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
{P.Maier,P.W.Trinder}@hw.ac.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 [19] and Eden [13] use elaborate runtime systems (RTS) 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 RTS [16]. 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 to keep up with the ever evolving GHC research compiler.

To preserve maintainability and ease development several recent parallel Haskells use Concurrent Haskell [17] as a systems language on a vanilla GHC rather than changing the GHC RTS; examples include CloudHaskell [6], and the `Par` Monad [15]. Our new language, *Haskell distributed parallel Haskell (HdpH)*, also follows this approach.

Table 1 compares the key features of general purpose parallel Haskells, and each of these languages is discussed in detail in Sect. 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 Haskells have implemented fault tolerance. Determinism identifies whether the language model guarantees that a function will be a function even if its body is evaluated in parallel.

The crucial differences between HdpH and other parallel Haskells can be summarised as follows. Both GHC and the `Par` Monad provide parallelism only on a single multicore, where HdpH scales onto distributed-memory architectures with many multicore nodes. CloudHaskell replicates Erlang style [1] explicit distribution. It is most

Table 1. Parallel Haskell comparison

	Low-level RTS			Haskell-level RTS		
Property \ Language	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	+	+	+	+	−	+

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 Sect. 3.3.
- HdpH is designed with *fault tolerance* in mind.
  - The separation of heaps provides *fault isolation*, making it possible to recover from remote node failure.
  - Fault tolerance necessitates non-determinism, as we will argue in Sect. 3.1.

  HdpH is a distributed-memory *parallel* language, but not (yet) a *distributed* programming language as it lacks crucial features such as support for distributed data and exceptions. We leave the implementation of such features and of fault tolerant skeletons to future work as discussed in Sect. 6.

Section 4 outlines the HdpH implementation design which aims to deliver acceptable performance while being maintainable. Implementing the system in vanilla (GHC) Concurrent Haskell is crucial to preserving maintainability, and enables the language design space to be explored more readily than modifying an RTS written in C. 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 runtime systems like GUM [20] and Eden/EDI [11,2], and kernel-based proposals [12,3].

Section 5 reports initial performance results for three simple data parallel or divide-and-conquer programs on up to 168 cores of a Beowulf cluster. The HdpH system is available for download [9].

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 [7] here. A comprehensive survey of parallel functional languages is available in [21].

2.1 Shared-Memory Languages

There have been a number of parallel Haskell extensions [21], and several have extended the GHC compiler. A crucial feature of GHC is its concurrency support: lightweight threads with extremely low thread management overheads.

The `Par` Monad [15] provides monadic control of concurrency, realising deterministic pure parallelism. Moreover, the `Par` Monad allows the lifting of 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- and distributed-memory implementations [16,20] are available. The specification of parallelism in GpH is non-monadic and 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 [19,14].

2.2 Distributed-Memory Languages

Eden extends Haskell with distributed-memory parallelism [13]. 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 RTS. 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 and F#. 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 [22].

A recent development that heavily influenced our work, was the design and implementation of CloudHaskell [6] It emulates Erlang's distributed programming model, explicitly targeting distributed-memory systems, and implementing all parallelism extensions (processes with explicit message passing and closure serialisation) entirely

on the Haskell level. 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 RTS to coordinate parallelism. That is, the RTS schedules and distributes work, synchronises and communicates with threads, and so forth. Implementations taking this approach include Dream/EDI [13] for Eden, GUM [20] for distributed-memory GpH, and the threaded GHC RTS [16] for shared-memory GpH.

The use of a low-level systems language may be necessary when the performance impact of layers of abstraction cannot be tolerated. However, low-level implementations of parallel coordination tend to suffer from high maintenance cost, as seemingly unrelated internal changes to the RTS, e. g., to the memory layout of closures, may break parallel coordination. In contrast, the implementations of CloudHaskell [6] and the `Par` Monad [15] leave the GHC RTS unchanged, and implement all coordination 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. The design is strongly influenced by GpH, by Eden's EDI layer, and by two recent developments that lift functionality normally provided by a low-level RTS to the Haskell level.

The `Par` Monad [15], a shallowly embedded domain specific language (DSL) for deterministic shared-memory parallelism. Section 3.1 adapts this DSL to distributed-memory parallelism, including semantic provisions for fault tolerance.

Closure serialisation in CloudHaskell [6]. Section 3.2 extends CloudHaskell's closure representation to support polymorphic closure transformations, which Section 3.3 exploits to implement high-level coordination abstractions.

3.1 Primitives

Figure 1 shows the basic primitives that HdpH exposes to the programmer, with shared-memory primitives inherited from the `Par` Monad [15] to the left, and distributed-memory primitives to the right.

The `Par` type constructor is a monad[1] 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 `IVar`s, which are essentially mutable variables that are writable exactly once. The programmer has access to these via three

[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 [8].

```
data Par a  -- Par monad
eval :: a -> Par a

fork :: Par () -> Par ()

data IVar a -- buffers
new :: Par (IVar a)
put :: IVar a -> a -> Par ()
get :: IVar a -> Par a
```

```
data NodeId     -- explicit locations
allNodes :: Par [NodeId]

data Closure a  -- explicit, serialisable closures
spark  :: Closure(Par ()) -> Par ()
pushTo :: Closure(Par ()) -> NodeId -> Par ()

data GIVar a    -- global handles to IVars
glob :: IVar (Closure a) -> Par (GIVar (Closure a))
rput :: GIVar (Closure a) -> Closure a -> Par ()
at   :: GIVar (Closure a) -> NodeId
```

Fig. 1. HdpH primitives. To the left types and primitives for shared memory inherited from the `Par` Monad [15]; to the right types and primitives for distributed memory.

operations: IVar creation (`new`), blocking read (`get`), and write (`put`). Note that `put` does not normalise its argument, unlike the `put` in [15]. 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 DSL towards distributed memory HdpH exposes abstract data types for explicit locations, explicit closures (discussed in detail in Sect. 3.2), and global IVars. An explicit location identifies a *node*, i. e., an operating system process running HdpH, within the GHC RTS, possibly on multiple cores. HdpH is *location-aware*: The programmer can test locations for equality, query the set of all locations (`allNodes`) and query the current location (by `myNode = fmap at (new >>= glob)`).

The basic primitives for generating distributed-memory parallelism are `spark` and `pushTo`. The former operates much like `fork`, generating a computation (henceforth referred to as a *spark*) 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`ed must be converted to an explicit closure first. The `pushTo` primitive is similar except that it eagerly pushes a closure containing a `Par` computation to a target node, where it is eagerly unwrapped and 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 (Sect. 4.2).

To retrieve the results of remote closures or synchronise distributed computations, HdpH introduces *global IVars*. These are simply global references to IVars, with three operations: Creation (`glob`) by globalising a local IVar, remote write (`rput`), and information (`at`) about the location of the underlying IVar. To ensure that the values written by `rput` are serialisable, these operations restrict the base type of their underlying IVars to closures. Hence all values transported between nodes, be it computations or results, are closures — so results may again be computations. Note that there is no remote read on global IVars — in this respect they are much like channels in Eden and CloudHaskell, supporting remote write but only local read.

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

```
fib :: Int -> Int
fib n
 | n <= 1    = 1
 | otherwise = x + y
     where x = fib (n-1)
           y = 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)
```

```
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)

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 sequential code and shared-memory parallel code in HdpH; to the right GpH code and distributed-memory parallel code in HdpH.

architectures using the GUM RTS. The second variant, `spfib`, uses the shared-memory primitives of the `Par` Monad, and is thus confined to the shared-memory GHC RTS. The third variant, `dpfib`, employs the HdpH primitives and can 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.

The HdpH primitives constitute a deliberately low-level language, much like the GpH primitives. Section 3.3 will show how to build common abstractions on top.

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 [15]. The latter forbids racing `put`; any attempt to do so, i. e., any attempt to `put` into an already filled IVar, will abort the program in the name of determinism. 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 `put`s aren't fatal. The price we pay for the more flexible semantics is that `Par` computations remain confined to the monadic world. HdpH only offers the monadic function `runParIO :: Par a -> IO a` to extract `Par` computations, in contrast to the pure `runPar :: Par a -> a` of [15].

There are good reasons for accepting non-determinism in the distributed-memory setting. Firstly, for a distributed computation to survive node failures, it must be able to speculatively restart supposedly failed tasks. Such a speculative restart must share the global IVar expecting its result with the original task, opening up a race if the original task happens to be alive (e. g., because the executing node didn't fail but was temporar-

ily unreachable or unresponsive). Secondly, many distributed algorithms are non-deterministic by nature — insisting on determinism would severely limit expressiveness.

3.2 Explicit Closures

CloudHaskell [6] introduced the idea of making a thunk `thk` serialisable[2] 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 serialisable as its code address. The latter trick requires a Haskell extension: a primitive type constructor `Static` for reflecting code addresses of 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.[3] 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. 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 the CloudHaskell `Closure` constructor 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

[2] In keeping with the programming languages community, we take *serialisation* to mean the process of encoding in-memory data structures into byte strings to be sent over the network. This is not to be confused with the notion of *serialisability* in concurrency theory.

[3] Like CloudHaskell, we emulate `Static` support by requiring the programmer to register all `Static` functions in a lookup table.

```
data Closure a = UnsafeMkClosure
                   a                     -- actual thunk
                   (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 thk = (unstatic fun) env
           return $ UnsafeMkClosure thk fun env

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

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

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

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

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 introduction.[4] We use strategies to force the evaluation of a closure (Sect. 3.3), and will address the other issues while introducing the enhanced HdpH closure representation in Fig. 3.

Dual closure representation. To avoid unnecessary serialisation HdpH maintains a dual representation of closures, extending CloudHaskell's closure representation with the actual thunk being represented, see the first argument of `UnsafeMkClosure` in Fig. 3. This representation avoids unnecessary serialisation 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, the actual thunk `thk` and its serialisable representation `fun` and `env`, 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 thunk `thk` 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 the reconstruction of `thk` 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 thunk `thk`. Normalising `thk`, 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`

[4] 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.

and `unClosure clo` would not have the same strictness properties if the actual thunk `thk` were normalised.

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 types, i. e., instances of class `Binary`. The function `toClosure` simply pairs a serialised value with the appropriate static deserialiser which exists thanks to the `Binary` context. Note that `toClosure` lazily delays serialising its argument until the resulting `Closure` is serialised or normalised. In particular, `unClosure . toClosure` is an identity that does not involve serialisation.

A more general safe `Closure` construction generates the arguments to the constructor `UnsafeMkClosure` automatically by macro expansion, using Template Haskell. This is done by function `mkClosure :: Q Exp -> Q Exp`, which safely converts a *quoted* thunk, 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 `$(...)`. To explain what `mkClosure` does, we show what the call in Fig. 2 expands to.

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

To start, `mkClosure` finds the variables captured by the given thunk and packs them into a tuple, here `(gv, t, n)`. Then, it constructs the explicit `Closure` expression `(UnsafeMkClosure thk fun env)`, where `thk` is the given thunk, `env` is the serialised environment, i. e., the serialised tuple of captured variables, and `fun` is a static deserialiser. The latter is actually the code address of a wrapper around the given thunk, abstracting over its serialised environment. That is, the wrapper is a λ-abstraction whose body is the given thunk 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|])
```

This implementation of `fmap`, however, does not compile because the argument to `mkClosure` captures the function `f`, requiring `f` to be serialisable. Yet arbitrary functions are not serialisable — that is the very reason for introducing the `Closure` type.

```
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 = unClosure clo' `deepseq` return clo'
  where clo' = 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 `Closure`s. Note how `mapClosure` is implemented in terms of `Closure` elimination and introduction; that is why our efforts in curbing 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 `Closure`s to functions on `Closure`s, see Appendix A.

3.3 Strategies and Skeletons

Directly using coordination primitives, like those in Sect. 3.1, does introduce parallelism but obscures code by intertwining computation and coordination aspects. To disentangle coordination and computation we aim to develop higher-level abstractions over the primitives, and Fig. 4 shows some simple examples.

Strategies are compositional building blocks for coordination developed for GpH in [19,14]. Following [14], 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 [14] 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 [14] 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.

Since all distributed-memory parallelism in HdpH involves explicit `Closure`s, we focus on `Closure` strategies. The most basic of these is `forceClosure` (see Fig. 4), which fully normalises the thunk inside a `Closure`. It does so by eliminating the `Closure` with `unClosure`, then converting the resulting thunk into a new `Closure` with `toClosure`, before normalising the thunk with `deepseq` and returning the new closure. Note how this is different from just `deepseq`ing the thunk and returning 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 all elements of a list in parallel. The list elements are of type `Closure a`, so we expect an argument of type `Strategy (Closure a)`; however, the strategy argument needs to be serialised itself, 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 Fig. 2. The strategy `parListNF`, which fully normalises 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 polymorphic closure transformations like `mapClosure` and polymorphic strategies like `parListNF`, we can build simple skeletons like GpH does. For example, Fig. 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 Appendix B.

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 muddied in the rest of the function that deals with the necessary closure conversions and eliminations.

4 Implementation Design

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, i. e., mutable data structures in Haskell, and agents, i. e., Haskell IO threads. Each node runs several thread schedulers, typically one per core. Each scheduler owns a dedicated thread pool (a concurrent deque) that may be accessed by other schedulers for stealing work. Each node runs a message handler, which shares access to the spark pool (another concurrent deque) with the schedulers. Each node also has a registry (a concurrent map) of global IVars that is shared between message handler and schedulers.

Inter-node communication is abstracted into a *communication layer*, that provides startup and shutdown functionality, node IDs, and seamless peer-to-peer send/receive

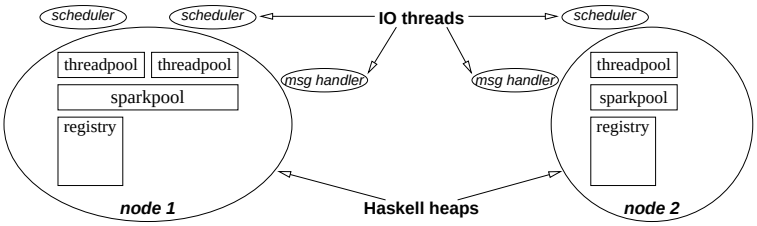

Fig. 5. HdpH system architecture; coupling a dual core and a uni-core node

of arbitrarily sized byte strings. Currently this layer is based on MPI; we plan ports to other network protocols with better support for fault tolerance.

Thread scheduling is based on the work-stealing scheme of the `Par` Monad [15], 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 (Fig. 6). A global reference records the type of the referred-to object as a phantom type, i. e., `ref :: GRef t` refers to an object of type `t`. A global reference is represented by a pair of a node ID identifying the host of the referred-to 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 referred-to object is established by the *registry*, a concurrently mutable lookup table, much like 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 `deref`erencing it). However, to avoid having to implement a global garbage collection, we add a third operation for `free`ing

```
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 clo' (at gv)
                where clo' = $(mkClosure [|lift (deref gv) >>=
                                           maybe
                                             (return ())
                                             (λv -> put v clo >> lift (free gv))|])
```

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

a global reference. Thus, we are faced with the problem that a global reference may be dead (because it has been `free`d earlier) when we attempt to `deref`erence it, which explains why `deref` returns a `Maybe` type.

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

Global references aren't exposed in HdpH. Instead they serve to implement *global IVars*: a `GIVar` is simply a global reference to an IVar (Fig. 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 IVar referred to by `gv`. That computation dereferences the global reference and, depending on the outcome, either returns immediately (in case the global reference was dead) or else writes `clo` to the referred-to IVar and `free`s the global reference `gv`. Note that the action 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 re-implements the spark management of GUM [20] at the Haskell level. Each node stores *sparks*, i. e., values of type `Closure (Par ())`, in a pool. Sparks enter the pool either on being `spark`ed or on being received in a SCHEDULE message. Sparks leave the pool either to be turned into local threads (by eliminating the `Closure`), or to be SCHEDULEd on another node, which entails serialising the `Closure`. Currently the spark selection strategy is purely age-based: the youngest ones are turned into threads, the oldest ones are SCHEDULEd 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 threshold, the FISH expires and a NOWORK message is returned to its originating node, which will wait for some time before sending the next FISH.

Executing `pushTo clo node` sends a PUSH message containing `clo` to `node`. 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 `fork`ing a thread.

nodes	cores	Fibonacci runtime	Fibonacci error	Fibonacci speedup	SumEuler (prim) runtime	SumEuler (prim) error	SumEuler (prim) speedup	SumEuler (parMap) runtime	SumEuler (parMap) error	SumEuler (parMap) speedup
sequential		424.58s	6%		355.69s	8%		*see columns to the left*		
1	6	75.64s	1%	5.6	62.31s	< 0.5%	5.7	62.65s	< 0.5%	5.7
2	12	42.29s	< 0.5%	10.0	32.72s	< 0.5%	10.9	32.71s	< 0.5%	10.9
3	18	28.32s	1%	15.0	22.07s	< 0.5%	16.1	22.14s	1%	16.1
4	24	20.25s	< 0.5%	21.0	16.42s	1%	21.7	16.47s	< 0.5%	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	< 0.5%	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

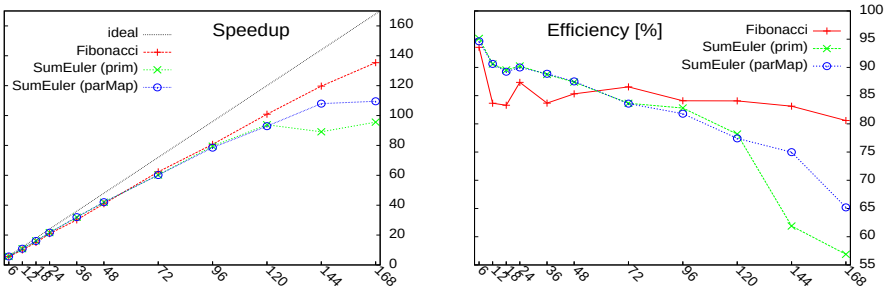

Fig. 7. Results of three benchmarks: runtime, absolute speedup and efficiency

4.3 Current Limitations of the HdpH Implementation

At the time of writing, two issues handicap the usability of HdpH. Firstly, the lack of `Static` support in GHC necessitates work-arounds that bloat the number of top-level declarations and burden the programmer with explicit `Static` registration. Secondly, the MPI-based communication layer is performing poorly on large messages, which severely affects data-intensive applications.

5 Preliminary Performance Results

To investigate the scalability and efficiency of HdpH we have benchmarked three simple parallel programs 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 0.3.0) 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; to limit variability we used only 6 cores per node. 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 Fig. 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 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 `[1..65536]` and reduce the result to a sum. Both are regular, 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 Appendix B.

Both SumEuler benchmarks scale worse than Fibonacci, with efficiency declining faster and maximum speedup 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,[5] 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 reason for this single-node efficiency boost is that HdpH completely avoids serialisation overheads when running on a single node.

6 Conclusion and Future Work

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 (Sect. 3). The HdpH implementation is designed for maintainability. It does not rely on a bespoke low-level RTS but is implemented in Concurrent Haskell as supported by the GHC (Sect. 4). Initial performance results for simple benchmarks are promising with good efficiency and absolute speedups (Sect. 5).

HdpH is still a young project, and development continues in several directions. We are experimenting with fault tolerant skeletons, e. g., task farms that guarantee evaluation of all tasks despite repeated node failures. We are exploring how to extend HdpH towards fully-fledged distributed programming, specifically how to handle distributed data and exceptions in the style of GdH [18]. We are also developing a profiler to analyse HdpH programs as well as the HdpH implementation. Finally, we plan to benchmark HdpH on realistic problems and compare its performance to other parallel Haskells.

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 [10]. Key requirements for SymGridPar2 are the scalability and reliability supported by the HdpH distributed-memory programming model.

Acknowledgements. Thanks to Andrew Black, Jeff Epstein, Hans-Wolfgang Loidl, and Rob Stewart for stimulating discussions. This research is supported by the projects

[5] The speedup and efficiency graphs suggest that the `parMap`-based SumEuler outperforms the other beyond 120 cores, but the measurement errors are too high to support such a conclusion.

HPC-GAP (EPSRC EP/G05553X), SCIEnce (EU FP6 RII3-CT-2005-026133), and RELEASE (EU FP7-ICT 287510).

References

1. Armstrong, J.L., Virding, S.R., Williams, M.C., Wikstrom, C.: Concurrent Programming in Erlang. Prentice-Hall, 2nd edn. (1996)
2. Berthold, J.: Explicit and implicit parallel functional programming: concepts and implementation. Ph.D. thesis, Philipps-Universität Marburg, Germany (2008)
3. Berthold, J., Al Zain, A., Loidl, H.W.: Scheduling light-weight parallelism in ArTCoP. In: PADL 2008, San Francisco, USA. pp. 214–229. LNCS 4902, Springer (2008)
4. Chakravarty, M.M.T., Leshchinskiy, R., Peyton-Jones, S.L., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: DAMP 2007, 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.L.: Towards Haskell in the cloud. In: Haskell 2011, Tokyo, Japan. pp. 118–129. ACM Press (2011)
7. Grelck, C., Scholz, S.B.: SAC - a functional array language for efficient multi-threaded execution. International Journal of Parallel Programming 34(4), 383–427 (2006)
8. Harrison, W.L.: The essence of multitasking. In: AMAST 2006, Kuressaare, Estonia. pp. 158–172. LNCS 4019, Springer (2006)
9. Haskell distributed parallel Haskell, `https://github.com/PatrickMaier/HdpH`
10. HPC-GAP: High Performance Computational Algebra and Discrete Mathematics, `http://www-circa.mcs.st-andrews.ac.uk/hpcgap.php`
11. Klusik, U., Ortega-Mallén, Y., Peña, R.: Implementing Eden — or: Dreams become reality. In: IFL 1998, London, UK. pp. 103–119. LNCS 1595, Springer (1999)
12. Li, P., Marlow, S., Peyton-Jones, S.L., Tolmach, A.P.: Lightweight concurrency primitives for GHC. In: Haskell 2007, Freiburg, Germany. pp. 107–118. ACM Press (2007)
13. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. J. Funct. Program. 15(3), 431–475 (2005)
14. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.W.: Seq no more: Better strategies for parallel Haskell. In: Haskell 2010, Baltimore, USA. pp. 91–102. ACM Press (2010)
15. Marlow, S., Newton, R., Peyton-Jones, S.L.: A monad for deterministic parallelism. In: Haskell 2011, Tokyo, Japan. pp. 71–82. ACM Press (2011)
16. Marlow, S., Peyton-Jones, S.L., Singh, S.: Runtime support for multicore Haskell. In: ICFP 2009, Edinburgh, Scotland. pp. 65–78. ACM Press (2009)
17. Peyton-Jones, S.L., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL 1996, St. Petersburg Beach, USA. pp. 295–308 (1996)
18. Pointon, R.F., Trinder, P.W., Loidl, H.W.: The design and implementation of Glasgow distributed Haskell. In: IFL 2000, Aachen, Germany. pp. 53–70. LNCS 2011, Springer (2001)
19. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton-Jones, S.L.: Algorithms + strategy = parallelism. J. Funct. Program. 8(1), 23–60 (1998)
20. 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 1996, Philadelphia, USA. pp. 79–88. ACM Press (1996)
21. Trinder, P.W., Loidl, H.W., Pointon, R.F.: Parallel and distributed Haskells. J. Funct. Program. 12(4&5), 469–510 (2002)
22. Wiger, U.: What is Erlang-style concurrency?, `http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/`

A Categorical Structure on Closures

We've seen in Section 3.2 that `Closure` isn't a functor, yet `mapClosure` looks suspiciously similar to `fmap`. Why is this? It turns out that `mapClosure` is indeed the morphism map of a functor into the subcategory induced by the `Closure` type constructor — the functor just does not originate in the standard category of Haskell types (which is required for instances of the `Functor` class).

Let **Hask** be the standard category of Haskell types, whose objects are Haskell types and whose morphisms are Haskell functions, with the standard composition `(.)` and unit `id`. Let **Closure** be the full subcategory of **Hask** induced by the `Closure` type constructor, i. e., objects are types of the form `Closure t`, for some type `t`, and morphisms, composition and unit are inherited from **Hask**. Now let **Clo** be a different category of closures whose objects are the objects of **Closure** (i. e., all types of the form `Closure t`) but whose morphisms are function closures. That is, a morphism $f : \texttt{Closure s} \to \texttt{Closure t}$ in **Clo** is a closure of type `Closure (s -> t)`. We provide unit `idClosure` and composition `compClosure` as defined below. It is easily verified that `idClosure` and `compClosure` satisfy the identity and associativity laws, making **Clo** a category indeed.

```
idClosure :: Closure (a -> a)
idClosure = $(mkClosure [|id|])

compClosure :: Closure (b -> c) -> Closure (a -> b) -> Closure (a -> c)
compClosure clo_g clo_f = $(mkClosure [|unClosure clo_g . unClosure clo_f|])
```

Now, we can define a functor F from **Clo** to **Closure** whose object map is the identity and whose morphism map is `mapClosure`, mapping any morphism in **Clo** (i. e., any function closure) to the corresponding morphism in **Closure** (i. e., lifting the function closure to a function on closures). Remains to show that F preserves identity and distributes over composition, which amounts to showing the validity of the following equations (again easily verified).

```
                  mapClosure idClosure = id
mapClosure (clo_g `compClosure` clo_f) = mapClosure clo_g . mapClosure clo_f
```

B Code for Sum of Euler's Totients

```
sumeuler_prim :: Int -> [Int] -> Par Integer
sumeuler_prim sparks xs = sum <$> (mapM join =<< mapM spark_sumeuler xss)
  where xss = deal sparks xs

        spark_sumeuler :: [Int] -> Par (IVar (Closure Integer))
        spark_sumeuler xs = do
          v <- new
          gv <- glob v
          spark $(mkClosure [|eval (sumeuler xs) >>= rput gv . toClosure|])
          return v

        join :: IVar (Closure Integer) -> Par Integer
        join = return . unClosure <=< get

sumeuler_parMap :: Int -> [Int] -> Par Integer
sumeuler_parMap sparks xs = sum <$> parMap $(mkClosure [|sumeuler|]) xss
  where xss = deal sparks xs
```

```
sumeuler :: [Int] -> Integer
sumeuler = sum . map totient

totient :: Int -> Integer
totient n = toInteger $ length (filter ( k -> gcd n k == 1) [1 .. n])

deal :: Int -> [a] -> [[a]]
deal players = Data.List.transpose . chunk players

chunk :: Int -> [a] -> [[a]]
chunk size [] = []
chunk size xs = ys : chunk size zs where (ys,zs) = splitAt size xs
```