Skeletons for Recursively Unfolding Process Topologies

Jost Berthold and Rita Loogen[a]

[a]Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Hans-Meerwein-Straße, D-35032 Marburg, Germany.

We discuss two different patterns for the generation of process topologies. All processes are either created by a single root process, or the topology unfolds recursively along a spanning tree. An obvious drawback of the first approach is the bottleneck in the root process, which becomes more serious when the number of processes increases. Difficulties of the second approach are the appropriate installation of the communication channels and the correct placement of processes on processors. We compare the generation policies for rings and toroids and analyse the impact of the topology generation on the runtimes of programs.

1. Introduction

Skeletons [3] provide commonly used patterns of parallel evaluation and simplify the development of parallel programs, because they can be used as complete building blocks in a given application context. In many skeletal parallel programming approaches a fixed number of skeletons is provided, together with optimised implementations for special target architectures, see e.g. [2] and [4] or, more recently, with implementations on top of communication libraries like MPI, see e.g. [5], [1].

In functional languages like Haskell or ML, skeletons can simply be specified as polymorphic higher-order functions. If parallelism or concurrency can be expressed, skeletons can even be *implemented* in the language itself. This is possible in languages like GpH (Glasgow parallel Haskell) [11], Concurrent Clean [8], Eden [7], or Concurrent ML [10]. Describing both the functional specification and the parallel implementation of a skeleton in the same language context has several advantages. Firstly, it constitutes a good basis for formal reasoning and correctness proofs. Secondly, it provides much flexibility, as skeleton implementations can easily be adapted to special cases, and if necessary, new skeletons can even be introduced by the programmer himself.

Topology skeletons define process systems with an underlying communication topology, e.g. pipes, rings, grids, hypercubes etc. In this paper, we discuss two different ways to generate process topologies. The simplest method is to create all processes and their interconnecting channels by a single root process. Alternatively, the topology can be unfolded recursively, each process creating its successor processes with respect to a generational spanning tree of the topology. In the first approach, the root process may become a bottleneck when the number of processes increases. Such a bottleneck is avoided in the second approach at the price of a more sophisticated installation of the communication channels and the need for explicit placement of processes on processors. In this paper, we describe and compare the two approaches for rings and toroids. We implement the skeletons in our parallel functional language Eden and briefly analyse the impact of the topology generation on the runtimes of programs.

2. A Short View on Eden

Eden [7], a parallel extension of the functional language Haskell, embeds functions into *process abstractions* with the special function `process` and explicitly *instantiates* (i.e. runs) them on remote processors using the operator `( # )`. Processes are distinguished from functions by their operational property to be executed remotely, while their denotational meaning remains unchanged as compared to the underlying function.

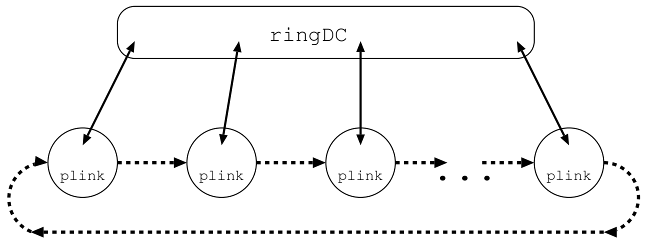
```
process :: (Trans a, Trans b) => (a -> b)    -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

For a given function `f` and an expression `e`, evaluation of the expression `(process f # e)` leads to the creation of a new (remote) process which evaluates the function application `f e`. The argument `e` is evaluated to normal form by the creator or parent process, i.e. the process evaluating the process instantiation. The result value of `e` is transmitted from the parent to the child and the child output `f e`, which will be completely evaluated by the child process, is transmitted from the child to the parent via implicit communication channels installed during process creation. The type class `Trans` provides implicitly used functions for these transmissions. Tuples are transmitted component-wise by independent concurrent threads, and lists are transmitted as streams, element by element.

Eden provides the dynamic creation of channels which allows to establish direct channel connections between arbitrary processes. An Eden process may explicitly generate a new *dynamic reply channel* and pass the channel's name to another process. The receiving process may then either use the name to return some information directly to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Eden introduces a unary type constructor `ChanName` for the names of dynamically created channels. It provides two operators to generate and use channel names.

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

Evaluating an expression `new (\ ch_name ch_vals -> e)` has the effect that a new channel name `ch_name` is declared as reference to the new input channel via which the values `ch_vals` will eventually be received in the future. The scope of both is the body expression `e`, which is the result of the whole expression. The channel name must be sent to another process to establish the direct communication. A process can reply through a channel name `ch_name` by evaluating an expression `(parfill ch_name e1 e2)`. Before `e2` is evaluated, a new concurrent thread for the evaluation of `e1` is generated, whose normal form result is transmitted via the dynamic channel. The result of the overall expression is `e2`. The generation of the new thread is a side effect. Its execution continues independently from the evaluation of `e2`.
Figure 1 shows the definition of a ring skeleton in Eden. All processes are created by the process evaluating the function `ring` and communicate in a unidirectional way. The number of ring processes is given by the first parameter. Parameter functions `split` and `combine` specify how to distribute the input to the ring processes and how to combine the results of the ring processes to the overall result. The node function `f` determines the behaviour of each ring process. It is applied to the corresponding part of the `input` and the value received from its ring predecessor, yielding an element of the list `toParent` which is part of the overall result, and a value that is passed to its ring successor. Note that the ring is closed by using the list of ring outputs `ringOuts` rotated by one position to the right by `rightrotate` as inputs `ringIns` in the node function applications. The Haskell function `zip` converts a pair of lists element by element into a list of pairs and `unzip` does the reverse. The `mzip` function corresponds to the `zip` function except that a lazy pattern is used to match the second argument. This is necessary, because the second argument of `mzip` is the recursively defined ring input. Laziness is essential in this example - a corresponding definition is not possible in an eager language.



```
ring ::  (Trans ri,Trans ro,Trans r) =>
         Int                        -- ring size
         -> (Int -> i -> [ri])      -- input split function
         -> ([ro] -> o)             -- output combine function
         -> ((ri,r) -> (ro,r))      -- ring process mapping
         -> i -> o                  -- input-output mapping
ring n split combine f input = combine toParent
  where
    (toParent,ringOuts) = unzip [plink f # inp | inp <- nodeInputs]
    inputs              = split n input
    nodeInputs          = mzip inputs ringIns
    ringIns             = rightRotate ringOuts
    rightRotate xs      = last xs : init xs

plink :: (Trans ri,Trans ro,Trans r) =>
         ((ri,r) -> (ro,r)) -> Process (ri,ChanName r) (ro,ChanName r)
plink f = process fun_link
  where fun_link (fromParent,nextChan) =  new (\ prevChan prev ->
              let (toParent,next) = f (fromParent,prev)
              in  parfill nextChan next (toParent,prevChan))
```

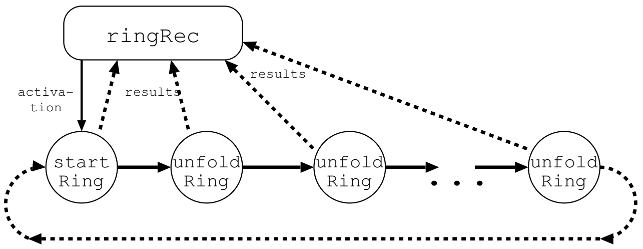
Figure 1. Eden Ring Skeleton

The function `plink` establishes direct channel connections between the ring processes. It embeds the node function `f` into a process which creates a new input channel `prevChan` that is passed to the neighbour ring process via the parent. The ring output `next` is sent via the received channel `nextChan`, while the ring input `prev` is received via its newly created input channel `prevChan`. The ring input/output from/to the parent is received and sent on static channel connections while the communication between ring processes occurs on dynamic reply channels. As all processes are created by a single parent process, the default round-robin placement policy of Eden is sufficient to guarantee an even distribution of processes on processors.

In the following section we will discuss alternative skeleton definitions where the topologies are recursively unfolded. For simplicity we restrict the discussion to rings and toroids (two-dimensional ring structures). The same techniques can be applied to higher-dimensional structures like hypergrids or hypercubes.

3. Recursively Unfolding Rings and Toroids

The single-source creation of process systems may lead to a serious bottleneck in the creator process when the number of processes increases. For this reason, we investigate the recursive unfolding of process topologies. We start with the discussion of a one-dimensional unidirectional ring skeleton



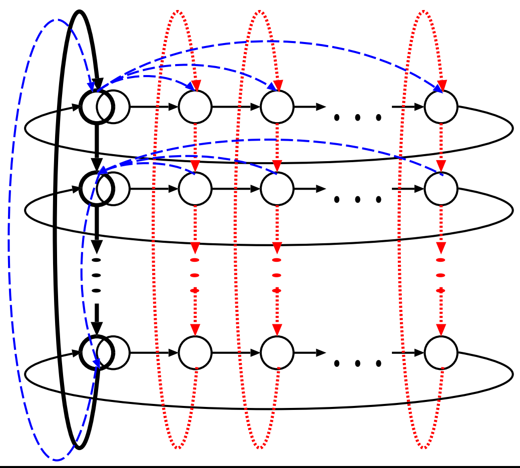
```
ringRec n split combine f input = plist `seq` combine toParent
  where (pChans, toParent)  = createChans n  -- result channels
        plist = (process (startRing  f (split n input))) # pChans

startRing :: (Trans ri, Trans ro, Trans r) =>
             ((ri,r) -> (ro,r)) -> [ri] -> [ChanName ro]  -> ()
startRing  f (i:is) (c:cs)
   = new (\ firstChan firstIns ->  -- channel to close the ring
          let (result,ringOut)  =  f (i,firstIns)
              recCall   = unfoldRing firstChan f is
              next      = (process recCall) # (cs,ringOut)
          in  parfill c result next )

unfoldRing :: (Trans ri, Trans ro, Trans r) =>
              ChanName r -> ((ri,r) -> (ro,r)) -> [ri] ->
              ([ChanName ro],r) -> ()
unfoldRing   firstChan f (i:is) ((c:cs),ringIn)
   = parfill c result next
   where  (result, ringOut) = f (i,ringIn)
          recCall           = unfoldRing  firstChan f is
          next | null is    = parfill firstChan ringOut ()
               | otherwise  = (process recCall) # (cs,ringOut)

createChans :: Trans a => Int -> ([ChanName a],[a])
createChans 0 = ([],[])
createChans n = new (\chX valX -> let (cs,xs) = createChans (n-1)
                                  in (chX:cs,valX:xs))
```

Figure 2. Recursively Unfolding Ring Skeleton

where each process but the last creates its successor process in the ring. Subsequently, we show how higher-dimensional structures can be built on this.

3.1. Rings

Figure 2 shows an alternative definition of the ring skeleton which has the same type interface as the original one of Figure 1. The input to the ring processes is now passed as a parameter and thus will be communicated together with the process instantiation, while the output of the ring processes is returned to the originator process via initially created dynamic reply channels `pChans` which are communicated to the ring processes. The static output of the ring processes is merely the unit value (). The explicit demand on the unit value `plist` by `plist 'seq'` leads to the immediate creation



Recursive Toroid Creation
Solid lines show the underlying ring skeletons (thick lines indicate the first column ring). Dotted lines indicate the vertical connections created using dynamic channels.

Dashed lines show how the dynamic reply channels from row 2 are passed through the ring connection to row 1, which sends on these channels.

Figure 3. Creation scheme of a torus topology using ring skeletons

of the ring processes when the ring skeleton is called. The first process evaluates the `startRing` function. It creates a dynamic reply channel which is passed through the sequence of ring processes and will be used by the last process to close the ring connection. It is assumed that the number of ring processes is at least two. Thus, the functions `startring` and `unfoldRing` are never called with an empty input list. The input from the parent process is passed through the sequence of ring processes where each ring process takes its part of the input and passes the rest list to its successor process.

The roles of static and dynamic channel connections are exchanged in the two versions of ring skeleton definitions. The previously static output connections to the parent are now modelled by dynamic reply channels while the previously dynamic ring connections can now be realised as static connections, except that the connection from the last to the first ring process is still implemented by a dynamic reply channel.

Experiments with application programs using the ring skeleton show that the recursive ring creation is slightly advantageous as the number of ring processes increases. For a small number of processes there is almost no impact on the runtimes of programs. The number of messages sent and received by the parent process is clearly reduced while the overall amount of messages remains almost the same.

3.2. Torus

The Eden torus skeleton defined in [6] creates all processes by a single process and establishes dynamic interconnection channels between them. In the following, we redefine this skeleton by using the previously defined ring skeletons, as the torus is nothing but a two-dimensional grid with ring connections in both dimensions. Figure 3 depicts the generation scheme for the torus topology used in our redefinition. The first column and all rows are created as unidirectional rings. The other column rings must be installed using dynamic channels.

Figure 4 shows the core of the recursively unfolding torus skeleton. Function `toroideRec` describes the toroid by its dimensions (no. of rows and columns) and the functionality of each node. To place all processes on different processor elements, the first column of the torus structure is created with a variant `ringP` of the recursively unfolding ring skeleton, which allows for placing ring processes with a constant stride. To place processes row by row, the first column is placed with stride `dim2`, i.e. the length of the rows.

```
toroideRec :: (Trans input, Trans output, Trans horiz, Trans vert) =>
  Int -> Int ->                                  -- dimensions
  ((input,horiz,vert) -> (output,horiz,vert)) -> -- node function
  [[input]] -> [[output]]                        -- resulting mapping
toroideRec dim1 dim2 f rows
  = rnf outChans ‘seq‘ start_it ‘seq‘   -- force channel & ring creation
    list2matrix dim2 outs               -- re-structure output
  where (outChans,outs) = createChans (dim1*dim2)
        ringInput = (list2matrix dim2 outChans, rows)
        -- creating first column ring
        start_it  = ringP dim1 dim2 (\_ -> uncurry zip ) spine
                          (gridlineR dim1 dim2 f) ringInput

-- ring function for 1st column ring
gridRow :: (Trans i, Trans o, Trans h, Trans v) =>
  Int -> Int ->                -- dimensions
  ((i,h,v) -> (o,h,v)) ->      -- node function
  (([ChanName o], [i]), [[ChanName v]]) -> ((), ([[ChanName v]]))
gridRow dim1 dim2 f ((ocs, row), allnextRowChans) =
  let (cChanNamevs, rowChans) = createChans dim2
      -- creating row ring
      start  = startRingDI staticIn (gridNode f) dummyCs mynextRowChans
      staticIn = mzip3 row ocs cChanNamevs
      mynextRowChans = allnextRowChans!!(dim1-2)
      (dummyCs, _ ) = createChans dim2
  in  rnf cChanNamevs ‘seq‘ rnf dummyCs ‘seq‘ start ‘seq‘
      ((), rowChans:take (dim1-2) allnextRowChans)

-- ring function for row rings
gridNode :: (Trans i, Trans o, Trans h, Trans v) =>
   ((i,h,v) -> (o,h,v)) ->
   ((i,ChanName o, ChanName (ChanName v)),ChanName v,h) -> ((),h)
gridNode f ((a,cResult,cv),cToBottom,fromLeft) =
   new ( \  cFromAbove fromAbove   ->
      let  (out,toRight,toBottom) = f   (a,fromLeft,fromAbove)
      in  parfill cv cFromAbove       -- send vertical input channel
          (parfill cResult out        -- send result for parent
          (parfill cToBottom toBottom -- send data on column ring
          ((), toRight)) ))           -- result and data on row ring
```

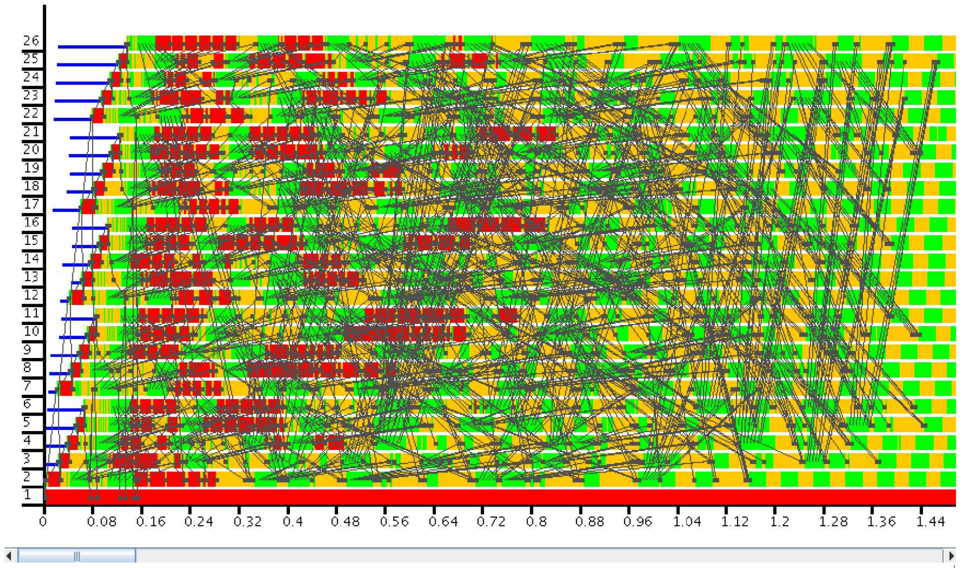
Figure 4. Core of Recursively Unfolding Torus Skeleton

The ring function `gridRow` for the first column ring creates a ring for each row. We do not use the normal interface of the `ring` skeleton but the internal startring function, because we want to embed the column processes into the row rings. A subtlety of the inner rings is the circular dependency of their dynamic input, i.e. the dynamic channels to establish the additional column rings. It is necessary to use a variant `startRingDI` which decouples static input, which is available

Multiplication of dense random 1000×1000 matrices with

Recursive Toroid

Overall Runtime: $12.1 sec.$



Multiplication of dense random 1000×1000 matrices with

Single-Source Toroid

Overall Runtime: $12.4 sec.$

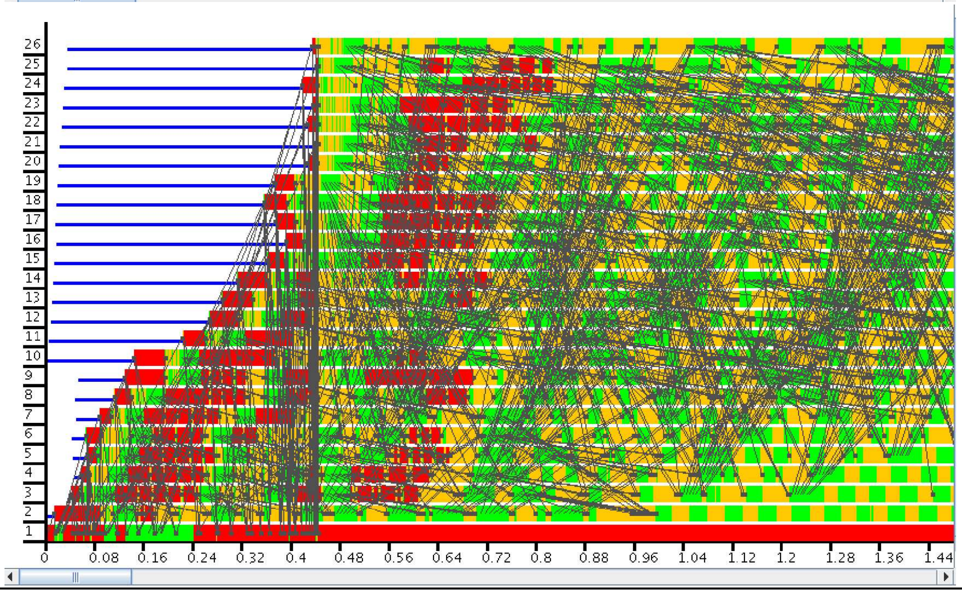


Figure 5. Start Phase of Matrix Multiplication Traces Using Toroid Skeletons, on 26 Processors

at instantiation time, and dynamic input, which will only not be produced after process instantiation. Otherwise, the inner rings would immediately deadlock on process instantiation. Each row ring process returns a channel name for its vertical input, which must be collected and passed to the previous row through the first column ring (as indicated in Figure 3 for the second row).

Measurements with a toroid-based matrix multiplication algorithm (by Gentleman, see [9]) show that runtimes are slightly better for the recursive version, due to a distributed startup sequence. Figure 5 shows traces of the start phase of the matrix multiplication program executed on 26 nodes of a Beowulf cluster, using either the original (single-source) or the recursive skeleton. Processors (resp. processes[1]) are shown as horizontal bars with colour-coded segments for their actions. We distinguish between the states blocked (red – dark grey), runnable (yellow – bright grey) and running (green – middle grey). Idle processors are shown as a smaller bar, and messages between processes are indicated by grey lines.

[1]Because of explicit placement, every processor executes exactly one process. So we identify nodes and processes.

While runtime is only slightly improved, the traces show the expected improvement in startup: process creation is carried out by different processors in a hierarchical fashion in the recursive skeleton implementation. One can observe how the first column unfolds, starting at processor 2 with stride 5, and how each of these processes unrolls one row. Process creation takes about 0.15 sec. in this version, whereas the single-source version below needs 0.4 sec. until all processes start to work (explaining the difference in runtime).

The improvement in startup pays especially for skeletons with a big number of processes. In any case, it substantially reduces the network traffic. The program investigated here already includes the input matrices in the process abstraction instead of communicating these big data structures via channels (which would be more time-consuming). However, the parent process in the single-source version has to send the channel names to each toroid process, which needs 125 messages. The parent process in the recursive version only sends 2 messages – creation and input to the `startring` process of the first column ring.

4. Conclusions and Future Work

The recursive unfolding of rings and toroids spreads the process creation overhead over several processes, thereby avoiding an eventual bottleneck in the originator process. Although the specification and implementation of the recursive unfolding is more sophisticated, case studies have proved that the effort definitely pays off. An interesting aspect which we want to elaborate further is the creation of higher-dimensional topologies by an appropriate nesting of lower-dimensional ones. Currently we have only shown how to define a recursive two-dimensional toroid skeleton using one-dimensional ring skeletons.

References

[1] M. Cole A. Benoit. eSkel – The Edinburgh Skeleton Library, University of Edinburgh 2002. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

[4] J. Darlington, Y. Guo, and H.W. To. Structured Parallel Programming: Theory meets Practice. In *Research Directions in Computer Science*. Cambridge University Press, 1996.

[5] H. Kuchen. The Münster Skeleton Library Muesli, University of Münster 2002. `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/`.

[6] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

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

[8] R. Plasmeijer, M. van Eekelen, M. Pil, and P. Serrarens. Parallel and Distributed Programming in Concurrent Clean. In *Research Directions in Parallel Functional Programming*, page 323ff. Springer, 1999.

[9] M.J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.

[10] J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, August 1999.

[11] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a portable implementation of Haskell. In *IFL'95 — Implementation of Functional Languages*, 1995.