Eden — Parallel Functional Programming with Haskell

Rita Loogen

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

Original Version appeared in: V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.): CEFP 2011, Springer LNCS 7241, 2012, pp. 142-206.

Abstract. Eden is a parallel functional programming language which extends Haskell with constructs for the definition and instantiation of parallel processes. Processes evaluate function applications remotely in parallel. The programmer has control over process granularity, data distribution, communication topology, and evaluation site, 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, so-called algorithmic skeletons, are provided as higher-order functions in a user-extensible skeleton library written in Eden. Eden is geared toward distributed settings, i.e. processes do not share any data, but can equally well be used on multicore systems. This tutorial gives an up-to-date introduction into Eden's programming methodology based on algorithmic skeletons, its language constructs, and its layered implementation on top of the Glasgow Haskell compiler.

Table of Contents

1 Introduction 2
2 Skeleton-based Programming in Eden 4
2.1 Map-and-Reduce 5
2.2 Divide-and-Conquer 12
2.3 Eden's Skeleton Library 16
3 Eden's Basic Constructs 17
3.1 Defining and Creating Processes 17
3.2 Coping With Laziness 21
3.3 Implementing the Divide-and-Conquer Skeleton `disDC` 23
4 Controlling Communication Costs 24
4.1 Reducing Communication Costs: Chunking 25
4.2 Communication vs Parameter Passing: Running Processes Offline 28
4.3 Tuning the Parallel Mergesort Program 29

5 Defining Non-Hierarchical Communication Topologies 32
5.1 The Remote Data Concept 33
5.2 A Ring Skeleton 34
5.3 A Torus Skeleton 37
6 Workpool Skeletons 40
6.1 Many-to-one Communication: Merging Communication Streams . 40
6.2 A Simple Master-Worker Skeleton 40
6.3 Classification of Parallel Map Implementations 43
6.4 A Nested Master-Worker Skeleton 44
7 Explicit Channel Management 45
7.1 Dynamic Channels 45
7.2 Implementing Remote Data with Dynamic Channels 47
8 Behind the Scenes: Eden's Implementation 47
8.1 Layered Parallel Runtime Environment 48
8.2 The Type Class Trans 50
8.3 The PA Monad: Improving Control over Parallel Activities 51
8.4 Process Handling: Defining Process Abstraction and Instantiation 53
9 Further Reading 55
10 Other Parallel Haskells (Related Work) 56
11 Conclusions 57
A Compiling, Running, Analysing Eden Programs 61
A.1 Compile Time Options 62
A.2 Runtime Options 62
A.3 EdenTV: The Eden Trace Viewer 62
B Auxiliary Functions 64
B.1 Unshuffle and Shuffle 64
B.2 SplitIntoN and Chunk 65
B.3 Distribute and OrderBy 66

1 Introduction

Functional languages are promising candidates for effective parallel programming, because of their high level of abstraction and, in particular, because of their referential transparency. In principle, any subexpression 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.

In these lecture notes we 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 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. Furthermore, process inputs and outputs are always completely evaluated before being sent in order to enable parallelism in the context of a host language with a demand-driven evaluation strategy. Algorithmic skeletons which specify common and sophisticated parallel communication patterns and topologies are provided as higher-order functions in a user-extensible skeleton library written in Eden. Skeletons provide a very simple access to parallel functional programming. Parallelization of a Haskell program can often simply be achieved by selecting and instantiating an appropriate skeleton from the skeleton library. From time to time, adaptation of a skeleton to a special situation or the development of a new skeleton may be necessary.

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 e.g. no need for global garbage collection. There is, however, a risk of loosing sharing, i.e. it may happen that the same expression is redundantly evaluated by several 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 and pass its name to another process. The latter can directly send data to the receiver process using the received channel name. An even easier-to-use way to define non-hierarchical process networks is the *remote data concept* where data can be released by a process to be fetched by a remote process. In this case a handle is first transferred from the owner to the receiver process (maybe via common predecessor processes). Via this handle the proper data can then directly transferred from the producer to the receiver process. 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 cyclic communication topologies like rings and tori can be defined in an elegant way. Eden supports an equational programming style where recursive process nets can simply be defined using recursive equations. Using the recently introduced PA (parallel action) monad, it is also possible to adopt a monadic programming style, in particular, when it is necessary to ensure that series of parallel activities are executed in a given order.

Eden has been implemented by extending the runtime system of the Glasgow Haskell compiler [24], a mature and efficient Haskell implementation, for parallel and distributed execution. The parallel runtime system (PRTS) uses suitable middleware (currently PVM [52] or MPI [43]) to manage parallel execution. Recently, a special multicore implementation which needs no middleware has been implemented [48]. Traces of parallel program executions can be visualised and analysed using the Eden Trace Viewer EdenTV.

This tutorial gives an up-to-date introduction into Eden's programming methodology based on algorithmic skeletons, its language constructs, and its layered implementation on top of the Glasgow Haskell compiler. Throughout the tutorial, exercises are provided which help the readers to test their understanding of the presented material and to experiment with Eden. A basic knowledge of programming in Haskell is assumed. The Eden compiler, the skeleton library, EdenTV, and the program code of the case studies are freely available from the Eden web pages, see

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

Plan of this tutorial. The next section provides a quick start to Eden programming with algorithmic skeletons. Section 3 introduces the basic constructs of Eden's coordination language, i.e. it is shown how parallelism can be expressed and managed. The next section presents techniques for reducing the communication costs in parallel programs. Section 5 shows how non-hierarchical communication topologies can be defined. In particular, a ring and a torus skeleton are presented. Section 6 explains how master-worker systems can be specified. An introduction to explicit channel management in Section 7 leads to Section 8 which introduces Eden's layered implementation. Hints at more detailed material on Eden are given in Section 9. After a short discussion of related work in Section 10 conclusions are drawn in Section 11. Appendix A contains a short presentation of how to compile, run, and analyse Eden programs. In particular, it presents the Eden trace viewer tool, EdenTV, which can be used to analyse the behaviour of parallel programs. Appendix B contains the definitions of auxiliary functions from the Eden `Auxiliary` library that are used in this tutorial.

The tutorial refers to several case studies and shows example trace visualisations. The corresponding traces have been produced using the Eden system, version 6.12.3, on the following systems: an Intel 8-core machine (2 × Xeon Quad-core @2.5GHz, 16 GB RAM) machine and two Beowulf clusters at Heriot-Watt University in Edinburgh (Beowulf I: 32 Intel P4-SMP nodes @ 3 GHz 512MB RAM, Fast Ethernet and Beowulf II: 32 nodes, each with two Intel quad-core processors (Xeon E5504) @ 2GHz, 4MB L3 cache, 12GB RAM, Gigabit Ethernet).

2 Skeleton-based Programming in Eden

Before presenting the Eden programming constructs we show how a quick and effective parallelization of Haskell programs can be achieved using pre-defined skeletons from the Eden skeleton library. (Algorithmic) skeletons [16] define common parallel computation patterns. In Eden they are defined as higher-order functions. In the following we look at two typical problem solving schemes for which various parallel implementations are provided in the Eden skeleton library: map-and-reduce and divide-and-conquer.

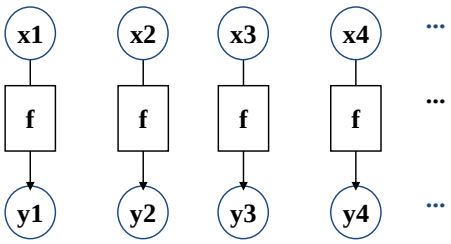


Fig. 1. Basic `map` evaluation scheme

2.1 Map-and-Reduce

Map-and-reduce is a typical data-parallel evaluation scheme. It consists of a `map` and a subsequent `reduce`.

Map, Parallel Map and Farm. The `map` function applies a function to each element of a list. In Haskell it can simply be expressed as follows

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

The `map` function is inherently parallel because in principle all function applications `(f x)` can be evaluated in parallel. It presents a simple form of data parallelism, because the same function is applied to different data elements (see Figure 1).

Eden's skeleton library contains several parallel implementations of `map`. The simplest parallel version is `parMap` where a separate process is created for each function application, i.e. as many processes as list elements will be created. The input parameter as well as the result of each process will be transmitted via communication channels between the generator process and the processes created by `parMap`. Therefore `parMap`'s type is

```
parMap :: (Trans a, Trans b) ⇒ (a → b) → [a] → [b]
```

The Eden-specific type context `(Trans a, Trans b)` indicates that both types `a` and `b` must belong to the Eden `Trans` type class of *transmissible* values. Most predefined types belong to this type class. In Haskell, type classes provide a structured way to define overloaded functions. `Trans` provides implicitly used communication functions.

If the number of list elements is much higher than the number of available processing elements, this will cause too much process creation overhead. Another skeleton called `farm` takes two additional parameter functions

`distribute :: [a] → [[a]]` and `combine :: [[b]] → [b]`.

It uses the `distribute`-function to split the input list into sublists, creates a process for mapping f on each sublist and combines the result lists using the

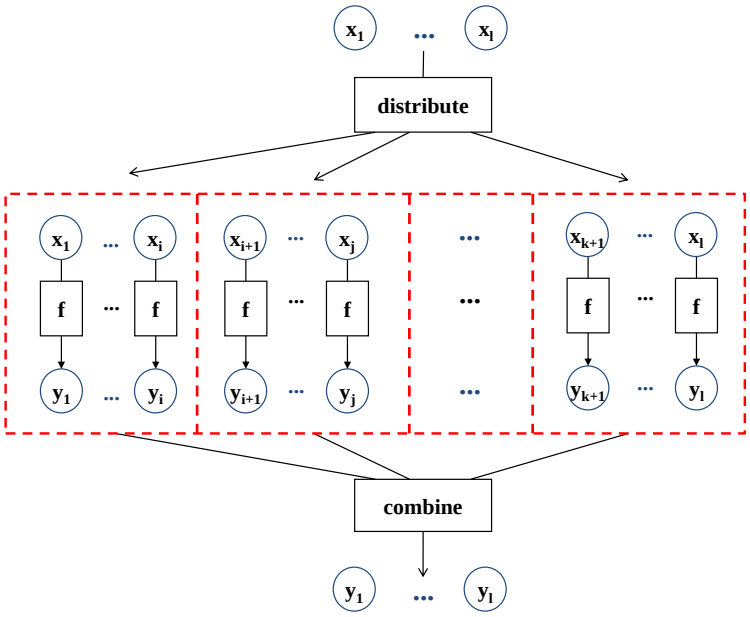


Fig. 2. Parallel `farm` evaluation scheme

`combine`-function. Of course, a proper use of farm to implement another parallel version of `map` requires that the following equation is fulfilled[1]:

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

Replacing the outer `map`-application with `parMap` leads to the definition of the `farm` skeleton in Eden:

```
farm :: (Trans a, Trans b) ⇒
        ([a] → [[a]])              -- ^ distribute
        → ([[b]] → [b])            -- ^ combine
        → (a → b) → [a] → [b] -- ^ map interface
farm distribute combine f
  = combine ∘ (parMap (map f)) ∘ distribute
```

The `farm` skeleton creates as many processes as sublists are generated by the parameter function `distribute` (see Figure 2, dotted lines indicate processes). In Eden's `Auxiliary` library the following functions for distributing and (re-)combining lists are defined. For the reader's convenience we have also put together the pure Haskell definitions of these functions in Appendix B.

- `unshuffle :: Int → [a] → [[a]]` distributes the input list in a round robin manner into as many sublists as the first parameter indicates.
- `shuffle :: [[a]] → [a]` shuffles the given list of lists into the output list. It works inversely to `unshuffle`.

[1] The programmer is responsible for guaranteeing this condition.

– `splitIntoN :: Int → [a] → [[a]]` distributes the input list blockwise into as many sublists as the first parameter determines. The lengths of the output lists differ by at most one. The inverse function of `splitIntoN` is the Haskell prelude function `concat :: [[a]] → [a]` which simply concatenates all lists in the given list of lists.

Eden provides a constant

```
noPe :: Int
```

which gives the number of available processing elements. Thus, suitable parallel implementations of `map` using `farm` are e.g.

```
mapFarmS, mapFarmB :: (Trans a, Trans b) ⇒
                      (a → b) → [a] → [b]
mapFarmS = farm (unshuffle  noPe) shuffle
mapFarmB = farm (splitIntoN noPe) concat
```

Reduce and Parallel Map-Reduce. In many applications, a reduction is executed after the application of `map`, i.e. the elements of the result list of `map` are combined using a binary function. In Haskell list reduction is defined by higher-order `fold`-functions. Depending on whether the parameter function is right or left associative, Haskell provides folding functions `foldr` and `foldl`. For simplicity, we consider in the following only `foldr`:

```
foldr :: (a → b → b) → b → [a] → b
foldr g e []     = e
foldr g e (x:xs) = g x (foldr g e xs)
```

Accordingly, the following composition of `map` and `foldr` in Haskell defines a simple map-reduce scheme:

```
mapRedr :: (b → c → c) → c → (a → b) → [a] → c
mapRedr g e f = (foldr g e) ∘ (map f)
```

This function could simply be implemented in parallel by replacing `map` with e.g. `mapFarmB`, but then the reduction will completely and sequentially be performed in the parent process. If the parameter function `g` is associative with type `b → b → b` and neutral element `e`, the reduction could also be performed in parallel by pre-reducing the sublists within the farm processes. Afterwards only the subresults from the processes have to be combined by the main process (see Figure 3). The code of this parallel map-reduce scheme is a slight variation of the above definition of the `farm`-skeleton where the distribution and combination of values is fixed and `mapRedr` is used instead of `map` as argument of `parMap`:

```
parMapRedr :: (Trans a, Trans b) ⇒
              (b → b → b) → b → (a → b) → [a] → b
parMapRedr g e f
  = if noPe == 1 then  mapRedr g e f xs  else
    (foldr g e) ∘ (parMap (mapRedr g e f)) ∘ (splitIntoN noPe)
```

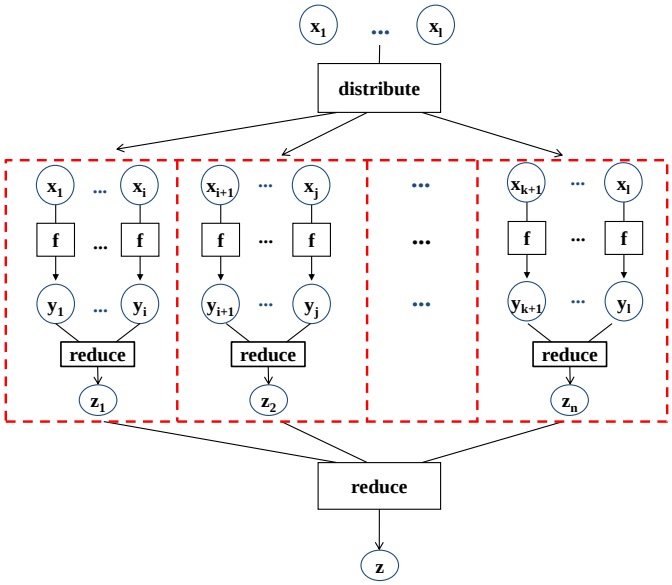


Fig. 3. Parallel map-reduce evaluation scheme

Note that parallel processes are only created if the number of available processor elements is at least 2. On a single processor element the sequential scheme `mapRedr` is executed.

With this skeleton the input lists of the processes are evaluated by the parent process and then communicated via automatically created communication channels between the parent process and the `parMap` processes. In Eden, lists are transmitted as streams which means that each element is sent in a separate message. Sometimes this causes a severe overhead, especially for very long lists. The following variant `offline_parMapRedr` avoids the stream communication of the input lists at all. Only a process identification number is communicated and used to select the appropriate part of the input list. The whole (unevaluated) list is incorporated in the worker function which is mapped on the identification numbers. As each process evaluates now the `(splitIntoN noPe)` application, this may cause some redundancy in the input evaluation but it substantially reduces communication overhead. In Subsection 4.2, we discuss this technique in more detail.

```
offline_parMapRedr :: (Trans a, Trans b) ⇒
              (b → b → b) → b → (a → b) → [a] → b
offline_parMapRedr g e f xs
  = if noPe == 1 then  mapRedr g e f xs  else
    foldr g e (parMap worker [0..noPe-1])
  where worker i = mapRedr g e f ((splitIntoN noPe xs)!!i)
```

```
module Main where

import System
import Control.Parallel.Eden
import Control.Parallel.Eden.EdenSkel.MapRedSkels

main :: IO ()
main = getArgs >>= \ (n:_) →
       print (cpi (read n))

-- compute pi using integration
cpi   :: Integer → Double
cpi n =  offline_parMapRedr (+) 0 (f ∘ index) [1..n] /
         fromInteger n
  where
    f       :: Double → Double
    f x     =  4 / (1 + x*x)
    index   :: Integer → Double
    index i =  (fromInteger i - 0.5) / fromInteger n
```

Fig. 4. Eden program for parallel calculation of π

Example: The number π can be calculated by approximating the integral

$$\pi = \int_0^1 f(x)\ dx \text{ where } f(x) = \frac{4}{1+x^2}$$

in the following way:

$$\pi = \lim_{n\to\infty} pi(n) \text{ with } pi(n) = \frac{1}{n}\sum_{i=1}^{n} f\left(\frac{i-0.5}{n}\right).$$

The function *pi* can simply be expressed in Haskell using our `mapRedr` function:

```
cpi   :: Integer → Double
cpi n =  mapRedr (+) 0 (f ∘ index) [1..n] / fromInteger n
  where
    f       :: Double → Double
    f x     =  4 / (1 + x*x)
    index   :: Integer → Double
    index i =  (fromInteger i - 0.5) / fromInteger n
```

The Haskell prelude function `fromInteger` converts integer numbers into double-precision floating point numbers.

A parallel version is obtained by replacing `mapRed` with `offline_parMapRedr`. The complete parallel program is shown in Figure 4. It is important that each Eden program imports the Eden module `Control.Parallel.Eden`. In addition, the program imports the part of the Eden skeleton library which provides parallel map-reduce skeletons. How to compile, run and analyse Eden programs is explained in detail in the appendix of this tutorial. Figure 5 shows on the left

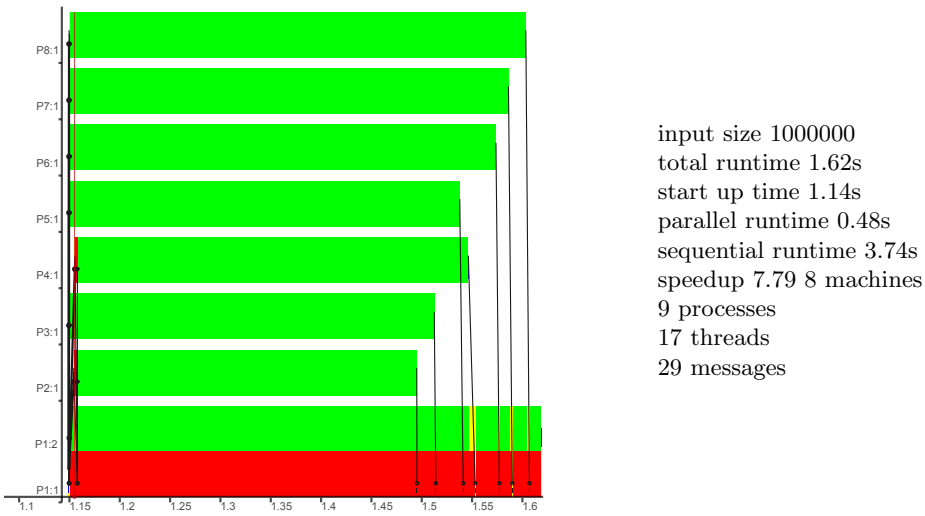


Fig. 5. Pi trace on 8 PEs, parallel map-reduce with 8 processes

the visualisation of a trace file by EdenTV and on the right some statistical data of this program run also provided by EdenTV. The trace has been produced for the input 1000000 with the parallel MPI-based runtime system on the Beowulf II. When using MPI the start-up time of the parallel runtime system is incorporated in the trace. The start-up time depends on the number of processor elements which are started. In the program run underlying the trace in Figure 5 the start-up took 1.14 seconds. Thus, it dominates the overall runtime which has been 1.62 seconds. The actual parallel runtime of the program has been 0.48 seconds. The sequential program takes 3.74 seconds with the same input on the same machine. The fraction

$$\frac{\text{sequential runtime}}{\text{parallel runtime}}$$

is called the speed-up of the parallel evaluation. The speed-up is usually bounded by the number of processor elements. In this example the speed-up has been 7.79 which is very good on 8 PEs.

The trace visualisation on the left shows the parallel program behaviour. It consists of 9 horizontal bars, the timelines for the 9 processes that have been executed. On the x-axis the time is shown in seconds. On the y-axis the process numbers are given in the form `P i:j` where `i` is the number of the processor element or machine, on which the process is executed and `j` is the local number of the process per machine. Note that the timelines have been moved to the left to omit the start-up time, i.e. the x-axis starts at 1.14 seconds.

The colours of the bars indicate the status of the corresponding process. Green (in grayscale: grey) means that the process is running. Yellow (light grey) shows that a process is runnable but not running which might be due to a garbage

collection or another process running on the same PE. Red (dark grey) indicates that a process is blocked, i.e. waiting for input. Messages are shown as black lines from the sender to the receiver process where the receiver is marked by a dot at the end of the line. The program starts 9 processes. Process `P 1:1`, i.e. Process 1 on PE 1 executes the main program. The `offline_parMapRedr` skeleton starts `noPe = 8` processes which are placed by default in a round-robin manner on the available PEs, starting with PE 2. Thus, the last process has also been allocated on PE 1 and is numbered `P 1:2`.

The trace picture shows that the child processes are always running (green) while the main process is blocked (red) most of the time waiting for the results of the child processes. 17 threads have been generated: one thread runs in the 9 processes each to compute the process output. In addition, 8 (short-living) threads have been started in the main process to evaluate and send the input (identification numbers) to the 8 child processes. In total 29 messages have been sent: In the beginning, 8 process creation messages and 7 acknowledgement messages are exchanged. Messages between `P 1:2` and `P 1:1` are not counted because they are not really sent, as both processes are executed on the same PE. Moreover, the main process `P 1:1` sends 7 input messages to the 7 remote processes. When the remote child processes terminate, they send their result back to the main process. Finally the main process computes the sum of the received values, divides this by the original input value and prints the result. ◁

Exercise 1: The following Haskell function `summePhi` sums Euler's totient or ϕ function which counts for parameter value n the number of positive integers less than n that are relatively prime to n:

```
summePhi   :: Int → Int
summePhi n =  sum (map phi [1..n])

phi   :: Int → Int
phi n =  length (filter (relprime n) [1..(n-1)])

relprime     :: Int → Int → Bool
relprime x y = gcd x y == 1
```

`sum` and `gcd` are Haskell prelude function, i.e. predefined Haskell function. `sum` sums all elements of a list of numbers. It is defined as an instance of the folding function `foldl'`, a strict variant of `foldl`:

```
sum :: Num a ⇒ [a] → a
sum =  foldl' (+) 0
```

`gcd` computes the greated common divisor of two integers.

1. Define `summePhi` as instance of a map-reduce scheme.
2. Parallelise the program using an appropriate map-reduce skeleton of the Eden skeleton library.
3. Run your parallel program on i machines, where $i \in \{1, 2, 4, ...\}$ (runtime option `-N`i) and use the Eden trace viewer to analyse the parallel program behaviour.

2.2 Divide-and-Conquer

Another common computation scheme is divide-and-conquer. Eden's skeleton library provides several skeletons for the parallelisation of divide-and-conquer algorithms. The skeletons are parallel variants of the following polymorphic higher-order divide-and-conquer function `dc` which implements the basic scheme: If a problem is *trivial*, it is *solved* directly. Otherwise, the problem is divided or (*splitted*) into two or more subproblems, for which the same scheme is applied. The final solution is computed by *combining* the solutions of the subproblems.

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

The easiest way to parallelise this `dc` scheme in Eden is to replace `map` with `parMap`. An additional integer parameter `lv` can be used to stop the parallel unfolding at a given level and to use the sequential version afterwards:

```
parDC :: (Trans a, Trans b) ⇒
         Int →   -- level
         DivideConquer a b
parDC lv trivial solve split combine
 = pdc lv
 where
  pdc lv x
   | lv == 0 = dc trivial solve split combine x
   | lv >  0 = if trivial x then solve x
               else combine x (parMap (pdc (lv-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 however not evenly distributed on the available processing elements (PEs). Note that each PE `i` places new locally created child processes in a round-robin manner on the PEs `(i mod noPe)+1`, `((i+1) mod noPe)+1` etc.

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 child processes will be created and may be allocated on PEs 2 to 8 as indicated by the tree in Figure 6. The main process on PE 1 places its two child processes on PEs 2 and 3. The child process on PE2 creates new processes on PEs 3 and 4, the one on PE 3 accordingly on PEs 4 and 5. The second process on PE 3 allocates its children on PE 6 and 7, where we assume that the two processes on PE 3 create their child processes one after the other and not in an interleaved way. In total, PEs 3, 4 and 5 would get two processes each, three processes would be allocated on PEs 6 and 7, while only one process would be placed on PE 8.

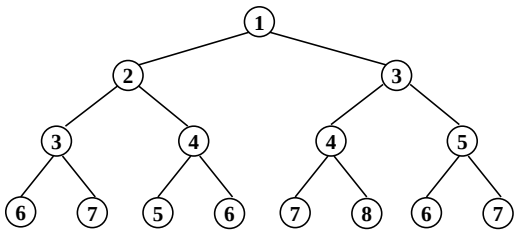


Fig. 6. Divide-and-conquer call tree with default process placement

Thus, the default process placement leads to an unbalanced process distribution on the available PEs. ◁

The Eden skeleton library provides more elaborated parallel divide-and-conquer implementations. In the following example, we use the `disDC` skeleton. In Subsection 3.3 we show the implementation of this skeleton in Eden. The `disDC` skeleton implements a so-called distributed expansion scheme. This works in a similar way like the above parallelization with `parMap` except for the following differences:

1. The skeleton assumes a fixed-degree splitting scheme, i.e. the split function always divides a problem into the same number of subproblems. This number, called the *branching degree*, is the first parameter of the `disDC` skeleton.
2. The creation of processes and their allocation is controlled via a list of PE numbers, called *ticket list*. This list is the second parameter of the skeleton. It determines on which PEs newly created processes are allocated und thus indirectly how many processes will be created. When no more tickets are available, all further evaluations take place sequentially. This makes the use of a level parameter to control parallel unfolding superfluous. Moreover, it allows to balance the process load on PEs. The ticket list `[2..noPe]` leads e.g. to the allocation of exactly one process on each PE. The main process starts on PE1 and `noPe-1` processes are created on the other available PEs. If you want to create as many processes as possible in a round-robin manner on the available PEs, you should use the ticket list `cycle ([2..noPe]++[1])`. The Haskell prelude function `cycle :: [a] → [a]` defines a circular infinite list by repeating its input list infinitely.
3. Each process keeps the first subproblem for local evaluation and and creates child processes only for the other subproblems.

Example: A typical divide-and-conquer algorithm is mergesort which can be implemented in Haskell as follows:

```
mergeSort      :: Ord a ⇒ [a] → [a]
mergeSort []   =  []
mergeSort [x]  =  [x]
mergeSort xs   =  sortMerge (mergeSort xs1) (mergeSort xs2)
                  where [xs1,xs2] = splitIntoN 2 xs
```

The function `mergeSort` transforms an input list into a sorted output list by subsequently merging sorted sublists with increasing length. Lists with at least two elements are split into into their first half and their second half using the auxiliary function `splitIntoN` from Eden's `Auxiliary` library (see also Appendix B). The sublists 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
```

In order to derive a simple skeleton-based parallelization one first has to define `mergeSort` as an instance of the `dc` scheme, i.e. one has to extract the parameter functions of the `dc` scheme from the recursive definition:

```
mergeSortDC  :: Ord a ⇒ [a] → [a]
mergeSortDC  =  dc trivial solve split combine
  where
    trivial    :: [a] → Bool
    trivial xs =  null xs || null (tail xs)

    solve :: [a] → [a]
    solve =  id

    split :: [a] → [[a]]
    split =  splitIntoN 2

    combine               :: [a] → [[b]] → [b]
    combine _ (xs1:xs2:_) =  sortMerge xs1 xs2
```

A parallel implementation of `mergeSort` is now achieved by replacing `dc` in the above code with `disDC 2 [2..noPe]`. Figure 7 shows the visualisation of a trace produced by a slightly tuned version of this parallel program for an input list with 1 million integer numbers. The tuning concerns the communication of inputs and outputs of processes. We will discuss the modifications in Subsection 4.3 after the applied techniques have been introduced.

The trace picture shows that all processes have been busy, i.e. in mode running (green / grey), during all of their life time. Processes are numbered `P i:1` where `i` is the number of the PE on which the process is evaluated and the `1` is the local process number on each PE. As exactly one process has been allocated on each PE, each process has the local process number `1`. The whole evaluation starts with the main process on PE 1 whose activity profile is shown by the lowest bar. The recursive calls are evaluated on the PEs shown in the call tree on the left in Figure 8. With ticket list `[2..noPe]` seven child processes will be created. The main process is allocated on PE 1 and executes the skeleton call. It

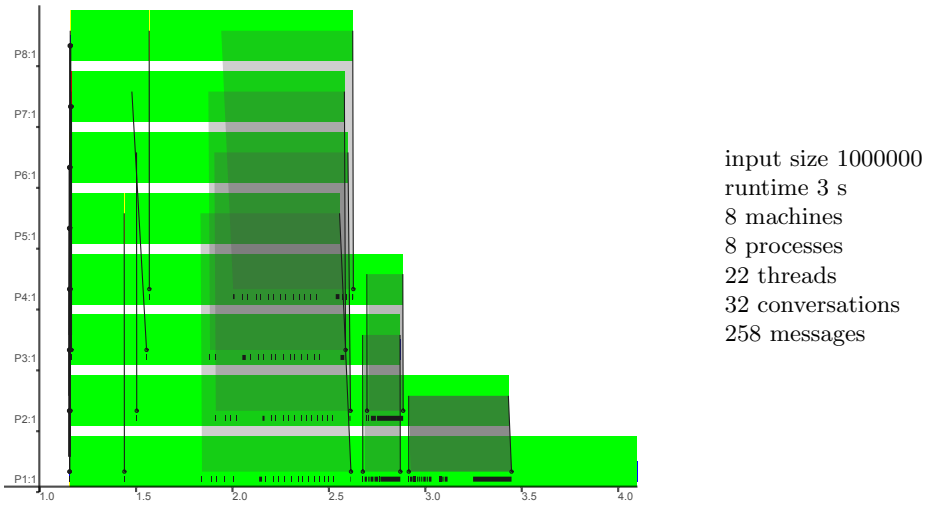


Fig. 7. Parallel `mergeSort` trace on 8 PEs, `disDC` skeleton

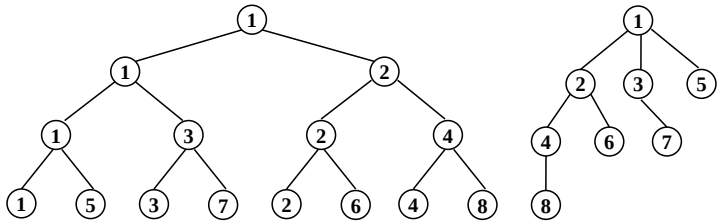


Fig. 8. Call tree (left) and process generation tree (right) of parallel `mergesort` execution

splits the input list into two halves, keeps the first half for local evaluation and creates a child process on PE 2 for sorting the second half. The remaining ticket list `[3..noPe]` is unshuffled into the two lists `[3,5,7]` and `[4,6,8]`. The first sublist is kept locally while the child process gets the second one. The main process and the child process on PE 2 proceed in parallel creating further subprocesses on PE 3 and PE 4, respectively, and unshuffling their remaining ticket lists into two sublists etc. The process generation tree on the right hand side in Figure 8 shows which process creates which other processes. As there is a one-to-one correspondence between processes and PEs processes are simply marked with the PE number.

In Figure 7 single messages are shown as black lines with a big dot at the receiver side while the communication of streams is shown as a shaded area. A stream communication consists of a series of messages. Only the last message of a stream is shown as a black line. The other messages are only indicated by a very short black line on the receiver side. In the statistics, messages and conversations

are counted. A stream communication is counted as a single conversation and as many messages as have been needed to communicate the stream. Thus in this example, we have only 32 conversations but 258 messages.

After about half of their runtime the upper half of processes (i.e. the leaf processes of the process generation tree in Figure 8) start to return their results in streams to their generator processes which merge the received lists with their own results using `sortMerge`. The whole merge phase occurs level-wise. Each process performs as many merge phases as its number of direct child processes. This can clearly be observed when relating the process generation tree and the message flow in the trace picture. Moreover, the trace clearly shows that after half of the overall runtime, the upper half of the processes finish already. In total, the PEs are badly utilised. This is the reason for the rather poor speedup which is only about 3 with 8 PEs, the runtime of the original sequential `mergeSort` with input size 1000000 being 9 sec in the same setting. ◁

Exercise 2: 1. Implement the following alternative parallelisation of the function `mergeSort`: Decompose the input list into as many sublists as processor elements are available. Create for each sublist a parallel process which sorts the sublist using the original `mergeSort` function. Merge the sorted sublists. Which skeleton(s) can be used for this parallelisation?
2. Run your parallel program on different numbers of processor elements, analyse the runtime behaviour using EdenTV, and compare your results with those achieved with the parallel divide-and-conquer version described before.

2.3 Eden's Skeleton Library

The functions `parMap`, `farm`, `mapFarmS`, `mapFarmB`, `parMapRedr`, and `parDC` defined above are simple examples for skeleton definitions in Eden. As we have seen, there may be many different parallel implementations of a single computation scheme. Implementations may differ in the process topology created, in the granularity of tasks, in the load balancing strategy, or in the communication policy. It is possible to predict the efficiency of skeleton instantiations by providing a *cost model* for skeleton implementations [38]. This aspect will however not be considered in this tutorial.

While many 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 in Haskell's prelude and libraries, Eden provides a well assorted skeleton library `Control.Parallel.Eden.EdenSkel`:

- `Control.Parallel.Eden.EdenSkel.Auxiliary` provides useful auxiliary functions like `unshuffle` and `shuffle` (see also Appendix B).
- `Control.Parallel.Eden.EdenSkel.DCSkels` comprises various divide and conquer skeletons like `parDC` or `disDC`.
- `Control.Parallel.Eden.EdenSkel.MapSkels` provides parallel map-like skeletons like `parMap`, `farm` or `offline_farm`.
- `Control.Parallel.Eden.EdenSkel.MapRedSkels` supplies parallel implementations of the map-reduce scheme like `parMapRedr` or a parallel implementation of Google map-reduce [6].
- `Control.Parallel.Eden.EdenSkel.TopoSkels` collects topology skeletons like pipelines or rings.
- `Control.Parallel.Eden.EdenSkel.WPSkels` puts together workpool skeletons like the master worker skeleton defined in Section 6.2.

3 Eden's Basic Constructs

Although many applications can be parallelised using pre-defined skeletons, it may be necessary to adjust skeletons to special cases or to define new skeletons. In these cases it is important to know the basic Eden coordination constructs for

- for defining and creating processes
- for generating non-hierarchical process topologies
- for modeling many-to-one communication.

Eden's basic constructs are defined in the Eden module `Control.Parallel.Eden` which must be imported by each Eden program.

3.1 Defining and Creating Processes

The central coordination constructs for the definition of processes are *process abstractions* and *instantiations*:

```
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

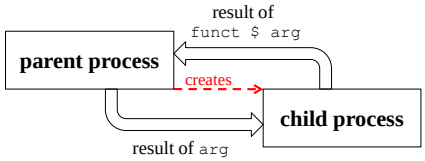


Fig. 9. Process topology after evaluating `(process funct) # arg`

parent process and will then be 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 scheme 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 (see Figure 9). 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 8. 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.

The coordination functions `process` and `( # )` are usually used in combination as in the definition of the following operator for parallel function application:

```
( $# )     :: (Trans a, Trans b) ⇒ (a → b) → a → b
f $# x  =  process f # x        -- ( $# ) = ( # ) ∘ process
```

The operator `($#)` induces that the input parameter `x`, as well as the result value, will be transmitted via channels. The types `a` and `b` must therefore belong to the class `Trans`.

In fact, this simple operator would be enough for extending Haskell with parallelism. The distinction of process abstraction and process instantiation may however be useful from a didactic point of view. A process abstraction defines process creation on the side of the child process while a process instantiation defines it on the side of the parent process. This is also reflected by the implementation of these constructs, shown in Section 8.

It is tempting to parallelise functional programs simply by using this parallel application operator at suitable places in programs. Unfortunately, in most cases this easy approach does not work. The reasons are manyfold as shown in the following simple example.

Example: In principle, a simple parallelisation of `mergeSort` could be achieved by using the parallel application operator `($#)` in the recursive calls of the original definition of `mergeSort` (see above):

```
mergeSort xs = sortMerge (mergeSort $# xs1)
                         (mergeSort $# xs2)
               where [xs1,xs2] = unshuffle 2 xs
```

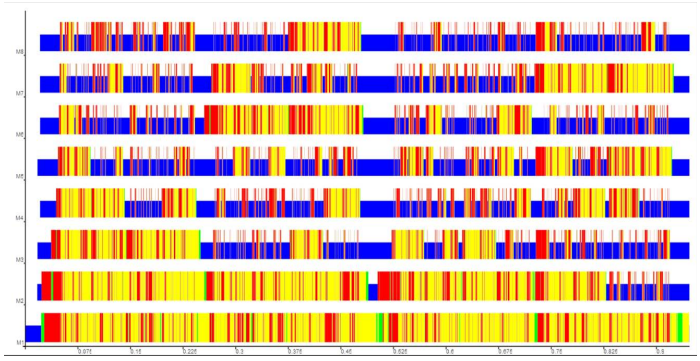
In this definition, two processes are created for each recursive call as long as the input list has at least two elements. In Figure 10 the activity profile of the 8 processor elements (machines view of EdenTV, see Appendix B) is shown for the execution of this simple parallel mergesort for an input list of length 1000. The processor elements are either idle (small blue bar), i.e. they have no processes to evaluate, busy with system activity (yellow/light grey bar), i.e. there are runnable processes but no process is being executed or blocked (red/dark grey bar), i.e. all processes are waiting for input. The statistics show that 1999 processes have been created and that 31940 messages have been sent. The parallel runtime is 0.95 seconds, while the sequential runtime is only 0.004 seconds, i.e. the parallel program is much slower than the original sequential program. This is due to the excessive creation of processes and the enourmous number of messages that has been exchanged. Moreover, this simple approach has a demand problem, as the processes are only created when their result is already necessary for the overall evaluation. In the following sections, we will present techniques to cope with these problems. ◁

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 can be created and connected by such streams.

Example: The sequence of all multiples of two arbitrary integer values n and m

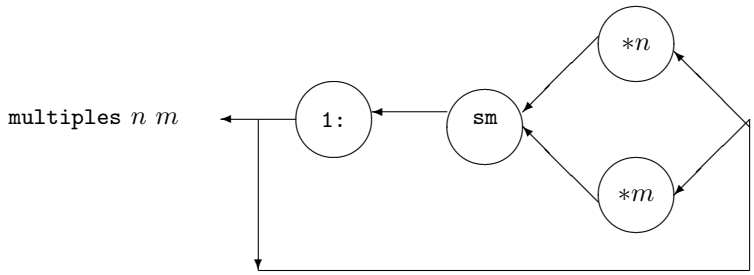
$$\langle n^i m^j \mid i, j \geq 0 \rangle$$

can easily be computed using the following cyclic process network:



input size 1000
par. time 0.95s
seq. time 0,004s
8 machines
1999 processes
31940 messages

Fig. 10. Trace visualisation of simple parallel mergesort, machines view



This network can be expressed in Eden as follows:

```
multiples :: Integer → Integer → [Integer]
multiples n m = ms
     where ms = 1: sm (map (*n) $# ms) (map (*m) $# ms)
```

The ordinary Haskell function `sm` works in a similar way as the `sortMerge` function used in the mergesort example but it eliminates duplicates when merging its sorted input 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
```

In this example two child processes will be created corresponding to the two applications of `($#)`. Each of these processes receives the stream of multiples from the parent process, multiplies each element with n or m, respectively, and sends each result back to the parent process. The parent process will evaluate the application of `sm` to the two result streams received from the two child processes. It uses two concurrent threads to supply the child processes with their input. Streaming is essential in this example to avoid a deadlock. The parallelism is

rather fine-grained with a low ratio of computation versus communication. Thus, speedups cannot be expected when this program is executed on two processors. ◁

Exercise 3: Define a cyclic process network to compute the sorted sequence of Hamming numbers $\langle 2^i 3^j 5^k \mid i, j, k \geq 0 \rangle$. Implement the network in Eden and analyse its behaviour with EdenTV.

3.2 Coping With Laziness

The laziness of Haskell has many advantages when considering recursive process networks and stream-based communication as shown above. 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 subcomputation. Thus, sometimes it is necessary to produce additional demand in order to unfold certain process systems. Otherwise, the programmer may experience *distributed sequentialism*. This situation is illustrated by the following attempt to define `parMap` using Eden's parallel application operator `($#)`:

Example: Simply replacing the applications of the parameter function in the `map` definition with parallel applications leads to the following definition:

```
parMap_distrSeq           :: (Trans a, Trans b) ⇒
                             (a → b) → [a] → [b]
parMap_distrSeq f []      =  []
parMap_distrSeq f (x:xs)  =  (f $# x) : parMap_distrSeq f xs
```

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 due to "parallel" evaluation, but slow down because of the process creation overhead added to the distributed, but sequential evaluation. Figure 11 shows the trace of the program for the parallel computation of π in which `parMap` has been replaced with `parMap_distrSeq` in the definition of the skeleton `offline_parMapRedr`. The input parameter has been 1000000 as in Figure 5. The distributed sequentialism is clearly revealed. The next process is always only created after the previous one has terminated. Note that the 8th process is allocated on PE 1. Its activity bar is the second one from the bottom. ◁

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:

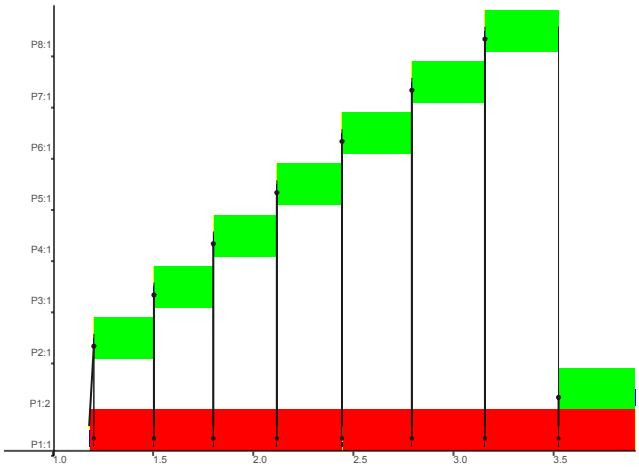


Fig. 11. Trace visualisation of pi program with `parMap_distrSeq`

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

The actual definition is shown in Section 8. 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]
```

In fact, `spawn` is defined as `spawnAt [0]`. The parameter `[0]` leads to the default round-robin process placement.

The counter part `spawnF` with a purely functional interface can be defined as follows:

```
spawnF :: (Trans a, Trans b) ⇒ (a → b) → [a] → [b]
spawnF =  spawn ∘ (map process)
```

The actual definition of `parMap` uses `spawn`:

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

The Haskell prelude function `repeat :: a → [a]` yields an infinite list by repeating its parameter.

Although `spawn` helps to eagerly create a series of processes, it may sometimes be necessary to add even more additional demand to support parallelism. For that purpose one can use the *evaluation strategies* provided by the library `Control.Parallel.Strategies` [41]. The next subsection and Section 5.2 contain examples.

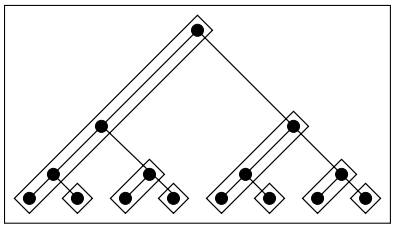


Fig. 12. Distributed expansion divide and conquer skeleton for a binary task tree

3.3 Implementing the Divide-and-Conquer Skeleton `disDC`

The skeleton `disDC` which we have used in Section 2 to parallelise the `mergeSort` algorithm implements a *distributed expansion scheme*, i.e. the process tree expands 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. Figure 12 shows the binary tree of task nodes produced by a divide-and-conquer 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. This tree corresponds with the call tree of the parallel `mergeSort` execution shown in Figure 8.

For the distributed expansion scheme explicit placement of processes is essential to avoid that too many processes are placed on the same PE while leaving others unused. Therefore `spawnAt` is used for process creation in the Eden implementation of the `disDC` skeleton shown in Figure 13.

Two additional parameters are used: the branching degree `k` and a `tickets` 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`.

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

The parallel creation of processes is explicitly supported using the explicit demand control function

```
childRes `pseq` rdeepseq myRes `pseq`
```

The function `pseq :: a → b → b` evaluates its first argument to weak head normal form before returning its second argument. Note that `` `pseq` `` denotes the infix variant of this function. The strategy function `rdeepseq` forces the complete

```
disDC :: (Trans a, Trans b) ⇒
         Int → [Int] →        -- ^ branch degree / tickets
         DivideConquer a b
disDC k tickets trivial solve split combine x
  = if null tickets then seqDC x
    else recDC tickets x
  where
    seqDC = dc trivial solve split combine
    recDC tickets x =
      if trivial x then solve x
      else childRes           `pseq`  -- explicit demand
           rdeepseq myRes     `pseq`  -- control
           combine x ( myRes:childRes ++ localRess )
      where
        -- child process generation
        childRes   = spawnAt childTickets childProcs procIns
        childProcs = map (process ∘ recDC) 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      = recDC myTs myIn
        localRess  = map seqDC localIns
```

Fig. 13. Definition of distributed-expansion divide-and-conquer skeleton

evaluation of its argument (to normal form) [41]. Both functions are originally provided in the library `Control.Deepseq` but re-exported from the Eden module. The above construct has the effect that first the child processes are created because the expression `childRes` is an application of `spawnAt`. As soon as all processes have been created, the strategy `rdeepseq` forces the evaluation of `myRes`, i.e. the recursive unfolding and generation of further processes. Using `pseq` the evaluation order of subexpressions is explicitly determined. Only then, the standard Haskell evaluation strategy is used to evaluate the overall result expression `combine x ( myRes:childRes ++ localRess )`.

4 Controlling Communication Costs

In many cases, it is not sufficient to simply instantiate a skeleton like `parMap`, `parMapRedr`, `farm` or `disDC` to parallelise a program. Often it is necessary to apply some techniques to reduce the communication costs of parallel programs, especially, when big data structures have to be transmitted between processes. In the following subsections, we explain two such techniques. We use a simple case study, raytracer, to show the effectiveness of these techniques. Details on

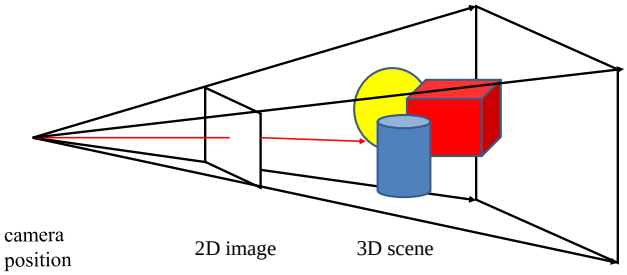


Fig. 14. Raytracing

the case study, especially the complete programs, can be found on the Eden web pages.

4.1 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 severely delimit the performance of the parallel program, as we have already mentioned in the examples discussed in Section 2.

Case Study (Raytracer): Ray tracing is a technique in computer graphics for generating a two-dimensional image from a scene consisting of three-dimensional objects. As the name indicates rays are traced through pixels of the image plane calculating their impacts when they encounter the objects (see Figure 14). The following central part of a simple raytracer program can easily be parallelised using the farm skeleton.

```
raytrace :: [Object] → CameraPos → [Impact]
rayTrace scene viewpoint
  = map impact rays
  where rays = generateRays viewPoint
        impact ray = fold earlier (map (hit ray) scene)
```

By replacing the outer `map` with `mapFarmS` (defined in Section 2, see page 7) 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 process creation, each process will evaluate it.

Figure 15 shows the trace visualisation (processes' activity over time) and some statistics produced by our trace viewer EdenTV (see Section A.3). The trace has been generated by a program run of the raytracer program with input size 250, i.e. 250^2 rays on an Intel 8-core machine ($2 \times$ Xeon Quadcore @2.5GHz, 16 GB RAM) machine using the PVM-based runtime system. As PVM works

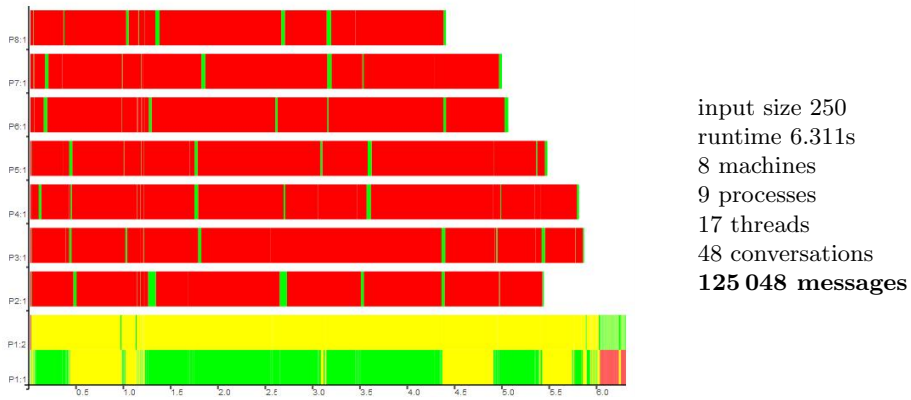
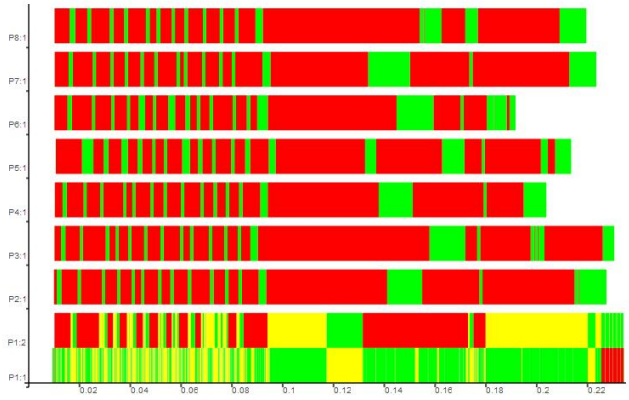


Fig. 15. Raytracer trace on 8 PEs, farm with 8 processes

with a demon which must be started before any program execution the startup time of the parallel program is neglectable. 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 (see the two bottom bars with process numbers `P 1:1` and `P 1:2`) . Alarming is the enormous number of 125048 messages that has been exchanged between the processes. When messages are added to the trace visualisation, the graphic becomes almost black. It is 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 48 and thus much less than the number of messages. ◇

In such cases it is advantageous to communicate a stream in larger segments. Note that this so-called *chunking* is not always advisable. In the simple cyclic network shown before 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`. It can equally well be used in other skeletons like e.g. `disDC` as shown in Subsection 4.3. The auxiliary function `chunk` decomposes a list into chunks of the size determined by its first parameter. See Appendix B for its definition which can be imported from the `Auxiliary` library. The function `chunkMap` applies `chunk` on the input list, applies a map skeleton `mapscheme` with parameter function



input size 250
chunk size 250
runtime 0.235s
8 machines
9 processes
17 threads
48 conversations
548 messages

Fig. 16. Raytracer trace on 8 PEs, farm with 8 processes and chunking

`(map f)` and finally concatenates the result chunks using the Haskell prelude 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))
```

Case Study (Raytracer continued): In our raytracer case study, we replace the `mapFarmS` skeleton with `chunkMap chunksize mapFarmS` where `chunksize` is an extra parameter of the program. Chunking substantially reduces communication costs, as the number of messages drops drastically when chunking is used. With input size 250, which means that 250^2 stream elements have to be sent to the farm processes and to be returned to the main process, and chunk size 250 the number of messages drops from $125048 (= 125000 + 48)$ downto $548 (= 500 + 48)$. This leads to much better results (see Figure 16). It becomes clear that processes are more active, but still are blocked a lot of time waiting for input from the main process. Only the main process (see bottom bar) is busy most of the time sending input to the farm processes. The number of messages has drastically been reduced, thereby improving the communication overhead and consequently the runtime a lot. A speedup of 26,86 in comparison to the previous version could be achieved. Nevertheless, the program behaviour can still be improved. ◇

Exercise 4: Give an alternative definition of the `mapFarmB` skeleton using

`chunkMap size parMap`

with a suitable `size` parameter. Assume that the length of the input list is a multiple of the number of processor elements.

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

4.2 Communication vs Parameter Passing: Running Processes Offline

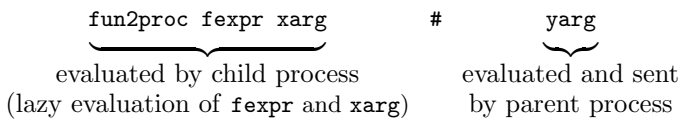
Eden processes have disjoint address spaces. Consequently, on process instantiation, the process abstraction will be transferred to the remote processing element including its whole environment, i.e. the whole heap reachable from the process abstraction will be copied. This is done even if the heap includes non-evaluated parts which may cause the duplication of work. A programmer is able to avoid work duplication by forcing the evaluation of unevaluated subexpressions 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 subexpression (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      :: (Trans b, Trans c) ⇒
                 (a → b → c) → a → Process b c
fun2proc f x =  process (\ y → f x y)
```

and the following process instantiation:



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 be 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      :: Trans b ⇒ (a → b) →  a → Process () b
rfi h x =  process (\ () → h x)
```

```
offline_farm :: Trans b ⇒
  Int              →       -- ^ number of processes
  ([a] → [[a]]) →          -- ^ input distribution
  ([[b]] → [b]) →          -- ^ result combination
  (a → b) → [a] → [b] -- ^ map interface
offline\_farm np distribute combine f xs
  = combine $ spawn (map (rfi (map f))
                         [select i xs | i ← [0..np-1]])
                    (repeat ())
  where select i xs = (distribute xs ++ repeat []) !! i
```

Fig. 17. Definition of `offline_farm` skeleton

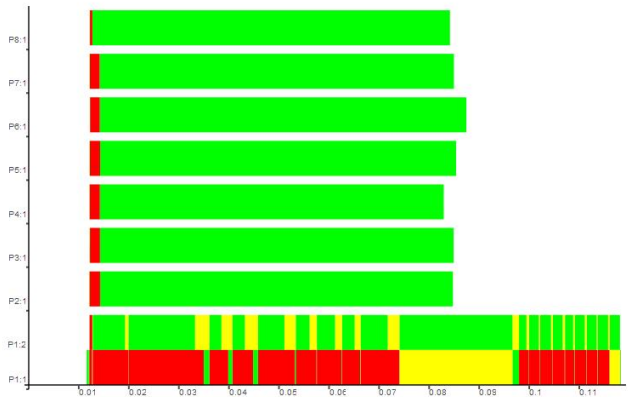
Now the argument of `h` will be evaluated on demand by the remote process itself. The empty tuple `()` (unit) is used as a dummy argument in the process abstraction. If the communication network is slow or if the result of the argument evaluation is large, instantiation via `rfi h x # ()` may be more efficient than using `(h $# x)`. We say that the child process runs *offline*.
The same technique has been used in Section 2 to define the `offline_parMapRedr` skeleton. In a similar way, the previously defined `farm` can easily be transformed into the *offline farm* defined in Figure 17, which can equally well be used to parallelise `map` applications. In contrast to the `farm` skeleton, this skeleton needs to know the number of processes that has to be created. Note that the offline farm will definitely create as many processes as determined by the first parameter. If input distribution does not create enough tasks, the selection function guarantees that superfluous processes will get an empty task list.

Although the input is not explicitly communicated to an `offline_farm`, chunking may still be useful for the result stream.

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. Thus, we save 8 stream communications and 258 messages. Figure 18 shows that the farm processes are now active during all their life time. The runtime could further be reduced by a factor of almost 2. ◇

4.3 Tuning the Parallel Mergesort Program

The tuning techniques "offline processes" and "chunking" have also been used to tune the parallel mergesort program presented in Section 2. Avoiding input communication using the offline technique requires slightly more modifications which can however be defined as a stand-alone offline distributed-expansion divide-and-conquer skeleton `offline_disDC`. The idea is to pass unique numbers to the processes which identify its position in the divide-and-conquer call tree. The processes use these numbers to compute the path from the root to their



input size 250
chunk size 250
runtime 0.119s
8 machines
9 processes
17 threads
40 conversations
290 messages

Fig. 18. Raytracer trace on 8 PEs, offline farm with 8 processes and chunking

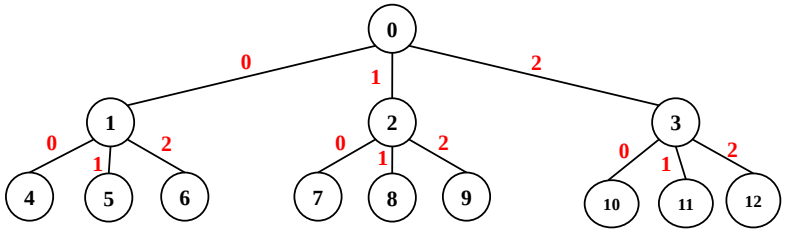


Fig. 19. Numbering of call tree nodes in offline divide-and-conquer skeleton

positions and to select the appropriate part of the input by subsequent applications of the `split` function starting from the original input. Figure 19 shows the node numbering of a call tree with branching degree $k = 3$. Auxiliary functions `successors` and `path` are used to compute the successor node numbers for a given node number and the path from the root node to a given node:

```
successors :: Int → Int → [Int]
successors k n = [nk + i | let nk = n*k, i ← [1..k]]

path :: Int → Int → [Int]
path k n | n == 0    = []
         | otherwise = reverse (factors k n)

factors :: Int → Int → [Int]
factors k n
  | n ≤ 0    = []
  | otherwise = (n+k-1) `mod` k : factors k ((n-1) `div` k)
```

For the example tree in Figure 19 we observe that `successors 3 2 = [7,8,9]` and `path 3 9 = [1,2]`. If we neglect the case that the problem size might not be

large enough to supply each process with work, the offline divide-and-conquer skeleton can be defined as follows:

```
offline_disDC :: Trans b ⇒
                 Int → [Int] → DivideConquer a b
offline_disDC k ts triv solve split combine x
  = disDC k ts newtriv newsolve newsplit newcombine 0
  where
    seqDC      = dc triv solve split combine
    newsplit   = successors k
    newtriv  n = length ts ≤ k^(length (path k n))
    newsolve n = seqDC  (select split x (path k n))
    newcombine n bs
      = combine (select split x (path k n)) bs

select :: (a → [a]) → a → [Int] → a
select split x ys = go x ys
  where go x []     = x
        go x (y:ys) = go (split x !! y) ys
```

The skeleton will create as many processes as the length of the ticket list. The `successors` function is used as `split` function for the offline divide-and-conquer skeleton. The initial input is set to zero, the root node number of the call tree. The predicate `newtriv` stops the parallel unfolding as soon as number of leaves in the generated call tree is greater than the length of the ticket list, i.e. the number of processes that has to be generated. When the parallel unfolding stops, the skeleton applies the normal sequential divide-and-conquer function `dc` to solve the remaining subproblems. The auxiliary function `select` computes the subproblem to be solved by node with number `n`. It successively applies the original `split` function and selects the appropriate subproblems with the Haskell list index operator `(!!) :: [a] → Int → a`, thereby following the path `path k n` from the root node to the current node. The `combine` function is also modified to locally select the current subproblem. In most cases this simplified version of the offline divide-and-conquer skeleton will work satisfactorily. However, the skeleton will bounce whenever the problem size is not large enough to allow for the series of `split` applications. Therefore, Figure 20 presents a modified version of the skeleton definition which checks whether splitting is still possible or not. If no more splitting is possible, the process has no real work to do, because one of its predecessor processes has the same problem to solve. In principle, it need not produce a result. Changing the internal result type of the skeleton to e.g. a `Maybe` type is however not advisable because this would e.g. de-activate chunking or streaming, if this is used for the result values in the original skeleton. Instead, the skeleton in Figure 20 internally produces two results, a flag that indicates whether the process created a valid subresult or whether it already the result its parent process simply can pass. In fact, all superfluous processes compute the result of a trivial problem which is assigned to one of its predecessor. The corresponding predecessor can then simply overtake the first of the (identical) results of its child processes. This offline divide-and-conquer skeleton has been used to produce the trace file in Figure 7. In addition, chunking of the result

```
offline_disDC :: Trans b ⇒
                 Int → [Int] → DivideConquer a b
offline_disDC k ts triv solve split combine x
  = snd (disDC k ts newtriv newsolve newsplit newcombine 0)
  where
    seqDC      = dc triv solve split combine
    newsplit   = successors k
    newtriv  n = length ts ≤ k^(length (path k n))
    newsolve n = (flag, seqDC localx)
      where (flag, localx) = select triv split x (path k n)
    newcombine n bs@((flag,bs1):_)
      = if flag then (True, combine localx (map snd bs))
                else (lab,  bs1)
      where (lab, localx) = select triv split x (path k n)

select :: (a → Bool) → (a → [a])  -- ^ trivial / split
          → a → [Int] → (Bool,a)
select trivial split x ys = go x ys
  where go x []     = (True, x)
        go x (y:ys) = if trivial x then (False, x)
                      else go (split x !! y) ys
```

Fig. 20. Offline version of the divide-and-conquer skeleton `disDC`

lists has been added by adapting the parameter functions of the `offline_disDC` skeleton, i.e. composing the function `chunk size` with the result producing parameter functions `solve` and `combine` and unchunking parameter list elements of `combine` as well as the overall result using `concat`. Note that the parameter functions `trivial`, `solve`, `split`, and `combine` are the same as in the definition of the function `mergeSortDC` (see Page 14, Section 2). Finally, the actual code of the parallel mergesort implementation is as follows:

```
par_mergeSortDC :: (Ord a, Trans a) ⇒ Int → [a] → [a]
par_mergeSortDC size
  = concat ∘
      (offline_disDC 2 [2..noPe] trivial
         ((chunk size) ∘ solve)   split
         (\ xs → (chunk size) ∘ (combine xs) ∘ (map concat)))
  where
    -- the same as in mergeSortDC
```

Exercise 5: Change the parallel mergesort implementation in such a way that the branching degree of the recursive call tree can be given as an additional parameter to the function `par_mergeSortDC`.

5 Defining Non-Hierarchical Communication Topologies

With the Eden constructs introduced up to now, communication channels are only (implicitly) established during process creation between parent and child

processes. These are called *static channels* and they build purely hierarchical process topologies. Eden provides additional mechanisms to define non-hierarchical communication topologies.

5.1 The Remote Data Concept

A high-level, natural and easy-to-use way to define non-hierarchical process networks like e.g. rings in Eden is the *remote data concept* [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 transmit 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.

The remote data feature has the following interface in Eden [20]:

```
type RD a  -- remote data

-- convert local data into remote data
release :: Trans a  ⇒ a → RD a

-- convert remote data into local data
fetch :: Trans a  ⇒ RD a → a
```

The following 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

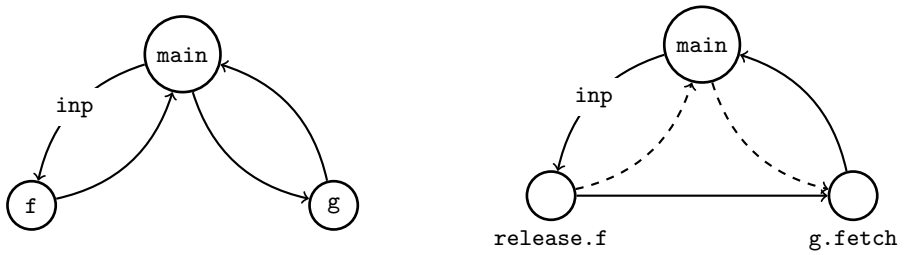
```
(g $# (f $# inp)).
```

This leads to the process network in Figure 21 (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 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`. 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 21 (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



(a) indirect connection (b) direct connection

Fig. 21. A simple process graph

```
ring      :: (Trans i, Trans o, Trans r) ⇒
             ((i,r) → (o,r))    --^ ring process function
             → [i] → [o]        --^ 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
```

Fig. 22. Definition of `ring` skeleton

version, i.e. it is passed via the main process from the process computing `f` to the one computing `g`. ◁

5.2 A Ring Skeleton

Consider the definition of a process ring in Eden given in Figure 22. The number of processes in the ring is determined by the length of the input list. The ring processes are created using the `parMap` skeleton.

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 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`. Thus, the program would get stuck without the lazy pattern, because the ring input will only be produced after process creation and process creation will not occur without the first input.

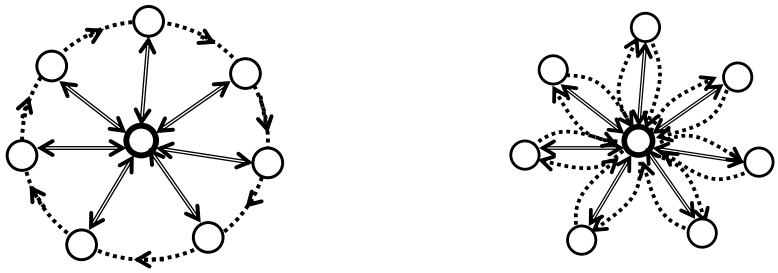


Fig. 23. Topology of process ring (left: intended topology, right: actual topology)

```
ringRD      :: (Trans i, Trans o, Trans r) ⇒
               ((i,r) → (o,r))    --^ ring process function
               → [i] → [o]        --^ 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)
```

Fig. 24. Ring skeleton definition with remote data

Unfortunately, this elegant and compact ring definition will not produce a ring topology but a star (see Figure 23). 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 where 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.

Fortunately, the process ring can easily be re-defined using the remote data concept as shown in Figure 24. 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. Thus, the worker functions of the parallel processes have a different type. In fact, the star topology is still used but only to propagate remote data handles. The proper data is passed directly from one process to its successor in the ring.

19.079864s (a) without demand control

2.922803s (b) with demand

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

Fig. 25. Warshall trace on 8 PEs, ring with 8 processes and chunking

This transfer occurs via additional so-called *dynamic* channels, which are not reflected in the worker function type. This is the mechanism used to implement the remote data concept in Eden. We will introduce Eden's dynamic channels and the implementation of remote data using dynamic channels in Section 7. Before we present a case study with the ring skeleton `ringRD`.

Case Study (Warshall's algorithm): Warshall's algorithm for computing shortest paths in a graph given by an adjacency matrix can be implemented using the ring skeleton. 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 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. Note that the final argument of this function is the one communicated via the ring.

```
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
  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 row with the information of the row received from its ring predecessor.

Unfortunately, a trace analysis reveals (see Figure 25(a)) that this program has a demand problem. In a first part, 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

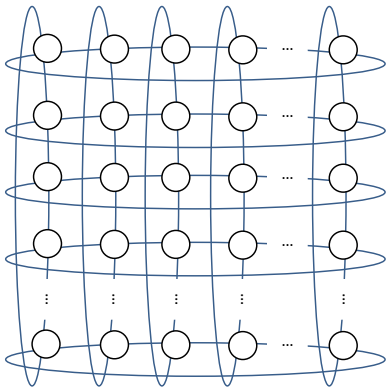


Fig. 26. Torus topology

of the ring can be compressed. The additional demand can be expressed by `rdeepseq nextrowk ‘pseq‘` which has to be included before the recursive call to ring_iterate:

```
(rowR, restoutput) = rdeepseq nextrowk ‘pseq‘
                     ring_iterate size k (i+1) nextrowk xs
```

This forces 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 25(b). Note the different scaling on the x-axes in both pictures. The improved program version needs only a sixth of the runtime of the first version. ◇

Exercise 6: Define a ring skeleton in such a way that each process creates its successor processor. Use remote data to close the ring and to establish the communication with the process which executes the ring skeleton.

5.3 A Torus Skeleton

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, i.e. the processes per row and per column form process rings (see Figure 26). In addition, each process has two extra connections to send and receive values to/from the parent. These are not shown in Figure 26. The torus topology can be used to implement systolic computations, 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 27 creates the desired toroidal topology of Figure 26 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 predecessors. It produces an output to the parent and two remote handles to release outputs to its successors. The shape of the torus is determined by the shape of the input.

The size of the torus will usually depend on the number of available processors (`noPe`). A typical value is e.g. $\lfloor\sqrt{\texttt{noPe}}\rfloor$. 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 value of type `c` from the parent, a stream `[a]` from the left predecessor and a stream `[b]` from its upper predecessor, and produces a final result `d` for its parent as well as result streams of type `[a]` and `[b]` for its right and lower successors, 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 [23]. Each node of the torus gets two blocks of the input matrices to be multiplied sequentially. It passes its blocks to the successor processes in the torus, the block of the first matrix to the successor in the row and the block of the second matrix to the successor 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. The torus can be instantiated with the following node function:

```
nodefunction :: Int                          --^ torus dimension
 → ((Matrix,Matrix), [Matrix], [Matrix]) --^ process input
 → ([Matrix],        [Matrix], [Matrix]) --^ process output
nodefunction n ((bA,bB), rows, cols)
  = ([bSum], bA:nextAs, bB:nextBs)
  where bSum = foldl' matAdd (matMult bA bB)
                             (zipWith matMult nextAs nextBs)
        nextAs = take (n-1) rows
        nextBs = take (n-1) cols
```

The result matrix block is embedded in a singleton list to avoid its streaming when being returned to the main process. Figure 28 shows a trace of the torus parallelisation of matrix multiplication created on the Beowulf cluster II for input matrices with dimension 1000. Messages are overlayed. In total, 638 messages have been exchanged. 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 matrix blocks without manipulating them, communication occurs immediately. Afterwards, the actual computations are performed. Finally the processes return their local result blocks to the main process on PE 1 (bottom bar). ◇

```
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 (repeat (ptorus f))) t_inss
  (outss,outssA,outssB) = unzip3 (map unzip3 t_outss)
  inssA   = map rightRotate outssA
  inssB   = rightRotate outssB
  t_inss  = lazyzipWith3 lazyzip3 inss inssA inssB

-- 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) →
               let (toParent, outA, outB)
                    = f (fromParent, fetch inA, fetch inB)
               in  (toParent, release outA, release outB))

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

lazyzip3 :: [a] → [b] → [c] → [(a,b,c)]
lazyzip3 = lazyzipWith3 (\ x y z → (x,y,z))
```

Fig. 27. Definition of `torus` skeleton

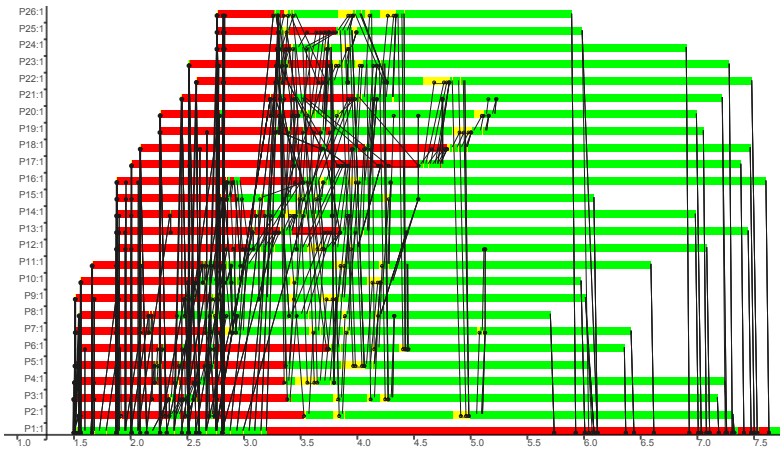


Fig. 28. Trace of parallel matrix multiplication with `torus` skeleton

6 Workpool Skeletons

Workpool skeletons provide a powerful and general way to implement problems with irregular parallelism, which are decomposed into tasks of varying complexity. For such problems it is feasible to implement a task or work pool which is processed by a set of worker processes. The tasks are dynamically distributed among the workers to balance the work load. Often a master process manages the task pool, distributes the tasks to the worker processes, and collects the results. Then the work pool is organised as a master-worker system. In such systems it is important that the master reacts immediately on the reception of a worker result by sending a new task to the respective worker, i.e. the master must receive worker results as soon as they arrive. Thus, many-to-one communication is necessary for the communication from the workers to the master.

6.1 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. In fact, merging several incoming streams guarantees that incoming values are passed to the single output stream as soon as they arrive. Thus, merging the results streams of the worker processes allows the master in a master-worker system to react quickly on the worker results which are also interpreted as requests for new tasks. As the incoming values arrive in an unpredictable order, `merge` introduces non-determinism. Nevertheless functional purity can be preserved in most portions of an Eden program. It is e.g. possible to use sorting in order to force a particular order of the results returned by a `merge` application and thus to encapsulate `merge` within a skeleton and save the deterministic context. In the next subsection we show how a determinstic master-worker skeleton is defined although the `merge` function is internally used for the worker-to-master communication.

6.2 A Simple Master-Worker Skeleton

The `merge` function is the key to enable dynamic load balancing in a master-worker scheme as shown in Figure 29. 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 30) 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

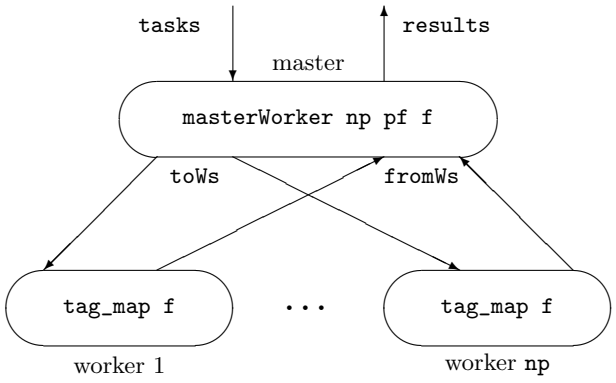


Fig. 29. Master-worker process topology

```
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 (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])
```

Fig. 30. Definition of a simple master worker skeleton

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 workers. 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]`). Note that the functions `distribute` and `orderBy` can be imported from Eden's `Auxiliary` library. Their definitions are also given in Appendix B.

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 30). The `prefetch` parameter determines the behaviour of the skeleton, between a completely dynamic (prefetch 1) and a completely static task distribution (prefetch $\geq \frac{\#\text{ tasks}}{\#\text{ workers}}$). Further tasks are sent to workers which have delivered a result. The list `newReqs` is extracted from the worker results which are tagged with the corresponding worker id. The requests are merged according

Fig. 31. Mandelbrot traces on 25 PEs, offline farm (left) vs master-worker (right)

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. Note that we have assumed a statically fixed task pool, which, in essence, results in another parallel map implementation with dynamic assignment.

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       -- ^ threshold for iterations
         → Complex Double → Complex Double
                      -- ^ coordinates
         → Integer    -- ^ size
         → 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
```

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` with a parallel map implementation. Figure 31 shows two traces that have been produced on the Beowulf cluster I at Heriot-Watt University in Edinburgh. 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 noPe (splitInto noPe) concat)`. The program yielding the trace on the right hand side replaced `map` by `masterWorker noPe 8`, where in both cases `noPe = 25`. 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

version	number of processes	task transfer	task distribution
parMap	number of tasks	communication	one per process
mapFarm{S/B}	noPe	communication	static
offline_farm	mostly noPe	local selection	static
masterWorker	mostly noPe	communication	dynamic

Fig. 32. Classification of parallel map implementations

`splitInto noPe` for a block-wise task distribution instead of `unshuffle 25`. The latter would lead to a well-balanced workload with the farm which is however a special property of this example problem. In general, 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 PE. The trace on the right hand side in Figure 31 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. ◇

Exercise 7: Define a master-worker skeleton `mwMapRedr` which implements a map-reduce scheme.

```
mwMapRedr :: Int                -- ^ number of processes
          → Int                 -- ^ prefetch
          → (b → b → b)         -- ^ reduce function
          → b                   -- ^ neutral element
          → (a → b)             -- ^ map function
          → [a] → b             -- ^ input - output mapping
```

The worker processes locally reduce their results using `foldr` and return requests to the master process to ask for new input values (tasks). When all tasks have been solved (how can this be detected?), the worker processes return their reduction result to the master who performs the final reduction of the received values.

6.3 Classification of Parallel Map Implementations

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 32. The simple `parMap` is mainly used to create a series of processes evaluating the same function. The static task distribution implemented in the `farm` variants 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.

Exercise 8: Write parallel versions of the Julia-Set program provided on the Eden web pages using the various parallel map skeletons. Use EdenTV to analyse and compare the runtime behavior of the different versions.

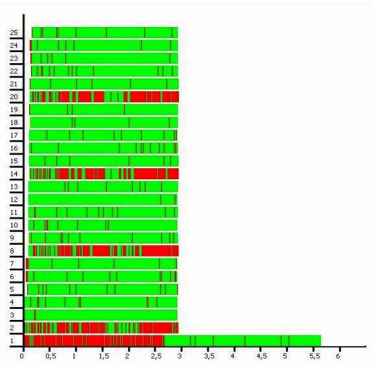
6.4 A Nested Master-Worker Skeleton

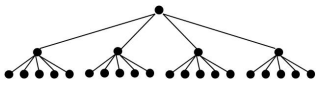
The master worker skeleton defined above is a very simple version of a workpool skeleton. It has a single master process with a central workpool, the worker processes have no local state and cannot produce and return new tasks to be entered into the workpool. Several more sophisticated workpool skeletons 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 [7, 50]. 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. Figure 33 shows the definition of the corresponding skeleton `mwNested`.

```
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
```

Fig. 33. Definition of nested workpool skeleton

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 34 shows an example trace produced for input size 2000 on 25 PEs with a two-level master worker system comprising four submasters 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





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

Fig. 34. Mandelbrot trace on 25 PEs with hierarchical master-worker skeleton (hierarchy shown on the right)

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 31). ◇

7 Explicit Channel Management

In Eden, process communication occurs via unidirectional one-to-one channels. In most cases, these channels are implicitly created on process creation. This mechanism is sufficient for the generation of hierarchical communication topologies. In Section 5, we have seen how non-hierarchical process topologies and corresponding skeletons like rings can easily be defined using the remote data concept. This concept is based on the lower-level mechanism of dynamically creating channels by receiver processes.

7.1 Dynamic Channels

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 result expression `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`.

```
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)
```

Fig. 35. Definition of ring skeleton with dynamic channels

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 direction opposite to the desired data transfer.

Example: It is of course also possible to define the ring skeleton directly using dynamic channels. Again, the ring function `f` is replaced 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 `ringDC` given in Figure 35.

Each ring process creates an input channel which is immediately returned to the parent process and passed to the predecessor process. It receives from the parent a channel to send data to the successor in the ring and uses this channel to send the ring output `ringOut` to its successor process using 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.

```
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)
```

Fig. 36. Definition of remote data with dynamic channels

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

7.2 Implementing Remote Data with Dynamic Channels

Remote data can be implemented in Eden using dynamic channels [20] as shown in Figure 36.

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 name to A, and thus to 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, is created and passed to process B. When process A receives a channel `c` (of type `ChanName a`) from B via `cc`, it sends the local data `x` via `c` to B. Conversely, in order to `fetch` remote data, represented by the remote data handle `cc`, process B creates a new (dynamic) channel `c` and sends the channel name via `cc` to A. The proper data will then be received via the channel `c`.

8 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. In principle, it shares its parallel runtime system (PRTS) with Glasgow parallel Haskell [58] 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 [47, 58, 14, 34] and describe Eden's implementation on top of the module

`Control.Parallel.Eden.ParPrim`.

This module provides primitive monadic operations which are used to implement the Eden constructs on a higher-level of abstraction [10].

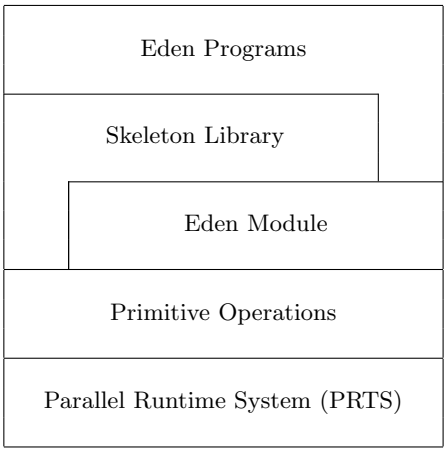


Fig. 37. Layer structure of the Eden system

8.1 Layered Parallel Runtime Environment

Eden's implementation has been organised in layers (see Figure 37) 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. 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.

The Eden module contains Haskell definitions of the high-level Eden constructs, thereby making use of the *primitive operations* shown in Figure 38. The primitive operations implement basic actions which have to be performed directly in the runtime system of the underlying sequential compiler GHC[2].

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 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.

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

Channel Administration:
- `createC#` creates a placeholder and an inport 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

Fig. 38. Primitive operations

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 imput 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`.

```
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
```

Fig. 39. Type class `Trans`

8.2 The Type Class Trans

As explained in Section 3, 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 definition of the `Trans` class is given in Figure 39. 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.

For streams, `write` is specialized in such a way that it evaluates each list element to normal form before transmitting it using `sendData Stream`. The corresponding instance declaration for lists is shown in Figure 40. For tuples (up to

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

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
```

Fig. 40. `Trans` instance declarations for lists and pairs

9 components), channel creation is overloaded as shown exemplarily for pairs in Figure 40. 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.

For self-defined data structures that are input or output of processes, the Eden programmer must provide instance declarations for the classes `NFData` and `Trans`. In most cases, it is sufficient to use the default definition for the `Trans` class and to define a normal form evaluation function `rnf`.

Example: For binary trees the following declarations would be sufficient:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

instance NFData a ⇒ NFData (Tree a) where
 rnf (Leaf x)     = rnf x
 rnf (Node x l r) = rnf x `seq` rnf l `seq` rnf r

instance Trans a ⇒ Trans (Tree a)
```

With these declarations, trees will be completely evaluated before being sent in a single message. ◁

8.3 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

```
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
```

Fig. 41. PA monad definition

operations within the PA monad and unwrap the parallel action only once. The definition of the PA monad is given in Figure 41. Note that the data constructor `PA` of the PA monad is not exported from the Eden module. Thus, the ordinary programmer can only use `return` and `bind` to specify series of parallel actions.

In Section 7, 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 as shown in Figure 42. In the PA variants of `fetch` and `release`, a channel is created, a thread is forked and in the `release` case the channel and in the `fetch` case the value received via the channel is returned.

The PA monad is especially advantageous when defining series of parallel activities like e.g. when each component of a data structure has to be released or fetched. In particular, this keeps the compiler from applying optimising transformations that are not safe for side-effecting operations.

Example: The following definitions transform 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
```

Note that the predefined Haskell function

```
mapM :: (Monad m) ⇒ (a → m b) → [a] → m [b]
```

lifts a monadic function to lists. ◁

Exercise 9: Define functions `releaseTree` and `fetchTree` to release node-wise the data elements in binary trees:

```
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
```

Fig. 42. Implementation of Remote Data using the PA monad

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

releaseTree :: Trans a ⇒ Tree a      → Tree (Rd a)
fetchTree   :: Trans a ⇒ Tree (Rd a) → Tree a
```

8.4 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.

A process abstraction of type `Process a b` is implemented by a function `f_remote` (see Figure 43) 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

```
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)
```

Fig. 43. Implementation of process abstraction

```
( # ) :: (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
```

Fig. 44. Implementation of process instantiation

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

```
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]
```

Fig. 45. Definition of `spawn`

generated. Then a primitive 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 45. Note that it is not necessary to provide a processor number for each process abstraction. The list with PE numbers is cycled to guarantee sufficient PE numbers.

Exercise 10: Define a function `spawnTree` to instantiate process abstractions given in a binary tree structure together with their inputs:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

spawnTree :: (Trans a, Trans b) ⇒
             Tree (Process a b, a) → Tree b
```

9 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 [39, 13]. Details on the parallel runtime system and Eden's concept of implementation can best be found in [8, 10, 4]. The technique of layered parallel runtime environments has been further developed and generalised by Berthold, Loidl and Al Zain [3, 12]. The Eden trace viewer tool EdenTV is available on Eden's web site. A short introductory description is given in [11]. Another tool for analysing the behaviour of Eden programs has been developed by de la Encina, Llana, Rubio and Hidalgo-Herreo [21, 22, 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 book by Gorlatch and Rabhi [38,

54]. Several parallel map implementations have been discussed and analysed in [33]. An Eden implementation of the large-scale *map-and-reduce* programming model proposed by Google [18] has been investigated in [6, 4]. Hierarchical master-worker schemes with several layers of masters and submasters have been presented in [7]. A sophisticated distributed workpool has been presented in [19]. Definitions and applications of further specific skeletons can be found in the following papers: topology skeletons [9], adaptive skeletons [25], divide-and-conquer schemes [5, 36]. Special skeletons for computer algebra algorithms are developed with the goal to define the kernel of a computer algebra system in Eden [37, 35]. Meta-programming techniques have been investigated in [51]. An operational and a denotational semantics for Eden have been defined by Ortega-Mallén and Hidalgo-Herrero [28, 29, 27]. These semantics have been used to analyze Eden skeletons [31, 30]. A non-determinism analysis has been presented by Segura and Peña [44, 55].

10 Other Parallel Haskells (Related Work)

Several extensions of the non-strict functional language Haskell [26] for parallel programming are available. These approaches differ in the degree of explicitness when specifying parallelism and the control of parallel evaluations. The spectrum reaches from explicit low-level approaches where the programmer has to specify and to control parallel evaluations on a low level of abstraction to implicit high-level approaches where in the extreme the programmer does not have to bother about parallelism at all. Between the extremes there are approaches where the programmer has to specify parallelism explicitly but parallel execution is managed by sophisticated parallel runtime systems. It is a challenge to find the right balance between control and abstraction in parallel functional programming. The following enumeration sketches some parallel extensions of Haskell from explicit to implicit approaches:

Haskell plus MPI uses the foreign function interface (FFI) of Haskell to provide the MPI [43] functionality in Haskell [49]. It supports an SPMD style, i.e. the same program is started on several processing elements (PEs). The different instances can be distinguished using their MPI rank and may exchange serializable Haskell data structures via MPI send and receive routines.

The Par Monad [56] is a monad to express deterministic parallelism in Haskell. It provides a `fork` to create parallel processes and write-once mutable reference cells called `IVars` for exchanging data between processes. A skeleton-based programming style is advocated to abstract from the low-level basic constructs. It is notable that the Par monad is completely implemented as a Haskell library including a work-stealing scheduler written in Haskell.

Eden (the subject of 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. Eden has been designed for distributed memory systems but can equally well be used on multicore systems.

Glasgow parallel Haskell (GpH) [58] and **Multicore Haskell** [42] share the same language definition (basic combinators `par` and `pseq` and evaluation strategies) but differ in their implementations. While GpH with its parallel runtime system GUM can be executed on distributed memory systems, Multicore Haskell with its threaded runtime system is tailored to shared-memory multicore architectures. The language allows to mark expressions using the simple combinator `par` for parallel evaluation. These expressions are collected as *sparks* in a spark pool. The runtime system decides which sparks will be evaluated in parallel. This is out of control of the programmer. Moreover, access to local and remote data is automatically managed by the runtime system. Evaluation strategies [57, 41] abstract from low-level expression marking and allow to describe patterns for parallel behaviour on a higher level of abstraction.

Data Parallel Haskell [46] 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. This is the most implicit and easy-to-use approach, but restricted to the special case of data parallelism.

Note that we excluded from this overview approaches to concurrent programming like Concurrent Haskell [45] and distributed programming like Cloud Haskell [32] or HdpH [40]. Although not dedicated to parallel programming these languages can also be used for that purpose but on a rather low level of abstraction.

11 Conclusions

These lecture notes have given a comprehensive overview of the achievements and the current status of the Eden project with a focus on Eden's skeleton-based programming methodology. Eden extends Haskell with constructs for the explicit definition and creation of processes. Communication between these processes occurs via uni-directional one-to-one channels which will be established automatically on process creation between parent and child processes, but can also be explicitly created for direct data exchange between arbitrary processes. Eden provides an elaborated skeleton library with various parallel implementations of common computation schemes like map, map-reduce, or divide-and-conquer, as well as skeletons defining communication topologies and master-worker systems. Communication costs are crucial in distributed systems. Techniques like chunking, running processes offline and establishing direct communication channels using remote data or dynamic channels can be used to reduce communication costs substantially. Application programmers will typically find appropriate predefined skeletons for parallelising their Haskell programs, but also have the possibility to modify and adapt skeletons for their special requirements. The Eden project is ongoing. Current activities comprise the further development of the

Eden skeleton library as well as the investigation of further high-level parallel programming constructs.

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.

Special thanks go to Yolanda Ortega-Mallén, Oleg Lobachev, Mischa Dieterle, Thomas Horstmeyer, and the anonymous reviewer for their valuable comments on a preliminary version of this tutorial.

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*, Vol. 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. *Explicit and Implicit Parallel Functional Programming: Concepts and Implementation*. PhD thesis, Philipps-Universität Marburg, Germany, 2008.
5. 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.
6. J. Berthold, M. Dieterle, and R. Loogen. Implementing Parallel Google Map-Reduce in Eden. In *Europar'09*, LNCS 5704, pages 990–1002. Springer, 2009.
7. 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.
8. 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.
9. 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.

10. 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).
11. 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.
12. 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.
13. S. Breitinger. *Design and Implementation of the Parallel Functional Language Eden*. PhD thesis, Philipps-Universität Marburg, Germany, 1998.
14. S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98*, LNCS 1490, pages 318–334. Springer, 1998.
15. J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
16. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
17. A. de la Encina. *Formalizando el proceso de depuración en programación funcional paralela y perezosa*. PhD thesis, Universidad Complutense de Madrid (Spain), 2008. In Spanish.
18. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
19. M. Dieterle, J. Berthold, and R. Loogen. 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, Springer, 2010.
20. 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.
21. 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.
22. 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.
23. W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 25(1):112–115, 1978.
24. GHC: The Glasgow Haskell Compiler. Website `http://www.haskell.org/ghc`.
25. K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003.
26. Haskell: A non-strict functional programming language. Website. `http://www.haskell.org/`.
27. M. Hidalgo Herrero. *Semánticas Formales para un Lenguaje Funcional Paralelo*. PhD thesis, Universidad Complutense de Madrid (Spain), 2004. In Spanish.
28. M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

29. 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.
30. 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.
31. 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.
32. S. P. Jeff Epstein, Andrew P. Black. Towards Haskell in the cloud. In *Haskell '11: Proceedings of the 4th ACM symposium on Haskell*, pages 118–129. ACM, 2011.
33. 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.
34. U. Klusik, Y. Ortega-Mallén, and R. Peña Marí. Implementing Eden – or: Dreams Become Reality. In *IFL'98*, LNCS 1595, pages 103–119. Springer, 1999.
35. O. Lobachev. *Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms*. PhD thesis, Philipps-Universität Marburg, Germany, 2011.
36. O. Lobachev, J. Berthold, M. Dieterle, and R. Loogen. Parallel FFT using Eden Skeletons. In *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83, Springer, 2009.
37. 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.
38. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *[53]*, chapter 4, pages 95–128. Springer, 2003.
39. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
40. P. Maier, P. Trinder, and H.-W. Loidl. Implementing a High-level Distributed-Memory parallel Haskell in Haskell. In *IFL'11: 23rd Int. Workshop on the Implementation of Functional Languages*, LNCS. Springer, 2011. to appear.
41. 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*. ACM Press, 2010.
42. 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. ACM Press, 2009.
43. MPI: The Message-Passing Interface. Website. `http://www.open-mpi.org/`.
44. R. Peña and C. Segura. Non-determinism Analysis in a Parallel-Functional Language. In *Implementation of Functional Languages (IFL 2000)*, LNCS 1268. Springer, 2001.
45. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of POPL '96*, pages 295–308. ACM Press, 1996.
46. S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, 2008.
47. 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.

48. B. Pickenbrock. Developing a Multicore Implementation of Eden. Bachelor thesis, Philipps-Universität Marburg, 2011. in german.
49. B. Pope and D. Astapov. Haskell-mpi, Haskell bindings to the MPI library. `https://github.com/bjpop/haskell-mpi`, 2010.
50. S. Priebe. Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton. In *European Conference on Parallel Computing (Euro-Par) 2006*, LNCS 4128, 2006.
51. S. Priebe. *Structured Generic Programming in Eden.* PhD thesis, Philipps-Universität Marburg, Germany, 2007.
52. PVM: Parallel Virtual Machine. Website. `http://www.epm.ornl.gov/pvm/`.
53. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer, 2003.
54. F. Rubio. *Programación Funcional Paralela Eficiente en Eden.* PhD thesis, Universidad Complutense de Madrid (Spain), 2001. In Spanish.
55. C. Segura. *Análisis de programas en lenguajes funcionales paralelos.* PhD thesis, Universidad Complutense de Madrid (Spain), 2001. In Spanish.
56. S. P. Simon Marlow, Ryan Newton. A monad for deterministic parallelism. In *Haskell '11: Proceedings of the 4th ACM symposium on Haskell*, pages 71–82. ACM, 2011.
57. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.
58. 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*, 1996.

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.

Typical command lines for compiling, running and analysing the simple program for computing π shown in Figure 4 are e.g.

```
prompt> ghc -parmpi --make -O2 -eventlog pi.hs
prompt> pi 1000000 +RTS -N8 -ls
prompt> edentv loogen=pi_1000000_+RTS_-N8_-ls.parevents
```

Because of the option `-parmpi` code for the MPI version of the parallel runtime system (PRTS) is produced. The option `-eventlog` enables the code to produce traces at runtime. The code is then run with input parameter 1000000. The runtime system options after `+RTS` have the effect that the runtime system is started on 8 processing elements (option `-N8`) and that a trace file is produced (option `-ls`). Finally the trace viewer EdenTV (see Section A.3) is started to visualise and analyse the produced trace file.

A.1 Compile Time Options

To compile Eden programs with parallel support one has to use the options `-parpvm` to use PVM [52] or `-parmpi` to use MPI [43] as middleware. The option `-eventlog` allows for the 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.2 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.

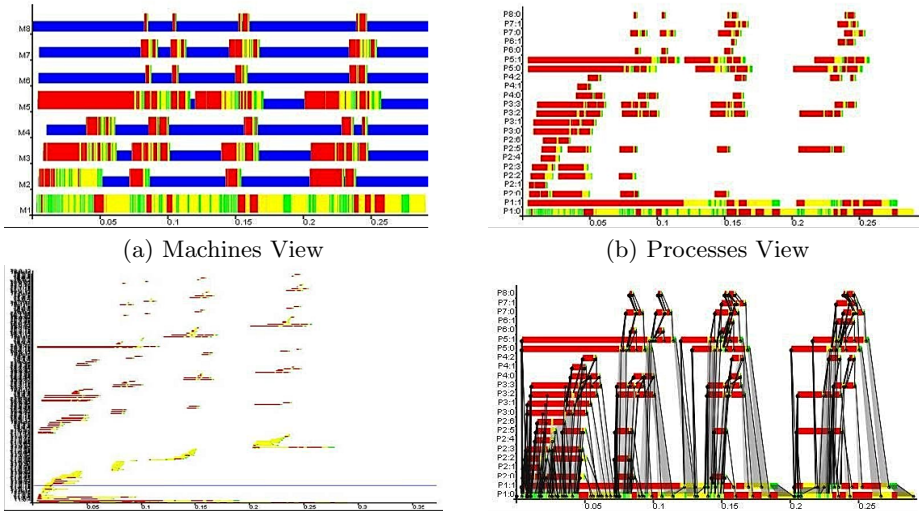
RTS option	effect	default
`-N<n>`	set number of PEs	number of PVM/MPI nodes
`-MPI@<file>`	specify MPI hostfile	`mpihosts`
`-qQ<n>`	set buffer size for messages	32K
`-ls`	enable event logging[3]	
`-qrnd`	random process placement	round-robin placement

A.3 EdenTV: The Eden Trace Viewer

The Eden trace viewer tool (EdenTV) [11] provides a *post-mortem analysis* of program executions on the level of the computational units of the parallel runtime system (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 or threads are represented by horizontal bars, respectively, with time on the x-axis.

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



(a) Machines View (b) Processes View

(c) Thread View (d) Processes with Message Overlay

Fig. 46. Examples of EdenTV diagrams

- 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 46 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 46 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 46(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 46(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 of EdenTV:
`http://www.mathematik.uni-marburg.de/~eden/?content=trace_main&navi=trace`

B Auxiliary Functions

This section contains the definitions of the auxiliary functions which have been used in the examples of this tutorial. These pure Haskell functions are provided in the Eden module `Control.Parallel.Eden.EdenSkel.Auxiliary`.

B.1 Unshuffle and Shuffle

The function `unshuffle :: Int → [a] → [[a]]` distributes the input list in a round robin manner into as many sublists as the first parameter determines.

```
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 inverse function `shuffle :: [[a]] → [a]` shuffles the given list of lists into the output list.

```
shuffle :: [[a]]  -- ^ sublists
           → [a]  -- ^ shuffled sublists
shuffle = concat ∘ transpose
```

Note that the function `transpose` is predefined in the standard library `Data.List`. The Haskell prelude function `concat :: [[a]] → [a]` simply concatenates all lists of the given list of lists.

The function `unshuffle` has the advantage that the result lists grow uniformly. Consequently, the function works *incrementally* in the sense that it produces

values on all output lists even if the input list is not completely available or an infinite stream. In the same way, `shuffle` is able to produce output even if the input lists are incomplete or infinite.

B.2 SplitIntoN and Chunk

The function `splitIntoN :: Int → [a] → [[a]]` distributes the input list blockwise into as many sublists as the first parameter determines. The lengths of the output lists differ by at most one. This property is achieved by using the following function `bresenham` which follows an idea from the Bresenham algorithm from computer graphics [15]. The function `bresenham` computes takes two integer parameters `n` and `p` and computes $[i_1, ..., i_p]$ such that $i_1 + ... + i_p = n$ and $|i_j - i_k| \leq 1$ for all $1 \leq j, k \leq n$.

```
bresenham :: Int      -- ^n
             → Int    -- ^p
             → [Int] -- ^[i1,...,ip]
bresenham n p = take p (bresenham1 n)
  where
    bresenham1 m = (m `div` p) : bresenham1 ((m `mod` p)+ n)

splitIntoN :: Int       -- ^ number of blocks
              → [a]     -- ^ list to be split
              → [[a]] -- ^ list of blocks
splitIntoN n xs = f bh xs
  where bh = bresenham (length xs) n
            f [] [] = []
            f [] _  = error "some␣elements␣left␣over"
            f (t:ts) xs = hs : (f ts rest)
                  where (hs,rest) = splitAt t xs
```

The Haskell prelude function `splitAt :: Int → [a] → ([a],[a])` splits a list into a prefix of the length determined by its first parameter and the rest list.

Note that `splitIntoN` works only for finite lists. Moreover, it does not work incrementally, i.e. the whole input list must be available before any output will be produced.

While `splitIntoN` divides a list into the given number of sublists, the following function `chunk` decomposes a list into sublists of the size given as first parameter. All sublists except of the last one have the given size.

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

In contrast to `splitIntoN`, `chunk` works incrementally and can also be applied to incomplete or infinite lists.

Note that the inverse function to `splitIntoN` and to `chunk` is the Haskell prelude function `concat :: [[a]] → [a]`.

B.3 Distribute and OrderBy

The functions `distribute` and `orderBy` are used in the definition of the master-worker skeleton (see Section 6.2). The function `distribute` distributes a task list into several task lists for the worker processes in the order determined by a stream of worker id's which are the workers' requests for new tasks.

```
distribute :: Int      -- ^ number of workers
              → [Int]  -- ^ request stream  with worker IDs
              → [t]    -- ^ task list
              → [[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 _      _      _  = []
```

The function `orderBy` combines the worker results in the order determined by a stream of worker id's.

```
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
```