogonal features. An immediate engineering task is to amalgamate topology awareness and fault tolerance into a single DSL. While HdpH was designed for architectures with $10^5$ cores, we only have made systematic measurements up to 32K cores for pragmatic reasons: access to all 90K cores of HECToR is simply too expensive. As COTS and HPC platforms grow, we expect that larger architectures will eventually become more affordable, which would help us continue to use HdpH for solving open problems in algebraic representation theory.

## 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 Lilia Georgieva, Sam Lindley, Daria Livesey, Greg Michaelson, Jeremy Singer and the anonymous referees for helpful feedback.

## References

[1] J. Allen. *Effective Akka*. O'Reilly, 2013.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer*. Morgan & Claypool, 2nd edition, 2013.

[4] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference, Anaheim, CA, USA*, 1997.

[5] P. B. Borwein, R. Ferguson, and M. J. Mossinghoff. Sign changes in sums of the Liouville function. *Mathematics of Computation*, 77(263): 1681–1694, 2008.

[6] F. Cappello. Fault tolerance in petascale/exascale systems. *Int. Journal HPC Applications*, 23(3):212–226, 2009.

[7] M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *DAMP 2007, Nice, France*, pages 10–18. ACM, 2007.

[8] J. Epstein, A. P. Black, and S. L. Peyton-Jones. Towards Haskell in the cloud. In *Haskell 2011, Tokyo, Japan*, pages 118–129. ACM, 2011.

[9] A. Foltzer *et al*. A meta-scheduler for the Par-monad: composable scheduling for the heterogeneous cloud. In *ICFP 2012, Copenhagen, Denmark*, pages 235–246. ACM, 2012.

[10] GAP Group. GAP – groups, algorithms, and programming, 2007. `http://www.gap-system.org`.

[11] M. Geck and J. Müller. James' conjecture for Hecke algebras of exceptional type, I. *J. Algebra*, 321(11):3274–3298, 2009.

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

[13] T. Hoff. Netflix: Continually test by failing servers with Chaos Monkey. `http://highscalability.com`, December 2010.

[14] V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform Grids. In *CCGrid 2010, Melbourne, Australia*, pages 123–134. IEEE, 2010.

[15] V. Kravtsov, P. Bar, D. Carmeli, A. Schuster, and M. T. Swain. A scheduling framework for large-scale, parallel, and topology-aware applications. *J. Parallel Distrib. Comput.*, 70(9):983–992, 2010.

[16] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars and handlers. In *POPL 2014, San Diego, USA*. ACM, 2014.

[17] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *HPDC'12, Delft, The Netherlands*, pages 137–148. ACM, 2012.

[18] S. Linton *et al*. Easy composition of symbolic computation software using SCSCP. *J. Symb. Comput.*, 49:95–119, 2013.

[19] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning, 2010.

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

[21] W. Ma and S. Krishnamoorthy. Data-driven fault tolerance for work stealing computations. In *ICS 2012, Venice, Italy*, pages 79–90. ACM, 2012.

[22] P. Maier and R. Stewart. HdpH source code, 2014. `https://github.com/PatrickMaier/HdpH`.

[23] P. Maier and P. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL 2011, Lawrence, KS, USA, Revised Selected Papers*, LNCS 7257, pages 35–50. Springer, 2012.

[24] P. Maier, D. Livesey, H.-W. Loidl, and P. Trinder. High-performance computer algebra: A Hecke algebra case study. In *Euro-Par 2014, Porto, Portugal*. Springer, 2014. To appear.

[25] P. Maier, R. Stewart, and P. W. Trinder. Reliable scalable symbolic computation: The design of SymGridPar2. *Computer Languages, Systems & Structures*, 40(1):19–35, 2014.

[26] S. Marlow, S. L. Peyton-Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP 2009, Edinburgh, Scotland*, pages 65–78. ACM, 2009.

[27] S. Marlow, R. Newton, and S. L. Peyton-Jones. A monad for deterministic parallelism. In *Haskell 2011, Tokyo, Japan*, pages 71–82. ACM, 2011.

[28] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *PGAS 2011, Galveston Island, TX, USA*, 2011.

[29] S. L. Peyton-Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL 1996, St. Petersburg Beach, USA*, pages 295–308, 1996.

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

[31] R. Stewart. Promela abstraction of HdpH-RS reliable scheduler extension, 2013. `https://raw.github.com/robstewart57/phd-thesis/master/spin_model/hdph_scheduler.pml`.

[32] P. W. Trinder *et al*. GUM: A portable parallel implementation of Haskell. In *PLDI 1996, Philadelphia, USA*, pages 79–88. ACM, 1996.

[33] P. W. Trinder *et al*. Algorithms + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[34] T. White. *Hadoop – The Definitive Guide: MapReduce for the Cloud*. O'Reilly, 2009.

[35] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, and H. E. Bal. A simple and efficient fault tolerance mechanism for divide-and-conquer systems. In *CCGrid 2004, Chicago, USA*, pages 735–734. IEEE, 2004.