Supervised Workpools for Reliable Massively Parallel Computing

Robert Stewart, Phil Trinder, and Patrick Maier

Heriot Watt University
Mathematical and Computer Sciences
Edinburgh, UK
{R.Stewart,P.W.Trinder,P.Maier}@hw.ac.uk

Abstract. The manycore revolution is steadily increasing the performance and size of massively parallel systems, to the point where system reliability becomes a pressing concern. Therefore, massively parallel compute jobs must be able to tolerate failures. For example, in the HPC-GAP project we aim to coordinate symbolic computations in architectures with 10^6 cores. At that scale, failures are a real issue. Functional languages are well known for advantages both for parallelism and for reliability, e.g. stateless computations can be scheduled and replicated freely.

This paper presents a software level reliability mechanism, namely supervised fault tolerant workpools implemented in a Haskell DSL for parallel programming on distributed memory architectures. The workpool hides task scheduling, failure detection and task replication from the programmer. To the best of our knowledge, this is a novel construct. We demonstrate how to abstract over supervised workpools by providing fault tolerant instances of existing algorithmic skeletons. We evaluate the runtime performance of these skeletons both in the presence and absence of faults, and report low supervision overheads.

1 Introduction

Changes in chip manufacturing technology is leading to architectures where the number of cores grow exponentially, following Moore's law. Many predict the proliferation of massively parallel systems currently exemplified by the large commodity off-the-shelf (COTS) clusters used in commercial data centres, or the high performance computing (HPC) platforms used in scientific computing. For example, over the last 4 years, the performance (measured in FLOPS) of the world's fastest supercomputer has risen 16-fold, according to TOP500[1]. This has been accompanied by a 13-fold increase in the total number of cores, and by power consumption more than tripling (from 2.3 to 7.9 MW). The latter trend in particular points to an ever increasing size of these systems, not only in terms of total number of cores, but also in the number of networked components (i.e. compute nodes and network switches).

[1] http://www.top500.org

Even with failure rates of individual components appearing negligible, the exponential growth in the number of components makes system reliability a growing concern [21]. Thus, massively parallel compute jobs either must complete quickly, or be able to tolerate component failures.

Depending on the problem domain, there may be several ways to react to failures. For instance, stochastic simulations can trade precision for reliability by simply discarding computations on failed nodes. Similarly, grid-based continuous simulations can recover information lost due to a node failure by "smoothing" information from its neighbours. However, there are problem domains, e.g. optimisation or symbolic computation, where trading precision is impossible because solutions are necessarily exact. In these domains, fault tolerance is more costly, as it can only be achieved by replicating the computations of failed nodes, which incurs overheads and requires book keeping. Nonetheless, replication-based fault tolerance is widely used, e.g. in MapReduce frameworks like Hadoop [2].

Functional languages have long been advocated as particularly suitable for parallel programming because they encourage a stateless coding style which simplifies the scheduling of parallelism. Furthermore statelessness also simplifies task replication, making functional languages even more attractive for massively parallel programming, where replication-based fault tolerance is indispensable.

This paper presents the design and implementation of a fault tolerant workpool in a functional language. A *workpool* is a well-known parallel programming construct that guarantees the parallel execution of independent *tasks*, relieving the programmer of concerns about task scheduling (and sometimes load balancing). A *fault tolerant* workpool additionally guarantees completion of all tasks (under some proviso), thus relieving the programmer of concerns about detecting node failures, replicating tasks, and the associated book keeping. We note that our workpool is able to recover from the failure of any number of nodes bar a single distinguished one, the node hosting the supervised workpool. Therefore, the workpool is not *high-availability*, setting it apart from high-availability behaviours in Erlang. However, our workpool is able to guarantee that system reliability matches the reliability of a single node, which is good enough for most massively parallel applications.

The workpool is implemented on top of *HdpH* [17], a Haskell domain specific language (DSL) for distributed-memory parallel programming. HdpH and the workpool are being developed within the HPC-GAP project[2]. The project aims to solve large computer algebra problems on massively parallel platforms like HECToR, the UK national supercomputing service with currently 90,000 cores, by coupling the *GAP* computer algebra system [11] with the *SymGridParII* coordination middleware [16].

We start by surveying fault tolerant approaches, languages and frameworks (Section 2), before making the following contributions:

1. We present the design and implementation of a novel fault-tolerant workpool in Haskell (sections 3.1 and 3.3), hiding task scheduling, failure detection and

[2] http://www-circa.mcs.st-andrews.ac.uk/hpcgap.php

task replication from the programmer. Moreover, workpools can be nested to form fault-tolerant hierarchies, which is essential for scaling up to massively parallel platforms like HECToR.

2. The implementation of high-level fault tolerant abstractions on top of the workpool: generic fault tolerant skeletons for task parallelism and nested parallelism, respectively (section 4).
3. We evaluate the fault tolerant skeletons on two benchmarks. These benchmarks demonstrate fault tolerance: computations do complete even in the presence of node failures. We also measure the overheads of the fault tolerant skeletons, both in terms of the cost of book keeping, and in terms of the time to recover from failure (section 5).

2 Related Work

Most existing fault tolerant approaches in distributed architectures follow a rollback-recovery approach, and new opportunities are being explored as alternative and more scalable possibilities. This section outlines non-language and language based approaches to fault tolerance.

2.1 Non Language-Based Approaches to Fault Tolerance

At the highest level, algorithmic methods have been proposed, with the injection of fault oblivious algorithms and self stabilising algorithms [5]. Various techniques have been used at the application level, such as reflective object-oriented programming [9].

The *Message Passing Interface* [13] is a predominant and efficient communication layer in HPC platforms. Thorough comparisons of fault tolerant MPI approaches and implementations have been made [12]. These include checkpointing the state of computation [4], extending or modifying the semantics of the MPI standard [10], and runtime resilience to overcome node failure [7], though the onus is on the user to handle faults programmatically.

On COTS platforms, computational frameworks such as MapReduce realise fault tolerance through replication, as implementations such as Hadoop [2] have shown. They are optimised for high throughput, but limit the programmer to one parallel pattern. In contrast, our supervised workpool is a fault tolerant construct that can be used for multiple parallel patterns, as described in Section 4.

2.2 Fault Tolerance in Erlang

Erlang [1] is a dynamically typed functional language designed for programming concurrent, real-time, distributed fault tolerant programs. Erlang provides a process-based model of concurrency with asynchronous message passing. Erlang processes do not shared memory, and all interaction is done through message passing.

Erlang has three mechanisms that provide fault tolerance in the face of failures [14]: monitoring the evaluation of expressions; monitoring the behaviour of other processes; and trapping evaluation errors of undefined functions. Additionally, Erlang provides primitives for creating and deleting links between processes. Process linking is symmetrical, and exit signals can be propagated up hierarchical supervision trees.

On top of these primitives, Erlang provides the Open Telecom Platform (OTP) which separates the Erlang framework from the application, and includes a set of fault tolerant *behaviours*. The Erlang/OTP *generic supervisor* behaviour provides fault tolerance for Erlang programs executing on multiple compute nodes. Processes residing in the Erlang VM inherit either the role of *supervisor*, or alternatively, a *child*. Supervisors receive exit signals from children, taking the appropriate action. They are responsible for starting, stopping, and monitoring child processes, and keep child processes alive by restarting them if necessary. Supervisors can be nested, creating hierarchical supervision trees.

In contrast to Erlang behaviours, which are designed for distributed computing, our supervised workpool is designed for distributed-memory *parallel* programming. Additionally, statically typed polymorphic skeletons can be constructed on top of the workpool.

2.3 Fault Tolerance in Distributed Haskell Extensions

Cloud Haskell. Cloud Haskell [8] is a domain specific language for distributed-memory computing platforms. It is implemented as a shallow embedding in Haskell, and provides a message communication model that is inspired by Erlang. It emulates the Erlang approaches (Section 2.2) of isolated process memory and explicit message passing, and provides process linking. Cloud Haskell inherits the language features of Haskell, including purity, types, and monads, as well as the multi-paradigm concurrency models in Haskell. A significant contribution of Cloud Haskell is a mechanism for serialising function closures, enabling higher order functions to be used in distributed computing environments. As Cloud Haskell tightly emulates Erlang, it is once again more designed for distributed rather than parallel computing.

HdpH. Haskell distributed parallel Haskell (HdpH) [17] is a distributed-memory parallel DSL for Haskell that supports high-level semi-explicit parallelism, and is designed for fault tolerance. HdpH is an amalgamation of two recent contributions to the Haskell community. Its closure serialisation and transmission over networks is inspired by Cloud Haskell (Section 2.3), and it uses the `Par` Monad [19], as a shallowly embedded DSL for parallelism. The write-once semantics of the original `Par` Monad are relaxed in HdpH slightly to support fault tolerance: we ignore successive writes rather than failing, which is described in Section 3.3.

HdpH extends the `Par` Monad for *distributed-memory* parallelism, rather than distributed systems as in Erlang or Cloud Haskell. Parallelism in the `Par` Monad is achieved with a `fork` primitive, and an `IVar` is a communication

abstraction to communicate results of parallel tasks (referred to in other languages, as futures [20] or promises [15]). HdpH extends CloudHaskell's closure representation, by supporting polymorphic closure transformations in order to implement high-level coordination abstractions. This extension is crucial for the implementation of generic fault tolerant skeletons as described in Section 4.

3 The Design & Implementation of a Supervised Workpool

3.1 Design

A workpool is an abstract control structure that takes units of work as input, returning values as output. The scheduling of the work units is coordinated by the workpool implementation. Workpools are a very common pattern, and are often used for performance, rather than fault tolerance. For example workpools can be used for limiting concurrent connections to resources, to manage heavy load on compute nodes, and to schedule critical application tasks in favour of non-critical monitoring tasks [26]. The supervised workpool presented in this paper extends the workpool pattern by adding fault tolerance to support the fault tolerant execution of HdpH applications. The fault tolerant design is inspired by the supervision behaviour and node monitoring aspects of Erlang (Section 2.2), and combines this with Haskell's polymorphic, static typing.

Most workpools schedule work units dynamically, e.g. an idle worker selects a task from the pool and executes it. For simplicity our current HdpH workpool uses static scheduling: each worker is given a fixed set of work units. The supervised workpool performs well for applications exhibiting regular parallelism, and also for limited irregular parallel programs, as shown in Section 5. A fault tolerant work stealing scheduler is left to future work (Section 6).

The Haskell implementation is made possible by the loosely coupled design in HdpH. Before describing the workpool in detail, we introduce terminology for HdpH and the workpool in Table 1.

IVar	A write-once mutable mutable reference.
GIVar	A global reference to an IVar, which is used to remotely write values to the IVar.
Task	Consists of an *expression* and a *GIVar*. The expression is evaluated, and its value is written to the associated GIVar.
Completed task	When the associated GIVar in a *task* contains the value of the task expression.
Closure	A serializable expression or value. Tasks and values are serialized as closures, allow them to be shipped to other nodes.
Supervisor thread	The Haskell thread that has initialized the workpool.
Process	An OS process executing the GHC runtime system.
Supervising process	The *process* hosting the *supervisor thread*.
Supervising node	The node hosting the supervising process.
Worker node	Every node that has been statically assigned a task from a given workpool.

Table 1: HdpH and workpool terminology

A fundamental principle in the HdpH supervised workpool is that there is a one-to-one correspondence between a *task* and an `IVar` — each task evaluates an expression to return a result which is written to its associated `IVar`. The *tasks* are distributed as closures to *worker nodes*. The *supervisor thread* is responsible for creating and globalising `IVar`s, in addition to creating the associated tasks and distributing them as closures. Here are the workpool types and the function for using it:

```
type SupervisedTasks a  = [(Closure (IO ()), IVar a)]
supervisedWorkpoolEval :: SupervisedTasks a -> [NodeId] -> IO [a]
```

The `supervisedWorkpoolEval` function takes as input a list of tuples, pairing tasks with their associated `IVar`s, and a list of `NodeId`s. The closured tasks are distributed to worker nodes in a round robin fashion to the specified worker `NodeId`s, and the workpool waits until all tasks are complete i.e. all `IVar`s are full. If a node failure is identified before tasks complete, the unevaluated tasks sent to the failed node are reallocated to the remaining available nodes. Detailed descriptions of the scheduling, node failure detection, and failure recovery is in Section 3.3.

The supervised workpool guarantees that given a list of *tasks*, it will fully evaluate their result provided that:

1. The *supervising node* is alive throughout the evaluation of all tasks in the workpool.
2. All expressions are computable. For example, evaluating an expression should not throw uncaught exceptions, such as a division by 0; all programming exceptions such as non-exhaustive case statements must be handled within the expression; and so on.

Our supervised workpool is non-deterministic, and hence is monadic. This is useful in some cases such as racing the evaluation of the same task on separate nodes, and also for fault tolerance — the write semantics of `IVar`s are described in Section 3.3. To recover determinism in the supervised workpool, expressions must be *idempotent*. An idempotent expression may be executed more than once which entails the same side effect as executing only once. E.g inserting a given key/value pair to a mutable map - consecutive inserts have no effect. Pure computations, because of their lack of side effects, are of course idempotent.

Workpools are functions and may be freely nested and composed. There is no restriction to the number of workpools hosted on a node, and Section 5.2 will present a divide-and-conquer abstraction that uses this flexibility.

3.2 Use Case Scenario

Figure 1 shows a workpool scenario where six closures are created, along with six associated `IVar`s. The closures are allocated to three worker nodes: `Node2`, `Node3` and `Node4` from the supervising node, `Node1`. Whilst these closures are

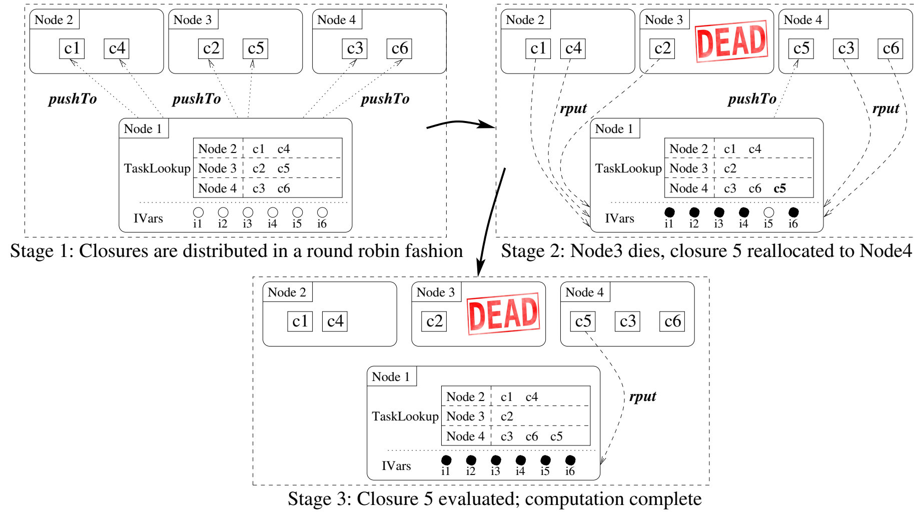


Fig. 1: Reallocating closures scenario

```
-- |HdpH primitives
type IVar a = TMVar a                            -- type synonym for IVar
data GIVar a                                     -- global handles to IVars
data Closure a                                   -- explicit, serialisable closures
pushTo :: Closure (IO ()) -> NodeId -> IO ()     -- explicit task placement
rput   :: GIVar (Closure a) -> Closure a -> IO () -- write a value to a remote IVar
get    :: IVar a -> IO a                         -- blocking get on an IVar
probe  :: IVar a -> STM Bool                     -- check if IVar is full or empty
```

Fig. 2: Type signatures of HdpH primitives

being evaluated, `Node3` fails, having completed only one of its two tasks. As `IVar i5` had not been filled, closure `c5` is reallocated to `Node4`. No further node failures occur, and once all six `IVar`s are full, the supervised workpool terminates. The mechanisms for detecting node failure, for identifying completed tasks, and the reallocation of closures are described in Section 3.3.

3.3 Implementation

The types of the relevant HdpH primitives are shown in Figure 2. The complete fault tolerant workpool implementation is available [23], and the most important functions are shown in Figure 3. All line numbers refer to this Figure. Two independent phases take place in the workpool:

1. Line 5 shows the `supervisedWorkpoolEval` function which creates the workpool, distributes tasks, and then uses the STM termination check described in item 2. The `distributeTasks` function on line 9 uses `pushTo` (from Figure 2) to ship tasks to the worker nodes and creates `taskLocations`, an instance of `TaskLookup a` (line 3). This is a mutable map from `NodeId`s to `SupervisedTasks a` on line 2, which is used for the book keeping of task

locations. The monitorNodes function on lines 22 - 30 then monitors worker node availability. Should a worker node fail, blockWhileNodeHealthy (line 29) exits, and reallocateIncompleteTasks on line 30 is used to identify incomplete tasks shipped to the failed node, using probe (from Figure 2). These tasks are distributed to the remaining available nodes.

2. Haskell's STM [22] is used as a termination check. For every task, an IVar is created. A TVar is created in the workpool to store a list of values returned to these IVars from each task execution. The getResult function on line 34 runs a *blocking* get on each IVar, which then writes this value as an element to the list in the TVar. The waitForResults function on line 42 is used to keep phase 1 of the supervised workpool active until the length of the list in the TVar equals the number of tasks added to the workpool.

```
1   -- |Workpool types
2   type SupervisedTasks a  = [(Closure (IO ()), IVar a)]
3   type TaskLookup a       = MVar (Map NodeId (SupervisedTasks a))
4
5   supervisedWorkpoolEval :: SupervisedTasks a -> [NodeId] -> IO [a]
6   supervisedWorkpoolEval tasks nodes = do
7           -- PHASE 1
8           -- Ship the work, and create an instance of 'TaskLookup a'.
9       taskLocations <- distributeTasks tasks nodes
10          -- Monitor utilized nodes; reallocate incomplete tasks when worker nodes fail
11      monitorNodes taskLocations
12
13          -- PHASE 2
14          -- Use STM as a termination check. Until all tasks are evaluated, phase 1 remains active.
15      fullIvars <- newTVarIO []
16      mapM_ (forkIO . atomically . getResult fullIvars . snd) tasks
17      results <- atomically $ waitForResults fullIvars (length tasks)
18
19          -- Finally, return the results of the tasks
20      return results
21
22  monitorNodes :: TaskLookup a -> IO ()
23  monitorNodes taskLookup = do
24      nodes <- fmap Map.keys $ readMVar taskLookup
25      mapM_ (forkIO . monitorNode) nodes
26      where
27       monitorNode :: NodeId -> IO ()
28       monitorNode node = do
29         blockWhileNodeHealthy node -- Blocks while node is healthy (Used in Figure 4)
30         reallocateIncompleteTasks node taskLookup -- reallocate incomplete tasks shipped to 'node'
31
32  -- |Takes an IVar, runs a blocking 'get' call, and writes
33  --  the value to the list of values in a TVar
34  getResult :: TVar [a] -> IVar a -> STM ()
35  getResult values ivar = do
36      v  <- get ivar
37      vs <- readTVar values
38      writeTVar results (vs ++ [v])
39
40  -- |After each write to the TVar in 'evalTask', the length of the list
41  --  is checked. If it matches the number of tasks, STM releases the block.
42  waitForResults :: TVar [a] -> Int -> STM [a]
43  waitForResults values i = do
44      vs <- readTVar values
45      if length vs == i then return vs else retry
```

Fig. 3: Workpool implementation & Use of STM as a termination check

The restriction to idempotent tasks in the workpool (Section 3.1) enables the workpool to freely duplicate and re-distribute tasks. Idempotence is permitted by the *write* semantics of `IVar`s. The first write to an `IVar` succeeds, and subsequent writes are ignored — successive `rput` attempts to the same `IVar` are non-fatal. To support this, the write-once semantics of `IVar`s in the `Par` Monad [19] are relaxed slightly in HdpH, to support fault tolerance. This enables identical closures to be raced on separate nodes. Should one of the nodes fail, the other evaluates the closure and `rput`s the value to the associated `IVar`. Should the node failure be intermittent, and a successive `rput` be attempted, it is silently ignored. It also enables replication of closures residing on overloaded nodes to be raced on healthy nodes.

It would be unnecessary and costly to reallocate tasks if they had been fully evaluated prior to the failure of the worker node it was assigned to. For this purpose, a `probe` primitive (Figure 2) is used to identify which `IVar`s are full, indicating the evaluation status of its associated task. As such, all `IVar`s that correspond to tasks allocated to the failed worker node are *probed*. Only tasks associated with empty `IVar`s are reallocated as closures.

Node Failure Detection A new transport layer for distributed Haskells [6] underlies the fault tolerant workpool. The main advantage of adopting this library is the typed error messages at the Haskell language level.

```
1  -- connection attempt
2  attempt <- connect myEndPoint remoteEndPointAddress <default args>
3  case attempt of
4    (Left (TransportError ConnectFailed)) -> -- unblocks 'blockWhileNodeHealthy', Figure 3 line 29
5    (Right connection) ->                    -- carry on
```

Fig. 4: Detecting node failure in the `blockWhileNodeHealthy` function

The `connect` function from the transport layer is shown in Figure 4. It is used by the workpool to detect node failure in the `blockWhileNodeHealthy` function on line 29 of Figure 3. Each node creates an endpoint, and endpoints are connected to send and receive messages between the nodes. Node availability is determined by the outcome of connection attempts using `connect` between the node hosting the supervised workpool, and each worker node utilized by that workpool. The transport layer ensures lightweight communications by reusing the underlying TCP connection. One logical connection attempt between the supervising node and worker nodes is made each second. If `Right Connection` is returned, then the worker node is healthy and no action is taken. However, if `Left (TransportError ConnectFailed)` is returned then the worker node is deemed to have failed, and `reallocateIncompleteTasks` (Figure 3, line 30) re-distributes incomplete tasks originally shipped to this node. Concurrency for monitoring node availability is achieved by Haskell IO threads on line 25 in Figure 3.

An alternative to this design for node failure detection was considered - with periodic *heartbeat* messages sent from the worker nodes *to* the process hosting the supervised workpool. However the bottleneck of message delivery would be

the same i.e. involving the endpoint of the process hosting the workpool. Moreover, there are dangers with timeout values for *expecting* heartbeat messages in *asynchronous* messaging systems such as the one employed by HdpH. Remote nodes may be wrongly judged to have failed e.g. when the message queue on the workpool process is flooded, and heartbeat messages are not popped from the message queue within the timeout period. Our design avoids this danger by synchronously checking each connection.

It is necessary that each workpool hosted on a node monitors the availability of worker nodes. With nested or composed supervised workpools there is a risk that the network will be saturated with `connect` requests to monitor node availability. Our architecture avoids this by creating just one Haskell thread per node that monitors availability of all other nodes, irrespective of the number of workpools hosted on a node. Each supervisor thread communicates with these monitoring threads to identify node failure. See the complete implementation [23] for details.

4 High Level Fault Tolerant Abstractions

The HdpH paper [17] describes the implementation of algorithmic skeletons in HdpH. This present work extends these by adding resilience to the execution of two generic parallel algorithmic skeletons.

HdpH provides high level coordination abstractions: evaluation strategies and algorithmic skeletons. The advantages of these skeletons are that they provide a higher level of abstraction [25] that capture common parallel patterns, and that the HdpH primitives for work distribution and operations on `IVar`s are hidden away from the programmer.

We show here how to use fault tolerant workpools to add resilience to algorithmic skeletons. Figure 5 shows the type signatures of fault tolerant versions of the following two generic algorithmic skeletons.

pushMap is a parallel skeleton to provide a parallel map operation, applying a function closure to the input list.

pushDivideAndConquer is another parallel skeleton that allows a problem to be decomposed into sub-problems until they are sufficiently small, and then reassembled with a combining function.

`IVar`s are globalised and closures are created from tasks in the skeleton code, and `supervisedWorkpoolEval` is used at a lower level to distribute closures, and to provide the guarantees described in Section 3.3. The tasks in the workpool are eagerly scheduled into the threadpool of remote nodes.

The two algorithmic skeletons have different scheduling strategies — `pushMap` schedules tasks in a round-robin fashion; `pushDivideAndConquer` schedules tasks randomly (but statically at the beginning, not on-demand).

```
pushMap
   :: [NodeId]         -- available nodes
   -> Closure (a -> b) -- function closure
   -> [a]              -- input list
   -> IO [b]           -- output list

pushDivideAndConquer
   :: [NodeId]                                        -- available nodes
   -> Closure (Closure a -> Bool)                     -- trivial
   -> Closure (Closure a -> IO (Closure b))           -- simplySolve
   -> Closure (Closure a -> [Closure a])              -- decompose
   -> Closure (Closure a -> [Closure b] -> Closure b) -- combine
   -> Closure a                                       -- problem
   -> IO (Closure b)                                  -- output
```

Fig. 5: Fault tolerant algorithmic parallel skeletons used in Appendix A and B of [24]

5 Evaluation

This section demonstrates the use of the fault tolerant mechanisms, and specifically the two fault tolerant algorithmic skeletons from Section 4. Implementations of two symbolic programs are presented: *Summatory Liouville* which is task parallel; and *Fibonacci* which is a canonical divide-and-conquer problem. The implementations of these can be found in the technical report for this paper [24].

The equivalent non-fault tolerant `pushMap` and `pushDivideAndConquer` skeletons are used for comparing the supervised workpool overheads in the presence and absence of faults, which are described in Section 5.4. These do not make use of the supervised workpool, and therefore do not protect against node failure.

5.1 Data Parallel Benchmark

To demonstrate the `pushMap` data parallel skeleton (Figure 5), Summatory Liouville [3] has been implemented in HdpH, adapted from existing Haskell code [27]. The Liouville function $\lambda(n)$ is the completely multiplicative function defined by $\lambda(p) = -1$ for each prime p. $L(n)$ 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 *scale-up* runtime results measure Summatory Liouville $L(n)$ for $n = [10^8, 2 \cdot 10^8, 3 \cdot 10^8..10^9]$. Each experiment is run on 20 nodes with closures distributed in a round robin fashion, and the chunk size per closure is 10^6. For example, calculating $L(10^8)$ will generate 100 tasks, allocating 5 to each node. On each node, a partial Summatory Liouville value is further divided and evaluated in parallel, utilising multicore support in the Haskell runtime [18].

5.2 Control Parallel Benchmark using Nested Workpools

The `pushDivideAndConquer` skeleton (Figure 5) is demonstrated with the implementation of Fibonacci. This example illustrates the flexibility of the supervised

workpool, which can be nested hierarchically in divide-and-conquer trees. At the point when a closure is deemed too computationally expensive, the problem is decomposed into sub-problems, turned into closures themselves, and pushed to other nodes. In the case of Fibonacci, costly tasks are decomposed into 2 smaller tasks, though the `pushDivideAndConquer` skeleton permits any number of decomposed tasks to be supervised.

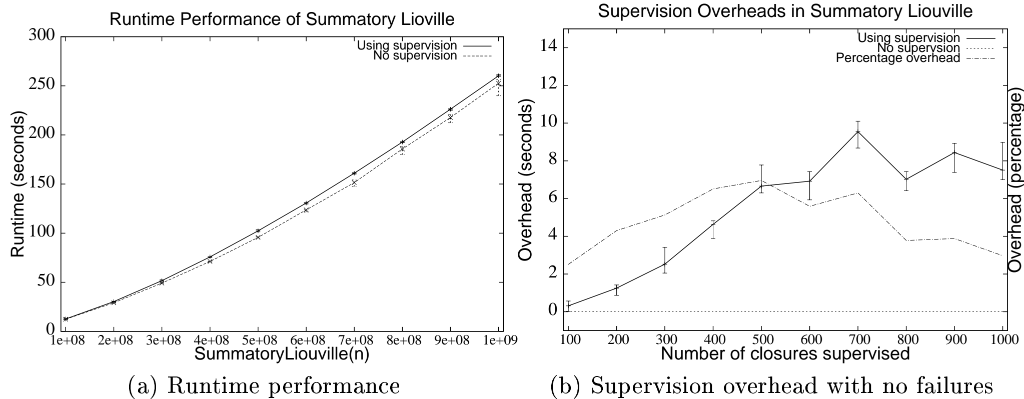
The runtime results measure Fibonacci $Fib(n)$ for $n = [45..55]$, and the sequential threshold for each n is 40. Unlike the `pushMap` skeleton, closures are distributed to random nodes from the set of available nodes to achieve fairer load balancing.

5.3 Benchmark Platform

The two applications were benchmarked on a Beowulf cluster. Each Beowulf node comprises two Intel quad-core CPUs (Xeon E5504) at 2GHz, sharing 12GB of RAM. Nodes are connected via Gigabit Ethernet and run Linux (CentOS 5.7 x86_64). HdpH version 0.3.2 was used and the benchmarks were built with GHC 7.2.1. Benchmarks were run on 20 cluster nodes; to limit variability we used only 6 cores per node. Reported runtime is median wall clock time over 20 executions, and reported error is the range of runtimes.

5.4 Performance

No Failure The runtimes for Summatory Liouville are shown in Figure 6(a). The chunk size is fixed, increasing the number of supervised closures as n is increased in $L(n)$. The overheads of the supervised workpool for Summatory Liouville are shown in Figure 6(b). The runtime for Fibonacci are shown in Figure 7.



(a) Runtime performance (b) Supervision overhead with no failures

Fig. 6: Runtime performance and supervision overheads with no failures for Summatory Lioville 10^8 to 10^9

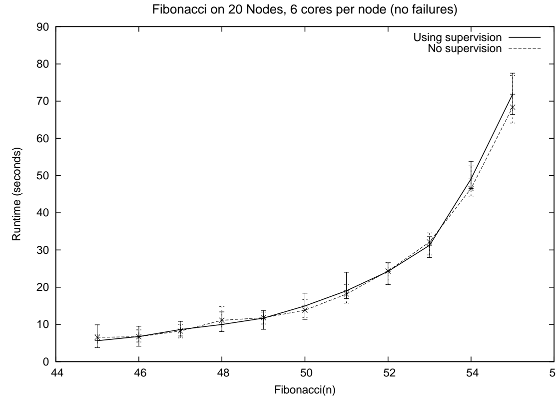


Fig. 7: Runtime performance for Fibonacci with no failures

The supervision overheads for Summatory Liouville range between 2.5% at $L(10^8)$ and 7% at $L(5 \cdot 10^8)$. As the problem size grows to $L(10^9)$, the number of generated closures increases with the chunk size fixed at 10^6. Despite this increase in supervised closures, near constant overheads of between 6.7 and 8.4 seconds are observed between $L(5 \cdot 10^8)$ and $L(10^9)$.

Overheads are not measurable for Fibonacci, as they are lower than system variability (owing probably to random work distribution).

The runtime for calculating $L(5 \cdot 10^8)$ is used to verify the scalability of the HdpH implementation of Summatory Liouville. The median runtime on 20 nodes (each using 6 cores) is 95.69 seconds, and on 1 node using 6 cores is 1711.51 seconds, giving a speed up of 17.9 on 20 nodes.

Recovery From Failure To demonstrate the fault tolerance and to assess the efficacy of the supervised workpool, recovery times have been measured when *one node* dies during the computation of Summatory Liouville. Nested workpools used by `pushDivideAndConquer` also tolerate faults. Due to the size of the divide-and-conquer graph for large problems, they are harder to analyse in any meaningful way.

To measure the recovery time, a number of parameters are fixed. The computation is $L(3 \cdot 10^8)$ with a chunk size of 10^6, which is initially deployed on 10 nodes, with one hosting the supervising task. The `pushMap` skeleton is used to distribute closures in a round robin fashion, so that 30 closures are sent to each node. An expected runtime utilising 10 nodes is calculated from 5 failure-free executions. From this, approximate timings are calculated for injecting node failure. The Linux `kill` command is used to forcibly terminate one running Haskell process prematurely.

The results in Figure 8 show the runtime of the Summatory Liouville calculation when node failure occurs at approximately [10%,20%..90%] of expected execution time. 5 runtimes are observed at each timing point. Figure 8 also reports the average number of closures that are reallocated relative to when node

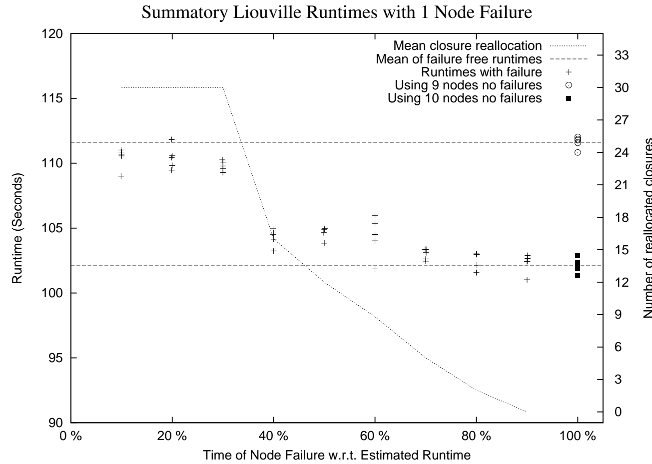


Fig. 8: Recovery time with 1 node failure

failure occurs. As described in Section 3.3, only non-evaluated closures are redistributed. The expectation is that the longer the injected node failure is delayed, the fewer closures will need reallocating elsewhere. Lastly, Figure 8 shows 5 runtimes using 10 nodes when *no* failures occur, and additionally 5 runtimes using 9 nodes, again with no failures.

The data shows that at least for the first 30% of the execution, no tasks are complete on the node, which can be attributed to the time taken to distribute 300 closures and for each node to begin evaluation. Fully evaluated closure values are seen at 40%, where only 16 (of 30) are reallocated. This continues to fall until the 90% mark, when 0 closures are reallocated, indicating that all closures had already been fully evaluated on the responsible node.

The motivation for observing failure-free runtimes using 9 and also 10 nodes is to evaluate the overheads of recovery time when node failure occurs. Figure 8 shows that when a node dies early on (in the first 30% of estimated total runtime), the performance of the remaining 9 nodes is comparable with that of a failure-free run on 9 nodes. Moreover, node failure occurring near the end of a run (e.g. at 90% of estimated runtime) does not impact runtime performance, i.e. is similar to that of a 10 node cluster that experiences no failures at all.

6 Conclusions and Future Work

As COTS and HPC platforms grow in size, faults will become more frequent, rather than failures being exceptional events as in existing architectures. Failures may be caused by hardware malfunction, intermittent network transmission, or software errors. Future dependability of such platforms should therefore rely on a multitude of fault tolerant approaches at all levels of the computational stack.

In this paper, we have presented a language based approach to fault tolerant distributed-memory parallel computation in Haskell: a fault tolerant workpool

that hides task scheduling, failure detection and task replication from the programmer. On top of this, we have developed fault tolerant versions of two algorithmic skeletons. They provide high level abstractions for fault tolerant parallel computation on distributed-memory architectures. To the best of our knowledge the supervised workpool is a novel construct.

The supervised workpool and fault tolerant parallel skeleton implementations exploit recent advances in distributed-memory Haskell implementations, primarily HdpH [17] . The workpool and the skeletons guarantee the completion of tasks even in the presence of multiple node failures, withstanding the failure of all but the supervising node. The work is targeting fault tolerant symbolic computation on 10^6 cores within the HPC-GAP project.

The supervised workpool has acceptable runtime overheads — between 2.5% and 7% using a data parallel skeleton. Moreover when a node fails, the recovery costs are negligible.

Future Work. The full HdpH language affords both explicit and implicit closure placement. In contrast, the current supervised workpool implementation permits only static explicit closure distribution. We are adapting the supervised workpool approach to provide fault tolerance to the work stealing scheduler in HdpH.

Acknowledgements. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.
2. A. Bialecki, C. Taton, and J. Kellerman. Apache Hadoop: a Framework for Running Applications on Large Clusters Built of Commodity Hardware. `http://hadoop.apache.org/`, 2010.
3. 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.
4. A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In *Super Computing*, page 25, 2003.
5. F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir. Toward Exascale Resilience. *High Performance Computing Applications*, 23(4):374–388, 2009.
6. D. Coutts and E. de Vries. The New Cloud Haskell. In *Haskell Implementers Workshop*. Well-Typed, September 2012.
7. M. development. Feature: Adding -disable-auto-cleanup to mpich2, 2010. `http://goo.gl/PNEaO`.

8. J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the Cloud. In *Haskell Symposium*, pages 118–129, 2011.
9. J.-C. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *Symposium on Fault-Tolerant Computing*, pages 489–498, 1995.
10. G. E. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Euro PVM/MPI*, pages 346–353, 2000.
11. The GAP Group. *GAP – Groups, Algorithms, and Programming.* http://www.gap-system.org.
12. W. Gropp and E. Lusk. Fault Tolerance in MPI Programs. *Special issue of the Journal High Performance Computing Applications*, 18:363–372, 2002.
13. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific And Engineering Computation. MIT Press, 1994.
14. J. Larson. Erlang for Concurrent Programming. In *ACM Queue*, volume 6, pages 18–23. ACM, September 2008.
15. B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI*, pages 260–267. ACM, 1988.
16. P. Maier, R. Stewart, and P. Trinder. Reliable Scalable Symbolic Computation: The Design of SymGridPar2. In *SAC 2013*. ACM. To appear.
17. P. Maier and P. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL 2011*. Springer, 2012.
18. S. Marlow, S. L. P. Jones, and S. Singh. Runtime Support for Multicore Haskell. In *ICFP*, pages 65–78, 2009.
19. S. Marlow, R. Newton, and S. L. P. Jones. A Monad for Deterministic Parallelism. In *Haskell Symposium*, pages 71–82, 2011.
20. J. Niehren, J. Schwinghammer, and G. Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, 2006.
21. B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST*. USENIX Association, 2007.
22. N. Shavit and D. Touitou. Software Transactional Memory. In *PODC*, PODC '95, pages 204–213. ACM, 1995.
23. R. Stewart, P. Maier, and P. Trinder. Implementation of the HdpH Supervised Workpool. http://www.macs.hw.ac.uk/~rs46/papers/tfp2012/SupervisedWorkpool.hs, July 2012.
24. R. Stewart, P. Maier, and P. Trinder. Supervised Workpools for Reliable Massively Parallel Computing. Technical report, Heriot-Watt University, 2012. http://www.macs.hw.ac.uk/~rs46/papers/tfp2012/TFP2012_Robert_Stewart.pdf.
25. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
26. F. Trottier-Hebert. Learn You Some Erlang For Great Good - Building an Application With OTP. http://learnyousomeerlang.com/building-applications-with-otp, 2012.
27. A. A. Zain, K. Hammond, J. Berthold, P. W. Trinder, G. Michaelson, and M. Aswad. Low-pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on Multicore Architectures. In *DAMP*, pages 25–36. ACM, 2009.

A Summatory Liouville source code using pushMap

```
farmSumLiouville :: Integer -> Int -> IO Integer
farmSumLiouville x chunkSize = do
  nodes <- allNodes
  sum <$> HdpH.Strategies.FT.pushMap -- fault tolerant
          -- HdpH.Strategies.pushMap -- not fault tolerant
           nodes
           $(mkClosure [|sumLEvalChunk|])
           chunked_list
    where
      chunked_list = zip lowers uppers
      lowers       = [1, toInteger (chunkSize + 1) .. x]
      uppers       = [toInteger chunkSize, toInteger (chunkSize*2) .. x]++[x]

sumLEvalChunk :: (Integer,Integer) -> Integer
sumLEvalChunk (lower,upper) =
     let chunkSize = 1000
         smp_chunks = chunk chunkSize [lower,lower+1..upper] :: [[Integer]]
         tuples      = map (head &&& last) smp_chunks
     in sum $ parMap rpar ( \(lower,upper) -> sumLEvalChunk' lower upper 0) tuples

sumLEvalChunk' :: Integer -> Integer -> Integer -> Integer
sumLEvalChunk' lower upper total
  | lower > upper = total
  | otherwise = let s = toInteger $ liouville lower
                in sumLEvalChunk' (lower+1) upper (total+s)

liouville :: Integer -> Int
liouville n
  | n == 1 = 1
  | length (primeFactors n) `mod` 2 == 0 = 1
  | otherwise = -1

primeFactors :: Integer -> [Integer]
primeFactors = -- omitted
```

B Fibonacci source code using pushDivideAndConquer

```
fib_dnc :: Int -> Int -> IO Integer
fib_dnc seqThreshold n = do
    res <-  skel  (toClosure n)
    return $ unClosure res
      where
       skel nClosured = do
             nodes <- allNodes
             HdpH.Strategies.FT.pushDivideAndConquer -- fault tolerant
             -- HdpH.Strategies.pushDivideAndConquer -- not fault tolerant
              nodes
              $(mkClosure [| dnc_trivial_abs seqThreshold |])
              $(mkClosure [| dnc_simplySolve |])
              $(mkClosure [| dnc_decompose |])
              $(mkClosure [| dnc_combineSolutions |])
              nClosured

dnc_trivial_abs :: Int -> Closure Int -> Bool
dnc_trivial_abs (seqThreshold) =
  \ clo_n -> unClosure clo_n <= max 1 seqThreshold

dnc_simplySolve =
  \ clo_n -> return $ toClosure (force $ fib $ unClosure clo_n)

dnc_decompose =
  \ clo_n -> let n = unClosure clo_n in [toClosure (n-1), toClosure (n-2)]

dnc_combineSolutions =
  \ _ [clo_x, clo_y] -> toClosure (unClosure clo_x + unClosure clo_y)

fib :: Int -> Integer
fib n | n <= 1    = 1
      | otherwise = fib (n-1) + fib (n-2)
```