Eden — Parallel Functional Programming with Haskell

Rita Loogen
loogen@informatik.uni-marburg.de
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, Germany

Contents

1 Introduction

Functional languages are promising candidates for parallel programming, because of their high level of abstraction and, in particular, because of their referential transparency. In principle, any sub-expression could be evaluated in parallel. As this implicit parallelism would lead to too much overhead, modern parallel functional languages allow the programmers to specify parallelism explicitly.
These lecture notes present Eden, a parallel functional programming language which extends Haskell with constructs for the definition and instantiation of parallel processes. The underlying idea of Eden is to enable programmers to specify process networks in a declarative way. Processes evaluate function applications remotely in parallel. The function parameters are the process inputs and the function result is the process output. Thus, a process maps input to output values. Inputs and outputs are automatically transferred via unidirectional one-to-one channels between parent and child processes. Programmers need not think about triggering low-level send and receive actions for data transfer between parallel processes. Nevertheless, the programmer has direct control over evaluation site, process granularity, data distribution, and communication topology, but need not manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer. Common and sophisticated parallel communication patterns and topologies, i.e. algorithmic skeletons, are provided as higher-order functions in a user-extensible skeleton library written in Eden.
Eden is tailored for distributed memory architectures, i.e. processes work within disjoint address spaces and do not share any data. This simplifies Eden's implementation as there is no need for global garbage collection. There is, however, a risk of loosing sharing, i.e. it may happen that expression evaluations are duplicated in parallel processes.
Although the automatic management of communication by the parallel runtime system has several advantages, it also has some restrictions. This form of communication is only provided between parent and child processes, but e.g. not between sibling processes. I.e. only hierarchical communication topologies are automatically supported. For this reason, Eden also provides a form of explicit channel management. A receiver process can create a new input channel name and pass

this to another process. The latter can directly send data to the receiver process via the received channel name. Moreover, many to one communication can be modeled using a pre-defined (necessarily non-deterministic) merge function. These non-functional Eden features make the language very expressive. Arbitrary parallel computation schemes like sophisticated master-worker systems or divide-and-conquer computations can be defined in an elegant way.
Eden has been implemented by extending the runtime system of the Glasgow Haskell compiler [42], the most mature and efficient Haskell implementation, for parallel and distributed execution. The parallel runtime system (PRTS) uses suitable middleware (currently PVM [39] or MPI [34]) to manage parallel execution. Recently, a special multicore implementation which needs no middleware has been implemented. Traces of parallel program executions can be visualised and analysed using the Eden Trace Viewer.
This tutorial gives an up-to-date introduction into Eden's language constructs, its programming methodology based on algorithmic skeletons, and its layered implementation on top of the Glasgow Haskell compiler.

Plan of this tutorial.

It is assumed that the reader has a basic knowledge of programming in Haskell. The first section introduces the basic constructs of Eden's coordination language, i.e. how parallelism can be expressed and managed. It is explained how simple process networks like process farms, process rings, and master-worker systems can be specified. The next section shows how more sophisticated algorithmic skeletons can be implemented in Eden. Section 4 explains how Eden has been implemented. Hints at more detailed material on Eden are given in Section 5. Related work is shortly discussed in Section 6. Conclusions are drawn in Section 7. The appendix contains a short explanation of how to compile, run, and analyse Eden programs. In particular, it presents the Eden trace viewer EdenTV, an important tool to analyse the behaviour of parallel programs. The tutorial refers to several case studies and shows example trace visualisations. Most corresponding traces have been produced on an Intel 8-core machine (2 × Xeon Quadcore @2.5GHz, 16 GB RAM) machine. The program code of the case studies can be found on the Eden pages, see

`http://www.mathematik.uni-marburg.de/~eden/`

2 The Basics of Eden

Eden uses Haskell as its computation language. It provides special coordination constructs

- for defining and creating processes

- for creating channels on the receiver side and using them on the sender side — this is used for defining non-hierarchical process topologies
- for merging incoming streams into a single one — this is used to model many-to-one communication.

2.1 Defining and Creating Processes

Eden provides two coordination constructs for the definition of processes: *process abstraction* and *instantiation*:

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

The purpose of function `process` is to convert functions of type `a -> b` into *process abstractions* of type `Process a b` where the type context `(Trans a, Trans b)` indicates that both types `a` and `b` must belong to the `Trans` class of *transmissible* values. Process abstractions are instantiated by using the infix operator `( # )`. An expression `(process funct) # arg` leads to the creation of a remote process for evaluating the application of the function `funct` to the argument `arg`. The argument expression `arg` will be evaluated concurrently by a new thread in the parent process and sent to the new child process. The child process will evaluate the function application `funct arg` in a demand driven way, using the standard lazy evaluation of Haskell. If the argument value is necessary to complete its evaluation, the child process will suspend, until the parent thread has sent it. The child process sends back the result of the function application to its parent process. Communication is performed through *implicit 1:1 channels* that are established between child and parent process on process instantiation, while process synchronisation is achieved by exchanging data through the communication channels, as these have non-blocking sending, but blocking reception. In order to increase the parallelism degree and to speed up the distribution of the computation, process in- and outputs will be evaluated to normal form before being sent (except for expressions with a function type, which are evaluated to weak head normal form). This implements a *pushing approach* for communication instead of a *pulling approach* where remote data would have to be requested explicitly.
Because of the normal form evaluation of communicated data, the type class `Trans` is a subclass of the class `NFData` (Normal Form Data) which provides a function `rnf` to force the normal form evaluation of data. `Trans` provides communication functions overloaded for `lists`, which are transmitted as streams, element by element, and for `tuples`, which are evaluated component-wise by concurrent threads in the same process. An Eden process can thus comprise a number of threads, which may vary during its lifetime. The type class `Trans` will be explained in

more detail in Section 4. A channel is closed when the output value has been completely transmitted to the receiver. An Eden process will end its execution as soon as all its output channels are closed or are detected to be unnecessary (during garbage collection). Termination of a process implies the immediate closure of its input channels, i.e., the closure of the output channels in the corresponding producer processes, thus leading to a termination cascade through the process network.

Example: The following function creates a parallel sorting network which transforms an input stream into a sorted output stream by subsequently merging sorted sublists with increasing length:

```
mergeSort      :: (Ord a, Trans a) => [a] -> [a]
mergeSort []   =  []
mergeSort [x]  =  [x]
mergeSort xs   =  sortMerge (process mergeSort # xs1)
                            (process mergeSort # xs2)
  where [xs1,xs2] = unshuffle 2 xs
```

Streams with at least two elements are split into two sub-streams using the auxiliary function `unshuffle`.

```
unshuffle :: Int      -- ^ number of sublists
             -> [a]   -- ^ input list
             -> [[a]] -- ^ distributed output
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]

takeEach :: Int -> [a] -> [a]
takeEach n []     =  []
takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

The sub-streams are sorted by recursive instantiations of `mergeSort` processes. The sorted sublists are coalesced into a sorted result list using the function `sortMerge` which is an ordinary Haskell function. The context `Ord a` ensures that an ordering is defined on type `a`.

```
sortMerge :: Ord a => [a] -> [a] -> [a]
sortMerge []           ylist        = ylist
sortMerge xlist        []           = xlist
sortMerge xlist@(x:xs) ylist@(y:ys)
   | x <= y = x : sortMerge xs ylist
   | x >  y = y : sortMerge xlist ys
```

The process system generated when `mergeSort` is applied to a list with more than two elements is a binary tree. ◁

As the coordination functions `process` and `( # )` are usually used in combination, one could instead use the following operation for parallel function application:

```
($#)      ::  (a -> b) -> a -> b
f $# x    = process f # x          -- ( $# ) = ( # ) .  process
```

Example: Using the `($#)` operator the third case in the definition of `mergeSort` (see above) can be rewritten as follows:

```
mergeSort xs = sortMerge (mergeSort $# xs1) (mergeSort $# xs2)
```
◁

2.2 Creating Farms of Processes

The laziness of Haskell has many advantages as we will see soon when considering recursive process networks and stream-based communication. Even though, it is also an obstacle to parallelism, because pure demand-driven (lazy) evaluation will activate a parallel evaluation only when its result is already needed to continue the overall computation, i.e. the main evaluation will immediately wait for the result of a parallel sub-computation. Thus, sometimes it is necessary to produce additional demand in order to unfold certain process systems. Otherwise, the programmer may experience *distributed sequentiality*. This situation is illustrated by the next example:

Example: Replacing the function application in the `map` function:

```
map :: (a -> b) -> [a] -> [b]
map f xs  =  [f x | x <- xs]
```

by a process instantiation, leads to a simple parallel `map` function, in which a different process is created for each element of the input list:

```
parMap_distrSeq :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMap_distrSeq f xs =  [f $# x | x <- xs]
```

The operator `($#)` determines that the input parameter `x`, as well as the result value, will be transmitted via channels. The types `a` and `b` should therefore belong to the class `Trans`.
The problem with this definition is that for instance the expression

```
sum (parMap_distrSeq square [1..10])
```

will create 10 processes, but only one after the other as demanded by the `sum` function which sums up the elements of a list of numbers. Consequently, the computation will not speed up by "parallel" evaluation, but slow down because of the process creation overhead added to the distributed, but sequential evaluation.
◁

To avoid this problem the (predefined) Eden function `spawn` can be used to eagerly and immediately instantiate a complete list of process abstractions with their corresponding inputs. Neglecting demand control, `spawn` can be denotationally specified as follows:

```
spawn   ::  (Trans a, Trans b) => [Process a b] -> [a] -> [b]
spawn   =  zipWith ( # ) -- definition without demand control
```

The variant `spawnAt` additionally locates the created processes on given processor elements (identified by their number).

```
spawnAt :: (Trans a, Trans b) => [Int] -> [Process a b] -> [a] -> [b]
```

Now a really parallel variant of the function `map` can be defined as follows:

```
parMap     ::  (Trans a, Trans b) => (a->b) -> [a] -> [b]
parMap f   =  spawn (repeat (process f))
```

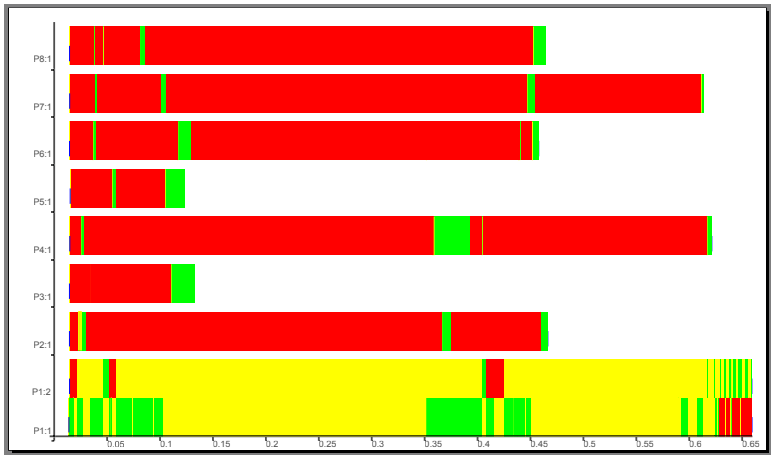
where the Haskell prelude function `repeat ::  a -> [a]` yields a list infinitely repeating the parameter element. This `parMap` function implements a very simple form of data parallelism. The parameter function is applied in parallel to all elements of a given input list. The number of created processes corresponds to the length of the input list. A more flexible version of a parallel map allows to adapt the number of processes to e.g. the number of available processing elements (in Eden provided by the predefined constant `noPe ::  Int`). The following function `farm` takes two additional function parameters: an input distribution function and a result combination function. The input distribution function distributes the input list into a list of lists the length of which determines the number of processes to be created. The result combination function combines the list of lists of results into a single result list. Of course, the distribution function and the combination function should be inverse to each other, i.e.

`combine .  (map (map f)) .  distribute = map f.`

```
farm ::  (Trans a, Trans b) =>
         ([a] -> [[a]])           -- ^ input distribution
         -> ([[b]] -> [b])        -- ^ result combination
         -> (a->b) -> [a] -> [b]  -- ^ map interface
farm distribute combine f
         = combine .  (parMap (map f)) .  distribute
```

The function `farm` implements a process farm with static task distribution. A typical distribution function would be e.g. `unshuffle noPe`. A corresponding combine function would be `shuffle = concat . transpose`.
The functions `parMap` and `farm` are simple examples for the definition of (algorithmic) skeletons [13] in Eden. Skeletons define common parallel computation patterns. In Eden they are simply defined as higher-order functions. Eden provides a library with many pre-defined skeletons. In Section 3 we will discuss some more sophisticated skeleton definitions in Eden.

Case Study (Raytracer): The following central part of a ray tracer can easily be parallelised using the farm skeleton.



input size 100
runtime 0.659667s
8 machines
9 processes
17 threads
48 conversations
20048 messages

Figure 1: Raytracer trace on 8 PE, farm with 8 processes

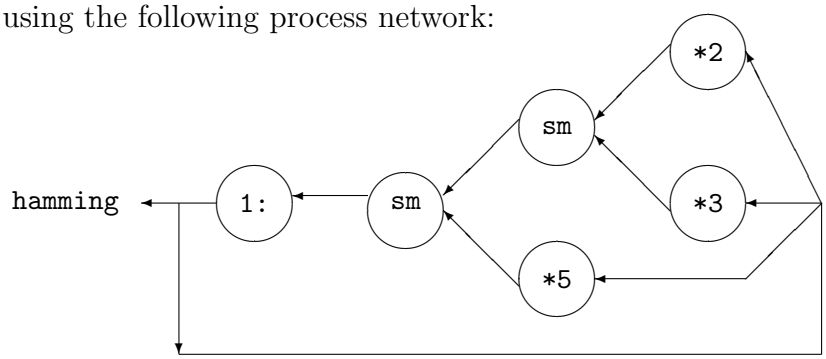
```
rayTrace scene viewpoint
  = map impact rays
  where rays = generateRays viewPoint
        impact ray = fold earlier (map (hit ray) scene)
```

The complete program codes of case studies can be found on the Eden web pages. By replacing the outer `map` by `(farm (unshuffle noPe) shuffle)` we achieve a parallel ray tracer which creates as many processes as processing elements are available. Each process computes the impacts of a couple of rays. The rays will be computed by the parent process and communicated to the remote processes. Each process receives the `scene` via its process abstraction. If the `scene` has not been evaluated before the processes are created, each process will evaluate it.
Figure 1 shows the trace visualisation (processes' activity over time) and some statistics produced by our trace viewer EdenTV (see Section A.2). The trace has been generated by a program run of the raytracer program with input size 100 on an Intel 8-core machine (2 × Xeon Quadcore @2.5GHz, 16 GB RAM) machine. The result is disappointing, because most processes are almost always blocked (red/dark grey) and show only short periods of activity (green/grey). 9 processes (the main process and 8 farm processes) and 17 threads (one thread per farm process and 9 threads in the main process, i.e. the main thread and 8 threads which evaluate and send the input for the farm processes) have been created. Two processes have been allocated on machine 1. Alarming is the enormous number of messages that is exchanged between the processes. When messages are added to the trace visualisation, the graphic becomes almost black. It becomes obvious that the extreme number of messages is one of the reasons for the bad behaviour of the program. Most messages are stream messages. A stream is counted as a single conversation. The number of conversations, i.e. communications over a single channel, is much less than the number of total messages. In the next section, we will show how to optimise process communication. ◇

2.3 Process Communication

Eden processes exchange data via unidirectional one-to-one communication channels. The type class `Trans` provides implicitly used functions for this purpose. As laziness enables infinite data structures and the handling of partially available data, communication streams are modeled as lazy lists, and circular topologies of processes are created and connected by such streams.

Example: The sequence of Hamming numbers $\langle 2^i 3^j 5^k \mid i, j, k \geq 0 \rangle$ can easily be computed using the following process network:



which can be expressed in Eden as follows:

```
hamming :: [Int]
hamming = 1: sm ((uncurry sm) $# ((map (*2) $# hamming,
                                  (map (*3) $# hamming))
                 ((map (*5) $# hamming)
```

The function `sm` is similar to the previously introduced function `sortMerge` but eliminates duplicates when merging sorted lists.

```
sm :: [Int] -> [Int] -> [Int]
sm [ ]       ys     = ys
sm xs        [ ]    = xs
sm xl@(x:xs) yl@(y:ys)
        | x < y     = x : sm xs yl
        | x == y    = x : sm xs ys
        | otherwise = y : sm xl ys
```

Four child processes will be created by the four applications of `($#)`. The main process will evaluate the outer application of `sm` itself. Note that the process network resulting from the above Eden definition does not exactly correspond to the one shown above. The problem is that in Eden implicitly created communication channels are always established between parent and child processes. The parent process of all Hamming child processes is the main process. Thus, the output of the process computing `(map (*2) hamming)` will be sent to the main process which passes it further to the process computing `(uncurry sm ...)`. We will later show how channels can be created explicitly to cope with this problem.

◁

2.3.1 Reducing Communication Costs: Chunking

The default stream communication in Eden produces a single message for each stream element. This may lead to high communication costs and in many cases it is advantageous to communicate a stream in larger segments. Note that this chunking is not always advisable. In the Hamming network it is e.g. important that elements are transferred one-by-one — at least at the beginning — because the output of the network depends on the previously produced elements. If there is no such dependency, the decomposition of a stream into chunks reduces the number of messages to be sent and in turn the communication overhead. The following definition shows how chunking can be defined for parallel `map` skeletons like `parMap` and `farm`. The auxiliary function `chunk` decomposes a list into chunks of a given size. The function `chunkMap` simply applies `chunk` on the input list and afterwards concatenates the result chunks using the predefined Haskell function `concat :: [[a]] -> [a]`.

```
chunkMap :: Int
         -> (([a] -> [b]) -> ([[a]] -> [[b]]))
         -> (a -> b) -> [a] -> [b]
chunkMap size mapscheme f xs
  = concat (mapscheme (map f) (chunk size xs))

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

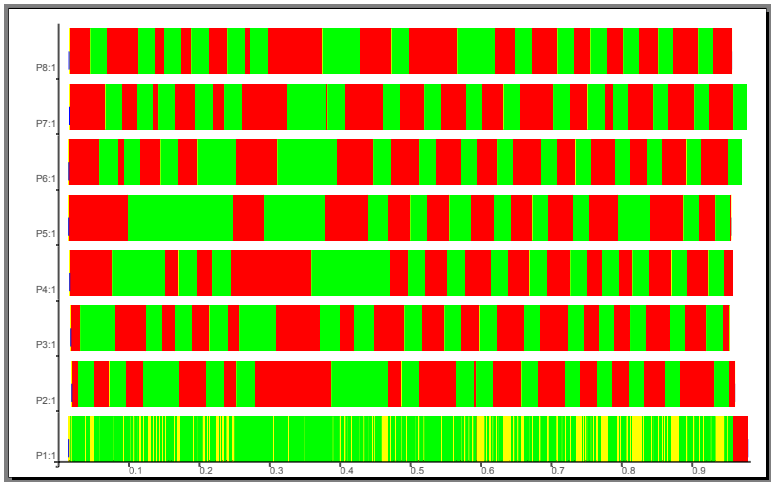
Chunking may substantially reduce communication costs, as the number of messages drops when chunking is used.

Case Study (Raytracer continued): Changing the raytracer program by

1. generating only `noPe-1` farm processes to avoid that two processes have to share a machine
2. reducing the number of messages by chunking the input and output streams of processes

leads to better results (see Figure 2). Note that the input size is 5 times the input size of the previously shown trace. It becomes clear that processes are now active half of the time, but still blocked the rest of time waiting for input from the main process. Only the main process is busy most of the time sending input to the farm processes. The number of messages has drastically been reduced, thereby improving the runtime behaviour a lot. However, the program behaviour can still be improved. ◇

We can even act more radically and reduce communication costs further for data transfers from parent to child processes.



input size 500
runtime 0.978566s
8 machines
8 processes
15 threads
42 conversations
292 messages

Figure 2: Raytracer trace on 8 PE, farm with 7 processes and chunking

2.3.2 Communication vs Parameter Passing: Running Processes Offline

Eden processes have disjoint address spaces. Consequently, on process instantiation, the process abstraction will be transferred to the remote processing element including its whole environment, i.e. the whole heap reachable from the process abstraction will be copied. This is done even if the heap includes non-evaluated parts which may cause the duplication of work. A programmer is able to avoid work duplication by forcing the evaluation of unevaluated sub-expressions of a process abstraction before it is used for process instantiation.
There exist two different approaches for transferring data to a remote child process. Either the data is passed as a parameter or sub-expression (without prior evaluation unless explicitly forced) or data is communicated via a communication channel. In the latter case the data will be evaluated by the parent process before sending.

Example: Consider the function `fun2proc` defined by

```
fun2proc :: (a -> b -> c) -> a -> Process b c
fun2proc f x = process (\ y -> f x y)
```

and the following process instantiation:

```
fun2proc fexpr xarg # yarg
```

When this process instantiation is evaluated, the process abstraction `fun2proc fexpr xarg` (including all data referenced by it) will be copied and sent to the processing element where the new process will be evaluated. The parent process creates a new thread for evaluating the argument expression `yarg` to normal form and a corresponding outport (channel). Thus, the expressions `fexpr` and `xarg`

will be evaluated lazily if the child process demands their evaluation, while the expression `yarg` will immediately evaluated by the parent process. ◁

If we want to evaluate the application of a function `h :: a -> b` by a remote process, there are two possibilities to produce a process abstraction and instantiate it:

1. If we simply use the operator `($#)`, the argument of `h` will be evaluated by the parent process and then passed to the remote process.

2. Alternatively, we can pass the argument of `h` within the process abstraction and use instead the following *remote function invocation*.

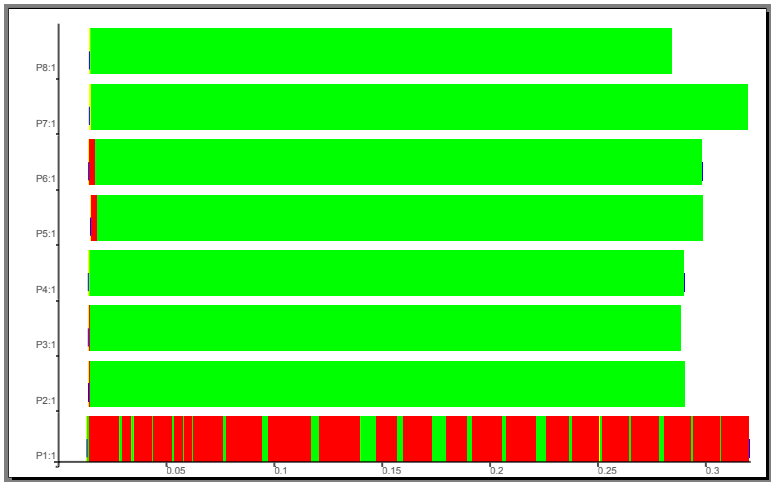
```
rfi      ::  (a -> b) -> a -> Process () b
rfi h x  =  process (\ () -> h x)
```

Now the argument of `h` will be evaluated on demand by the remote process itself. The empty tuple `()` (unit) is used as a dummy argument in the process abstraction. If the communication network is slow or if the result of the argument evaluation is large, instantiation via `rfi h x # ()` may be more efficient than using `(h $# x)`. We say that the child process runs *offline*.
With this technique the previously defined `farm` can easily be transformed into the following *offline farm*, which can equally well be used to parallelise `map` applications.

```
offline_farm ::  (Trans a, Trans b) =>
                 ([a] -> [[a]]) ->      -- input distribution
                 ([[b]] -> [b]) ->      -- result combination
                 (a->b) -> [a] -> [b]   -- map interface
offline_farm distribute combine f xs
    = combine $ spawn (map (rfi (map f)) (distribute xs)) (repeat ())
```

Although the input is not explicitly communicated to an `offline_farm`, chunking may be useful, because it also causes the result stream to be sent in chunks.

Case Study (Raytracer continued 2): Using the offline farm instead of the farm in our raytracer case study eliminates the explicit communication of all input rays to the farm processes. The processes now evaluate their input by themselves. Only the result values are communicated. Figure 3 shows that the farm processes are now active during all their life time. The runtime is much smaller. ◇



input size 500
runtime 0.320841s
8 machines
8 processes
15 threads
35 conversations
160 messages

Figure 3: Raytracer trace on 8 PE, offline farm with 7 processes and chunking

2.3.3 Dynamic channels and Remote Data

With the Eden constructs introduced up to now, communication channels are only (implicitly) established between parent and child processes during process creation. These are called *static channels* and they build purely hierarchical process topologies.

Example: Consider the following definition of a process ring in Eden. The number of processes in the ring is determined by the length of the input list. The ring processes are created using the `parMap` function.

```
ring      :: (Trans i, Trans o, Trans r) =>
             ((i,r) -> (o,r))   --^ ring process function
             -> [i] -> [r]      --^ input - output mapping
ring f is =  os
  where
    (os,ringOuts)  = unzip (parMap f (lazyzip is ringIns))
    ringIns        = rightRotate ringOuts

lazyzip                 :: [a] -> [b] -> [(a,b)]
lazyzip []      _       =  []
lazyzip (x:xs) ~(y:ys) =  (x,y) : lazyzip xs ys

rightRotate    :: [a] -> [a]
rightRotate [] =  []
rightRotate xs =  last xs : init xs
```

The auxiliary function `lazyzip` corresponds to the Haskell prelude function `zip` but is lazy in its second argument (because of using the lazy pattern `~(y:ys)`). This is crucial because the second parameter `ringIns` will not be available when

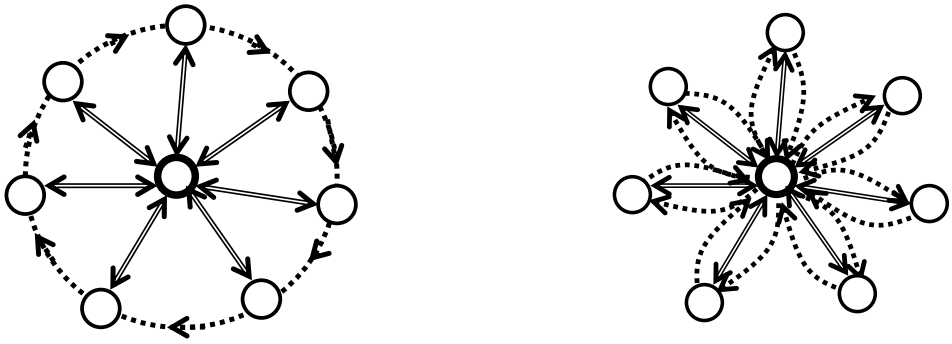


Figure 4: Topology of process ring (left: intended topology, right: actual topology)

the `parMap` function creates the ring processes. Note that the list of ring inputs `ringIns` is the same as the list of ring outputs `ringOuts` rotated by one element to the right using the auxiliary function `rightRotate`.
Unfortunately, this elegant and compact ring definition will not produce a ring topology but a star (see Figure 4). The reason is that the channels for communication between the ring processes are not established in a direct way, but only indirectly via the parent process. One could produce the ring as a chain of processes when each ring process creates its successor but this approach would cause the input from and output to the parent process to run through the chain of predecessor processes. Moreover it is not possible to close this chain to form a ring. ◁

To overcome this problem Eden provides functions to explicitly create and use *dynamic* channel connections between arbitrary processes:

```
new      ::  Trans a => (ChanName a -> a -> b) -> b
parfill  ::  Trans a =>  ChanName a -> a -> b  -> b
```

By evaluating `new (\ name val -> e)` a process creates a dynamic channel `name` of type `ChanName a` in order to receive a value `val` of type `a`. After creation, the channel should be passed to another process (just like normal data) inside the expression result `e`, which will as well use the eventually received value `val`. The evaluation of `(parfill name e1 e2)` in the other process has the side-effect that a new thread is forked to concurrently evaluate and send the value `e1` via the channel. The overall result of the expression is `e2`.
These functions are rather low-level and it is not easy to use them appropriately. Let us suppose that process A wants to send data directly to some process B

by means of a dynamic channel. This channel must first be generated by the process B and sent to A before the proper data transfer from A to B can take place. Hence, the dynamic channel is communicated in the opposite direction to the desired data transfer.

Example: Using dynamic channels the ring channels can be established in the intended way. The ring definition must only slightly be modified. It suffices to replace the application of the ring function `f` with a modified version `plink f` which introduces dynamic channels to transfer the ring input. Instead of passing the ring data via the parent process, only the channel names are now passed via the parent process from successor to predecessor processes in the ring. The ring data is afterwards directly passed from predecessors to successors in the ring. Note the the orientation of the ring must now be changed which is done by using `leftrotate` instead of `rightrotate` in the definition of `ring`.

```
ringDC      :: (Trans i, Trans o, Trans r) =>
               ((i,r) -> (o,r))   -- ^ ring process function
               -> [i] -> [r]      -- ^ input - output mapping
ringDC f is =  os
  where
    (os,ringOuts)  = unzip (parMap (plink f) (lazyzip is ringIns))
    ringIns        = leftRotate ringOuts

leftRotate        :: [a] -> [a]
leftRotate []     =  []
leftRotate (x:xs) =  xs ++ [x]

plink :: (Trans i, Trans o, Trans r) =>
         ((i,r) -> (o,r))          -- ^ ring process function
         -> ((i, ChanName r)
              -> (o, ChanName r))  -- ^ .. with dynamic channels
plink f (i, outChan)
  = new (\ inChan ringIn ->
         parfill outChan ringOut (o, inChan))
  where (o, ringOut) = f (i, ringIn)
```

Each ring process creates an input channel which is immediately returned to the parent process and passed to the predecessor process. Each ring process receives from the parent a channel to send data to the successor in the ring. It uses this channel to send the ring output `ringOut` to its successor process by a concurrent thread created by the `parfill` function. The original ring function `f` is applied to the parent's input and the ring input received via the dynamic ring input channel. It produces the output for the parent process and the ring output `ringOut` for the successor process in the ring. In fact, the star topology is still used but only to propagate channel names. The proper data is now passed directly from one process to its successor in the ring.

Although this definition leads to the intended topology, the correct and effective use of dynamic channels is not obvious. ◁

A more abstract, more natural and easier-to-use way to define non-hierarchical process networks like rings in Eden are *remote data* [1]. The main idea is to replace the data to be communicated between processes by handles to it, called remote data. These handles can then be used to convey the real data directly to the desired target. Thus, a remote data of type `a` is represented by a handle of type `RD a` with interface functions `release` and `fetch`. The function `release` produces a remote data handle that can be passed to other processes, which will in turn use the function `fetch` to access the remote data. The data transmission occurs automatically from the process that releases the data to the process which uses the handle to fetch the remote data.
Remote data can be defined in Eden as follows [15]:

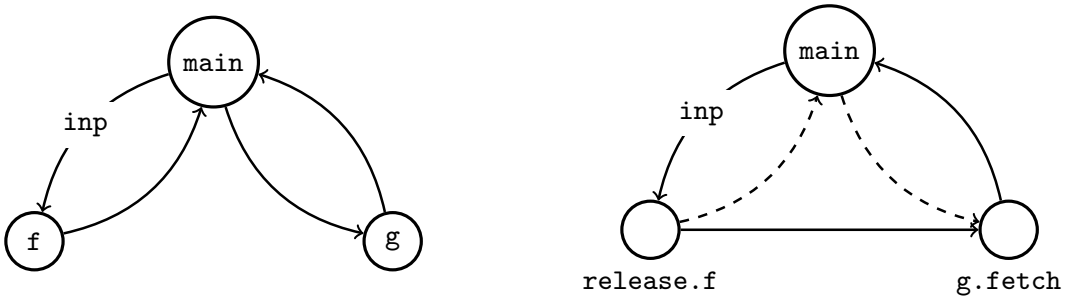
```
type RD a = ChanName (ChanName a) -- remote data

-- convert local data into remote data
release ::  Trans a => a -> RD a
release x = new (\ cc c -> parfill c x cc)

-- convert remote data into local data
fetch ::  Trans a => RD a -> a
fetch cc = new (\ c x -> parfill cc c x)
```

Notice how the remote data approach preserves the direction of the communication (from process A to process B) by introducing another channel transfer from A to B. This channel will be used by B to send its (dynamic) channel data to A, and thus establish the direct data communication. More exactly, to `release` local data `x` of type `a`, a dynamic channel `cc` of type `RD a`, i.e. a channel to transfer a channel name. When a channel `c` (of type `ChanName a`) is received via `cc`, the local data `x` is sent via `c`. Conversely, in order to `fetch` remote data, represented by the remote data handle `cc`, a new (dynamic) channel `c` is created and the channel name is sent via `cc`. The proper data will be received via the channel `c`. The next simple example illustrates how the remote data concept is used to establish a direct channel connection between sibling processes.

Example: Given functions `f` and `g`, the expression `(g . f) a` can be calculated in parallel by creating a process for each function. One just replaces the function calls by process instantiations

```
(g $# (f $# inp)).
```

This leads to the process network in Figure 5 (a) where the process evaluating the above expression is called `main`. Process `main` instantiates a first process for calculating `g`. In order to transmit the corresponding input to this new process, `main` instantiates a second process for calculating `f`, passes its input to this process



(a) indirect connection (b) direct connection

Figure 5: A simple process graph

and receives the remotely calculated result, which is passed to the first process. The output of the first process is also sent back to `main`. As in the ring example the drawback of this approach is that the result of the process calculating `f` is not sent directly to the process calculating `g`, thus causing unnecessary communication costs.
In the second implementation, we use remote data to establish a direct channel connection between the child processes (see Figure 5 (b)):

```
(g . fetch) $# ((release . f) $# inp)
```

The output produced by the process calculating `f` is now encapsulated in a remote handle that is passed to the process calculating `g`, and fetched there. Notice that the remote data handle is treated like the original data in the first version, i.e. it is passed via the main process from the process computing `f` to the one computing `g`. ◁

Example: Also the process ring considered before can easily be re-defined using the remote data concept:

```
ringRD      :: (Trans i, Trans o, Trans r) =>
               ((i,r) -> (o,r))   --^ ring process function
               -> [i] -> [r]      --^ input - output mapping
ringRD f is =  os
  where
    (os,ringOuts)  = unzip (parMap (toRD f) (lazyzip is ringIns))
    ringIns        = rightRotate ringOuts

toRD :: (Trans i, Trans o, Trans r) =>
         ((i,r) -> (o,r))          -- ^ ring process function
         -> ((i, RD r)
              -> (o, RD r))        -- ^ .. with remote data
toRD f (i, ringIn) = (o, release ringOut)
      where (o, ringOut) = f (i, fetch ringIn)
```

Again the original ring function is embedded using the auxiliary function `toRD` into a function which replaces the ring data by remote data and introducing calls to `fetch` and `release` at appropriate places. The orientation of the original ring can be kept. Note that the double number of channel names are now passed to establish the intended ring topology. However, the remote data definition is much easier than using dynamic channels directly. ◁

We conclude this subsection with another example which shows the definition of the Hamming process network using remote data in order to establish the intended process topology shown before.

Example: In order to introduce direct communication between the processes in the Hamming process network, one has to introduce appropriate calls to `fetch` and `release` where sibling processes should directly communicate:

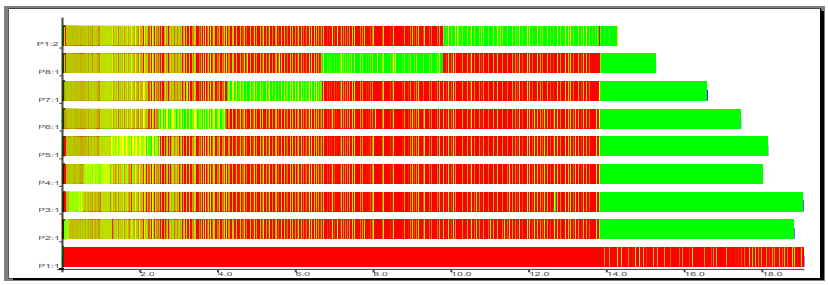
```
hamming :: [Int]
hamming = 1: sm ((toRDuncurry sm) $# ((release.(map (*2))) $# hamming,
                                      ((release.(map (*3))) $# hamming))
                ((map (*5)) $# hamming)

toRDuncurry :: ([a] -> [a] -> [a]) -> ([RD a],[RD a]) -> [a]
toRDuncurry  f (rd1, rd2) = f (fetch rd1) (fetch rd2)
```

The function `uncurry` is replaced with a special variant, which expects remote data as input. The data must be fetched before the original parameter function can be applied to it. Communication between parent and child processes need not be done using remote data handles. ◁

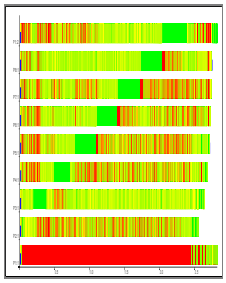
The remote data concept enables the programmer to easily build complex topologies by combining simpler ones [15].

Case Study (Warshall's algorithm): Warshall's algorithm for computing shortest paths in a graph given by an adjacency matrix can be implemented using a process ring. Each ring process is responsible for the update of a subset of rows. The whole adjacency matrix is rotated once around the process ring. Thus, each process sees each other row of the whole adjacency matrix. The kernel of the implementation is the iteration function executed by the ring processes for each row assigned to them.

```
ring_iterate :: Int -> Int -> Int ->
                            [Int] -> [[Int]] -> ( [Int], [[Int]])
ring_iterate size k i rowk (rowi:xs)
    | i > size  =  (rowk, [])    --  Finish Iteration
    | i == k    =  (rowR, rowk:restoutput) - send own row
    | otherwise =  (rowR, rowi:restoutput) - update row
 where
  (rowR, restoutput) = ring_iterate size k (i+1) nextrowk xs
```



19.079864s
(a) without demand control



2.922803s
(b) with demand

Runtimes are shown above. The other statistical information is for both versions the same: input size 500, 8 machines, 9 processes, 41 threads, 72 conversations, 4572 messages

Figure 6: Warshall trace on 8 PE, ring with 8 processes and chunking

```
nextrowk | i == k    = rowk -- no update, if own row
         | otherwise = updaterow rowk rowi (rowk!!(i-1))
```

In the kth iteration the process with row k sends this row into the ring. During the other iterations each process updates its own rows with the information of the row received from its ring predecessor.
Unfortunately, a trace analysis reveals (see Figure 6(a)) that the parallel version has a demand problem. In a first phase, an increasing phase of activity runs along the ring processes, until the final row is sent into the ring. Then all processes start to do their update computations. By forcing the immediate evaluation of `nextrowk`, i.e. the update computation, the sequential start-up phase of the ring can be compressed. The additional demand can be expressed by `rnf nextrowk` `` `pseq` `` which has to be included before the recursive call to ring_iterate. This demands the evaluation of `nextrowk` to normal form before the second argument of `pseq` is evaluated. The effect of this small code change is enormous as shown in Figure 6(b).

◇

2.3.4 Many-to-one communication: Merging Communication Streams

Many-to-one communication is an essential feature for many parallel applications, but, unfortunately, it introduces non-determinism and, in consequence, spoils the purity of functional languages. In Eden, the predefined function

```
merge :: [[a]] -> [a]
```

merges (in a non-deterministic way) a list of streams into a single stream. Functional purity can be preserved in most portions of an Eden program. It is e.g.

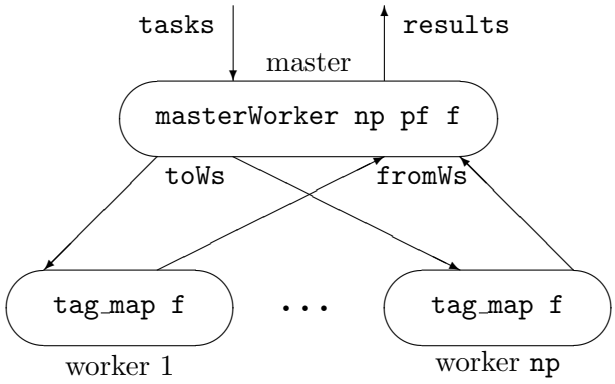


Figure 7: Master-worker process topology

```
masterWorker :: (Trans a, Trans b) =>
                Int -> Int -> (a->b) -> [a] -> [b]
masterWorker np prefetch f tasks
 = orderBy fromWs reqs
 where
    fromWs   = spawn workers toWs
    workers  = [process (tag_map f) | n <-[1..np]]
    toWs     = distribute np tasks reqs
    newReqs  = merge [[i | r <- rs] | (i,rs) <- zip [1..np] fromWs]
    reqs     = initReqs ++ newReqs
    initReqs = concat (replicate prefetch [1..np])
```

Figure 8: Eden implementation of a simple master worker scheme

possible to use sorting in order to force a particular order of the results returned by a `merge` application and thus to encapsulate `merge` in a deterministic context. The `merge` function can be used to react quickly to requests coming in an unpredictable order from a set of processes, and thus enable dynamic load balancing of parallel programs in a master-worker scheme as shown in Figure 7. The master process distributes tasks to worker processes, which solve the tasks and return the results to the master.

The Eden function `masterWorker` (evaluated by the "master" process) (see Figure 8) takes four parameters: `np` specifies the number of worker processes that will be spawned, `prefetch` determines how many tasks will initially be sent by the master to each worker process, the function `f` describes how tasks have to be solved by the workers, and the final parameter `tasks` is the list of tasks that have to be solved by the whole system. The auxiliary pure Haskell function `distribute :: Int -> [a] -> [Int] -> [[a]]` is used to distribute the tasks to the work-

ers. Its first parameter determines the number of output lists, which become the input streams for the worker processes. The third parameter is the request list `reqs` which guides the task distribution. The request list is also used to sort the results according to the original task order (function `orderBy :: [[b]] -> [Int] -> [b]`):

```
distribute :: Int  -- ^number of workers
              -> [t]   -- ^ task list
              -> [Int] -- ^ request stream  with worker IDs
              -> [[t]] -- ^ each inner list for one worker
distribute np reqs tasks = [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 _      _      _  = []

orderBy :: [[r]]    -- ^ nested input list
           -> [Int] -- ^ request stream gives distribution
           -> [r]   -- ^ ordered result list
orderBy rss [] = []
orderBy rss (r:reqs)
  = let (rss1,(rs2:rss2)) = splitAt r rss
    in  (head rs2): orderBy (rss1 ++ ((tail rs2):rss2)) reqs
```

Initially, the master sends as many tasks as specified by the parameter `prefetch` in a round-robin manner to the workers (see definition of `initReqs` in Figure 8). Further tasks are sent to workers which have delivered a result. The list `newReqs` is extracted from the worker results tagged with the corresponding worker id which are merged according to the arrival of the worker results. This simple master worker definition has the advantage that the tasks need not be numbered to re-establish the original task order on the results. Moreover, worker processes need not send explicit requests for new work together with the result values.

Case Study (Mandelbrot): The kernel function of a program to compute a two-dimensional image of the Mandelbrot set for given complex coordinates `ul` (upper left) and `lr` (lower right) and the number of pixels in the horizontal dimension `dimx` as a string can be written as follows in Haskell:

```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
  = header ++ (concat $ map xy2col lines)
  where
    xy2col         :: [Complex Double] ->  String
    xy2col line    =  concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
    (dimy, lines)  =  coord ul lr dimx
```

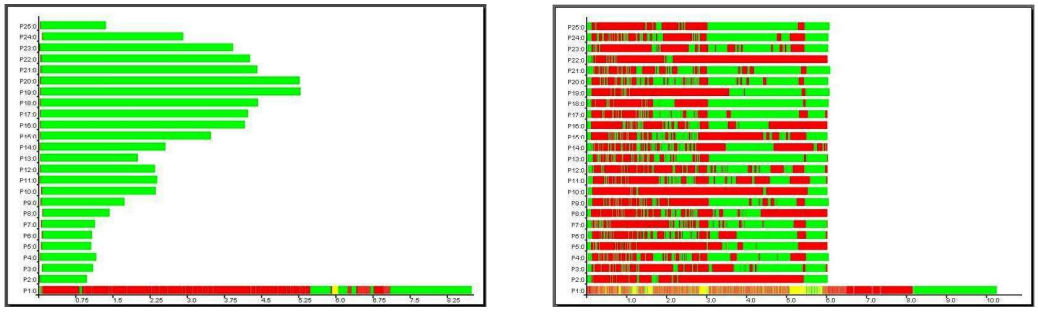


Figure 9: Mandelbrot traces on 25 PEs, offline farm with chunked tasks (left) vs master-worker (right)

The first parameter is a threshold for the number of iterations that should be done to compute the color of a pixel.
The program can easily be parallelised by replacing `map` in the expression `map xy2col lines` by a parallel map implementation. Figure 9 shows two traces that have been produced on a Beowulf cluster at Heriot-Watt University in Edinburgh (32 Intel P4-SMP nodes @ 3 GHz 512MB RAM, Fast Ethernet). The Mandelbrot program was evaluated with an input size of 2000 lines with 2000 pixels each using 25 PEs. The program that produced the trace on the left hand side replaced `map` by `offline_farm (chunk (2000 ‘div‘ 25)) concat`. The program yielding the trace on the right hand side replaced `map` by `masterWorker 25 8`.
In the offline farm, all worker processes are busy during their life time, but we observe an unbalanced workload which reflects the shape of the mandelbrot set. To be honest, the uneven workload has been enforced by using `chunk (2000 ‘div‘ 25)` for task distribution instead of `unshuffle 25`. The latter would even lead to a well-balanced workload which is however a special property of this example problem. In general, irregular parallelism cannot easily be balanced. The master-worker system uses a dynamic task distribution. In our example a prefetch value of 8 has been used to initially provide 8 tasks to each PEs. The trace in Figure 9 reveals that the workload is better balanced, but worker processes are often blocked waiting for new tasks. Unfortunately, the master process on machine 1 (lowest bar) is not able to keep all worker processes busy. In fact, the master-worker parallelisation needs more time than the badly balanced offline farm. In addition, both versions suffer from a long end phase in which the main or master process collects the results. ◇

3 Algorithmic Skeletons

Algorithmic skeletons [13] provide commonly used patterns of parallel evaluation which can simply be used by programmers to parallelise their applications. This

version	number of processes	task transfer	task distribution
parMap	number of tasks	communication	one per process
farm	mostly noPe	communication	static
offline_farm	mostly noPe	local selection	static
masterWorker	mostly noPe	communication	dynamic

Figure 10: Classification of parallel map implementations

facilitates the development of parallel programs. The functions `parMap`, `farm`, and `masterWorker` defined above are simple examples for skeleton definitions in Eden. In general, an algorithmic skeleton can be implemented in several ways. Implementations may differ in the process topology created, in the granularity of tasks, in the load balancing strategy or in the target architecture used to run the program. These details are transparent to the skeleton user, but determine the efficiency of the program. Nevertheless it is possible to predict the efficiency of skeleton instantiations by providing a *cost model* for each particular skeleton implementation [29].
While most skeleton systems provide pre-defined, specially supported sets of skeletons, the application programmer has usually not the possibility of creating new ones. In Eden, skeletons can be *used*, *modified* and *newly implemented*, because (like in other parallel functional languages) skeletons are no more than polymorphic higher-order functions which can be applied with different types and parameters. Thus, programming with skeletons in Eden follows the same principle as programming with higher-order functions. Moreover, describing both the functional specification and the parallel implementation of a skeleton in the same language context constitutes a good basis for formal reasoning and correctness proofs, and provides greater flexibility.
In a way similar to the rich set of higher-order functions provided by Haskell's prelude, Eden provides a well assorted skeleton library. Next we present a small selection of typical Eden skeletons.

3.1 Parallel Maps and Workpools

In Section 2 we have defined several parallel implementations of `map`, a simple form of data parallelism. A given function has to be applied to each element of a list. The list elements can be seen as tasks, to which a worker function has to be applied. The parallel map skeletons we developed up to now can be classified as shown in Figure 10. The simple `parMap` is mainly used to create a series of processes evaluating the same function. The static task distribution implemented in `farm` and `offline_farm` is especially appropriate for regular parallelism, i.e. when all tasks have same complexity or can be distributed in such a way that

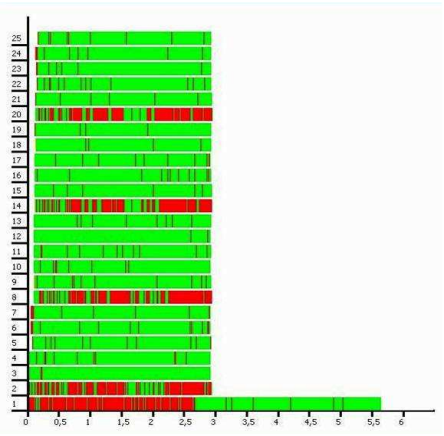
the workload of the processes is well-balanced. The master worker approach with dynamic task distribution is suitable for irregular tasks.
The master worker skeleton defined above is a very simple version of a workpool skeleton. It has a single master process with a central workpool, the worker processes have no local state and cannot produce and return new tasks to be entered into the workpool. Several more sophisticated workpool skeletons are provided in the Eden skeleton library. We will here exemplarily show how a hierarchical master worker skeleton can elegantly be defined in Eden. For details see [6, 38]. As a matter of principle, a nested master-worker system can be defined by using the simple master worker skeleton defined above as the worker function for the upper levels. The simple master worker skeleton must only be modified in such a way that the worker function has type `[a] -> [b]` instead of `a -> b`. The nested scheme is then simply achieved by folding the zipped list of branching degrees and prefetches per level. This produces a regular scheme. The proper worker function is used as the starting value for the folding. Thus it is used at the lowest level of worker processes.

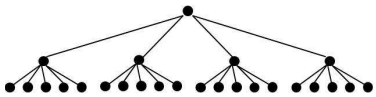
```
mwNested :: (Trans a, Trans b) =>
            [Int]           -- ^ branching degrees per level
            -> [Int]        -- ^ prefetches per level
            -> ([a] -> [b]) -- ^ worker function
            -> [a] -> [b]   -- ^ tasks, results
mwNested ns pfs wf = foldr fld wf (zip ns pfs)
  where
    fld :: (Trans a, Trans b) =>
           (Int,Int) -> ([a] -> [b]) -> ([a] -> [b])
    fld (n,pf) wf = masterWorker' n pf wf
```

Case Study (Mandelbrot continued): Using a nested master-worker system helps to improve the computation of Mandelbrot sets on a large number of processor elements. Figure 11 shows an example trace produced for input size 2000 on 25 PEs with a two-level master worker system comprising four sub-masters serving five worker processes each. Thus, the function `mwNested` has been called with parameters `[4,5]` and `[64,8]`. The trace clearly shows that the work is well-balanced among the 20 worker processes. Even the post-processing phase in the main process (top-level master) could be reduced, because the results are now collected level-wise. The overall runtime could substantially be reduced in comparison to the simple parallelisations discussed previously (see Figure 9). ◇

3.2 Divide-and-conquer

Divide-and-conquer is a well-known algorithmic technique in sequential programming: it divides or (*splits*) an instance of a problem into two or more smaller





branching degrees [4,5]:
1 master, 4 sub-masters, 5 workers per submaster
prefetches [64,8]:
64 tasks per submaster, 8 tasks per worker

Figure 11: Mandelbrot trace on 25 PEs with hierarchical master-worker skeleton (hierarchy shown on the right)

instances of the original problem. If these subproblems *are trivial* enough to be *solved* directly, the solution of the original instance is obtained by *combining* the sub-solutions. If the subproblems are still too large to be solved readily, they are (recursively) divided into still smaller instances. The Haskell version of this strategy is the following polymorphic higher-order function:

```
type DivideConquer a b
  = (a -> Bool) -> (a -> b)            -- ^ trivial? / solve
      -> (a -> [a]) -> (a -> [b] -> b) -- ^ split / combine
      -> a -> b                        -- ^ input / result
seqDC :: DivideConquer a b
seqDC is_trivial solve split combine = dc
  where dc x = if trivial x then solve x
                            else combine x (map dc (split x))
```

The resulting structure is a tree of task nodes where successor nodes correspond to the sub-problems, the leaves representing trivial tasks. Notice that the call tree may be non-homogeneous, and that trivial solutions (i.e. leaves) may appear at any level of the tree.
The most direct way to parallelise the `seqDC` scheme in Eden is to replace `map` by `parMap`. An additional integer parameter `d` can be used to stop the parallel unfolding at a given level and to use the sequential version instead:

```
parDC :: (Trans a, Trans b) =>
         Int ->  -- depth
         DivideConquer a b
parDC 0 = seqDC
parDC d trivial solve split combine = pdc d
  where
```

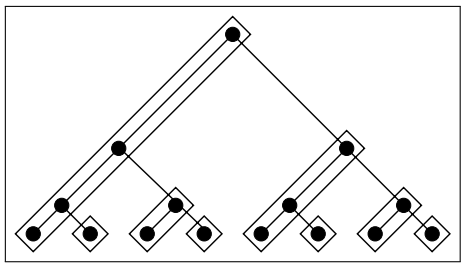
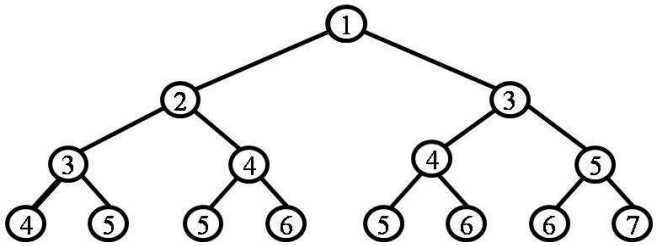


Figure 12: Distributed expansion divide and conquer skeleton for a binary task tree

```
pdc d x = if trivial x then solve x
                       else combine x results
  where results = parMap (pdc (d-1)) (split x)
```

In this approach a dynamic tree of processes is created with each process connected to its parent. With the default round robin placement of child processes, the processes are not evenly distributed on the available PEs.

Example: If 8 PEs are available and if we consider a regular divide-and-conquer tree with branching degree 2 and three recursive unfoldings, then 14 processes will be created and allocated on PEs 1 to 7, as indicated by the following tree:



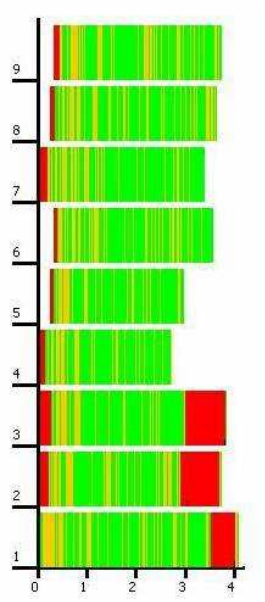
While no process is allocated on PE 8, PE 5 gets four processes, and three processes are allocated on PEs 4 and 6 each. ◁

The following parallel divide-and-conquer implementation provides a better load balance. It is assumed that every non-trivial task is split into a *fixed* number of subtasks. Now the process tree is created in a *distributed* fashion: One of the tree branches is processed locally, while the others are instantiated as new processes, as long as processor elements (PEs) are available. These branches will recursively produce new parallel subtasks, resulting in a *distributed expansion* of the computation. Figure 12 shows the binary tree of task nodes produced by a d&c strategy splitting each non-trivial task into two subtasks, in a context with 8 PEs. The boxes indicate which task nodes will be evaluated by each PE.
In this setting, explicit placement of processes is essential to avoid that too many processes are placed on the same PE while leaving others unused.

```
disDC :: (Trans a, Trans b) =>
              Int -> [Int] ->    -- branch degree / tickets
              DivideConquer a b
disDC k [] = seqDC
disDC k tickets is_trivial solve split combine
  = rec_disDC tickets
  where
    rec_disDC tickets =
        if trivial x then solve x
        else ...  -- code for demand control omitted
             combine x ( myRes:childRes ++ localRess )
        where
          -- child process generation
          childRes   = spawnAt childTickets childProcs procIns
          childProcs = map (process . rec_disDC) theirTs
          -- ticket distribution
          (childTickets, restTickets) = splitAt (k-1) tickets
          (myTs: theirTs)             = unshuffle k restTickets
          -- input splitting
          (myIn:theirIn)      = split x
          (procIns, localIns) = splitAt (length childTickets) theirIn
          -- local computations
          myRes     = rec_disDC myTs myIn
          localRess = map seqDC is_trivial solve split combine localIns
```

Figure 13: Eden definition of distributed expansion divide-and-conquer skeleton

Figure 13 shows the code for the distributed expansion d&c skeleton for k-ary task trees. Two additional parameter are needed: the branching degree and a ticket list with PE numbers to place newly created processes. As explained above, the left-most branch of the task tree is solved locally (`myIn`), other branches (`theirIn`) are instantiated using the Eden function `spawnAt` (see Section 2).
Observe how the ticket list is used to control the placement of newly created processes: First, the PE numbers for placing the immediate child processes are taken from the ticket list; then, the remaining tickets are distributed to the children in a round-robin manner using the function `unshuffle :: Int -> [a] -> [[a]]` which distributes the elements of a given list into as many lists as indicated by the first parameter. Computations corresponding to children will be performed locally (`localIns`) when no more tickets are available. The explicit process placement via ticket lists is a simple and flexible way to control the distribution of processes as well as the recursive unfolding of the task tree. If too




```
type LongInteger = (Int,[Int])
karat :: Int
         -> LongInteger
         -> LongInteger
         -> LongInteger
karat depth is1 is2
  = disDC 3 tickets
          trivial
          solve
          split
          combine
          (is1,is2)
  where tickets = [2..noPe]
```

Figure 14: Karatsuba trace on 9 PEs and code with distributed expansion dc skeleton

few tickets are available, computations are performed locally. Duplicate tickets can be used to allocate several child processes on the same PE.

Case Study (Karatsuba): Karatsuba's algorithm [24] computes the product of two large integers using a regular divide-and-conquer approach. An integer is represented in base b as a list of "digits" d_i, where for all i we have $0 \leq d_i < b$. If n is the length of the input integer representations, a divide step produces three subproblems where integers of length $n/2$ have to be multiplied. This algorithm perfectly fits into the regular divide-and-conquer scheme. In terms of the $\int$ skeleton `disDC`, the Karatsuba algorithm can be implemented as shown in Figure 14 where the functions `trivial`, `split` and `combine` are straightforward Haskell formulations of the corresponding mathematical definitions. The granularity of the three subtasks is not uniform, since the multiplications potentially uses integers of different lengths. In addition, trivial nodes (number size below a certain threshold) may appear at any depth in the tree.
Figure 14 shows the trace produced by the Karatsuba program parallelised with the distributed expansion divide-and-conquer skeleton on 7 dual-core linux workstations with 2 GB RAM and a Fast Ethernet LAN. The input were 2 integers with 32768 digits each.

◇

3.3 Torus

The next skeleton is an example of skeleton that produces a particular process topology. A torus is a two-dimensional topology in which each process is connected to its four neighbours. The first and last processes in each row and column are considered neighbours. In addition, each node has two extra connections to send and receive values to/from the parent. This topology is an example of systolic skeleton, where processes alternate parallel computation and global synchronisation steps. At each round, every node receives messages from its left and upper neighbours, computes, and then sends messages to its right and lower neighbours.
The implementation that we propose in Eden uses lists instead of synchronization barriers to simulate rounds. The remote data approach is used to establish direct connections between the torus nodes. The `torus` function defined in Figure 15 creates the desired toroidal topology by properly connecting the inputs and outputs of the different `ptorus` processes. Each process receives an input from the parent, and two remote handles to be used to fetch the values from its neighbours. It produces an output to the parent and two remote handles to release outputs to its neighbours. The shape of the torus is determined by the shape of the input. The source code of the skeleton is shown in Figure 15.
The size of the torus will usually depend on the number of available processors (`(`noPe). A typical value is $\lfloor\sqrt{\texttt{noPe}}\rfloor$. In this case each torus node can be placed on a different PE. The first parameter of the skeleton is the worker function, which receives an initial datum of type `c` from the parent, a list `[a]` from the left neighbour and a list `[b]` from its upper neighbour, and produces a final result `d` for its parent as well as results `[a]` and `[b]` for its right and lower neighbours, respectively. Functions `lazyzip3` and `lazyzipWith3` are lazy versions of functions of the `zip` family, the difference being that these functions use irrefutable patterns for parameters, corresponding to the torus interconnections.

Case Study (Matrix Multiplication): A typical application of the torus skeleton is the implementation of a block-wise parallel matrix multiplication [18]. Each node of the torus gets two blocks of the input matrices to be multiplied sequentially. It passes its blocks to the neighbour processes in the torus, the block of the first matrix to the neighbour in the row and the block of the second matrix to the neighbour in the column. It receives corresponding blocks from its neighbour processes and proceeds as before. If each process has seen each block of the input matrices which are assigned to its row or column, the computation finishes.
Figure 16 shows a trace of the torus parallelisation of matrix multiplication created on the Beowulf cluster for input matrices with dimension 1000. Messages are shown. It can be seen that all communication takes place in the beginning of the computation. This is due to Eden's push policy. Data is communicated as soon as it has been evaluated to normal form. As the processes simply pass ma-

```
torus :: (Trans a, Trans b, Trans c, Trans d) =>
         -> ((c,[a],[b]) -> (d,[a],[b])) -- ^ node function
         -> [[c]] -> [[d]]               -- ^ input-output mapping
torus f inss = outss
  where
    t_outss    = zipWith spawn (repeat (ptorus f)) t_inss
    t_inss     = lazyzipWith3 lazyzip3 inss inssA inssB
    (outss, outssA, outssB) = unzip3 (map unzip3 t_outss)
    inssA      = map rightRotate outssA
    inssB      = rightRotate outssB

-- each individual process of the torus
ptorus :: (Trans a, Trans b, Trans c, Trans d) =>
          ((c,[a],[b]) -> (d,[a],[b])) ->
          Process (c,RD [a],RD [b])
                  (d,RD [a],RD [b])
ptorus f = process \ (fromParent,          inA,           inB) ->
                     (toParent,   release outA,  release outB)
  where
    (toParent, outA, outB) = f (fromParent, fetch inA, fetch inB)


lazyzipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
lazyzipWith3 f (x:xs) ~(y:ys) ~(z:zs) = f x y z : mzipWith3 f xs ys zs
lazyzipWith3 _ _ _ _ = []

lazyzip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
lazyzip3 = lazyzipWith3 (:)
```

Figure 15: Eden torus skeleton

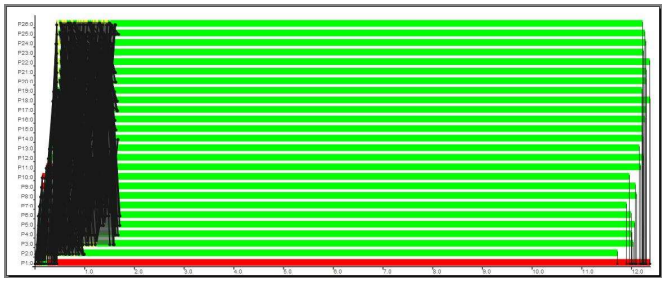


Figure 16: Trace of parallel matrix multiplication with torus skeleton

trix blocks without manipulating them, communication takes place immediately. Afterwards, the proper computations take place.

◇

4 Behind the Scenes: Eden's Implementation

Eden has been implemented by extending the runtime system (RTS) of the Glasgow Haskell compiler (GHC) with mechanisms for process management and communication. It shares its parallel runtime system (PRTS) with Glasgow parallel Haskell [43] but due to the disjoint address spaces of its processes does not need to implement a virtual shared memory and global garbage collection in contrast to GpH. In the following, we abstract from low-level implementation details like graph reduction and thread management which are explained elsewhere [37, 43, 12, 26] and describe Eden's implementation on top of EDI (Eden's implementation language) [9], a low-level coordination language, which provides primitive monadic operations which have been used to implement the Eden constructs on a higher-level of abstraction.
Eden's implementation has been organised in layers (see Fig. 17) to achieve more flexibility and to improve the maintainability of this highly complex system. The main idea has been to lift aspects of the runtime system (RTS) to the level of the functional language, i.e. defining basic workflows on a high level of abstraction in the Eden module and concentrating low-level RTS capabilities in a couple of primitive operations (module ParPrim.hs). In this way, part of the complexity has been eliminated from the imperative RTS level.
Every Eden program must import the Eden module, which contains Haskell definitions of Eden's language constructs. These Haskell definitions use primitive operations which are functions implemented in C that can be accessed from Haskell. The extension of GHC for Eden is mainly based on the implementation of these primitive operations, which provide the elementary functionality for Eden and form a low-level coordination language by themselves. We called this language EDI, the Eden implementation language.

4.1 Primitive Operations: Interface to the PRTS

The Eden module contains Haskell definitions of the high-level Eden constructs, thereby making use of the primitive operations shown in Fig. 18. The primitive operations implement basic actions which have to be performed directly in the runtime system of the underlying sequential compiler GHC[1].
Each Eden channel connects an outport of the sender process to an inport of the receiver process. There is a one-to-one correspondence between the threads of

[1]Note that, in GHC, primitive operations and types are distinguished from common functions and types by `#` as the last sign in their names.

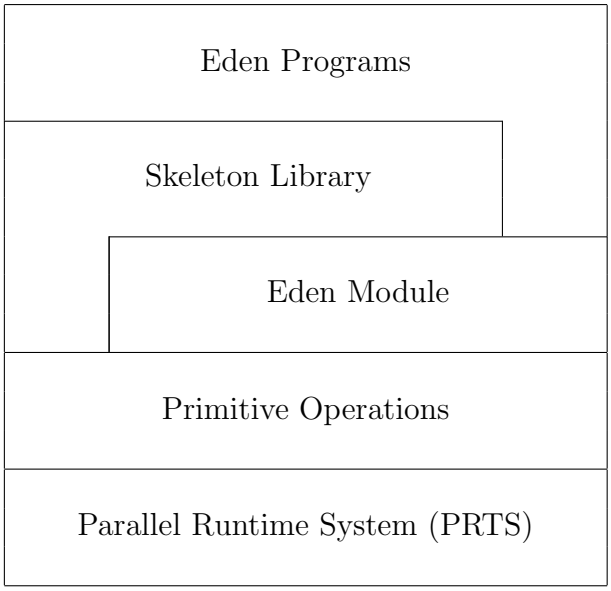


Figure 17: Layer structure of the Eden system

Channel Administration:

- `createC#` creates a placeholder and a new import for a new communication channel
- `connectToPort#` connects a communication channel in the proper way

Communication:

- `sendData#` sends data on a communication channel

Thread Creation:

- `fork#` forks a concurrent thread

General:

- `noPE#` determines number of processing elements in current setup
- `selfPE#` determines own processor identifier

Figure 18: Primitive Operations for Eden

---

a process and its outports. Each thread of a process evaluates some expression to normal form and sends the result via its outport. The primitive channels within the parallel runtime system are identified by three primitive integer values identifying the receiver side, i.e. the inport connecting a channel with a process. The three integers are (1) the processor element number, (2) the process number and (3) a specific port number:

```
data ChanName' a = Chan Int# Int# Int#
```

This type is only internally visible and used by the primitive channel administration functions. The wrapper functions of the primitive operations have the following types:

```
createC       :: IO ( ChanName' a, a )
connectToPort :: ChanName' a -> IO ()
```

Note that the wrapper functions always yield a result in the IO monad. Function `createC` creates a primitive channel and a handle to access the data received via this channel. Function `connectToPort` connects the outport of the thread executing the function call to a given channel i.e. the corresponding inport.
There is only a primitive for sending data but no one for receiving data. Receiving is done automatically by the runtime system which writes data received via an inport immediately into a placeholder in the heap. The wrapper function of the primitive `sendData#` has the following type:

```
sendData  :: Mode -> a -> IO ()
data Mode =  Connect | Stream | Data | Instantiate Int
```

There are four send modi and corresponding message types. Note that the messages are always sent via the outport associated with the executing thread. A `Connect` message is initially sent to connect the outport to the corresponding inport. This makes it possible to inform a sender thread when its results are no longer needed, e.g. when the placeholders associated with the inport are identified as garbage.
A `Data` message contains a data value which is sent in a single message. The mode `Stream` is used to send the values of a data stream. The `Instantiate i` message is sent to start a remote process on PE `i`.

4.2 The Type Class Trans

As explained in Section 2, the type class `Trans` comprises all types which can be communicated via channels. It is mainly used to overload communication for streams and tuples. Lists are transmitted in a *stream*-like fashion, i.e. element by element. Each component of a tuple is communicated via a separate primitive channel. This is especially important for recursive processes which depend on part of their own output (which is re-fed as input).

On the level of the Eden module, a channel is represented by a communicator, i.e. a function to write a value into the channel:

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

This simplifies overloading of the communication function for tuple types. The declaration of the `Trans` class is given in Figure 19. The context NFData (normal form data) is needed to ensure that transmissible data can be fully evaluated (using the overloaded function `rnf` (reduce to normal form)) before sending it. An overloaded operation `write :: a -> IO ()`) is used for sending data. Its default definition evaluates its argument using `rnf` and sends it in a single message using `sendData` with mode `Data`.
The function `createComm` creates an Eden channel and a handle to access the values communicated via this channel. The default definition creates a single primitive channel. The default communicator function is defined using the auxiliary function `sendVia` which connects to the primitive channel before sending data on it. Note that the communicator function will be used by the sender process while the channel creation will take place in the receiver process. The explicit connection of the sender outport to the inport in the receiver process helps to guarantee that at most one sender process will use a channel.

```
class NFData a => Trans a where
  write    :: a -> IO ()
  write x  =  rnf x `pseq` sendData Data x

  createComm :: IO (ChanName a, a)
  createComm = do (cx,x) <- createC
                  return (Comm (sendVia cx) , x)

-- Auxiliary send function
sendVia :: (NFData a, Trans a) =>
               (ChanName' a) -> a -> IO()
sendVia c d = do connectToPort c
                 sendData Connect d
                     write d
```

Figure 19: Type Class `Trans`

For streams, `write` is specialized in such a way that it evaluates each list element to normal form and transmits it using `sendData Stream`:

```
instance Trans a => Trans [a] where
  write list@[] = sendData Data list
  write (x:xs)  = do rnf x `pseq` sendData Stream x
                     write xs
```

For tuples (up to 9 components), channel creation is overloaded as shown exemplarily for pairs:

```
instance (Trans a, Trans b) => Trans (a,b) where \\
  createComm = do (cx,x) <- createC
                  (cy,y) <- createC
                  return (Comm (write2 (cx,cy)),(x,y))

-- auxiliary write function for pairs
write2 :: (Trans a, Trans b) =>
          (ChanName' a, ChanName' b) -> (a,b) -> IO ()
write2 (c1,c2) (x1,x2) = do fork (sendVia c1 x1)
                            sendVia c2 x2
```

Two primitive channels are created. The communicator function creates two threads to allow the concurrent and independent transfer of the tuple components via these primitive channels. For tuples with more than 9 components, the Eden programmer has to provide a corresponding `Trans` instance by himself or the default communicator will be used, i.e. a 10-tuple would be sent in a single message via a single channel.

4.3 The PA Monad: Improving Control over Parallel Activities

The Eden module provides a parallel action monad which can be used to improve the control of series of parallel actions. The parallel action monad wraps the IO monad. In particular, it is advantageous to define a sequence of side-effecting operations within the PA monad and unwrap the parallel action only once.

```
newtype PA a = PA { fromPA :: IO a }

instance Monad PA where
 return b       = PA $ return b
 (PA ioX) >>= f = PA $ do
   x  <- ioX
   fromPA $ f x

runPA :: PA a -> a
runPA = unsafePerformIO . fromPA
```

4.4 Process Handling: Defining process abstraction and instantiation

Process abstraction with `process` and process instantiation with `( # )` are implemented in the Eden module. While process abstractions define process creation

on the side of the newly created process, process instantiation defines the activities necessary on the side of the parent process. Communication channels are explicitly created and installed to connect processes using the primitives provided for handling Eden's dynamic input channels.

```
data (Trans a, Trans b) =>
      Process a b = Proc (ChanName b -> ChanName' (ChanName a) -> ())
process   :: (Trans a, Trans b) =>
             (a -> b) -> Process a b
process f =  Proc f_remote
  where
    f_remote (Comm sendResult) inCC
      = do  (sendInput, invals) = createComm
            connectToPort inCC
            sendData Data sendInput
            sendResult (f invals)
```

Figure 20: Haskell definitions of Eden process abstraction

A process abstraction of type `Process a b` is implemented by a function `f_remote` (see Fig. 20) which will be evaluated remotely by a corresponding child process. It takes two arguments: the first is an Eden channel (comprising a communicator function `sendResult`) via which the result of the process should be returned to the parent process. The second argument is a primitive channel `inCC` (of type `ChanName' (ChanName a)`) to return its input channels (communicator function) to the parent process. The exact number of channels between parent and child process does not matter in this context, because the operations on dynamic channels are overloaded. The definition of `process` shows that the remotely evaluated function, `f_remote`, creates its input channels via the function `createComm`. Moreover, it connects to the primitive input channel of its parent process and sends the communicator function of its input channels to the parent. Finally the process output, i.e. the result of evaluating the function within the process abstraction `f` to the inputs received via its input channels `invals`. The communicator function `sendResult` will trigger the evaluation of the process result to normal form before sending it.
Process instantiation by the operator `( # )` defines process creation on the parent side. The auxiliary function `instantiateAt` implements process instantiation with explicit placement on a given PE which are numbered from 1 to `noPe`. Passing 0 as a process number leads to the default round robin placement policy for processes. Process creation on the parent side works somehow dually to the process creation on the child side, at least with respect to channel management. First a new input channel for receiving the child process' results is generated. Then a primitive

```
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
pabs # inps
  = runPA $ instantiateAt  0  pabs inps

instantiateAt :: (Trans a, Trans b) =>
                 Int -> Process a b -> a -> PA b
instantiateAt pe (Proc f_remote) inps
  = PA $
    do (sendresult, result)     <- createComm
       (inCC,  Comm sendInput)  <- createC
       sendData (Instantiate pe) (f_remote sendresult  inCC)
       fork  (sendInput inps)
       return result
```

Figure 21: Haskell definitions of Eden process instantiation

```
spawnAt :: [Int] -> [Process a b] -> [a] -> [b]
spawnAt pos ps is
    = runPA $ sequence
                [instantiateAt st p i |
                 (st,p,i) <- zip3 (cycle pos) ps is]

spawn = spawnAt [0]
```

Figure 22: Haskell definition of `spawn`

channel for receiving the child process' input channel(s) is created. The process instantiation message sends the application of the process abstraction function `f_remote` applied to the created input channels to the processor element where the new child process should be evaluated. Finally a concurrent thread is forked which sends the input for the child process using the communicator function received from the child process. The result of the child process is returned to the environment.

The functions `spawnAt` and `spawn` can easily be defined using the PA monad and the primitive function `instantiateAt`, see Figure 22.

4.5 Implementation of Remote Data

In Section 2, the remote data concept has been implemented using Eden's dynamic channel operations. In fact, the implementation immediately uses the primitive operations and provides definition variants in the PA monad:

```
type RD a = ChanName (ChanName a)

releasePA :: Trans a
           => a            -- ^ The original data
           -> PA (RD a)    -- ^ The Remote Data handle
releasePA val = PA $ do
  (cc, Comm sendValC) <- createComm
  fork (sendValC val)
  return cc

release :: Trans a => a -- ^ The original data
          -> RD a      -- ^ The Remote Data handle
release = runPA . releasePA

fetchPA   :: Trans a => RD a -> PA a
fetchPA (Comm sendValCC) = PA $ do
  (c,val) <- createComm
  fork (sendValCC c)
  return val

fetch   :: Trans a
          => RD a   -- ^ The Remote Data handle
          -> a      -- ^ The original data
fetch = runPA . fetchPA
```

The PA variants of `fetch` and `release` are especially useful if structured data is to be released in such a way that there is a release for each data component. Consider the following definitions for transforming a list of local data into a corresponding remote data list and vice versa:

```
releaseAll :: Trans a
              => [a]    -- ^ The original list
              -> [RD a] -- ^ List of Remote Data handles,
                        -- ^ one for each list element
releaseAll as = runPA $ mapM releasePA as

fetchAll :: Trans a
            => [RD a] -- ^ The Remote Data handles
            -> [a]    -- ^ The original data
fetchAll ras = runPA $ mapM fetchPA ras
```

Advanced Eden programmers are encourages to use the PA monad to improve their control over sequences of parallel actions.

4.6 Implementation of Merge

The non-deterministic merge function is implemented using the `nmergeIO :: [[a]] -> IO [a]` provided by Concurrent Haskell.

```
merge     :: [[a]] -> [a]
merge xss =  unsafePerformIO (nmergeIO xss)
```

5 Further Reading

Comprehensive and up-to-date information on Eden is provided on its web site
`http://www.mathematik.uni-marburg.de/~eden`.
Basic information on its design, semantics, and implementation as well as the underlying programming methodology can be found in [30]. Details on the parallel runtime system and Eden's concept of implementation can best be found in [7, 9]. The technique of layered parallel runtime environments has been further developed and generalised by Berthold, Loidl and Al Zain [3, 11]. The Eden trace viewer tool EdenTV is available on Eden's web site. A short introductory description is given in [10]. Another tool for analysing the behaviour of Eden programs has been developed by de la Encina, Llana, Rubio and Hidalgo-Herreo [16, 17] by extending the tool Hood (Haskell Object Observation Debugger) for Eden. Extensive work has been done on skeletal programming in Eden. An overview on various skeleton types (specification, implementation, and cost models) have been presented as a chapter in the mentioned book by Gorlatch and Rabhi [29]. Several parallel map implementations have been discussed and analysed in [25]. An Eden implementation of the large-scale *map-and-reduce* programming model proposed by Google [14] has been investigated in [5]. Hierarchical master-worker schemes with several layers of masters and sub-masters have been presented in [6]. A sophisticated distributed workpool has been presented in [33]. Definitions and applications of further specific skeletons can be found in the following papers: topology skeletons [8], adaptive skeletons [19], divide-and-conquer schemes [4, 35]. Special skeletons for computer algebra algorithms are developed with the goal to define the kernel of a computer algebra system in Eden [28, 27]. An operational and a denotational semantics for Eden have been defined by Ortega-Mallén and Hidalgo-Herrero [20, 21]. These semantics have been used to analyze Eden skeletons [23, 22]. A non-determinism analysis has been presented by Segura and Peña [36].

6 Other Parallel Haskells (Related Work)

Several extensions of Haskell for parallel programming exist. The following enumeration follows the spectrum from explicit low-level approaches to implicit high-level approaches:

Haskell plus MPI[2] uses the foreign function interface (FFI) of Haskell to provide the MPI functionality in Haskell. It supports an SPMD style, i.e. the same program is started on several PEs. The different instances can be distinguished using their MPI rank and may exchange serializable Haskell data structures via MPI send and receive routines.

Eden (as described in these lecture notes) abstracts from low-level sending and receiving of messages. Communication via channels is automatically provided by the parallel runtime system. It allows, however, to define processes and communication channels explicitly and thus to control parallel activity and data distribution.

Glasgow parallel Haskell [43] and **Multicore Haskell** [32] share the same language definition but differ in their implementations. While GpH shares its parallel runtime system with Eden and can be executed on distributed memory systems, Multicore Haskell is tailored to shared-memory multicore architectures. The language allows to mark expressions for parallel evaluation. These are collected as *sparks* in a spark pool. The runtime system decides which sparks will be evaluated in parallel. Access to local and remote data is automatically managed by the runtime system. Evaluation strategies [31] abstract from low-level expression marking and allow to describe pattern for parallel behaviour on a higher level of abstraction.

Data Parallel Haskell [41] Data Parallel Haskell extends Haskell with support for nested data parallelism with a focus to utilise multicore CPUs. It adds parallel arrays and implicitly parallel operations on those to Haskell.

7 Conclusions

These lecture notes give an overview of achievements and the current status of the Eden project. Several simple case studies reflect Eden's programming methodology which is based on the implementation and use of algorithmic skeletons. The Eden project is ongoing. Current activities comprise the further development of the Eden skeleton library as well as the investigation of alternative high-level parallel programming constructs.

[2] See `https://github.com/bjpop/haskell-mpi`.

A Compiling, Running, Analysing Eden Programs

The Eden compiler, an extension of the Glasgow Haskell compiler (GHC), is available from the Eden homepage under URL

`http://www.mathematik.uni-marburg.de/~eden`

Prerequisites and installation instructions are provided. To compile Eden programs with parallel support one has to use the options `-parpvm` to use PVM[3] or `-parmpi` to use MPI[4] as middleware. The option `-eventlog` leads to production of trace files (event logs) when compiled Eden programs are executed. All GHC options, e.g. optimisation flags like `-O2`, can equally well be used with the Eden version of GHC.

A.1 Runtime Options

A compiled Eden program accepts in addition to its arguments *runtime system options* enclosed in

`+RTS <your options> -RTS`

With these options one can control the program setup and behaviour, e.g. on how many (virtual) processor elements (PEs) the program should be executed, which process placement policy should be used etc. The following table shows the most important Eden specific runtime options. All GHC RTS options can also be used. By typing `./myprogram +RTS -?` a complete list of available RTS options is given.

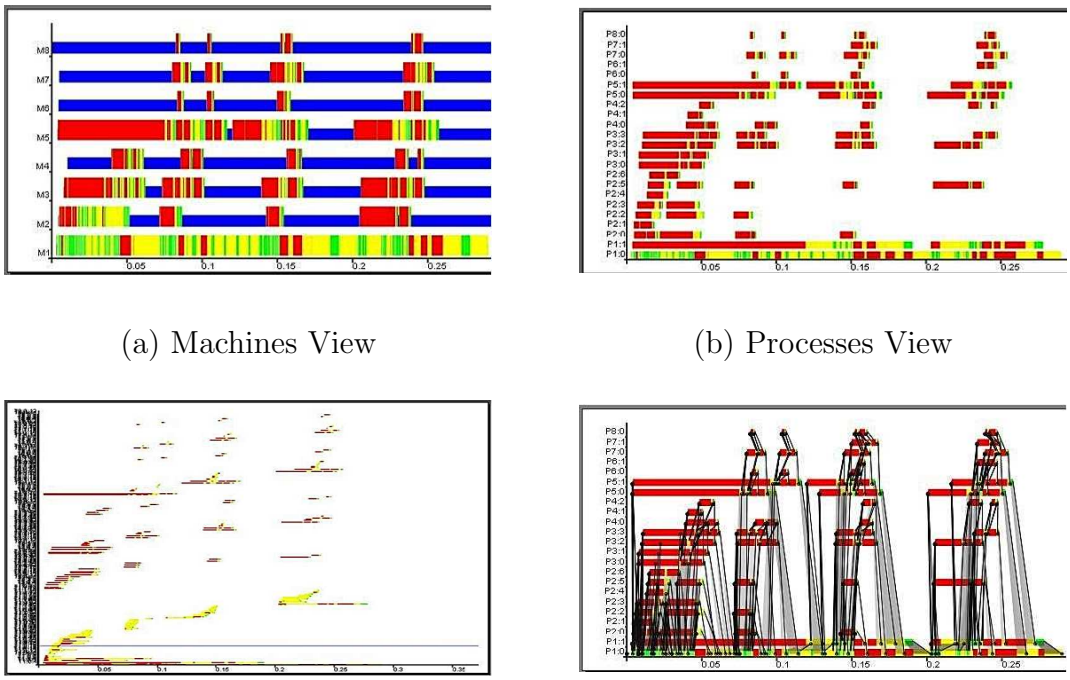
RTS option	effect	default
`-N<n>`	set number of PEs	number of PVM/MPI nodes
`-MPI@<file>`	specify MPI hostfile	`mpihosts`
`-qQ<n>`	set buffer size for messages	32K
`-ls`	enable event logging[5]	
`-qrnd`	random process placement	round-robin placement

A.2 EdenTV: The Eden Trace Viewer

The Eden trace viewer tool (EdenTV) [10] provides a *post-mortem analysis* of program executions on the level of the computational units of the PRTS. The latter is instrumented with special trace generation commands activated by the compile-time option `-eventlog` and the run-time option `+RTS -ls`. In the space-time diagrams generated by EdenTV, machines (i.e. processor elements), processes and threads are represented by horizontal bars, with time on the x-axis.

[3] `http://www.epm.ornl.gov/pvm/`
[4] `http://www.open-mpi.org/`

The machines diagrams correspond to the view of profiling tools observing the parallel machine execution, if there is a one-to-one correspondence between virtual and physical processor elements which will usually be the case. The processes per machine diagrams show the activity of Eden processes and their placement on the available machines. The threads diagrams show the activity of all created threads, not only the threads of Eden processes but also internal system threads. The diagram bars have segments in different colours, which indicate the activities of the respective logical unit (machine, process or thread) in a period during the execution. Bars are green when the logical unit is running, yellow when it is runnable but currently not running, and red when the unit is blocked. If trace visualisations are shown in greyscale, the colors have the following correspondences: light grey = yellow, grey = green, dark grey = red. In addition, a machine can be idle which means that no processes are allocated on the machine. Idleness is indicated by a small blue bar. The thread states are immediately determined from the thread state events in the traces of processes. The states of processes and machines are derived from the information about thread states.
Figure 23 shows examples of the machines, processes and threads diagrams for a divide-and-conquer program implementing the bitonic-merge-sort algorithm [2]. The trace has been generated on 8 Linux workstations connected via fast Ethernet. The program sorted a list of 1024 numbers with a recursion depth limit of 4.
The example diagrams in Figure 23 show that the program has been executed on 8 machines (virtual processor elements). While there is some activity on machine 1 (where the main program is started) during the whole execution, machines 6 to 8 are idle most of the time (smaller blue bar). The corresponding processes graphic (see Figure 23(b)) reveals that several Eden processes have been allocated on each machine. The activities in Machine 2 have been caused by different processes. The diagrams show that the workload on the parallel machines was low — there were only small periods where threads were running. The yellow-colored periods indicate system activity in the diagrams. The threads view is not readable because too many threads are shown. It is possible to zoom the diagrams to get a closer view on the activities at critical points during the execution.
Messages between processes or machines are optionally shown by grey arrows which start from the sending unit bar and point at the receiving unit bar (see Figure 23(d)). Streams can be shown as shadowed areas. The representation of messages is very important for programmers, since they can observe hot spots and inefficiencies in the communication during the execution as well as control communication topologies.
When extensive communication takes places, message arrows may cover the whole activity profile. For this reason, EdenTV allows to show messages selectively, i.e. between selectable (subsets of) processes. EdenTV provides many additional information and features, e.g. the number of messages sent and received by processes and machines is recorded. More information is provided on the web pages



(a) Machines View

(b) Processes View

(c) Thread View

(d) Processes with Message Overlay

Figure 23: Examples of EdenTV Diagrams

of EdenTV:

`http://www.mathematik.uni-marburg.de/~eden/?content=trace_main&navi=trace`

Acknowledgements

The author thanks the co-developers of Eden Yolanda Ortega-Mallén and Ricardo Peña from Universidad Complutense de Madrid for their friendship and continuing support. It is thanks to Jost Berthold that we have an efficient implementation of Eden in the Glasgow Haskell compiler. I am grateful to all other actual and former members of the Eden project for their manifold contributions: Alberto de la Encina, Mercedes Hildalgo Herrero, Christóbal Pareja, Fernando Rubio, Lidia Sánchez-Gil, Clara Segura, Pablo Roldan Gomez (Universidad Complutense de Madrid) and Silvia Breitinger, Mischa Dieterle, Ralf Freitag, Thomas Horstmeyer, Ulrike Klusik, Dominik Krappel, Oleg Lobachev, Johannes May, Bernhard Pickenbrock, Steffen Priebe, Björn Struckmeier, Nils Weskamp (Philipps-Universität Marburg). Last but not least, thanks go to Hans-Wolfgang Loidl, Phil Trinder, Kevin Hammond, and Greg Michaelson for many fruitful discussions, successful cooperations, and for giving us access to their Beowulf clusters.

References

[1] M. Alt and S. Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2004.

[2] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[3] J. Berthold. Towards a Generalised Runtime Environment for Parallel Haskells. In *Computational Science — ICCS'04*, LNCS 3038. Springer, 2004. (Workshop on Practical Aspects of High-level Parallel Programming — PAPP 2004).

[4] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed Memory Programming on Many-Cores A Case Study Using Eden Divide-&-Conquer Skeletons. In *ARCS 2009, Workshop on Many-Cores*. VDE Verlag, 2009.

[5] J. Berthold, M. Dieterle, and R. Loogen. Implementing Parallel Google Map-Reduce in Eden. In *Europar'09*, LNCS 5704, pages 990–1002. Springer, 2009.

[6] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical Master-Worker Skeletons. In *Practical Aspects of Declarative Languages (PADL 2008)*, LNCS 4902, pages 248 – 264. Springer, 2008.

[7] J. Berthold, U. Klusik, R. Loogen, S. Priebe, and N. Weskamp. High-level Process Control in Eden. In *EuroPar 2003 – Parallel Processing*, LNCS 2790, pages 732–741. Springer, 2003.

[8] J. Berthold and R. Loogen. Skeletons for Recursively Unfolding Process Topologies. In *Parallel Computing: Current & Future Issues of High-End Computing, ParCo 2005*, pages 835–842. NIC Series, Vol. 33, 2006.

[9] J. Berthold and R. Loogen. Parallel Coordination Made Explicit in a Functional Setting. In *Implementation and Application of Functional Languages (IFL 2006), Selected Papers*, LNCS 4449, pages 73–90. Springer, 2007. (*awarded best paper* of IFL'06).

[10] J. Berthold and R. Loogen. Visualizing Parallel Functional Program Runs – Case Studies with the Eden Trace Viewer –. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, pages 121–128. NIC Series, Vol. 38, 2007.

[11] J. Berthold, A. A. Zain, and H.-W. Loidl. Scheduling light-weight parallelism in ArtCoP. In *Practical Aspects of Declarative Languages (PADL 2008)*, LNCS 4902, pages 214 – 229. Springer, 2008.

[12] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98*, volume 1490 of *LNCS*, pages 318–334. Springer, 1998.

[13] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] M. Dieterle, T. Horstmeyer, and R. Loogen. Skeleton Composition Using Remote Data. In *Practical Aspects of Declarative Programming 2010 (PADL 2010)*, LNCS 5937, pages 73–87. Springer, 2010.

[16] A. Encina, L. Llana, F. Rubio, and M. Hidalgo-Herrero. Observing Intermediate Structures in a Parallel Lazy Functional Language. In *Principles and Practice of Declarative Programming (PPDP 2007)*, pages 109–120. ACM, 2007.

[17] A. Encina, I. Rodríguez, and F. Rubio. pHood: A Tool to Analyze Parallel Functional Programs. In *Implementation of Functional Languages (IFL'09)*, pages 85–99. Seton Hall University, New York, USA, 2009. Technical Report, SHU-TR-CS-2009-09-1.

[18] W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 25(1):112–115, 1978.

[19] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003.

[20] M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

[21] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation Semantics for Parallel Haskell Dialects. In *Asian Symposium on Programming Languages and Systems (APLAS 2003)*, pages 303–321. LNCS 2895, Springer, 2003.

[22] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Analyzing the Influence of Mixed Evaluation on the Performance of Eden Skeletons. *Parallel Computing*, 32(7–8):523–538, 2006.

[23] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Comparing Alternative Evaluation Strategies for Stream-Based Parallel Functional Languages. In *Implementation and Application of Functional Languages (IFL 2006), Selected Papers*, LNCS 4449, pages 55–72. Springer, 2007.

[24] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Inc., Rockville, MD, 1978.

[25] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden — Low-Effort Parallel Programming. In *Implementation of Functional Languages (IFL 2000), Selected Papers*, LNCS 2011, pages 71–88. Springer, 2001.

[26] U. Klusik, Y. Ortega-Mallén, and R. Peña Marí. Implementing Eden – or: Dreams Become Reality. In *IFL'98*, volume 1595 of *LNCS*, pages 103–119. Springer, 1999.

[27] O. Lobachev. *Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms*. PhD thesis, Philipps-Universität Marburg, Germany, 2011. to appear.

[28] O. Lobachev and R. Loogen. Towards an Implementation of a Computer Algebra System in a Functional Language. In *AISC/Calculemus/MKM 2008*, LNAI 5144, pages 141–174, 2008.

[29] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *[40]*, chapter 4, pages 95–128. Springer, 2003.

[30] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[31] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. W. Trinder. Seq no more: Better strategies for parallel Haskell. In *Haskell Symposium 2010*, Baltimore, MD, USA, Sept. 2010. ACM Press.

[32] S. Marlow, S. L. Peyton-Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP 2009 — Intl. Conf. on Functional Programming*, pages 65–78, Edinburgh, Scotland, Aug. 2009. ACM Press.

[33] R. L. Mischa Dieterle, Jost Berthold. A Skeleton for Distributed Work Pools in Eden. In M. Blume and G. Vidal, editors, *10th Fuji International Symposium on Functional and Logic Programming (FLOPS) 2010*, LNCS 6009, pages 337–353, Sendai, Japan, 2010. Springer.

[34] MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.

[35] M. D. R. L. Oleg Lobachev, Jost Berthold. Parallel FFT using Eden Skeletons. In *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83, Novosibirsk, Russia, 2009. Springer.

[36] R. Peña and C. Segura. Non-determinism Analysis in a Parallel-Functional Language. In *Implementation of Functional Languages (IFL 2000)*, LNCS 1268. Springer, 2001.

[37] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[38] S. Priebe. Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *European Conference on Parallel Computing (Euro-Par) 2006*, LNCS 4128, Dresden, Germany, 2006.

[39] PVM: Parallel Virtual Machine. Web page. `http://www.epm.ornl.gov/pvm/`.

[40] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[41] G. K. M. M. C. Simon Peyton Jones, Roman Leshchinskiy. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, 2008.

[42] The GHC Developer Team. The Glasgow Haskell Compiler. Website `http://www.haskell.org/ghc`.

[43] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.