

Implementing a High-level Distributed-Memory Parallel Haskell in Haskell

Patrick Maier and Phil Trinder

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK

Abstract. We present the initial design, implementation and preliminary evaluation of a new distributed memory parallel Haskell, HdpH. The language is a shallowly embedded parallel extension of Haskell that supports high-level semi-explicit parallelism, is scalable, and has the potential for fault tolerance. The HdpH implementation is designed for maintainability without compromising performance too severely. To provide maintainability the implementation is modular and layered and, crucially, coded in vanilla Concurrent Haskell. Initial performance results are promising for three simple data parallel or divide-and-conquer programs, e.g. an absolute speedup of 135 on 168 cores of a Beowulf cluster.

1 Introduction

The multicore revolution is driving renewed interest in parallel functional languages. Early parallel Haskell variants like GpH [20] and Eden [14] use elaborate runtime systems to support their high-level coordination constructs - evaluation strategies and algorithmic skeletons respectively. More recently the multicore Glasgow Haskell Compiler (GHC) implementation also extends the runtime system [18]. However these bespoke runtime systems have development and maintainability issues: they are complex stateful systems engineered in low-level C and use message passing for distributed architectures. Worse still they must be continuously re-engineered as the GHC research compiler evolves.

To preserve maintainability and ease development several recent parallel Haskell variants use Concurrent Haskell as a systems language on a vanilla GHC rather than changing the GHC runtime system. Examples include CloudHaskell [6], and the `Par Monad` [17]. Our new language, *Haskell distributed parallel Haskell (HdpH)*, also takes this approach.

Table 1 compares the key features of general purpose parallel Haskell variants, and each of these languages is discussed in detail in Section 2. Most of the entries in the table are self-explanatory. Fault Tolerance means that the language implementation isolates the heaps of each distributed node, and hence has the *potential* to tolerate individual node failures - few Haskell variants have implemented fault tolerance. Determinism identifies whether the language model guarantees that a function will be a function even if its body is in parallel.

The crucial differences between HdpH and other parallel Haskell variants can be summarised as follows. Both GHC and the `Par Monad` provide parallelism only on a single

Property \ Language	Low-level RTS			Haskell-level RTS		
	GpH-GUM	Eden	GHC	Par Monad	CloudHaskell	HdpH
Scalable (distributed memory)	+	+	-	-	+	+
Fault Tolerance (isolated heaps)	-	(+)			+	+
Polymorphic Closures	+	+	+	+	-	+
Pure, i.e. Non-Monadic API	+	+	+	-	-	-
Determinism	(+)	-	(+)	+	-	-
Implicit Task Placement	+	+	+	+	-	+
Automatic Load Balancing	+	+	+	+	-	+

Table 1. Parallel Haskell Comparison

multicore, where HdpH scales onto distributed memory architectures with many multicore nodes. CloudHaskell replicates Erlang style [1] explicit distribution. 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 evaluation strategies or algorithmic skeletons.

Section 3 describes the key features of HdpH as follows.

- HdpH is *scalable* with a distributed memory model that manages computations on more than one multicore node.
- HdpH provides *high-level semi-explicit parallelism* with
 - Implicit task placement: the programmer is not required to explicitly place tasks on specific nodes. Idle nodes seek work automatically.
 - Automated and dynamic load management: the programmer is not required to ensure that all nodes are utilised effectively. The implementation continuously manages load.
 - Polymorphism: polymorphic closures can be transferred between nodes.
 - Powerful coordination abstractions: specifically both evaluation strategies and algorithmic skeletons can be defined using a small set of polymorphic coordination primitives, see examples in Section 3.3.
- HdpH has the potential to be *fault tolerant* as there is no implicit sharing between the heap on one node and any other node. Hence the implementation can recover from the failure of remote nodes.
- The combination of stateful computations and fault tolerance implies that HdpH must be non-deterministic, as discussed in section 3.1.

Section 4 outlines the HdpH implementation design which aims to deliver acceptable performance while being maintainable. Implementing the language in vanilla (GHC) Concurrent Haskell is crucial to preserving maintainability, and enables the language design space to be explored far more readily than a low-level implementation. ♠**PWT:** reference a discussion in section 4?♠ Moreover the implementation is layered and modular with coordination aspects such as communication, thread management, global address management, scheduling etc. realised in independent modules. This design represents a middle ground between monolithic RTS like GUM [22] and Eden/EDI [12,3], and kernel-based proposals [13,2].

Section 5 reports initial performance results for three simple data parallel or divide-and-conquer programs on a 32 node, 256 core Beowulf cluster. The HdpH system is available for download [10], and an extended version of this paper is available [15];

2 Related Work

This section outlines parallel functional languages and implementations that have influenced the design and implementation of HdPH. As HdPH is primarily control-oriented, we do not consider data oriented parallel languages like DPH [4] or SAC [8] here. A comprehensive survey of parallel functional languages is available in [21].

2.1 Shared-Memory Languages

Haskell extensions for explicit concurrency have proved invaluable since 1996 [19]. More recently the focus of attention has shifted to parallel and distributed applications. A crucial feature of GHC concurrency are light-weight threads with extremely low thread management overheads

While explicit threads provide a suitable mechanism for describing independent stateful computations, the necessity of explicit control makes them less suitable for defining parallel, stateless computations. To provide a higher level of abstraction, the `Par` Monad [17] provides monadic control of concurrency, realising deterministic pure parallelism. Moreover, the `Par` Monad allows to lift system-level functionality (in the form of a work-stealing scheduler) to the Concurrent Haskell level. Performance results demonstrate that the overhead associated with the `Par` Monad remains low.

The `GpH` extension of Haskell focuses on pure parallelism and keeps many details of the parallel execution hidden from the programmer. Both shared [18] and distributed memory [22] implementations are available. The specification of parallelism in `GpH` is less intrusive than in the `Par` Monad. Effective parallel programming requires specifying both evaluation order and evaluation degree. To do so elegantly evaluation strategies, i. e., high-level coordination abstractions have been developed [20,16].

2.2 Distributed-Memory Languages

Eden extends Haskell with distributed memory parallelism [14]. It supports process abstractions, analogous to lambda abstractions, and uses process application to instantiate parallelism. Placement of the generated threads and synchronisation between them is implicit, and managed by the runtime system. A higher level of abstraction is provided through skeletons, capturing specific patterns of parallel execution, implemented using these parallelism primitives.

Erlang is a distributed functional language originally developed by Ericsson for constructing server-side telecommunications [1], and has experienced rapid uptake in a range of industrial sectors. Erlang broadly follows the Actor model and is widely recognised as a beacon language for distributed computing, influencing the design of many languages and frameworks, for example Scala, F#, and most relevantly Cloud-Haskell [6]. The key aspects of Erlang style concurrency are first class processes that may fail without damaging others, fast process creation and destruction, scalability, fast asynchronous message passing, copying message-passing semantics (share-nothing concurrency), and selective message reception [23].

A recent development that heavily influenced our work, was the design and implementation of CloudHaskell [6,7]. It emulates Erlang style distribution, explicitly targeting distributed memory systems, and implementing all parallelism extensions (processes with explicit message passing and automatic serialisation) entirely on the Haskell level. Overall, it is more explicit in its handling of parallelism and thus gives the programmer more control. Initial performance results indicate that the overhead from using Haskell as a system language is acceptable.

2.3 Parallel Functional Language Implementations

Many parallel functional language implementations use a sophisticated and low-level runtime system to coordinate the parallel coordination. For example the runtime system must distribute and schedule work, communicate between and synchronise threads etc. Example implementations taking this approach include GUM for distributed memory GpH [22], Dream/EDI for distributed memory Eden [14] and the threaded runtime system for GHC [18].

In an area where performance is the main issue it is appealing to use a low-level systems language and to avoid multiple layers that may impact performance. However, the main shortcoming of this approach is that constructing and maintaining the implementation is challenging and internal changes to the runtime system, e. g., to the structure of closures, may directly impact the parallel implementation.

In contrast, the implementations of CloudHaskell [6] and the `Par Monad` [17] leave the runtime system unchanged, and implement all functionality on the Haskell level. Using Haskell's advanced abstraction mechanisms ensures ease of maintainability and more readable implementations. Moreover GHC's light-weight threads deliver good performance.

3 Language Design

This section presents the initial design of HdpH. Our design is strongly influenced by GpH, by Eden's EDI layer, and by two recent developments that lift functionality normally provided by the RTS to the Haskell level.

1. The `Par Monad` [17] as a shallowly embedded DSL for deterministic parallelism; Section 3.1 introduces our extended `Par` monad for distributed-memory parallelism.
2. Closure serialisation in CloudHaskell [6]; Section 3.2 presents our extension of CloudHaskell's closure representation for supporting polymorphic closure transformations, which Section 3.3 capitalises on to implement high-level coordination abstractions.

3.1 Primitives

Figure 1 shows the basic primitives which HdpH exposes to the programmer, with shared-memory primitives mostly inherited from the `Par Monad` [17] to the left, and distributed-memory primitives to the right.

```

data Par a -- Par monad      data NodeId    -- explicit locations
eval :: a -> Par a         data Closure a -- explicit, serialisable closures

fork :: Par () -> Par ()   spark  :: Closure(Par ()) -> Par ()
                                pushTo :: Closure(Par ()) -> NodeId -> Par ()

data IVar a -- buffers     data GIVar a   -- global handles to IVars
new  :: Par (IVar a)      glob  :: IVar (Closure a) -> Par (GIVar (Closure a))
put  :: IVar a -> a -> Par ()
get  :: IVar a -> Par a   rput  :: GIVar (Closure a) -> Closure a -> Par ()

```

Fig. 1. HdpH primitives. To the left types and primitives for shared memory inherited from the `Par Monad`; to the right types and primitives for distributed memory.

The `Par` type constructor is a monad¹ for encapsulating a parallel computation. The basic primitive for generating shared-memory parallelism is `fork`, which forks a new thread and returns nothing. To communicate the results of computations (and to block waiting for their availability), threads employ `IVars`, which are essentially mutable variables that are writable exactly once. The programmer has access to these via 3 operations: `IVar` creation (`new`), blocking read (`get`), and write (`put`). Note that `put` does not normalise its argument, unlike the `put` in [17]. Instead the programmer can force expressions to weak-head normal form explicitly using `eval`; full normalisation can be defined by combining `eval` with `deepseq`.

To extend the constructs for distributed memory HdpH exposes types for explicit locations, explicit closures (discussed in detail in Section 3.2), and global `IVars`. Explicit locations identify *nodes*, i.e., an operating system process running HdpH using the GHC RTS and possibly utilising multiple cores.

The basic primitives for generating distributed-memory parallelism are `spark` and `pushTo`. The former operates much like `fork`, generating a thread that *may* be executed on a different node. However, it can't just take a `Par` computation as an argument because such a computation can't be serialised. Instead, the argument to be `spark`d must be converted to an explicit closure first. The `pushTo` primitive is similar except that it eagerly pushes a `Par` computation wrapped in an explicit closure to a target node, where it is eagerly executed. In contrast, `spark` just stores its argument in a local *spark pool*, where it sits waiting to be distributed or scheduled by an on-demand work-stealing scheduler, as detailed in Section 4.2.

To retrieve the results of a remote closure or synchronise computations on remote closures HdpH introduces *global IVars*. These are simply global references to `IVars`, offering two operations: Creation (`glob`) by globalising a local `IVar`, and remote write (`rput`). To ensure that the values written by `rput` are serialisable, `glob` and `rput` restrict the base type of their underlying `IVars` to closures. Hence all values transported between nodes, be it computations or result values, are closures. Thus result values may be computations! There is no remote read on global `IVars` — in this respect they are much like channels in Eden and CloudHaskell, supporting remote writes but only local reads.

For comparison and demonstration we present three parallel Fibonacci functions in Figure 2. All three functions take two arguments: the second is the argument to the

¹ `Par` is a continuation monad like Claessen's Poor Man's Concurrency monad [5]; alternatively `Par` could be based on Harrison's resumption monad [9].

```

pfib :: Int -> Int -> Int
pfib t n
  | n <= t    = fib n
  | otherwise = x `par` y `pseq` x + y
    where x = pfib t (n-1)
          y = pfib t (n-2)

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

spfib :: Int -> Int -> Par Int
spfib t n
  | n <= t    = return $ fib n
  | otherwise = do
    v <- new
    fork $ spfib t (n-1) >>=
      eval >>=
        put v
    y <- spfib t (n-2)
    x <- get v
    return (x + y)

dpfib :: Int -> Int -> Par Int
dpfib t n
  | n <= t    = return $ fib n
  | otherwise = do
    v <- new
    gv <- glob v
    spark $(mkClosure [|dpfib t (n-1) >>=
      eval >>=
        rput gv . toClosure|])
    y <- dpfib t (n-2)
    clo_x <- get v
    return (unClosure clo_x + y)

```

Fig. 2. Fibonacci numbers. To the left GpH code and shared-memory parallel code in HdpH; to the right sequential code and distributed-memory parallel code in HdpH.

Fibonacci function, and the first a granularity threshold below which to generate no parallelism.

The first variant, `pfib`, uses the GpH `par` and `pseq` primitives. It can be executed either on a shared-memory multicore using the GHC RTS or on distributed-memory architectures using the GUM RTS. The second variant, `spfib`, uses the shared-memory primitives of the `Par Monad`, and can thus only be executed on a shared-memory machine using the GHC RTS. The third variant, `dpfib`, employs the HdpH primitives and can thus be executed on shared or distributed-memory architectures using the HdpH implementation.

There are many similarities between `spfib` and `dpfib`; the difference is that `spfib` can simply `fork` the first recursive call, whereas `dpfib` must globalise the `IVar v`, yielding global `IVar gv`, and wrap the first recursive call in an explicit closure generated by the Template Haskell splice `$(mkClosure [|...|])`, before `spark`ing. Moreover, `dpfib` must convert the result of the `spark`ed computation to an explicit closure with `toClosure` before writing to `gv`, and that closure must be eliminated again with `unClosure` before adding the results of both recursive calls.

Non-determinism, fault tolerance and the semantics of IVars. There is a subtle difference in the semantics of `put` in HdpH versus the `Par Monad` [17]. To preserve determinism in the `Par Monad` the semantics are to abort the program if there is an attempt to `put` into a full `IVar`. HdpH opts for a different and non-deterministic semantics: `put`ting into a full `IVar` has no effect. That is, only the first `put` succeeds but subsequent `puts` aren't fatal.

A distributed language like HdpH has good reasons for introducing non-determinism. If HdpH is to survive node failures it must be able to restart supposedly failed computations, and such restarts are speculative because it is not possible to tell for certain whether a node has failed or whether it is only temporarily unreachable. A speculative restart will share the global `IVars` expecting its result with the original computation, opening up a race if the original computation is still alive. Another reason is that many distributed algorithms are non-deterministic by nature; insisting on determinism would

limit the expressiveness of HdpH in this domain. The price we pay for this gain in expressiveness is that HdpH computations cannot be run outside the IO monad.

3.2 Explicit Closures

CloudHaskell [6] introduced the idea of making a thunk `thk` serialisable by constructing an *explicit closure* consisting of an environment `env` storing the variables captured by `thk`, and a function `fun` such that `fun env = thk`. Because all variables captured by `thk` have been abstracted out to `env`, `fun` does not itself capture any variables, that is all its free variables are top-level, which implies that `fun` itself could be defined at top-level. CloudHaskell pulls two tricks to make explicit closures serialisable, i.e., an instance of class `Binary`. It assumes (1) that `env` is already serialised and represented as a byte string that will be deserialised by `fun`, and (2) that `fun` is top-level, hence is serialisable as its code address. The latter trick requires a Haskell extension: a primitive type constructor `Static` for reflecting code addresses of top-level terms, plus term formers `static :: a -> Static a` for obtaining the address of a term which could be top-level, and `unstatic :: Static a -> a` for resolving such an address. Though not (yet) implemented in GHC 7, we proceed with our language design in this section as if `Static` were fully supported.² Details about `Static` can be found in [6] and are not relevant for the rest of this paper, save for the fact that `Static` is an instance of the classes `Binary` and `NFData`.

CloudHaskell represents explicit closures as a pair of a serialised environment of type `Env`, a synonym for byte strings, and a static deserialiser, i.e., the address of a deserialiser.

```
data Closure a = MkClosure (Static (Env -> a)) Env
unClosure :: Closure a -> a
unClosure (MkClosure fun env) = (unstatic fun) env
```

Other than serialisation the only operations on closures that CloudHaskell exposes are introduction by the constructor, and elimination by `unClosure`. As introduction is lazy it delays serialising the environment until demanded, either by the closure being serialised or eliminated. Oddly, closure introduction and elimination are asymmetric: `unClosure . MkClosure` is not an identity, because `unClosure` eliminates not only the constructor but also the closure representation. Moreover, once that representation is gone there is no way of getting it back. This does not matter as long as closures are just used to ferry computations from one node to another, to be unpacked and executed at the target node.

We consider that the CloudHaskell `Closure` constructor is too limited. Firstly, the constructor should be generalised to support both computation with closures as well as their transportation. Ideally, `Closure` should be a *functor*, so closures can be transformed without eliminating them. In addition there should be a special closure transformation that *forces*, i.e., evaluates to normalform, its content. Finally, we'd like to avoid unnecessary serialisation, e.g., when eliminating a closure immediately after in-

² We currently use work-arounds to emulate `Static`, similar to CloudHaskell.

```

data Closure a = UnsafeMkClosure
    a                -- actual closure value
    (Static (Env -> a)) -- static deserialiser
    Env              -- serialised environment

instance Binary (Closure a) where
    put (UnsafeMkClosure _ fun env) = put fun >> put env
    get = do fun <- get
            env <- get
            let val = (unstatic fun) env
            return $ UnsafeMkClosure val fun env

instance NFData (Closure a) where
    rnf (UnsafeMkClosure _ fun env) = rnf fun `seq` rnf env

unClosure :: Closure a -> a
unClosure (UnsafeMkClosure val _ _) = val

toClosure :: (Binary a) => a -> Closure a
toClosure val = UnsafeMkClosure val (static decode) (encode val)

mapClosure :: Closure (a -> b) -> Closure a -> Closure b
mapClosure clo_f clo_x = $(mkClosure [|unClosure clo_f $ unClosure clo_x|])

```

Fig. 3. HdpH closure representation and operations on closures.

roduction³ We use strategies to force the evaluation of a closure (Section 3.3), and will address the other issues while introducing the enhanced HdpH closure representation in Figure 3.

Dual closure representation. To avoid unnecessary serialisation HdpH maintains a dual representation of closures, extending CloudHaskell’s closure representation with the actual closure. In Figure 3 the first argument of the `UnsafeMkClosure` constructor is the non-serialisable closure being represented. With this representation unnecessary serialisation is avoided as `Closure` elimination is just a projection on the first argument, involving no serialisation.

Dual representation implies the obligation to maintain the invariant that the two representations are semantically and computationally equivalent. When constructing closures explicitly this obligation rests on the programmer, which is why the constructor is termed `UnsafeMkClosure`.

The `Binary` instance maintains the invariant by serialising only the serialisable representation `fun` and `env`, reconstructing the actual closure `val` upon deserialisation by applying the static deserialiser `fun` to the serialised environment `env` in the same way as CloudHaskell eliminates its explicit closures. Note that this reconstruction of `val` is lazy, and hence delayed until the explicit `Closure` is eliminated.

The `NFData` instance normalises only the serialisable representation `fun` and `env`, not the actual closure `val`. Normalising `val`, too, would break the dual representation invariant because the actual closure would be in normal form but the serialisable representation would not. Specifically, `unClosure $ decode $ encode $ clo` and `unClosure clo` would not have the same strictness properties if the actual closure `val` were normalised.

³ This is not a concern if all closures are guaranteed to be serialised because they are to be shipped across the network. However, computing with closures tends to create lots of intermediate closures, so treating them efficiently becomes important.

Safe closure construction. As using `UnsafeMkClosure` is cumbersome and error-prone, there are *safe* `Closure` constructions that guarantee the dual representation invariant. The simplest such construction is `toClosure`, albeit only for serialisable values. The function `toClosure` simply pairs a serialised value with the appropriate static deserialiser which exists thanks to the `Binary` context. Note that `toClosure` is lazy, i. e., its argument will not be serialised until the resulting `Closure` is serialised or normalised. In particular, `unClosure . toClosure` is an identity which does not involve serialisation.

A more general safe `Closure` construction is to generate the arguments to the constructor `UnsafeMkClosure` automatically by macro expansion, using Template Haskell. This is done by the function `mkClosure :: Q Exp -> Q Exp`, which safely converts a *quoted* actual closure, i. e., an expression in Template Haskell’s quotation brackets `[|...|]`, into a quoted `Closure`, to be spliced into the code using Template Haskell’s splicing parentheses `$(...)`.⁴ To explain what `mkClosure` does, we show what the call in Figure 2 expands to.

```
mkClosure [|dpfib t (n-1) >>= eval >>= rput gv . toClosure|]
= [|let val = dpfib t (n-1) >>= eval >>= rput gv . toClosure
     env = encode (gv, t, n)
     fun = static (\env -> let (gv, t, n) = decode env in
                          dpfib t (n-1) >>= eval >>= rput gv . toClosure)
     in UnsafeMkClosure val fun env|]
```

To start, `mkClosure` finds the variables captured by the actual closure and packs them into a tuple, here `(gv, t, n)`. Then, it produces the explicit `Closure` expression `(UnsafeMkClosure val fun env)`, where `val` is the actual closure, `env` is the serialised environment, i. e., the serialised tuple of captured variables, and `fun` is a static deserialiser. The latter is actually the address of a wrapper around the actual closure, abstracting over its serialised environment. That is, the wrapper is a λ -abstraction whose body is the actual closure yet the captured variables (here `gv`, `t` and `n`) are now let-bound as a result of deserialising the parameter `env`. The wrapper itself does not capture any variables, hence `static` is applicable.

Note how `mkClosure` eliminates two pitfalls that `UnsafeMkClosure` exposed programmers to: (1) It guarantees the dual representation invariant, and (2) it ensures that the tuple of captured variables is serialised and deserialised in exactly the same shape and order.

Transforming closures. One might think that `Closure` should be an instance of the `Functor` class. It would appear that `fmap` can be implemented by generating an explicit `Closure` which applies a function another closure, like so:

```
fmap :: (a -> b) -> Closure a -> Closure b
fmap f clo_x = $(mkClosure [|f $ unClosure clo_x|])
```

The problem is that this `fmap` does not compile because the argument to `mkClosure` captures the function `f`, which implies that `f` must be serialisable. However, arbitrary functions are not serialisable — that is the very reason for inventing explicit `Closures`.

⁴ Truth in advertising: `mkClosure` is planned but not yet implemented. Nevertheless, we will use it in examples because it improves code readability.

```

type Strategy a = a -> Par a

using :: a -> Strategy a -> Par a
x `using` strat = strat x

forceClosure :: (Binary a, NFData a) => Strategy (Closure a)
forceClosure clo = val' `deepseq` return clo
  where clo@(UnsafeMkClosure val' _ _) = toClosure $ unClosure clo

parList :: Closure (Strategy (Closure a)) -> Strategy [Closure a]
parList clo_strat = mapM spawn >=> mapM get
  where spawn :: Closure a -> Par (IVar (Closure a))
        spawn clo = do
          v <- new
          gv <- glob v
          spark $ $(mkClosure [|clo `using` unClosure clo_strat|] >>= rput gv|)
          return v

parListNF :: (Binary a, NFData a) => Strategy [Closure a]
parListNF = parList $(mkClosure [|forceClosure|])

parMap :: (Binary a, Binary b, NFData b) => Closure (a -> b) -> [a] -> Par [b]
parMap clo_f xs = do clo_ys <- map f clo_xs `using` parListNF
  return $ map unClosure clo_ys
  where f = mapClosure clo_f
        clo_xs = map toClosure xs

```

Fig. 4. Task-farm skeleton implemented via closure strategies.

Nonetheless, the idea of an `fmap`-like `Closure` transformation can be salvaged if we insist on the function argument being a `Closure` itself. Figure 3 shows the resulting functor-like transformation `mapClosure`, promoting a function `Closure` to a function on `Closures`. Note how `mapClosure` is implemented in terms of `Closure` elimination and introduction, which is why our efforts in eliminating unnecessary serialisation overhead are relevant.

In fact, `mapClosure` is not just a functor-like transformation; it actually *is* (the morphism part of) a functor, just not the type of functor that would fit into the `Functor` class. Instead, it is a functor mapping function `Closures` to functions on `Closures`, see the technical report [15] for details.

3.3 Strategies and Skeletons

Directly using coordination primitives like those in Section 3.1, effectively parallelises functions but produces obscure code where computation and coordination aspects are intertwined. To disentangle coordination and computation we seek to develop higher-level abstractions over the primitives, and Figure 4 shows some simple examples.

Evaluation Strategies are compositional building blocks for coordination developed for `GpH` in [20,16]. Following [16], strategies in `HdpH` are identity functions in the `Par` monad, i.e. functions of type `a -> Par a` whose denotational semantics is the identity. A strategy may cause sequential or parallel evaluation of their argument as a side effect. Being based on the `Par` monad rather than the `Eval` monad of [16] has implications because the `Par` monad can't be escaped as easily as the `Eval` monad. For example, strategy application with `using` must stay in `Par`. Moreover, the strategy composition `dot` that was used extensively in the strategies library of [16] cannot be expressed without leaving the monad, nor can strategies for infinite data structures like

rolling buffers for lazy streams. Nevertheless, many useful strategy combinators can be written.

Since all distributed-memory parallelism in HdpH involves explicit `Closures`, we focus on `Closure` strategies. The most basic of these is `forceClosure` which fully evaluates a `Closure` and is shown in Figure 4. It evaluates the `Closure` by eliminating the constructor with `unClosure` and converting the resulting value back to another `Closure` with `toClosure` which is then forced to normalform with `deepseq`. Note how this is different from just `deepseq`ing the actual value of the original `Closure`, which would result in an evaluated closure that would revert to its unevaluated state upon serialisation.

The `parList` strategy combinator applies a strategy to a list in parallel. The list is of type `[Closure a]`, so we expect an argument of type `Strategy (Closure a)`; however, the strategy argument itself needs to be serialised, see the definition of `spawn`, so it must be wrapped in another `Closure`. The implementation of `parList` is straightforward: `Spawn` all strategy applications with `mapM spawn` producing a list of `IVars`, then read the results back with `mapM get`; the structure of the code for `spawn` itself is similar to that of `dpfib` in Figure 2. The strategy `parListNF`, which fully evaluates a list of closures in parallel, is derived by “applying” `parList` to `forceClosure` after wrapping the latter in another `Closure`.

Skeletons are polymorphic higher order functions that abstract common parallel programming patterns, e. g., task farms. Thanks to support for polymorphic closure transformations like `mapClosure` and polymorphic strategies like `parListNF`, we can implement simple skeletons as in GpH. For example, Figure 4 shows the task farm skeleton `parMap`, which applies a function `Closure` to all elements of a list in parallel using the strategy `parListNF`. A sample use of `parMap` to implement a data-parallel computation can be found in the technical report [15].

In the implementation of `parMap`, we can still observe the separation of computation: the `clo_ys <- map f clo_xs` part of the first line, and coordination: the `\using\ parListNF`, though it is muddled in the rest of the function that deals with the closure conversions and eliminations.

4 Implementation Design

♠**PWT**: We should list all of the features of GHC Concurrent Haskell we rely on: `Templates`, etc. ♠ For maintainability HdpH is implemented in a layered fashion with coordination aspects such as communication, global reference management, spark management, scheduling, etc. realised in independent modules. Figure 5 depicts the HdpH architecture in terms of state (mutable data structures in Haskell) and agents (Haskell IO threads). Each node runs several thread schedulers, typically one per core. Each scheduler has a dedicated thread pool - a concurrent deque, that may be accessed by other schedulers periodically. Each node runs a message handler, which shares access to the spark pool, another concurrent deque, with the schedulers. Each node has a registry, a concurrent map, of global `IVars` that is shared between message handler and schedulers.

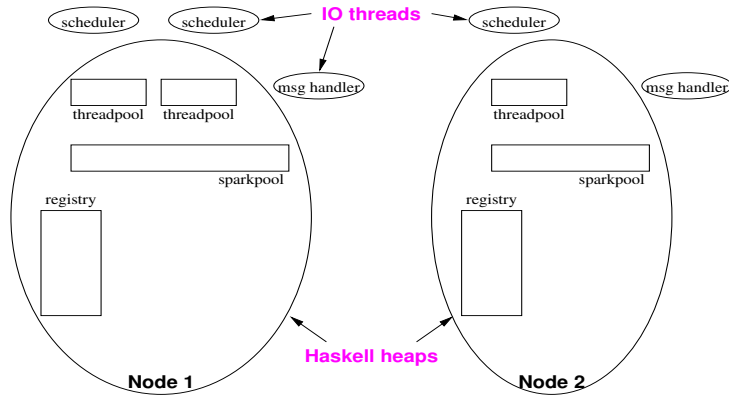


Fig. 5. HdpH system architecture; coupling a dual core and a uni-core node.

Inter-node communication is abstracted into a *communication layer*, that provides startup and shutdown functionality, node IDs, and seamless peer-to-peer send/receive of arbitrarily sized byte strings. Currently this layer is based on a standard MPI library; we plan to port to other networking libraries (with better support for fault tolerance).

Thread scheduling is based on the work-stealing scheme of the `Par Monad` [17], except that HdpH implements a two-tier work pool. Idle schedulers first try to steal threads from other thread pools; if that fails they try to pick sparks from the spark pool.

4.1 Global references and global IVars

Global references provide a type-safe way of accessing remotely hosted objects (Figure 6). A global reference records the type of the referenced object as a phantom type, i.e., `ref :: GRef t` references an object of type `t`. A global reference is represented by a pair of a node ID identifying the host of the referenced object and a name that is unique on that host (and stays unique over the life span of the host). This yields globally unique identifiers with cheap projection, `at`, to the node component, and straightforward serialisation and normalisation.

The link between a global reference (whose host is the current node) and its referenced object is established by the *registry*, a concurrently mutable table, much like

```

data GRef a
instance Eq (GRef a) where { ... }
instance Binary (GRef a) where { ... }
instance NFData (GRef a) where { ... }

at :: GRef a -> NodeId
globalise :: a -> IO (GRef a)
deref :: GRef a -> IO (Maybe a)
free :: GRef a -> IO ()

type GIVar a = GRef (IVar a)
glob :: IVar (Closure a) -> Par (GIVar (Closure a))
glob = lift . globalise
rput :: GIVar (Closure a) -> Closure a -> Par ()
rput gv clo = pushTo (at gv) $(mkClosure [|lift (deref gv) >>=
    maybe
      (return ())
      (\v -> put v clo >> lift (free gv))|])

```

Fig. 6. API of global references and implementation of global IVars.

the GALA table in the GUM RTS, except that the registry is implemented in Haskell (currently as mutable reference to an immutable finite map). There are two basic operations on global references: (1) Introducing a fresh one (by `globalise`ing a local object) and (2) eliminating an existing one (by `dereferencing` it). However, to avoid having to implement a global garbage collection, we add a third operation for `freeing` a global reference. Thus, we are faced with the problem that a global reference may be dead (because it has been `freed` earlier) when we attempt to `dereference` it, that explains the `Maybe` return type.

In many ways, global references are like stable names: they provide stable, global and type-safe identifiers for the objects they reference. There is one essential difference: The life time of a stable name is tied to the life time of its referenced object — stable names whose objects have vanished may be garbage collected and re-used later. In contrast, the life time of a global reference is decoupled from the life time of the object it points to (since the object may live in a different heap). Hence global references must never be re-used.

Global references aren't exposed in `HdpH` (apart from the function `at`). Instead they serve to implement *global IVars*: a `GIVar` is simply a global reference to an `IVar` (Figure 6) and inherits the properties of global references, including serialisability. Moreover, `glob` simply lifts the respective operation on global references to the `Par` monad.

The semantics of `rput` is more complex: it pushes a computation to the node hosting the global `IVar` (i. e., the node hosting the referenced `IVar`). That computation dereferences the global reference and, depending on the outcome, either returns immediately (in case the global reference was dead) or else takes the dereferenced `IVar`, writes to it and then `free`s the original global `IVar`. The semantics of actions on dead references is consistent with the semantics for `IVars`. If `rput` encounters a dead global `IVar` `gv` then `gv` must have been filled by an earlier, successful `rput`, and in that case `put` would fail silently, just as `rput` does.

4.2 Spark management

`HdpH` spark management is essentially a re-implementation of GUM RTS functionality at the Haskell level. Each node stores *sparks*, i. e., values of type `Closure (Par ())`, in a pool, that they enter either on being `spark`ed by a scheduler, or after the message handler has received a `SCHEDULE` message. Sparks leave the pool either to be turned into local threads (by eliminating the `Closure`), or to be `SCHEDULE`d on another node, which entails serialising the `Closure`. Currently the spark selection strategy is purely age-based: the youngest sparks are turned into threads, and the oldest are `SCHEDULE`d away.

When the spark pool is running low, a `FISH` message is sent to a random node (or to a node known to have had excess sparks recently). If a node receives a `FISH`, it either replies with a `SCHEDULE` (in case it has excess sparks to give away) or forwards the `FISH` to a random node. To avoid `FISH` messages circulating forever, each `FISH` counts the number of times it is forwarded. If the counter reaches a configurable threshold, the `FISH` expires and a `NOWORK` message is returned to its originator, who will then wait for some configurable amount of time before sending the next `FISH`.

nodes	cores	Fibonacci			SumEuler (prim)			SumEuler (parMap)		
		runtime	error	speedup	runtime	error	speedup	runtime	error	speedup
	<i>sequential</i>	424.58s	6%		355.69s	8%		<i>see columns to the left</i>		
1	6	75.64s	1%	5.6	62.31s	< $\frac{1}{2}$ %	5.7	62.65s	< $\frac{1}{2}$ %	5.7
2	12	42.29s	< $\frac{1}{2}$ %	10.0	32.72s	< $\frac{1}{2}$ %	10.9	32.71s	< $\frac{1}{2}$ %	10.9
3	18	28.32s	1%	15.0	22.07s	< $\frac{1}{2}$ %	16.1	22.14s	1%	16.1
4	24	20.25s	< $\frac{1}{2}$ %	21.0	16.42s	1%	21.7	16.47s	< $\frac{1}{2}$ %	21.6
6	36	14.09s	1%	30.1	11.13s	1%	31.9	11.12s	1%	32.0
8	48	10.37s	1%	41.0	8.48s	1%	42.0	8.47s	< $\frac{1}{2}$ %	42.0
12	72	6.81s	1%	62.3	5.91s	2%	60.2	5.91s	1%	60.2
16	96	5.26s	2%	80.7	4.47s	3%	79.5	4.53s	1%	78.5
20	120	4.21s	2%	100.9	3.79s	5%	93.8	3.83s	9%	92.9
24	144	3.55s	2%	119.7	3.99s	13%	89.1	3.29s	16%	108.0
28	168	3.14s	7%	135.4	3.72s	7%	95.6	3.25s	7%	109.5

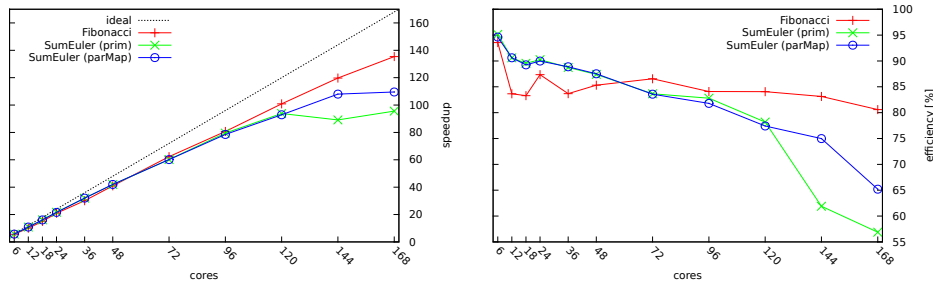


Fig. 7. Results of 3 benchmarks: runtime, absolute speedup and efficiency.

Executing `pushTo clo` node sends `clo` to node in a `PUSH` message. Upon receiving a `PUSH` the message handler eliminates the `Closure` and executes the resulting computation, without waiting for a scheduler to become available. Thus `pushTo` is suitable for very short and urgent actions like writing to an `IVar` or forking a thread.

5 Preliminary Performance Results

To evaluate the scalability and efficiency of HdpH we benchmark three simple parallel programs on a 32-node Beowulf cluster. Each node comprises two Intel quad-core processors (Xeon E5504) at 2GHz, sharing 4MB of L3 cache and 12GB of RAM. Nodes are connected via Gigabit Ethernet and run Linux (CentOS 5.7 x86_64). HdpH and the benchmarks were built with GHC 7.2.1 and linked against the MPICH2 library (version 1.2.1p1). Benchmarks were run on up to 28 cluster nodes, utilising up to 6 cores per node.⁵ Reported runtime is median wall clock time over 7 executions. Reported error is standard deviation relative to median runtime; percentages in the low single digits indicate high quality measurements.

Figure 7 summarises our results in terms of runtime, absolute speedup and efficiency. The *Fibonacci* benchmark is a regular divide-and-conquer algorithm computing `dpfib 30 50` from Figure 2. The program generates 17710 sparks with an average granularity of 25 milliseconds. With efficiency declining very slowly, Fibonacci scales very well, yielding a maximum speedup of 135 on 168 cores. The reason is that a regular

⁵ To avoid high variability we follow common practice in not maximising the number of nodes and cores.

divide-and-conquer algorithm tends to generate work on many nodes, so work stealing via random fishing tends to be very effective.

The two *SumEuler* benchmarks map Euler’s totient function over the list `[1..65536]` and reduce the result to a sum. Both are regular ♠PWT: Are they regular? Computing the totient of a larger number takes longer than for a small number.♠♠PM: I am not chunking the input list but dealing it to 1024 tasks in a round-robin fashion, so each task will have big and small numbers. Code is now in the appendix.♠, flat data-parallel algorithms, where the main thread deals the input list in a round-robin fashion to 1024 sparks (with a granularity of about 350 milliseconds each), and sums up the results. The two *SumEuler* benchmarks differ in that one is implemented solely in terms of the `HdpH` primitives whereas the other relies on the `parMap` skeleton (and hence on polymorphic closure operations). Code for both versions can be found in the technical report [15].

Both *SumEuler* benchmarks scale worse than Fibonacci — efficiency is declining faster, maximum speedup is limited to about 110 on 168 cores — because the main node is bound to become a bottleneck. Remarkably though, both *SumEuler* benchmarks perform virtually the same,⁶ suggesting that the overhead of `parMap` is negligible.

Finally, we observe that all benchmarks achieve their peak efficiency, about 95%, on a single node. Efficiency drops steeply (to 80–90%) when adding a second node and then declines more slowly and steadily. The likely cause for the boosted efficiency is that `HdpH` completely avoids serialisation overheads when running on a single node.

6 Conclusion

We have presented the initial design, implementation and preliminary evaluation of a new distributed memory parallel Haskell, `HdpH`. The language supports high-level semi-explicit parallelism, is scalable, and has the potential for fault tolerance (Section 3). The `HdpH` implementation is designed for maintainability without compromising performance too severely. To provide flexibility and maintainability the implementation is modular and layered and, crucially, coded in vanilla Concurrent Haskell (Section 4). Initial performance results for three simple data parallel or divide-and-conquer programs are promising. The current implementation delivers good efficiency and a peak absolute speedup of 135 on 168 cores spread over 28 nodes of a Beowulf cluster (Section 5).

In future work we plan to further explore the language design space represented by `HdpH`, developing a richer set of algorithmic skeletons and evaluation strategies. We will also tune the `HdpH` implementation performance. For this and general programming profiling and visualisation tools are essential. We will also adopt GHC extensions as they become available, for example a `Static` implementation.

In the medium term we plan to use `HdpH` as the implementation language for the `SymGridPar2` middleware providing parallel execution of large GAP computational algebra problems [11]. Key requirements for `SymGridPar2` are the scalability and reli-

⁶ Speedup and efficiency graphs suggest that the `parMap`-based *SumEuler* outperforms the other beyond 120 cores; however, measurement errors are too high to support that conclusion.

ability supported by the HdpH distributed memory programming model. To provide reliability we envisage developing Erlang-style fault tolerance abstractions.

Acknowledgements. Thanks to Andrew Black, Jeff Epstein, Hans-Wolfgang Loidl, and Rob Stewart for stimulating discussions. This research is supported by the EPSRC HPC-GAP (EP/G05553X), EU FP6 SCIENCE (RII3-CT-2005-026133), and EU FP7 RELEASE (FP7-ICT 287510) projects.

References

1. Armstrong, J., Viriding, S., Williams, M., Wikstrom, C.: Concurrent Programming in Erlang. Prentice-Hall, 2nd edn. (1996)
2. Berthold, J., Al Zain, A., Loidl, H.W.: Scheduling light-weight parallelism in ArTCoP. In: PADL '08, San Francisco, USA. pp. 214–229. ACM Press (2008)
3. Berthold, J.: Explicit and implicit parallel functional programming : concepts and implementation. Ph.D. thesis, Philipps-Universitt Marburg, Germany (Jul 2008)
4. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: DAMP '07, Nice, France. pp. 10–18. ACM Press (2007)
5. Claessen, K.: A poor man's concurrency monad. J. Funct. Program. 9(3), 313–323 (1999)
6. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the cloud. In: Haskell '11, Tokyo, Japan. pp. 118–129. ACM Press (2011)
7. Epstein, J.: Functional programming for the data centre. Master's thesis, Computer Laboratory, University of Cambridge (Jun 2011)
8. Grelck, C., Scholz, S.B.: SAC — A Functional Array Language for Efficient Multi-threaded Execution. International Journal of Parallel Programming 34(4), 383–427 (Aug 2006)
9. Harrison, W.L.: The essence of multitasking. In: AMAST 2006, Kuressaare, Estonia. pp. 158–172. LNCS 4019, Springer (2006)
10. Haskell distributed parallel Haskell. <https://github.com/PatrickMaier/HdpH>
11. HPC-GAP: High Performance Computational Algebra and Discrete Mathematics. Web page (Aug 2011), <http://www-circa.mcs.st-andrews.ac.uk/hpcgap.php>
12. Klusik, U., Ortega-Mallén, Y., Peña, R.: Implementing Eden - or: Dreams become reality. In: IFL '98. pp. 103–119. LNCS 1595, Springer (1998)
13. Li, P., Marlow, S., Peyton Jones, S., Tolmach, A.: Lightweight concurrency primitives for GHC. In: Haskell '07, Freiburg, Germany. pp. 107–118. ACM Press (2007)
14. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. J. Funct. Program. 15(3), 431–475 (2005)
15. Maier, P., Trinder, P.: Implementing a high-level distributed-memory parallel Haskell in Haskell. Tech. Rep. HW-MACS-TR-0091, Heriot-Watt University (2011), http://www.macs.hw.ac.uk/~pm175/papers/Maier_Trinder_IFL2011_XT.pdf
16. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: Better strategies for parallel Haskell. In: Haskell '10, Baltimore, USA. pp. 91–102. ACM Press (2010)
17. Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. In: Haskell '11, Tokyo, Japan. pp. 71–82. ACM Press (2011)
18. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore Haskell. In: ICFP '09, Edinburgh, Scotland. pp. 65–78. ACM Press (2009)
19. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL '96, St Petersburg, USA. pp. 295–308. ACM Press (1996)
20. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithms + strategy = parallelism. J. Funct. Program. 8(1), 23–60 (1998)

21. Trinder, P.W., Loidl, H.W., Pointon, R.F.: Parallel and distributed Haskells. *J. Funct. Program.* 12(4&5), 469–510 (2002)
22. Trinder, P.W., Hammond, K., Mattson Jr., J.S., Partridge, A.S., Peyton Jones, S.L.: GUM: a portable parallel implementation of Haskell. In: *PLDI '96*, Philadelphia, USA. pp. 78–88. ACM Press (1996)
23. Wiger, U.: What is Erlang-style concurrency?, <http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/>