

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

Patrick Maier, Phil Trinder, and Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Abstract. We present 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. 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. As such it represents a middle ground between monolithic RTS like GUM and Eden/EDI, and kernel-based proposals. Initial performance results are promising with absolute speedups of about 23 on 30 Beowulf nodes for simple programs.

1 Introduction

Parallel functional languages (PFLs) combine high-level computation with high-level coordination. High level coordination implies that the implementation must manage many low-level coordination aspects such as thread creation, placement, and the communication and synchronisation between threads. To do so effectively the language implementation must be sophisticated.

Early effective PFL implementations exploited monolithic and low-level runtime systems, for example the GUM runtime system for GpH [21] and the Dream and EDI systems for Eden [10,3]. Some recent PFL RTS also take this approach, for example multicore GHC [16]. Major issues with this approach are the complexity of the RTS, and it's maintainability as the sequential language implementation evolves.

Concurrent and parallel RTS are now sufficiently complex that it has been proposed that they are implemented using a kernel, cf. [11,2]. Here there is just a small low-level kernel to maintain, and as the majority of the system is in a high-level language it is faster to develop and easier to modify and maintain. However this approach has not yet been successful, largely due to the performance overheads.

However, there is a recent trend towards implementing parallel RTS using a concurrent declarative language as a systems language. Examples include Data parallel Haskell [4], CloudHaskell [6], and the `Par` monad [15]. The advantages of ease of development and maintenance are broadly similar to the kernel approach.

The new *Haskell distributed parallel Haskell (HdpH)* language we propose here takes this third approach, using Concurrent **Haskell** as the systems language to implement a **distributed-memory parallel Haskell** extension. HdpH provides semi-explicit

distributed-memory parallel coordination by combining an extended version of the closure serialisation from CloudHaskell [6] with an extended version of the `Par` parallelism monad from [15]. The high-level coordination is implemented using proven technologies from GUM [21] like global addresses and randomised work stealing.

The paper makes the following research contributions:

- We present the initial design of the new Haskell distributed parallel Haskell (HdpH) language, with the following key features. HdpH supports *high-level semi-explicit parallelism*, and provides evaluation strategies and algorithmic skeletons defined using a small set of coordination primitives (examples provided). HdpH is *scalable*, supporting execution on distributed memory platforms. HdpH has the potential for fault tolerance 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 (Section 3).
- We outline the HdpH implementation design which aims to deliver acceptable performance while remaining maintainable. Crucial to maintainability the implementation uses vanilla (GHC) Concurrent Haskell as a systems language. Moreover the implementation is layered and modular with coordination aspects such as communication, thread management, global address management, scheduling etc. realised in independent modules (Section 4).
- We report initial performance results for simple data parallel and divide-and-conquer programs on a 32 node, 256 core Beowulf cluster (Section 5).

2 Related Work

This section outlines other parallel functional languages that have influenced the design or implementation of HdpH. As HdpH is primarily control-oriented, we do not consider data oriented parallel languages like DpH: [4] or SAC [8] here. However a comprehensive survey of parallel functional languages is available in [20].

2.1 Shared-Memory Languages

Haskell extensions for explicit concurrency have been designed and implemented as early as 1996 [17], targeting in particular GUI applications. More recently the focus of attention shifted to distributed applications. One salient feature of GHC’s support for concurrency is the low overhead associated with thread creation, providing light-weight threads.

While explicit threads provide a suitable mechanism for describing separate, stateful computations, the necessity of explicit control code makes them less suitable for defining parallel, stateless computations. To provide a higher level of abstraction, the `Par` monad [15,13] 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 details of the parallel execution hidden from the programmer [18]. The specification of parallelism

is less intrusive in this notation. However, efficient use of parallelism requires to also specify evaluation order and evaluation degree. To this end, the abstraction of evaluation strategies has been developed [19,14]. It uses the type class `NFData` to define a function that performs full evaluation of a data structure. Support for these parallel extensions on multi-core machines is included with the default version of GHC [16], and evaluation strategies are available from the `parallel` package.

2.2 Distributed-Memory Languages

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 concurrent or distributed computing. That is, it has influenced the design of the concurrent coordination provided in many languages and frameworks, for example Scala, F#, and most relevantly CloudHaskell [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 [22].

Eden extends Haskell with distributed memory parallelism [12]. 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.

A recent development that heavily influenced our work, was the design and implementation of CloudHaskell [6,7]. It explicitly targets distributed memory systems, and implements 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

Most implementations of the above systems (GUM for distributed memory GpH [18], Dream/EDI for distributed memory Eden [12] and the threaded runtime system for shared-memory GpH [15]) use a monolithic runtime system, providing support for parallelism. In an area where performance is the main issue, the usage of a low-level systems language and the avoidance of several layers, that might impact performance, is natural. However, the main shortcoming of this approach is that maintaining the implementation is difficult and internal changes to the runtime system, e. g., to the structure of closures, may directly impact the parallel implementation.

In contrast, the implementation of CloudHaskell [6] leaves the runtime system unchanged, and implements all functionality on the Haskell level. This ensures ease of

maintainability, as well as more readable code by using Haskell’s advanced abstraction mechanisms. For high performance it uses GHC’s light-weight threads.

3 Initial Language Design

We present an initial design for a (shallowly embedded) distributed-memory parallel extension of Haskell with the following additional features.

- Parallelism is semi-explicit, i. e., potentially parallel code is marked as such but the system decides at runtime when and where to execute such code; there is no guarantee of parallel execution.
- The language admits high-level abstractions, e. g., polymorphic skeletons (Section 3.3).
- The whole language can be implemented on top of the current GHC; no special runtime system is required.

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 [15] 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.

3.1 Primitives

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

To recap [15]: The `Par` type constructor is a monad for encapsulating a parallel computation. The basic primitive for generating shared-memory parallelism is `fork`,

```
data Par a -- par comp yielding an 'a'      data Closure a -- closure yielding an 'a'
fork :: Par () -> Par ()                    spark :: Closure (Par ()) -> Par ()

data IVar a -- IVar expecting an 'a'       data GIVar a -- global IVar expecting an 'a'
new  :: Par (IVar a)                       glob :: (NFData a, Binary a) => IVar a -> Par (GIVar a)
get  :: IVar a -> Par a
put  :: NFData a => IVar a -> a -> Par ()   rput :: (NFData a, Binary a) => GIVar a -> a -> Par ()
```

Fig. 1. HdpH primitives. LHS: Primitives for shared memory inherited from [15]. RHS: Primitives for distributed memory.

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` has an `NFData` context because it fully evaluates its second argument before writing it to the `IVar`.

The functions of most distributed-memory primitives resembles those of their shared-memory counterparts. They also live in the `Par` monad which we have suitably extended (details in Section 4.5).

The basic primitive for generating distributed-memory parallelism is `spark`. It operates much like `fork`, generating a thread (also known as *spark*) that may be executed elsewhere (i. e., on a different host, or in a different Haskell RTS). 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 a closure first; we'll present the operations for that in Section 3.2 below.

Remains the problem of communicating the results of computations across hosts. To this end, we introduce *global IVars*. For the moment, one may think of a `GIVar` as the global address of an `IVar` with two operations: Creation (`glob`) by globalising a local `IVar`, and remote write (`rput`). Since the latter needs to serialise the value to be written, it requires a `Binary` context besides the `NFData` context inherited from `put`. The reason why `glob` requires the same context will become apparent in Section 4.3 which details the implementation of global `IVars`. Note there is no remote read on a `GIVar` — in this respect they are much like channels in Eden and CloudHaskell, supporting remote writes but only local reads.

```

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)

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

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

dpfib :: Int -> Int -> Par Int
dpfib t n
  | n <= t    = return $ fib n
  | otherwise = do
    v <- new
    gv <- glob v
    let val = dpfib t (n-1) >>= rput gv
        let env = encode (gv, t, n)
            let fun = static (\env ->
                let (gv, t, n) = decode env
                    in dpfib t (n-1) >>= rput gv)
        spark (unsafeMkClosure val fun env)
    y <- dpfib t (n-2)
    x <- get v
    return (x + y)

```

Fig. 2. Fibonacci numbers. LHS: GpH code and shared-memory parallel code in the `Par` monad. RHS: sequential code and distributed-memory parallel code in the `Par` monad.

To demonstrate the use of the primitives, we present three variants of computing the Fibonacci function on integers in parallel, see figure 2. All three parallel variants there take two arguments: the second argument being the actual argument to the Fibonacci

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

The first variant, `pfib`, is based on the GpH primitives `par` and `pseq`. It can be executed either on a shared-memory multicore (using e. g., the GHC RTS) or on a distributed-memory cluster (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 distributed-memory primitives and can thus be executed on shared- and distributed-memory architectures (using the GHC RTS plus a communication layer, e. g., MPI).

Comparing `spfib` and `dpfib`, the similarities are obvious. The difference is that `spfib` can simply fork the recursive call `spfib t (n-1) >>= put v`, whereas `dpfib` must globalise the `IVar v`, yielding global `IVar gv`, and spark a `Closure` wrapping the recursive call `dpfib t (n-1) >>= rput gv`.

The steps involved in converting the term `dpfib t (n-1) >>= rput gv` into a `Closure` are outlined below, more details on closure construction in Section 3.2.

1. Let-bind the given term to `val`.
2. Identify the non-toplevel free variables of the given term, pack them into a tuple — here `(gv, t, n)` — and serialise, producing the serialised environment `env`.
3. Write a wrapper which abstracts the given term over its serialised environment. That is, the wrapper is a function whose body is `dpfib t (n-1) >>= rput gv`, yet the non-toplevel free variables `gv, t` and `n` are now let-bound as a result of deserialising the parameter `env`.
4. Apply `static` to the wrapper, producing the static deserialiser `fun`.
5. Finally, package `val, fun` and `env` into a `Closure` by applying the constructor `unsafeMkClosure` (the use of which is perfectly safe here).

Currently, this explicit closure conversion is cumbersome and error prone (e. g., accidentally swapping `t` and `n` in the let-binding in the wrapper). However, much of this code could be generated by Template Haskell, as demonstrated by the CloudHaskell project.

3.2 Closures

CloudHaskell [6] introduced the idea of making a computation serialisable (at the language level) by converting it into an *explicit closure*, represented as a pair consisting of an (already serialised) environment together with a *static* (hence serialisable) function which deserialises the environment and performs the computation. Below, we reproduce the closure representation of CloudHaskell via a type constructor `Closure`, which is parametrised by the type of the actual computation it represents.

```
data Closure a = MkClosure (Static (Env -> a)) Env

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

Here, `Env` is used as a synonym for the type of lazy byte strings. The `Static` type constructor and its associated term formers `static` and `unstatic` are described in detail in the CloudHaskell paper. Though not (yet) implemented in GHC 7, we proceed with the language design in this section as if `Static` were fully supported; Section 4.1

sketches some work-arounds to achieve (rather clunky) support for `Static` within the current GHC (similar to how CloudHaskell does this).

The only operations on closures that CloudHaskell exposes (apart from serialisation) are introduction (via the constructor) and elimination (via `unClosure`). Note that introduction is lazy, i. e., will delay serialising the environment until demanded, either by the closure itself 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 actual closure representation. And 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. However, the `Closure` type constructor ought to aspire to more. It ought to support proper computation with, rather than just transportation of, closures. Ideally, `Closure` ought to be a *functor*, thus allowing to transform closures without eliminating them. On top of that, 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 introduction¹ (which in CloudHaskell's representation triggers serialisation of the environment). Forcing will be achieved by using *strategies*, cf. next section; we will address the other issues now while introducing our enhanced closure representation, presented in Figure 3.

Cutting serialisation overhead. Our `Closure` type constructor `C` has 3 arguments, where the second and third are familiar from CloudHaskell's closure representation. The first argument is the actual closure being represented, which solves the problem of unnecessary serialisation, because closure elimination is just a projection on the first argument, involving no serialisation.

Thus, `Closure` maintains a dual representation of an actual closure, which implies the obligation to maintain the invariant that the two representations remain in sync. The `Binary` instance does this by serialising only the explicit representation (i. e., the second and third constructor arguments), re-constructing the actual value upon deserialisation by eliminating the representation (in the same way as CloudHaskell does closure elimination); note that this re-construction is lazy, hence delayed until the whole `Closure` is eliminated.

The only place where it is not possible to guarantee the invariant is the `Closure` constructor `C`, which is why this constructor is not exposed. Instead, we expose the alias `unsafeMkClosure`, implying the programmer's obligation to maintain the representation invariant. A safe way to create a closure is via `toClosure`, albeit only for serialisable values, pairing a serialised value with a suitable static deserialiser (which exists thanks to the `Binary` context).

Finally, we stipulate the following contract for `NFData/Binary` instances: The functions `rnf` and `(rnf . encode)` must have the same strictness properties. This serves to make the primitives `put` and `rput` exhibit the same level of strictness, and

¹ 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.

```

type Env      -- serialised environment: lazy byte string

data Static a -- serialisable static value of type 'a'
instance Binary (Static a) where ...
instance NFData (Static a) where ...

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

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

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

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

toClosure :: (Binary a) => a -> Closure a
toClosure val = C val fun env where env = encode val
                                      fun = static decode

unsafeMkClosure :: a -> Static (Env -> a) -> Env -> Closure a
unsafeMkClosure val fun env = C val fun env

mapClosure :: Closure (a -> b) -> Closure a -> Closure b
mapClosure clo_f clo_x = unsafeMkClosure val fun env
  where
    val = unClosure clo_f $ unClosure clo_x
    env = encode (clo_f, clo_x)
    fun = static (\env -> let (clo_f, clo_x) = decode env
                          in unClosure clo_f $ unClosure clo_x)

```

Fig. 3. Closure representation and operations on closures.

also explains why the `NFData` instance for closures forces only the closure representation but not the actual closure value.

Transforming closures. Figure 3 shows the functor-like closure 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 reducing the serialisation overhead are relevant.

Why do we not make `Closure` an instance of `Functor` by implementing the standard `fmap :: (a -> b) -> Closure a -> Closure b` instead of `mapClosure`? Looking at the code of `mapClosure` the answer is obvious: To construct the result closure, both its arguments need to be serialised into the environment — which rules out plain function arguments. Still, `mapClosure` 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 Appendix A for details.

3.3 Strategies and Skeletons

Direct use of the primitives in Section 3.1, while sufficient to effectively parallelise code (as demonstrated on the Fibonacci example), tends to bloat the resulting parallel code and obscure the actual algorithm. With the aim of separating parallel coordination from actual computation, we develop higher-level abstractions on top of the primitives, see the code in Figure 4.

```
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'@(Closure 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
                  let val = (clo `using` unClosure clo_strat) >>= rput gv
                      let env = encode (clo, clo_strat, gv)
                          let fun = static (\env ->
                                          let (clo, clo_strat, gv) = decode env
                                              in (clo `using` unClosure clo_strat) >>= rput gv)
                      spark $ unsafeMkClosure val fun env
                  return v

parListNF :: (Binary a, NFData a) => Strategy [Closure a]
parListNF = parList (unsafeMkClosure val fun env) where val = forceClosure
                                                         env = encode ()
                                                         fun = static (\_ -> 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. Strategies for closures and skeleton parMap.

Strategies are compositional building blocks for coordination (without specifying the computation) and were introduced for the GpH programming model in [19,14]. Following the approach of [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) which may cause evaluation (sequentially or in parallel) 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 (via `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 are expressible still.

Since all (distributed-memory) parallelism in HdpH involves explicit closures, we focus on closure strategies. The most basic of these is `forceClosure`, which *forces* a closure of type `a` (provided there is a suitable `Binary/NFData` context). It does so by eliminating the closure (via `unClosure`) and converting the resulting value into another closure (via `toClosure`), the actual value of which is then forced (via `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. Here, 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 code for `spawn`), so it must be wrapped in another `Closure`. The implementation of `parList` is straightforward: Spawn all strategy applications (via `mapM spawn`) producing a list of `IVars`, then read the results back (via `mapM get`); the structure of the code for `spawn` itself is similar to that of `dpfib` in Figure 2.

The strategy `parListNF`, which forces a list of closures in parallel, is derived by “applying” `parList` to `forceClosure`. Technically, this involves converting `forceClosure` into a trivial closure (note the empty environment) to satisfy `parList`’s argument restriction.

Skeletons are (usually 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 much like is done 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 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` part), though it is muddled by the remaining code (the purpose of which is to deal with all the closure conversions and eliminations forced upon us by using explicit closures).

3.4 Comparison of HdpH Language Features

The table below sets out how HdpH differs from other parallel Haskell: the shared-memory `Par` monad, GpH both with its SMP and its GUM RTS, Eden and CloudHaskell.

	Par (SMP)	GpH-SMP	GpH-GUM	Eden	CloudHaskell	HdpH
distributed memory	–	–	+	+	+	+
separate heaps	–	–	–	+	+	+
polymorphic closures	+	+	+	+	–	+
lazy work stealing	+	+	+	?	–	+
standard RTS	+	+	–	–	+	+

4 Implementation Design

To achieve modularity, the implementation of HdpH is designed as a stack of monads on top of the IO monad. We'll describe the layers of the stack in the following sections, starting at the bottom.

4.1 Support for Static

This layer is a hack (following ideas from the CloudHaskell project) to provide the support for `Static` that is currently missing from GHC. `Static` values are represented as unique names (hence serialisable), so that the argument of the `Static` type constructor degenerates to a mere phantom type tracking the type of an associated static term. The functionality of `unstatic` is achieved by lookup into the *static table*, which is similar to the reference table described in Section 4.3. The main difference is that the static table is initialised once at the beginning of the program and stays immutable thereafter, so there are no issues with concurrent access or dead references.

While `unstatic` behaves as it should (with a little help from `unsafePerformIO`) it is not possible to emulate the term former `static` exactly. Instead of 'synthesising' the `Static` value of a given term, the programmer has to explicitly assign a name (and ensure its uniqueness). Moreover, the `main` function must collect all `Static` values declared throughout the whole program (including all transitively imported modules) and initialise the static table explicitly.

4.2 Communication layer

This layer wraps a standard MPI library, providing startup and shutdown functionality, node IDs, and seamless communication between nodes. The objects communicated are lazy bytestrings of arbitrary (potentially even infinite) size.

4.3 Global References

Global references provide a type-safe way of accessing remotely hosted objects. Figure 5 presents the core of our implementation of global references. At the type level, a global reference tracks the type of its referenced object via a phantom type, i.e., `ref :: GRef t` is pointing to an object of type `t`. At the value level, a global reference is represented by a pair consisting of a node ID identifying the *host* of the referenced object and a name that is unique on that host (over the life span of the Haskell RTS). This yields unique global references with easy access to the host info (by calling `at`). Serialisation and normalisation is straightforward, relying on the respective properties of `NodeId` and `UniqueName`.

The link between a global reference (whose host is the current node) and its referenced object is established by a mutable *reference table*, much like the GALA table in the GUM RTS (except that the reference table is implemented in Haskell). The fact that the table is mutable (even concurrently) means that operations on global references generally need to live in a monad, the `GRefM` monad, and be atomic. There are two

```

type GRefRep = (NodeId, UniqueName)          data GRefM a -- global reference monad
data GRef a = GRef GRefRep                  globalise :: a -> GRefM (GRef a)
instance Binary (GRef a) where ...          deref :: GRef a -> GRefM a
instance NFData (GRef a) where ...         free :: GRef a -> GRefM ()
at :: GRef a -> NodeId

```

Fig. 5. Global references and operations in the GRefM monad.

basic operations on global references: (1) Introducing a fresh one (by `globalise`ing a local object) and (2) eliminating an existing one (by `deref`encing it). However, to avoid having to implement a global garbage collection, we add a third operation for `free`ing 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 `deref`ence it, in which case we abort with a random error. Sounds reckless, yet as we will see, it is consistent with the semantics of the programming model.

```

type GIVar a = GRef (IVar a, Env -> a)
glob :: (NFData a, Binary a) => IVar a -> Par (GIVar a)
glob v = lift $ globalise (v, dec_x)
  where
    dec_x :: Env -> a
    dec_x env_x = x `deepseq` x where x = decode env_x
rput :: (NFData a, Binary a) => GIVar a -> a -> Par ()
rput gv x = do
  here <- lift $ myNode
  if at gv == here
  then do (v, _) <- lift $ deref gv
          lift $ free gv
          put v x
  else pushTo (at gv) (unsafeMkClosure undefined fun env)
  where
    env_x = encode x
    env = encode (gv, env_x)
    fun = static (\env -> let (gv, env_x) = decode env
                          in do (v, dec_x) <- lift $ deref gv
                             lift $ free gv
                             put_ v $ dec_x env_x)

```

Fig. 6. Implementation of global IVars.

Given global references, we can implement *global IVars* and their operations. A `GIVar` is a global reference to a pair consisting of an `IVar` and a deserialiser for the `IVar`'s content; global IVars inherit serialisability from global references.

The operations `glob` and `rput` on global IVars live in the `Par` monad (which sits above the `GRefM` monad); their implementations are shown in Figure 6. Globalising an `IVar` is straightforward: Pair the `IVar` with the normalising deserialiser `dec_x` (which explains the `NFData/Binary` context on `glob`) and `globalise` the pair.

The code for the remote write is more complex. First, it checks whether the host of the given global `IVar` is the current node. If so, it finds the corresponding local

IVar, frees the global IVar and writes the given value to the local IVar, thereby normalising it. Otherwise, it pushes a closure to the host of the global IVar — the function `pushTo :: NodeId -> Closure (Par ()) -> Par ()` being similar to `spark`, except for PUSHing the given closure eagerly to the given target node. Said closure wraps the given global IVar `gv` and the *already serialised* value `env_x`; note that the actual closure value is *undefined* — this is admissible here because the closure is guaranteed to be serialised (ignoring its actual value). When the closure is executed on the host of the global IVar `gv`, it finds the associated local IVar `v` and deserialiser `dec_x` by dereferencing, frees `gv`, *deserialises* `env_x` and writes the result to `v`. The local write uses `put_`, a variant of `put` which is head- rather than hyper-strict, yet the value to be written is still in normalform, thanks to the normalising deserialiser `dec_x`.

Note that dead references are no problem for global IVars. The only way for an `rput` to encounter a dead global IVar `gv` is if `gv` had been written to by an earlier, successful `rput`. However, the semantics of IVars prohibits multiple writes, cf. [15], so aborting on dereferencing a dead global IVar is consistent with the semantics.

4.4 Spark management

To a large extent, spark management is a re-implementation of GUM RTS functionality at the Haskell level. On each node, sparks are closures of type `Par ()` stored in a pool, which they enter either upon being *sparked*, or on being received in a SCHEDULE message. Sparks leave the pool either to be turned into a local thread (by eliminating the closure), or to be SCHEDULEd on another node (which entails serialising the closure). The currently implemented spark selection strategy is purely age-based: The youngest sparks are turned into threads, the oldest are SCHEDULEd away.

When the spark pool is running low, the node sends a FISH message, which may travel across the network (randomly, at present) for a configurable number of hops, looking for work. If a node with excess sparks receives a FISH it replies with a SCHEDULE. If the hops expire before the FISH finds work, a NOWORK message is returned to its sender, who then decides to wait some time before sending the next FISH.

Finally, a node may send a PUSH message containing a spark to be executed immediately on the target node; this feature is used to implement eager remote writes.

4.5 Scheduler

The top layer of the monad stack is our `Par` monad. This is an extension of the `Par` monad in [15], itself a variation of Claessen’s Poor Man’s Concurrency monad [5]. With the monad comes the `runPar :: Conf -> StaticDecl -> Par a -> IO (Maybe a)` function to run a `Par` computation and eliminate the whole monad stack down to `IO`. Apart from the computation, `runPar` expects some configuration parameters (for controlling fishing, etc.) and a declaration of all `Static` values used throughout the computation (to initialise the static table). The return type is wrapped in a `Maybe` because only the main node returns the result of the computation; all other nodes return nothing.

Internally, the `Par` monad hides a scheduler, which works off a two-tier workpool: It prefers to schedule threads, but if there are no runnable threads it will pick a spark and turn it into a thread. If there are no sparks either the scheduler will go to sleep

(only to wake up once there is a spark or a runnable thread). In fact, there may be several schedulers (ideally one per core) running concurrently and sharing access to the workpool. However, at the moment concurrent schedulers cause extreme variance in performance (probably due to contention).

5 Preliminary Performance Results

We report experiments based on two programs. The first is computing the Fibonacci function of 50 using the divide-and-conquer algorithm of Figure 2. The actual call is `dpfib 30 50`, i. e., the granularity threshold is 30, resulting in 17710 sparks. The second is computing the sum of Euler’s totient function between 1 and 65536 using a skeleton-based data parallel algorithm shown in Appendix B. The actual call is `dpsumeuler 1 65536 128`, where the last parameter is the task size, i. e., each spark gets to compute totients of a list of 128 integers, resulting in 512 sparks.

The experiments were undertaken on a Beowulf cluster consisting of 32 hosts, each with 8 cores and 12 GB of RAM, connected via Gigabit Ethernet. The experiments use the following system parameters. With regards to placement we allocate only one core per node (i. e., per GHC RTS).² We do exploit the multicore platform by running several RTS (up to 3) on a single host. For work allocation we allow FISH messages to travel as many hops as there are nodes, thereby ensuring that the random fishing strategy finds work with high probability, even if work is only available at a single node. Moreover, we set the timeout after receiving a NOWORK message to as little as 1 millisecond, thereby making idle nodes fish for work constantly.

nodes		dpfib		dpsumeuler	
nodes	hosts	runtime[s]	speedup	runtime[s]	speedup
sequential		422.2		363.7	
1	1	450.4	0.94	370.7	0.98
2	2	230.7	1.83	189.2	1.92
3	3	155.1	2.72	125.7	2.89
4	4	117.1	3.61	94.8	3.84
6	6	79.1	5.34	65.7	5.54
8	8	59.5	7.10	49.1	7.41
12	12	41.4	10.2	34.1	10.7
16	16	31.2	13.5	26.8	13.6
24	24	21.3	19.8	19.0	19.1
30	30	17.8	23.7	16.3	22.3
45	30	13.8	30.6	13.2	27.6
60	30	12.8	33.0	13.1	27.8
75	30	13.5	31.3	13.7	26.5
90	30	14.6	28.9	16.6	21.9

Fig. 7. Performance results for `dpfib` and `dpsumeuler` on a Beowulf cluster.

² While our design permits to exploit multicores by running several schedulers on the same thread pool, this appears to lead to contention, producing extremely variable runtimes.

Figure 7 summarises the results, where the figures are computed from median run-times (over 3 or 5 runs). We make the following observations.

- Lazy work distribution works well, despite using only random fishing. It works particularly well for a regular divide-and-conquer algorithm like `dpfib`, because this algorithm floods the parallel machine quickly with sparks generating more sparks, so that work is never far to find. Yet, even with a data parallel algorithm like `dpsumeuler`, where only the root node produces sparks, the random fishing strategy appears to find work easily, at least up to 16 nodes; only beyond 16 nodes do the speedups of `dpsumeuler` drop below those of `dpfib`.
- The main reason for the drop in performance beyond 45 nodes is due to the startup and shutdown overhead of the MPI communications layer. The times for this range from about 0.1 second on a single node to 1.0 second on 30 nodes but soar to 2.5, 4.7, 7.1 and 10.4 seconds on 45, 60, 75 and 90 nodes, respectively. Thus, startup/shutdown overhead dominates the runtime beyond 45 nodes.
- The reason for the high overhead when running on a single node is likely down to the fact that measurements were taken before the optimisations for cutting out unnecessary serialisation were implemented. This is particularly obvious when comparing the overheads of `dpfib` and `dpsumeuler`, bearing in mind that the former generates about 30 times as many sparks as the latter.

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. 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. As such it represents a middle ground between monolithic RTS like GUM and Eden/EDI, and kernel-based proposals like [11,2]. Initial performance results are promising with absolute speedups of about 23 on 30 Beowulf nodes for simple data parallel and divide-and-conquer programs.

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 [9]. Key requirements for SymGridPar2 are the scalability and reliability supported by the HdpH distributed memory programming model. To provide reliability we envisage developing Erlang-style fault tolerance abstractions.

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

References

1. Armstrong, J., Virding, 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. 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 — Workshop on Declarative Aspects of Multicore Programming. pp. 10–18. ACM Press, Nice, France (Jan 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 — Symposium on Haskell. ACM Press, Tokyo, Japan (Sep 2011), to appear
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. HPC-GAP: High Performance Computational Algebra and Discrete Mathematics. Web page (Aug 2011), <http://www-circa.mcs.st-andrews.ac.uk/hpcgap.php>
10. Klusik, U., Ortega-Malln, Y., Pea, R.: Implementing Eden - or: Dreams Become Reality. In: IFL'98. Springer Verlag LNCS 1595 (1998)
11. Li, P., Marlow, S., Peyton Jones, S., Tolmach, A.: Lightweight concurrency primitives for ghc (June 2007), Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop
12. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. *J. Funct. Program.* 15(3), 431–475 (2005)
13. Marlow, S.: *Parallel and Concurrent Programming in Haskell, version 1.1 edn.* (Jun 2011)
14. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: Better strategies for parallel haskell. In: Haskell '10 — Symposium on Haskell. pp. 91–102. ACM Press, Baltimore, MD, USA (Sep 2010)
15. Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. In: Haskell '11 — Symposium on Haskell. ACM Press, Tokyo, Japan (Sep 2011), to appear
16. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore Haskell. In: ICFP '09 — Intl. Conf. on Functional Programming. pp. 65–78. ACM Press, Edinburgh, Scotland (Sep 2009)
17. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL'96 — Symposium on Principles of Programming Languages. pp. 295–308. ACM, St Petersburg, Florida (January 1996), http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/concurrent-haskell.ps.gz
18. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96 — Programming Languages Design and Implementation. pp. 79–88. Philadelphia, PA, USA (May 1996), <http://www.cee.hw.ac.uk/~dsg/gph/papers/ps/gum.ps.gz>
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., Loidl, H.W., Pointon, R.F.: Parallel and distributed Haskell. *J. Funct. Program.* 12(4&5), 469–510 (2002), special Issue on Haskell
21. 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. ACM Press (1996)
22. Wiger, U.: What is Erlang-style concurrency? (2010), <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, i.e., a morphism $f : \text{Closure } s \rightarrow \text{Closure } t$ is 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 = let val = id
             env = encode () -- empty environment
             fun = static (\_ -> id)
             in unsafeMkClosure val fun env

compClosure :: Closure (b -> c) -> Closure (a -> b) -> Closure (a -> c)
compClosure clo_g clo_f = let val = unClosure clo_g . unClosure clo_f
                           env = encode (clo_g, clo_f)
                           fun = static (\env ->
                                         let (clo_g, clo_f) = decode env
                                         in unClosure clo_g . unClosure clo_f)
                           in unsafeMkClosure val fun env
```

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., function closure) to the corresponding morphism in **Closure** (i.e., function on closures). Remains to show that F preserves identity and distributes over composition, which amounts to show the validity of the following equations (again easily verified).

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

B Code for sum of Euler's totients

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

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

dpsumeuler :: Int -> Int -> Int -> Par Integer
dpsumeuler lower upper k = sum <$> parMap clo_sumeuler chunked_list
  where chunked_list = chunk k [upper, upper - 1 .. lower]
        val = sumeuler
        env = encode () -- empty environment
        fun = static (\_ -> sumeuler)
        clo_sumeuler = unsafeMkClosure val fun env
```

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