Parallel Coordination Made Explicit in a Functional Setting

Jost Berthold and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,loogen}@informatik.uni-marburg.de

Abstract. We present a low-level coordination language for Haskell which can be used as an implementation language for parallel Haskell extensions. It has been developed in the context of the latest Eden implementation (based on the Glasgow-Haskell-Compiler, GHC, version 6) and it is thus referred to as the "EDen Implementation language", EDI. EDI provides a small set of directly implemented primitive operations for basic thread control, system information, and communication. We explore the expressiveness and performance of both Eden and its low-level implementation language EDI in comparison. It turns out that hardly any differences in performance can be observed. The main advantage of EDI in comparison to Eden is more accurate control of parallel execution. Our long-term goals are maintenance and structured implementation of Eden and a solid low-level implementation language, which can be used for other parallel Haskells as well.

1 Introduction

The area of parallel functional programming exhibits a variety of approaches, the common bases of which are referential transparency of functional programs and the ability to independently evaluate subexpressions. While some approaches pursue the target of (semi-)automatic parallelisation for special data structures (i.e. *data parallelism*), other dialects are more explicit in parallel coordination and allow what we call *general-purpose parallelism*, able to capture task-oriented parallelism. It is generally accepted [2, 17] that functional languages allow a clean distinction between a computation (or "base") language and independent coordination constructs for parallelism control.

The parallel functional language Eden [7] adds constructs for the dynamic creation of processes and communication channels to the non-strict functional computation language Haskell. The Eden programming model is semi-explicit general-purpose parallelism: Parallel processes are programmer-controlled, while communication is system-controlled. Eden has been implemented by layers on top of the Glasgow Haskell compiler (GHC) [14]. The central part is the Eden module, which implements the Eden constructs in Haskell using a few primitive operations provided by the parallel extension of the GHC runtime environment (RTE).

Any explicit parallel runtime support must express *operational* properties of the execution entities and will – in the end – rely on an imperative-style description. Parallelism support in its basic form must be considered as imperative and thus encapsulated in monads. Yet programmers might wish for a higher level of abstraction in their parallel programs and, for instance, use algorithmic skeletons [13] (higher-order functions for common parallel patterns), because they are not interested in gory details of implementation. Some parallel languages and libraries offer a fixed set of predefined skeletons and special, highly optimised implementations. On the other hand, with a more explicit general-purpose parallel language, a programmer can express *new* skeletons specific to the application.

Whether to hide or show the imperative basics is a question of language design. Eden tries to achieve a compromise between extremes in these matters: it exposes the execution unit of parallel processes to the programmer, but sticks to a functional model for their use. Eden processes differ from functions by additional strictness and remote evaluation. Further Eden language features allow for reactive systems and arbitrary programmer-controlled communication, which is (necessarily) opposed to referential transparency.

In this paper, the Eden implementation primitives will be considered as a language of their own, the *EDen Implementation language*, EDI for short. In contrast to Eden, EDI uses explicit communication and the IO monad to encapsulate side-effects. We compare expressiveness and performance of Eden and EDI. While the differences in performance can be neglected, the programming styles are substantially different. EDI allows an accurate control of parallelism, useful for system programming, whereas the higher abstraction of Eden is favourable for application programming, but often obscures what exactly is happening during parallel execution. The primary goal of this work is a structured Eden implementation, using a low-level implementation language which can be used for other parallel Haskells as well.

The paper is organised as follows: Section 2 describes Eden and its implementation. The primitive operations used in Eden's implementation constitute the Eden implementation language EDI. Section 3 discusses skeleton programming in Eden and EDI. Selected Eden skeletons have been re-programmed in EDI. Moreover, pitfalls of EDI programming are discussed. The paper ends with a discussion of related work in Section 4 and conclusions in Section 5.

2 Eden Language and Implementation

2.1 Eden Language Constructs

The parallel Haskell extension Eden [7] is an explicit general-purpose language for parallel programming, which gives programmers control over parallel processes. Eden allows to define *process abstractions* by a constructing function `process` and to explicitly *instantiate* (i.e. run) them on remote processors using the operator `( # )`. Processes are distinguished from functions by their operational property of remote execution.

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

For a given function `f`, evaluation of the expression `(process f) # arg` leads to the creation of a new (remote) process which evaluates the application of function `f` to argument `arg`. The argument is evaluated locally and sent to the new process.

Processes are encapsulated units of computation which communicate their inputs and results via *channels*. All values are reduced to normal form prior to sending, which implies additional strictness for processes. If input or output of a process is a tuple, each component will be evaluated and communicated by an own concurrent thread. Lists will be communicated element by element, values of other types will be communicated in single messages.

Communication between processes is automatically managed by the system and hidden from the programmer, but additional language constructs allow to explicitly create and access communication channels and to create arbitrary process networks. In the next subsection, we are showing how this feature is used to handle the hidden communication explicitly in the lower levels of the Eden system.
The task of parallel programming is simplified by a library of predefined skeletons [6]. Skeletons are higher-order functions defining parallel interaction patterns shared in many parallel applications. The programmer may use such known schemes from the library to achieve an instant parallelisation of a program.

2.2 Layers of the Eden Implementation

The implementation of Eden extends the runtime environment (RTE) of the Glasgow-Haskell-Compiler (GHC) [14] by a small set of primitive operations for process creation and communication between processes. These primitives merely provide very simple basic actions for process creation, data transmission between the machines' heaps, and system information. More complex operations are encoded in a functional module, called the *Eden module*. This module relies on the side-effecting primitive operations to encode Eden's process creation and communication semantics. The code on module level abstracts from many administrative issues, profiting from Haskell's support in genericity and code reuse. Moreover, it will protect the basic primitives from being misused. This leads to an organisation of the Eden system in layers (see Fig. 1): program level – skeleton library – Eden module – primitive operations – parallel runtime environment. This will greatly improve the maintainability of the highly complex system.

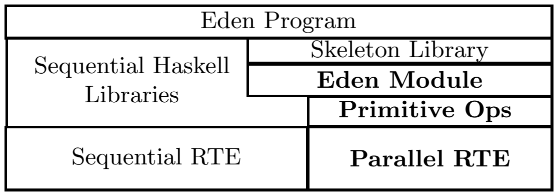

Fig. 1. Layered Eden implementation

The basic layer implementing the primitive operations is the GHC runtime environment, extended for parallel execution on clusters using MPI [9] or

PVM [12] as a middleware. The runtime system manages communication channels and thread termination; this will not be discussed further in this paper.

Primitive Operations. The current implementation of Eden is based on six primitives for system information, communication, and thread creation. The lowest level of the Eden module (shown in Fig. 2) consists of embedding the primitives in the IO monad to encapsulate the side-effects, and adds Haskell data types for communication mode and channels.

`noPe     :: IO Int`	number of processor elements
`selfPe   :: IO Int`	ID of own processor element
`createC  :: IO ( ChanName' a, a )`	channel name creation
`connectToPort :: ChanName' a -> IO ()`	channel installation
`sendData :: Mode -> a -> IO ()`	send data on implicitly given channel
`fork     :: IO () -> IO ()`	new thread in same process
`data ChanName' = Chan Int# Int# Int#`	a single channel: IDs from RTE
`data Mode = Stream | Data`	data modes: Stream or Single data
`    | Connect | Instantiate Int`	special modes: Connection, Instantiation

Fig. 2. Primitive operations to implement Eden

The first two primitives provide system information like the total number of processor elements (`noPe`) or the number of the processor element running a thread (`selfPe`).

For communication between processes, `createC` creates a new channel on the receiver side. It returns a channel name, containing three RTE-internal IDs: (PE, processID, portID) and (a handle for) the channel contents. Primitives `connectToPort` and `sendData` are executed on the sender side to connect a thread to a channel and to asynchronously send data. The send modes specify how the receiver sends data: either as an element of a stream (mode `Stream`), or in a single message (mode `Data`), or (optionally) just opening the connection (mode `Connect`). The purpose of the `Connect` mode is to provide information about future communication between processes to the runtime system. If every communication starts by a `Connect` message, the runtime system on the receiver side can terminate threads on the sender side evaluating unnecessary data.

For thread management, there is only the primitive `fork`, which creates a new thread (in the same process). Spawning a new process is implemented as sending data with the send mode `Instantiate`. The `Int` argument allows to explicitly place the new process on a certain processor. If it is zero, the RTE automatically places new processes in round-robin manner.

Eden Module: Overloaded Communication. The primitives for communication are used inside the Eden Module to implement Eden's specific data

```
newtype ChanName a = Comm (a -> IO())

class NFData a => Trans a where
    -- overloading for channel creation:
     createComm :: IO (ChanName a, a)
     createComm = do (c,v) <- createC
                     return (Comm (sendVia c), v)
    -- overloading for streams:
     write      :: a -> IO()
     write x = rnf x `seq` sendData Data x

sendVia ch d = do connectToPort ch
                  write d
```

Fig. 3. Type class `Trans` of transmissible data

```
-- list instance (stream communication)
instance Trans a => Trans [a]
 where write  l@[]  = sendData Data l
       write (x:xs) = do (rnf x `seq` sendData Stream x)
                         write xs

-- tuple instances (concurrency by component)
instance (Trans a, Trans b) => Trans (a,b)
  where createComm = do (c1,v1) <-createC
                        (c2,v2) <-createC
                        return (Comm (send2Via c1 c2), (v1,v2))

send2Via :: ChanName' a -> ChanName' b -> (a,b) -> IO ()
send2Via c1 c2 (v1,v2) = do fork (sendVia c1 v1)
                            sendVia c2 v2
```

Fig. 4. Eden Module: Overloading for communication

transmission semantics. The module defines type class `Trans` of transmissible data, which contains overloaded functions, namely `createComm` to create a high-level channel (type `ChanName`), and `write` to send data over channels.

As shown in Fig.3, the high-level channel `ChanName` is a *data communicator*, a function which performs the required send operation. It is composed by supplying the created primitive channel as a first argument to the auxiliary function `sendVia`. The latter, evaluated on sender side, first connects to the channel, and then calls function `write` to evaluate its second argument to normal form[1] and send it to the receiver in `Data` mode.

[1] The NFData class provides an evaluation strategy [15] `rnf` to force normal-form evaluation of any data type.

The two functions in `Trans` are overloaded as follows: `write` is overloaded for streams, which are communicated elementwise, and `createComm` is overloaded for tuples, which are evaluated concurrently by one thread for each component. Fig. 4 shows the instance declarations for lists and pairs. `write` communicates lists elementwise in `Stream` mode, and `createComm` for pairs creates two primitive channels, using the auxiliary function `sendVia` for `fork`ing threads.

Eden Module: Process Abstraction and Instantiation. The Eden constructs `process` and `( # )` render installation of communication channels between parent and child process, as well as communication, completely implicit, whereas the module internally uses *explicit* communication channels provided by `Trans` and the primitive operations.

```
data Process a b = Proc (ChanName b -> ChanName' (ChanName a) -> IO())

process :: (Trans a, Trans b) => (a -> b) -> Process a b
process f = Proc f_remote
   where f_remote (Comm sendResult) inCC
          = do (sendInput, input) <- createComm -- input communicator
               connectToPort inCC            -- sent back...
               sendData Data sendInput       --     ...to parent
               sendResult (f input)          -- sending result

( # ) :: (Trans a, Trans b) => Process a b -> a -> b
p # x = unsafePerformIO (instantiateAt 0 p x)

instantiateAt :: (Trans a, Trans b) =>
                 Int -> Process a b -> a -> IO b
instantiateAt pe (Proc f_remote) procInput
    = do (sendResult,  r )      <- createComm -- result communicator
         (inCC, Comm sendInput) <- createC    -- input comm. (reply)
         sendData (Instantiate pe)       -- spawn process
                  (f_remote sendResult inCC)
         fork (sendInput procInput)      -- send input concurrently
         return r                        -- return placeholder

-- variant of ( # ) which immediately delivers a whnf
data Lift a = Lift a
deLift (Lift x) = x

createProcess :: (Trans a, Trans b) => Process a b -> a -> Lift b
createProcess p i
      = unsafePerformIO (instantiateAt 0 p i >>= \x ->
                         return (Lift x))
```

Fig. 5. Eden Module: Process abstraction and instantiation

Fig. 5 shows the definition of process abstractions and instantiations in the Eden module. Process abstractions embed a function `f_remote` that is executed by a newly created remote process. This function takes a communicator `sendResult` to return the results of the process to the parent process, and a primitive channel `inCC` to send a communicator function (of type `ChanName a`) for its input channels to the parent process. The remote process first creates input channels, i.e. the corresponding communicator functions and the handle to access the received input. It connects to the channel `inCC` and sends the input communicator with mode `Data` on it. Afterwards, the process will evaluate the expression `(f input)` and send the result to the parent process, using the communicator function `sendResult`.

The instantiation operator `( # )` relies on the function `instantiateAt`, which defines the parent side actions for the instantiation of a new child process. The embedded function `f_remote` is applied to a previously created result communicator and a primitive channel for receiving the input, and the resulting IO action is sent to the designated processor *unevaluated*. A new thread is forked to send the input to the new process. As its name suggests, `instantiateAt` may place the new process on the PE specified by the parameter `pe`; or else uses the automatic round-robin placement if the parameter is 0.

Additionally the Eden module provides a variant `createProcess` of the instantiation, which differs in the type of the result value, *lifted* to immediately deliver a value in weak head normal form (whnf). This is e.g. necessary to create a series of processes without waiting for process results (see the `parMap` skeleton explained in the next section).

Eden coordination constructs have a purely functional interface, as opposed to the primitive operations encapsulated in the IO monad. Instantiation and process behaviour are described as a sequence of IO actions based on the primitives but, finally, the functional type of the instantiation operator `( # )` will be obtained by `unsafePerformIO`, the back door out of the IO monad.

3 Imperative Coordination in a Declarative Setting

Eden provides a purely declarative interface, but aims to give the programmer *explicit* control of parallelism in the program. Eden programs can be read twofold, from a computational and from a coordinational perspective:

– Instantiation of a previously defined process abstraction denotationally differs from function application by the additional strictness due to Eden's eager communication policy, but yields the same result as application of a strict function.
– Process abstraction and instantiation will hide any process communication, but expose the degree of parallelism of an algorithm directly by the number of instantiations.

However, the additional strictness introduced by eager communication is a crucial point for tuning parallel programs. On the one hand, it is required to start subcomputations at an early stage and in parallel. On the other hand, adding too much artificial strictness to a program can easily lead to deadlock situations. A complex Eden program normally uses a suitable skeleton library, optimised for the common case and circumventing common pitfalls of parallelism. Eden can also describe new specialised *skeletons*, and programming these is a different matter. Efficiently programming skeletons in Eden requires intimate knowledge of Eden specifics and a clear concept of the evaluation order in a demand-driven evaluation. Concentrating on the coordination view of Eden, programming skeletons can profit from a more *explicit* approach, as offered by Eden's implementation language EDI. EDI can be considered – necessarily at a lower level – as a fully-fledged alternative Eden-type language, which renders communication and side-effects explicit and will force to use the IO monad for parallel execution.

3.1 Low-Level Parallel Programming in EDI

Evaluation and Communication decoupled. In contrast to Eden's communication semantics, EDI communication is completely independent of the underlying computation. If a communicated value is not needed by the sender for a local computation, it will be left unevaluated by sending. This, of course, is not intended for parallel processes supposed to compute subresults. Programs in EDI therefore use evaluation strategies [15] to explicitly initiate the computation of a value to be sent. Although EDI does *not* encode coordination by strategies, using the class `NFData` and its normal form evaluation strategy `rnf` is a necessary part of EDI programming. We present and evaluate some parallel skeletons programmed in EDI and compare them with Eden skeletons.

Parallel Map. The higher-order function `map` applies a given function to all elements of a list. In a straightforward parallelisation, a process is created for each element of the resulting list. This can be expressed easily in Eden using process abstraction and instantiation, or programmed explicitly in EDI.

```
-- Eden's parallel map
parMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMap f xs = map deLift ([ createProcess (process f) x | x <- xs ]
                            ‘using‘ whnfspine)

-- auxiliary function for demand control
whnfspine :: Strategy [a]
whnfspine [] = ()
whnfspine (x:xs) = x ‘seq‘ whnfspine xs
```

The Eden version shown here uses the instantiation operator `createProcess`, which encodes all necessary communication and concurrency. Additional demand by `‘using‘ whnfspine` is necessary to force the immediate creation of all processes. Please note the use of `createProcess` instead of `( # )`, which is necessary because the strategy `whnfspine` would otherwise wait for the whnf of each process' result prior to forcing the creation of the next process.

```
-- monadic Edi parmap using primitive operations only:
parMapIO :: NFData b => (a -> b) -> [a] -> IO [b]
parMapIO f xs = do cs <- createCs (length xs)
                   sequence_ [ sendData (Instantiate 0) (doF ch x)
                               | (x,ch) <- zip xs (fst cs) ]
                   return (snd cs)
     where doF c x = do connectToPort c
                        let fx = f x
                        (rnf fx `seq` sendData Data fx)

createCs :: NFData a => Int -> IO ([ChanName' a],[a])
createCs n = do cList <- sequence (replicate n createC)
                let lists@(cs, vs) = unzip cList
                (rnf cs `seq` return lists)
```

The EDI version is explicitly monadic (but might, of course, escape from the IO monad by `unsafePerformIO` at top level). Prior to spawning the child processes, the caller creates a set of channels (by a simple abstraction `createCs` over the single channel creation `createC`). Each remote computation (defined by function `doF`) will receive one of these channels for sending back the result. The second parameter of `doF` is the *input*, potentially unevaluated. Whilst the Eden process instantiation spawns an own concurrent thread in the calling machine to send this input in normal form, the EDI version acts as a *demand-driven* parallel map (`parmap_dm`), useful to avoid bottlenecks in the caller. The latter can, of course, be modelled in Eden as well, by adding a dummy argument to the function applied to the list elements:

```
parmap_dm:: (Trans a, Trans b) => (a -> b) -> [a] -> IO [b]
parmap_dm f xs = map deLift
                ([ createProcess (process (\() -> f x)) () | x <- xs ]
                 `using` whnfspine)
```

An advantage of the EDI code is that the `Lift - deLift` trick as well as the explicit demand control using the strategy `whnfspine` is no longer necessary to create a series of processes.

Figure 6 shows runtime and speedup measurements for a small test program with the two demand-driven `parMap` versions, also including the previous Eden implementation (based on GHC 5) for comparisons. The program computes the sum of Euler Totients, $\sum_1^n \varphi(k)$ for $n = 25000$. Of course, the test program does not spawn an own process for every number $\varphi(k)$ to be computed – the task granularity would be much too fine. Numbers are distributed evenly among few processes, one on each available processor. And since the values are summed up afterwards (`map` is followed by a parallel `fold`), each process(or) computes the partial sum in parallel as well.

The *sequential* base performance of the previous Eden 5 system apparently is much worse (44% longer runtime); therefore speedup degrades slightly for the new implementation. The negligible difference between Eden 6 and EDI shows that the overhead for the module code is minor, and only the way input data is transmitted is relevant, depending on the concrete application.

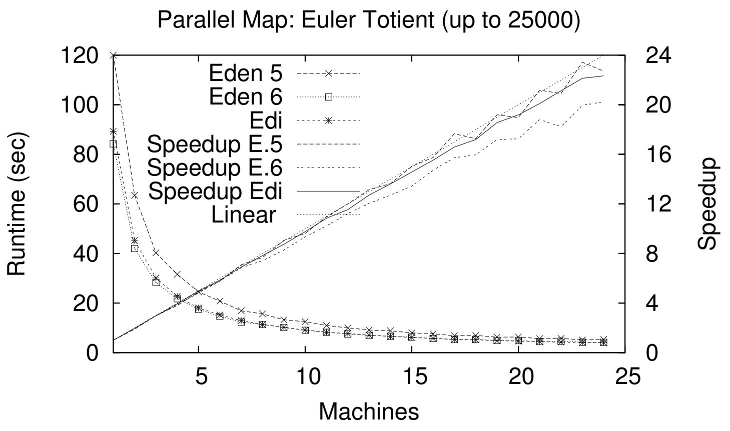

Fig. 6. Parallel `map`/`fold` example, Eden 5, Eden 6 and EDI

Nondeterminism, Concurrency and Parallelism. In the previous example, tasks have been distributed statically, in advance. When subtasks are of highly irregular complexity, or when the number of subtasks may vary depending on the input, *dynamic* load balancing is one of the most desired properties of a parallel `map` skeleton. The purely functional coordination constructs of Eden are not sufficient to describe dynamic task distribution; therefore Eden offers a nondeterministic additional construct `merge` for merging a list of streams into a single stream. Data is added to the output stream as soon as it is available in any of the input streams, in nondeterministic order. As shown in Fig. 7, this can be used for a *workpool* scheme, i.e. a `map` skeleton in master/worker scheme, where a worker process gets a new task every time it returns a result. A prefetch parameter determines the number of initial tasks assigned to a worker. It should be used to avoid workers running out of work.

In this simple version, the computation results are returned unsorted, in the order in which they have been sent back by the workers. In order to indicate which worker has completed a task, every worker tags its results with its id, a number between 1 and `np`. All result streams `fromWorkers` are merged nondeterministically in the master process. The worker numbers are then separated from the proper results, and serve as requests for new work. The auxiliary function `distribute` takes as arguments the list of requests and the available tasks, and distributes the tasks to `np` sublists, as indicated by the requests list. The number of initial requests is determined by skeleton parameter `prefetch`. A crucial property of the function `distribute` is that it must be "incremental", i.e. can deliver partial result lists without the need to evaluate requests not yet available.

A recent extension to this skeleton may even be nested and applied to computations where the results computed by workers may lead to new additional tasks [11].

```
edenWP :: (Trans t, Trans r) =>
      Int -> Int -> (t -> r) -> [t] -> [r]
edenWP np prefetch f tasks = results
  where fromWorkers = map deLift
                      (zipWith createProcess workerProcs toWorkers)
                                           `using` whnfspine
        workerProcs = [process (zip [n,n..] . map f) | n<-[1..np]]
        toWorkers          = distribute tasks requests
        (newReqs, results) = (unzip . merge) fromWorkers
        requests           = initialReqs ++ newReqs
        initialReqs        = concat (replicate prefetch [1..np])
        distribute :: [t] -> [Int] -> [[t]]
        distribute tasks reqs = [taskList reqs tasks n | n<-[1..np]]
        where taskList (r:rs) (t:ts) pe
                          | pe == r    = t:(taskList rs ts pe)
                          | otherwise  =    taskList rs ts pe
              taskList _      _      _  = []
```

Fig. 7. Eden workpool skeleton using `merge`

However, the workpool skeleton can also be implemented without the need for Eden's `merge` construct, nor the sophisticated `distribute`. Instead, we can use a nondeterministic construct of Concurrent Haskell: a channel which is read by concurrent sender threads inside the master. A channel (data type `Chan`) in Concurrent Haskell models a potentially infinite stream of data which may be consumed concurrently by different threads. Due to nondeterministic scheduling, channel operations are in the IO monad, like the EDI coordination constructs. Figure 8 shows a workpool skeleton which returns its result in the IO monad.

```
ediWP :: (NFData t, NFData r) =>
     Int -> Int -> (t -> r) -> [t] -> IO [r]
ediWP np prefetch f tasks = do
          (wInCCs, wInCs) <- createCs np
          (wOutCs, wOuts) <- createCs np
          sequence_ [ sendData (Instantiate 0) (worker f wOutC wInCC)
                      | (wOutC,wInCC) <- zip wOutCs wInCCs ]
          taskChan <- newChan
          fork (writeList2Chan taskChan
                   ((map Just tasks) ++ (replicate np Nothing)))
          sequence_ [ fork (inputSender prefetch inC taskChan answers)
                      | (inC,answers) <- zip wInCs wOuts ]
          return (concat wOuts)
```

Fig. 8. EDI workpool skeleton, using concurrent `inputSender` threads

The master needs channels not only to receive the results, but also to initiate input communication with the workers, thus two sets of `np` channels are created. A set of worker processes is instantiated with these channels as parameters. As shown in Fig.9, each worker creates a channel to receive input, sends it to the parent, and then connects to the given output channel to send the results as a stream.

We use a `Maybe` type in order to indicate termination. The `taskChan` is created and (concurrently) filled with the tagged task list (`map Just tasks`), followed by `np` termination signals (`Nothing`). The task channel is concurrently read by several input senders, one for every worker process, which will be forked next. Every input sender consumes the answers of one worker and emits one new task per answer, after an initial `prefetch` phase. The master process collects the answers using `concat`, the Haskell prelude function to concatenate a list of lists. A slight variant of this would be to sort the answers list in the order indicated by tags which are added to tasks to memorise their initial order.

```
worker :: (NFData t, NFData r) =>
          (t -> r) -> ChanName' [r] -> ChanName'(ChanName'[t]) -> IO ()
worker f outC inCC
      = do (inC, inTasks) <- createC -- create channel for input
           connectToPort inCC        -- send channel to parent
           sendData Data inC
           connectToPort outC        -- send result stream
           sendStream ((map f) inTasks)
  where sendStream :: NFData r => [r] -> IO ()
        sendStream   []   = sendData Data []
        sendStream (x:xs) = do (rnf x `seq` sendData Stream x)
                               sendStream xs

inputSender :: (NFData t) =>
               Int -> ChanName' [t] -> Chan (Maybe t) -> [r] -> IO ()
inputSender prefetch inC concHsC answers
       = do connectToPort inC
            react ( replicate prefetch dummy  ++ answers)
 where dummy = undefined
       react :: [r] -> IO ()
       react [] = return ()
       react (_:as) = do
                  task <- readChan concHsC -- get a task
                  case task of
                    (Just t) -> do (rnf t `seq` sendData Stream t )
                                   react as
                    Nothing  -> sendData Data [] -- and done.
```

Fig. 9. `worker` process and `inputSender` thread for EDI workpool

It should be noted that the EDI version of the workpool looks slightly more specialised and seems to use more concurrent threads than the – considerably shorter – Eden version. Since EDI uses explicit communication, the separate threads to supply the input become obvious. The Eden version works in quite the same way, but the concurrent threads are created implicitly by the process instantiation operation `createProcess`. Apart from one separate thread filling the channel with available tasks, both versions have exactly the same degree of concurrency; it is not surprising that both workpool implementations are similar in runtime and speedup.

Once the master process uses concurrent threads and the IO monad, it may easily be extended in different ways. One very useful extension would be to include a state in the master process, e.g. a "current optimal" solution for a branch-and-bound algorithm, or a dynamically increasing task pool, or using a stack instead of a FIFO queue for task management. Depending on the particular requirements for the master state, its implementation in a purely functional style may become quite cumbersome (see [8] for a case study). The explicitness of parallelism, communication and concurrency inflates the EDI code, but is advantageous when implementing specialised versions of skeletons.

A Ring Skeleton. The examples given up to now are showing, more or less, how Eden and EDI are interchangeable and comparable in performance. There are however situations where Eden's implicit concurrency and eagerness can lead to unwanted behaviour, and the source code usually does not clearly indicate the errors.

```
ring :: (Trans a, Trans b,
          Trans r) =>
         Int ->
         (Int -> i -> [a]) ->
         ([b] -> o) ->
         ((a,[r]) -> (b,[r]))
         -> i -> o

ring size makeInput processOutput ringWorker input = ...
```

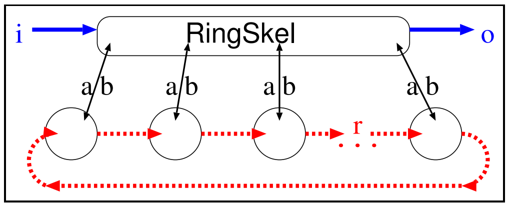

Fig. 10. A ring skeleton in Eden, type and communication structure

A ring of interconnected processes can be defined using Eden channels [1]. Fig. 10 shows the type signature of a highly parameterised ring skeleton, and depicts its process and communication structure. Parameters are the ring size, a function `makeInput` preparing the initial input to all ring processes, a similar function (`processOutput`) to construct the final output, and the functionality of the ring processes. All ring processes are identical and receive two inputs, one (of type `a`) from the caller and one (of type `[r]`, a stream) from their predecessor in the ring.

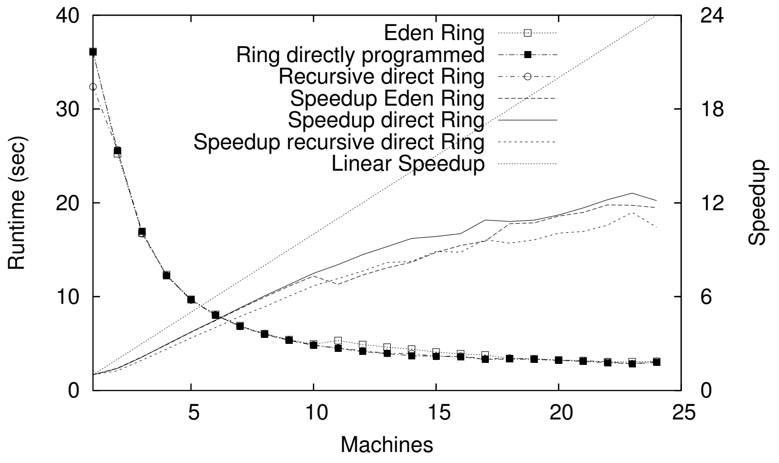

Fig. 11. Ring example: Warshall's algorithm (500 node graph), Eden vs. directly programmed, specialised ring

This skeleton may also be specified at a lower level in EDI, with the advantage that the communication, explicit anyway, may be optimised for the special application, e.g. when input is statically determined, or when the ring output is not relevant. As in the previous examples, there are no big runtime differences in the general case. Fig. 11 shows measurements for an example program, Warshall's algorithm to compute the complex hull of a directed graph.

This skeleton description is coherent at first sight, but some questions may arise when using it. The given type restricts the ring communication to a stream. This is a sensible restriction since, with a non-stream type, the ring necessarily degenerates to a pipeline, or simply deadlocks. Likewise, Eden constructs can express the case where the initial input (of type `a`) to the ring processes is static and thus embeddable into the process abstraction, as shown for `parMap`.

A more subtle detail can lead to problems when the general ring skeleton is used in a special context: If the initial ring process input (or output) happens to be a tuple, the programmer might expect that each component will be evaluated concurrently, as usual in Eden. However, the ring implementation adds an additional parameter to the input: Channels to the ring neighbours must be exchanged prior to computation. The ring process abstraction internally is of type `Process (a,ChanName [r]) (b,ChanName [r])` and, thus, does *not* use concurrency for components of their external input and output – the ring will immediately deadlock if the components of type `a` expose non-local data dependencies. A different Eden implementation of the ring, specialised to avoid this problem, is possible, but the difficulty is to find out the reason for the deadlock. Neither the calling program, nor the skeleton source code will clearly indicate the problem; it will remain hidden in the overloaded communication inside the Eden module.

Downside of Explicitness. As we have shown previously, the explicitness of EDI can help to optimise skeletons for particular cases and save time in spotting errors due to Eden's complex implicit communication semantics. On the other hand, programming in EDI considerably inflates the code and may have other pitfalls. Evaluation control prior to communication is the most important of these, since the implemented `sendData` primitive does not imply any prior evaluation. As EDI is purely monadic and deliberately simple, the programmer has to specify *every* single action.

Another possible source of errors is the all-purpose character of `sendData`, which uses the same primitive for data transmission, communication management, and process instantiation, distinguished only by the different send modes. Sending data by the wrong mode may lead to, e.g., a bogus process without any effect, as shown here:

```
badIdea_no1 :: Int -> a -> IO ()
badIdea_no1 pe data = sendData (Instantiate pe) data
```

If the data sent is, say, a number, its remote evaluation will have no effect at all, although its type is perfectly correct, due to the liberal typing of the primitive. In the example above, an auxiliary function for instantiation should force that the data sent is an action of type `IO()`.

```
spawnProcessAt :: Int -> IO () -> IO ()
spawnProcessAt pe action = sendData (Instantiate pe) action
```

Moreover, for data communication, threads are supposed to connect to a channel prior to communication and might cause obscure runtime errors if the wrong connections are created. Although the simple channels of EDI are strongly typed, this two-step communication allows to create erroneous communication sequences not discovered at compile time. The following (perfectly well-typed) function expects a *wrong* channel type and then does not connect prior to sending in one case, or alternatively uses the wrong send mode.

```
badIdea_no2 :: ChanName' Double -> [Double]  -- types do not match
               -> IO ()
badIdea_no2 c (n:ns)= do sendData Stream n   -- not yet connected
                         badIdea_no2 c ns
badIdea_no2 c   []  = do connectToPort c
                         sendData Stream []  -- wrong send mode
```

When evaluating this function, a run-time error will occur because the receiver's heap becomes corrupted.

As above, combining connection and send operation by a type-enforcing auxiliary function can detect the error. The applied evaluation strategy can as well be included in such a combined function.

```
sendEvalDataOver :: Strategy a -> ChanName' a -> a -> IO()
sendEvalDataOver eval ch d = do connectToPort c
                                (eval d ‘seq‘
                                   sendData Data d)
```

The only disadvantage here is that a separate function for sending lists is needed, since the send mode becomes hard-coded.

In order to streamline the interface between Haskell and the runtime system, the primitive `sendData` has been given the liberal type `Mode -> a -> IO ()`, which is why erroneous usage of the primitive will not be detected at compile time. Hence, the solution to these problems consists in typed auxiliary functions which will restrict the argument types in such a way that the primitives will be used as intended. Obviously, it is necessary to superimpose a layer of type-checking auxiliary functions over the primitive operations to improve error detection during type checking in EDI.

4 Related Work

EDI considered as a language provides extensions to existing concepts of Concurrent Haskell [5], as implemented in GHC. Thread concurrency is *extended* by process parallelism, communication in EDI is handled using channel communication instead of the shared synchronised heap cells (`MVars`) of Concurrent Haskell. As we have already underlined by one of our examples, both approaches can be sensibly combined. Latest efforts in Haskell implementations aim to extend Concurrent Haskell's thread concurrency to OS level for multiprocessor support in the threaded GHC runtime system [3]. Combining this future multicore support with the distributed-memory-parallelism provided by EDI is one of our long-term goals.

In the field of parallel functional languages, many language concepts follow more implicit approaches than Eden and, necessarily, its implementation language. Although intended as a low-level implementation language, EDI can be used as a language for distributed programming with explicit asynchronous communication.

Glasgow Distributed Haskell (GdH) [10] is the closest relative to EDI in this respect and provides comparable language features, especially location-awareness and dynamically spawning remote IO actions. However, GdH has been designed with the explicit aim to extend the virtual shared memory model of Glasgow Parallel Haskell (GpH) [16] by features of explicit concurrency (Concurrent Haskell [5]). Our implementation primarily aimed at a simple implementation concept for Eden and thus does not include the shared-memory-related concepts of GdH. Indeed, we think that GdH can be implemented with minimal extensions to our implementation.

Port-based distributed Haskell (PdH) [4] is an extension of Haskell for distributed programming. PdH offers a dynamic, server-oriented port-based communication for first-order values between different Haskell programs. In contrast to our implementation, its primary aim is to obtain open distributed systems, interconnecting different applications – integrating a network library and a stock Haskell compiler.

5 Conclusions and Future Work

We have presented a new implementation for the parallel functional language Eden, based on a lean low-level interface (EDI) to a sophisticated parallel Haskell runtime environment. Although essentially following previous concepts, the new implementation makes the side-effecting primitive operations explicit and allows to express parallel coordination in an imperative manner, while the computation language remains purely functional.

While EDI provides a low-level flexible and powerful approach to controlling coordination in a functional setting, Eden abstracts from many details, thereby simplifying the development of parallel programs, but partly losing coordination control. Runtime comparisons show that programs written in Eden and EDI will show the same performance as long as their behaviour is equivalent. This is because Eden is implemented on top of EDI. From the programmer's point of view, the Eden level of abstraction would be an asset if everything worked out fine. On the other hand, getting things right is much more difficult in Eden than on the EDI level of abstraction.

We have briefly mentioned the spectrum of parallel functional languages expressible by EDI and using our framework. Our Eden implementation based on EDI can be used to easily obtain prototype implementations for other parallel extensions of Haskell, mainly extensions at higher abstraction levels.

One of our research goals is to keep alive and advance a general-purpose parallel Haskell. The comparison of Eden and EDI undertaken in this paper is a step towards redesigning Eden and will need further investigation. Several other areas lend themselves to further research. Combining the concepts we developed for the runtime with state-of-the-art hardware techniques, such as multicore support, or modern wide-area network infrastructure (Grid Technology), is the most important goal. Likewise, by applying these concepts to a different computation language, the influences of the host language will emerge, and parallelism extensions can be cleanly separated from their sequential base or concrete application.

References

1. J. Berthold and R. Loogen. The Impact of Dynamic Channels on Functional Topology Skeletons. In A. Tiskin and F. Loulergue, editors, *HLPP 2005: 3rd International Workshop on High-level Parallel Programming and Applications*, Coventry, UK, 2005.
2. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming.* Springer, 1999.
3. T. Harris, S. Marlow, and S. P. Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell.* ACM Press, September 2005.

4. F. Huch and V. Stolz. Implementation of Port-based Distributed Haskell. In M. Mohnen and P. W. M. Koopman, editors, *IFL'01: Implementation of Functional Languages, 13th International Workshop, Draft Proceedings*, Stockholm, Sweden, 2001.
5. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, USA, 1996. ACM Press.
6. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 4. Springer, 2003.
7. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
8. R. Martínez and R. Pena. Building an Interface Between Eden and Maple: A Way of Parallelizing Computer Algebra Algorithms. In P. W. Trinder, G. Michaelson, and R. Pena, editors, *IFL'03: Implementation of Functional Languages, 15th International Workshop, Selected Papers*, LNCS 3145, Edinburgh, UK, 2003. Springer.
9. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.
10. R. Pointon, P. Trinder, and H.-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *IFL'00: Implementation of Functional Languages, 12th International Workshop, Selected Papers*, LNCS 2011, Aachen, Germany, 2000. Springer.
11. S. Priebe. Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par 2006: Parallel Processing. 12th International Euro-Par Conference*, LNCS 4128, Dresden, Germany, 2006.
12. PVM: Parallel Virtual Machine. Web page. *http://www.epm.ornl.gov/pvm/*.
13. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
14. The GHC Developer Team. The Glasgow Haskell Compiler. Website *http://www.haskell.org/ghc*.
15. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.
16. P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96: Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*. ACM Press, 1996.
17. P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4, 5):469–510, 2002.