THE IMPACT OF DYNAMIC CHANNELS ON FUNCTIONAL TOPOLOGY SKELETONS

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

ABSTRACT
Parallel functional programs with implicit communication often generate purely hierarchical communication topologies during execution: communication only happens between parent and child processes. Messages between siblings must be passed via the parent. This causes inefficiencies that can be avoided by enabling direct communication between arbitrary processes. The Eden parallel functional language provides *dynamic channels* to implement arbitrary communication topologies. This paper analyses the impact of dynamic channels on Eden's *topology skeletons*, i.e. skeletons which define process topologies such as rings, toroids, or hypercubes. We compare topology skeletons with and without dynamic channels with respect to the number of messages. Our case studies confirm that dynamic channels usually decrease the number of messages by up to 50% and can reduce runtime by up to 50%. Detailed analysis of Eden TV (trace viewer) execution profiles reveals the reasons for these substantial runtime gains.

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. Skeletons are often provided as special language constructs or templates, and the creation of new skeletons is considered as a system programming task or as a compiler construction task [5,12]. Therefore, many systems offer a closed collection of skeletons which the application programmer can use, but without the possibility of creating new ones, so that adding a new skeleton usually implies a considerable effort.

In a functional language like Haskell or ML, a skeleton can be specified as a polymorphic higher-order function. In parallel functional languages like GpH (Glasgow parallel Haskell) [24], Concurrent Clean [17], Eden [11], para-functional programming [8], or Concurrent ML [20], skeletons can be *implemented* in the language itself. 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.

In this paper, we consider the functional specification and implementation of topology skeletons and show how to improve their implementation substantially by using dynamically established communication connections which we call dynamic channels for short. Topology skeletons define parallel evaluation schemes with an underlying communication topology like a ring, a torus or a hypercube. Many parallel algorithms [9] rely on such underlying communication topologies. As any skeleton, topology skeletons can easily be expressed in a functional language. A

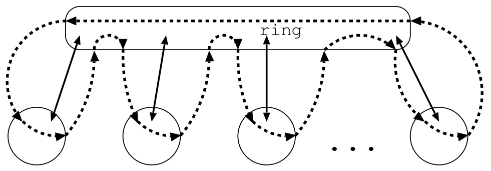


Figure 1: Hierarchical Ring Skeleton

simple ring can e.g. be defined in Haskell as follows:

```
ring :: ((i,[r]) -> (o,[r]))  -- ring process mapping
         -> [i] -> [o]        -- input-output mapping

ring f inputs = outputs
  where  (outputs, ringOuts) = unzip [ f inp | inp <- nodeInputs]
         nodeInputs          = mzip inputs ringIns
         ringIns             = rightRotate ringOuts
         rightRotate xs      = last xs : init xs
```

The function `ring` takes a node function `f` and a list `inputs` whose length determines the dimension of the ring. The node function `f` is applied to each element of the list `inputs` and a list of values which is received from its left ring neighbour. It yields an element of the list `outputs` which is the overall result and a list of values passed to its right ring neighbour. Note that the ring is closed by using the list of output lists `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[a].

A *parallel* ring can be obtained by evaluating each application of the node function `f` in parallel. Implicit and semi-explicit lazy parallel functional languages like GpH (Glasgow parallel Haskell) [24], Concurrent Clean [17], or Eden [11] introduce parallelism by primitives to spawn subexpressions for parallel evaluation by a separate thread or process. Necessary arguments and the results will automatically be communicated by the parallel runtime system underlying the implementation of such languages.

Unfortunately, the one-by-one pattern of parallel thread or process creation induces a purely hierarchical communication topology. Figure 1 shows the ring topology resulting from the above definition, if the node function applications are spawned for parallel evaluation. The solid arrows show the connections between the node processes and the parent process `ring` via which the inputs and outputs are

[a]Laziness is essential in this example - a corresponding definition is not possible in an eager language.

passed. The dashed lines show ring connections that are also established between the nodes and the parent which will pass the data as indicated. This unnecessarily increases the number of messages and causes a bottleneck in the parent process.

The parallel functional language Eden [11] offers means to create arbitrary channels between processes to achieve better performance by eliminating such communication bottlenecks. The expressiveness of Eden for the definition of arbitrary process topologies has been emphasised by [6], but in a purely conceptual manner, without addressing any performance issues. The use of dynamic channels has been investigated in [15], explaining how non-hierarchical process topologies can systematically be developed using dynamic reply channels. In the current paper, we focus on a detailed analysis of topology skeletons in Eden using trace information collected during parallel program executions. We compare topology skeletons defined with and without dynamic channels and analyse the benefits and overhead induced by the use of dynamic channels. Our case studies show that dynamic channels lead to substantial runtime improvements due to a reduction of message traffic and the avoidance of communication bottlenecks in parent processes. A new trace viewer tool [21] is used to visualise the interaction of all machines, processes and threads, and allows us to spot inefficiencies in programs in a post-mortem analysis.

2. Dynamic Channels in Eden

Eden [11], 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 some argument 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 locally by the creator or parent process, i.e. the process evaluating the process instantiation. The value of `e` is transmitted from the parent to the child and the child output `f e` is transmitted from the child to the parent via implicit communication channels installed during process creation. The type class[b] `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.

Example 1 Ring
In the following, we slightly refine the ring specification of the introduction and discuss two definitions of a process ring skeleton in Eden (see Figure 2): The number of ring processes is no longer deduced from the length of the input list, but given as a parameter. Input `split` and output `combine` functions generalise the skeleton and allow the input to be of arbitrary type `i` instead of the list type `[i]`.

[b]In Haskell, type classes provide a structured way to define overloaded functions.

```
ring, ringDC :: (Trans i,Trans o,Trans r) =>
         Int                       -- ring size
         -> (Int -> i -> [i])      -- input split function
         -> ([o] -> o)             -- output combine function
         -> ((i,[r]) -> (o,[r]))   -- ring process mapping
         -> i -> o                 -- input-output mapping

ring n split combine f input = combine toParent
  where
   (toParent,ringOuts) = unzip [process f # inp | inp <- nodeInputs]
   ...
```

Figure 2: Type of Eden ring skeletons and definition of static ring

The static `ring` skeleton has been obtained by replacing the function application `(f inp)` with the process instantiation `((process f) # inp)`. It uses only hierarchical interprocess connections and produces the topology shown in Figure 1 with the problems explained in the introduction. The non-hierarchical skeleton `ringDC` will be defined in Example 2 using dynamic channels. ◁

For this reason, 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*). Both possibilities exclude each other, and a runtime error will occur if a channel name is used more than once[c]. 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`. This is essential, because `e1` could yield a (possibly infinite) stream which would be communicated element by element. Or, `e1` could even (directly or indirectly) depend on the evaluation of `e2`.

[c]The current implementation detects the multiple use of channel names only for stream channels, but not for single-value channels.

```
-- ring process using dynamic channels
plink :: (Trans i,Trans o,Trans r) =>
          ((i,[r]) -> (o,[r])) -> Process (i,ChanName [r]) (o,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))
```

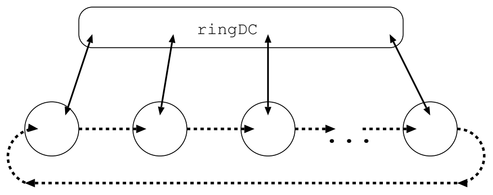


Figure 3: Ring Skeleton Using Dynamic Channels

Example 2   Ring, continued
In the version of Figure 3, the static ring connections are replaced by dynamic reply channels which have to be sent in the other direction to achieve the same information interchange. Therefore the above definition of `ring` is only modified in two places to define `ringDC`: The reply channels are rotated in the opposite direction — `rightRotate` is replaced by an appropriately defined `leftRotate`. More importantly, the process abstraction `process f` is replaced by a call to the function `plink f` (defined in Figure 3) which establishes the dynamic channel connections. The function `plink` 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. It receives a channel name `nextChan` to which the ring output `next` is sent, while the ring input `prev` is received via its newly created input channel. The mapping of the ring process remains as before, but the ring input/output is received and sent on dynamic channel connections instead of via the parent process. The obvious reduction in the amount of communications will be quantified in the following. ◁

3. Topology Skeletons

Topology skeletons define process systems with an underlying communication topology. In this section, we consider rings, toroids, and hypercubes. The concept of Eden dynamic channels allows to specify such topology skeletons exactly in the intended way, using direct connections between siblings. The following analysis quantifies the impact of dynamic channels in a theoretical manner. In the next section we will justify these considerations by measurements for chosen applications.

3.1. Analysis of the Ring Skeleton

The number of messages between all processes is compared for the ring skeletons of Section 2 with and without dynamic channels. In general, a process instantiation needs one system message from the parent for process creation. *Tuple inputs* *and outputs* of a process are evaluated componentwise by independent concurrent threads. Communicating input channels (destination of input data from the parent) needs $1+tsize(\texttt{i})$ administrative messages from the child (where $tsize(\texttt{a})$ denotes the number of components in type `a`). For simplicity, we only compute the amount of messages inside the system in the case where data items fit into single messages[d].

Let n denote the ring size, i_k and o_k be the number of input and output items for process k, and r_k the amount of data items which process k passes to its neighbour in the ring. Input data for the ring process is a pair and thus needs $3 = 1+2$ channel messages from each ring process. In case of the ring without dynamic channels, the total number of messages is:

$$Total_{noDC} = \overbrace{\sum_{k=1}^{n}(1 + i_k + r_k)}^{\text{sent by parent}} + \sum_{k=1}^{n} \overbrace{(3 + o_k + r_k)}^{\text{sent by child k}}$$

As seen in the introduction, ring data is communicated twice, via the parent. Thus the parent either sends or receives every message counted here!

Using dynamic channels, each ring process communicates one channel name via the parent (needs 2 messages) and communicates directly afterwards:

$$Total_{DC} = \overbrace{\sum_{k=1}^{n}(1 + i_k + 2)}^{\text{sent by parent}} + \sum_{k=1}^{n} \overbrace{(3 + o_k + 2 + r_k)}^{\text{sent by child k}}$$

It follows that using dynamic channels saves $(\sum_{k=1}^{n} r_k) - 4n$ messages, and we avoid the communication bottleneck in the parent process.

3.2. Toroid

As many algorithms in classical parallel computing are based on grid and toroid topologies, we extend our definition to the second dimension: a toroid is nothing more than a two-dimensional ring. In principle, the skeletons for those topologies work exactly the same way as the presented ring. In Figure 4, we only show the definition of the version which does not use dynamic channels. The version with dynamic channels can be derived as has been shown for the ring skeleton. It can also be found in [15]. The auxiliary functions `mzipWith3`, `mzip3` and `unzip3` are straightforward generalisations of the Haskell prelude functions `zipWith`, `zip` and `unzip` for triples. The prefix `m` marks versions with lazy argument patterns. Considering again the amount of messages, we get the following:

Let n denote the torus size (identical in the two dimensions), $i_{k,l}$ and $o_{k,l}$ be the number of input and output items for torus process (k,l). The amount of data items it passes through the torus connections shall be denoted $v_{k,l}$ and $h_{k,l}$ (vertical, horizontal). The input of a torus process is a triple and thus needs 4 administrative messages. If the skeleton does not use dynamic channels, the total number of

[d]When a data item does not fit into a single message due to the limited message size, it is split into several packages that are sent in separate partial messages.

```
toroid :: (Trans a,Trans b, Trans c, Trans d) =>
         Int -> Int            -- torus size (2 sizes)
          -> ((c,a,b)->(d,a,b)) -- node processes mapping
          -> [[c]] -> [[d]]     -- input-output mapping
toroid nf nc f toChildren =  outssToParent
 where
   (outssToParent,outssA,outssB) = unzip3 (map unzip3 outss)
   outss   = [[(process f) # outAB | outAB <- outs'] | outs' <- outss']
   outss'  = mzipWith3 mzip3 toChildren outssA' outssB'
   outssA' = mzipWith (:) nf (map last outssA) (map init outssA)
   outssB' = last outssB:init outssB
```

Figure 4: Static definition of toroid skeleton

messages is

$$Total_{noDC} = \overbrace{\sum_{k=1}^{n}\sum_{l=1}^{n}(1 + i_{k,l} + v_{k,l} + h_{k,l})}^{\text{sent by parent}} + \sum_{k=1}^{n}\sum_{l=1}^{n}\overbrace{(4 + o_{k,l} + v_{k,l} + h_{k,l})}^{\text{sent by child (k,l)}}$$

Again, the parent process is involved in every message counted here.

Using dynamic channels, torus processes exchange two channels via the parent (4 messages) and communicate directly afterwards; giving:

$$Total_{DC} = \overbrace{\sum_{k=1}^{n}\sum_{l=1}^{n}(1 + i_{k,l} + 4)}^{\text{sent by parent}} + \sum_{k=1}^{n}\sum_{l=1}^{n}\overbrace{(4 + o_{k,l} + 4 + v_{k,l} + h_{k,l})}^{\text{sent by child (k,l)}}$$

It follows that we save $(\sum_{k=1}^{n}\sum_{l=1}^{n}(v_{k,l} + h_{k,l})) - 8n^2$ messages.

3.3. Hypercube

The presented skeletons can be generalised even more to create 3-, 4-, and n-dimensional communication structures between a hyper-grid of processes. The ring skeleton is the one-dimensional instance of such a multi-dimension skeleton, and the well-known classical hypercube reduces to simply restricting the size to 2. We present a non-recursive hypercube definition where all processes are created by the parent process.

The nodes of the hypercube communicate with one partner in every dimension, thus the type of the node function includes this communication as a *list of streams* `[[r]]`, each stream sent by an independent concurrent thread. The `hypercube` skeleton creates all hypercube nodes and distributes the returned channels to the respective communication partners, where the call `(invertBit n d)` returns the communication partner of node `n` in dimension `d` by inverting bit position `d` in integer `n`. The process abstraction `hyperp` embeds the node function into a process abstraction, which expects and returns a list of channels, one for every dimension. Functions

```
hypercube :: (Trans i, Trans o, Trans r) =>
             Int                           -- dimension
             -> ((i,[[r]]) -> (o,[[r]])) -- node function
             -> [i] -> [o]  -- input/output (to/from all nodes)
hypercube dim nodefct inputs = outs
  where (outs,outChans) = unzip [ hyperp dim nodefct # proc_in
                                | proc_in <- proc_ins ]
        proc_ins =  zip inputs inChans
        inChans  = [ [ outChans!!(invertBit n d)!!d | d <- [0.. dim-1] ]
                     | n <- [0..2^dim -1] ]

hyperp :: (Trans i, Trans o, Trans r) =>
          Int                           -- dimension
          -> ((i,[[r]]) -> (o,[[r]])) -- node function
          -> Process (i, [ChanName [r]]) (o, [ChanName [r]])
hyperp dim nodefct =
        process (\ (input, toNeighbCs) ->
                    let (output, toNeighbs) = nodefct (input, fromNeighbs)
                        (fromNeighbCs, fromNeighbs) = createChans dim
                        sendOut = multifill toNeighbCs toNeighbs output
                    in (sendOut, fromNeighbCs) )

createChans :: Trans x => Int -> ([ChanName x], [x])
multifill   :: Trans x => [ChanName x] -> [x] -> b -> b
```

Figure 5: Definition of hypercube skeleton with dynamic channels

`createChans` and `multifill` are obvious generalisations of `new` and `parfill`, which work with lists of channels and values instead of single channels.

In contrast to the previous skeletons ring and toroid, the hypercube skeleton cannot use tuples with one component for every dimension. Dynamic reply channels are instead exchanged in form of a list. As an important consequence, a completely analogous skeleton with static communication channels *cannot be defined in Eden* (unless using *Channel Structures* [2], which are currently not implemented). By using dynamic reply channels, communication between neighbours is handled in separate streams and independent threads. When communicated via the parent, the *list of streams* between hypercube neighbours would be communicated as a *stream of lists*, sent by only one thread. Thus, a hypercube version with static connections would be applicable only for algorithms where neighbours interact in a strictly regular order. Otherwise, a deadlock may easily occur and it is impossible for the hypercube nodes to concurrently interact in different dimensions.

4. Case Studies

Throughout this section, we will show several runtime trace visualisations with the Eden Trace Viewer [21], a new tool for the post-mortem analysis of Eden program executions. Trace information is collected during runtime by an instrumented runtime environment, using the Pablo Trace Library [19] and Pablo's self-explanatory format SDDF (Self-Defining Data Format). Context information for all machines, processes and threads is gathered when loading a trace file, and the Eden Trace Viewer visualises the state transitions of these units of computation in different zoomable views, as well as the communication between processes.

4.1. Warshall-Algorithm Using a Ring

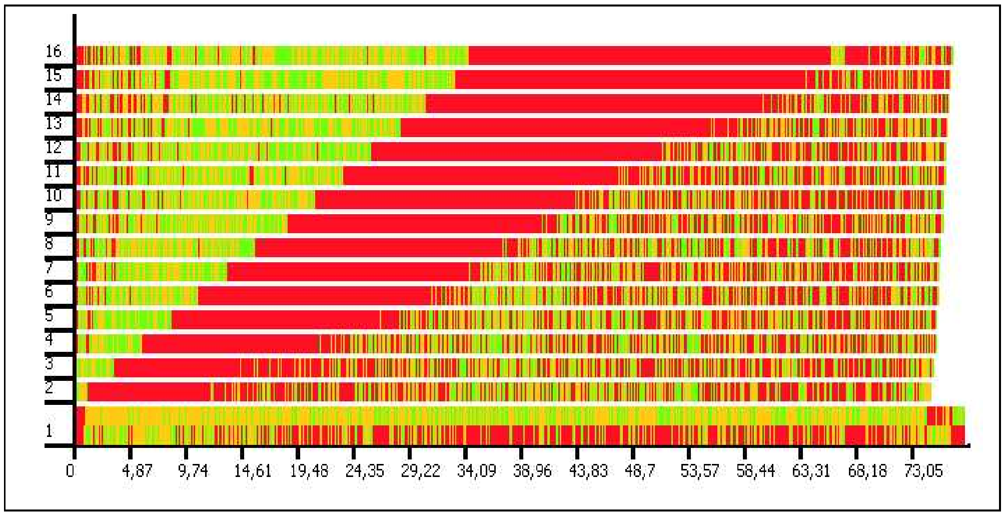
A simple use of ring structures is to pass global data around between nodes of a parallel computation. The program measured here is a parallel implementation of Warshall's algorithm to compute minimum distances for all nodes of a graph (adapted from [16]). The parallel processes communicate in a ring and each process evaluates rows of minimum distances for a subset of the graph nodes. Starting with the row for the first graph node, intermediate results are passed to the next ring process and flow through the whole ring for one round. While a row flows through the ring, each process updates its own rows by possible paths via the respective node, before eventually passing its own intermediate result to the ring neighbour. In a second phase, the remaining rows are received, and local rows are again updated accordingly to yield the final result.

The trace visualisations of Figure 6 and 7 show the *processes per machine* view of the Eden Trace Viewer for an execution of the warshall program on 16 processors of a Beowulf cluster, the input graph consisting of 500 nodes. Each process is represented by a horizontal bar with colour-coded segments for its actions. We distinguish between the process states blocked (red – dark grey), runnable (yellow – bright grey) and running (green – middle grey).

As expected from the analysis, the dynamic channel version uses about 50% of the messages of the static version (8676 instead of 16628) – network traffic is considerably reduced. Figures 8 and 9 show a zoom of the initial 5 seconds of both traces, with the messages exchanged between the processes drawn as grey lines between the horizontal bars. The figures show the high message traffic in the static version, where the parent process turns out to be a massive bottleneck. The direct ring communication can clearly be observed in the dynamic version.

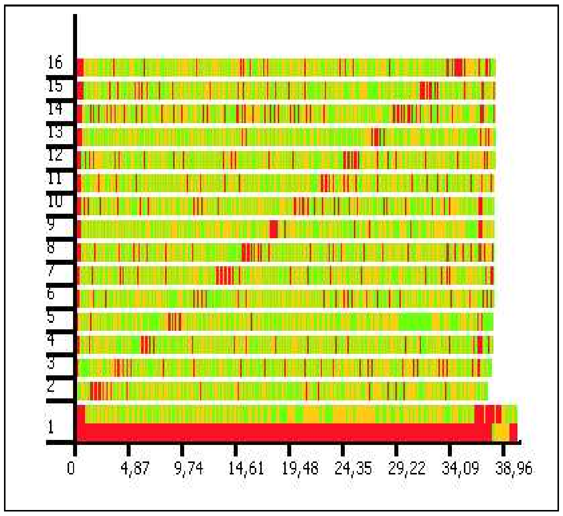
As the number of messages drops to 50%, the runtime decreases by approximately 50% as well, but the traces show that this is not a direct correspondence. The algorithm contains an inherent data dependency: each process must wait for the updated results of its predecessor, leading to a gap between the two phases passing through the ring. This is also observable in the dynamic version, but ring processes communicate in a distributed manner and show a good workload distribution with only short blocked or idle phases. In the static version, the time each ring process waits for data from the heavily-loaded parent is *accumulated* through the whole ring. The trace of this version shows a successively increasing wait phase while data flows through the ring.

Both versions scale well when the number of processors is increased. When moving from 8 to 24 processor elements, a relative speedup of 2.3 has been obtained.



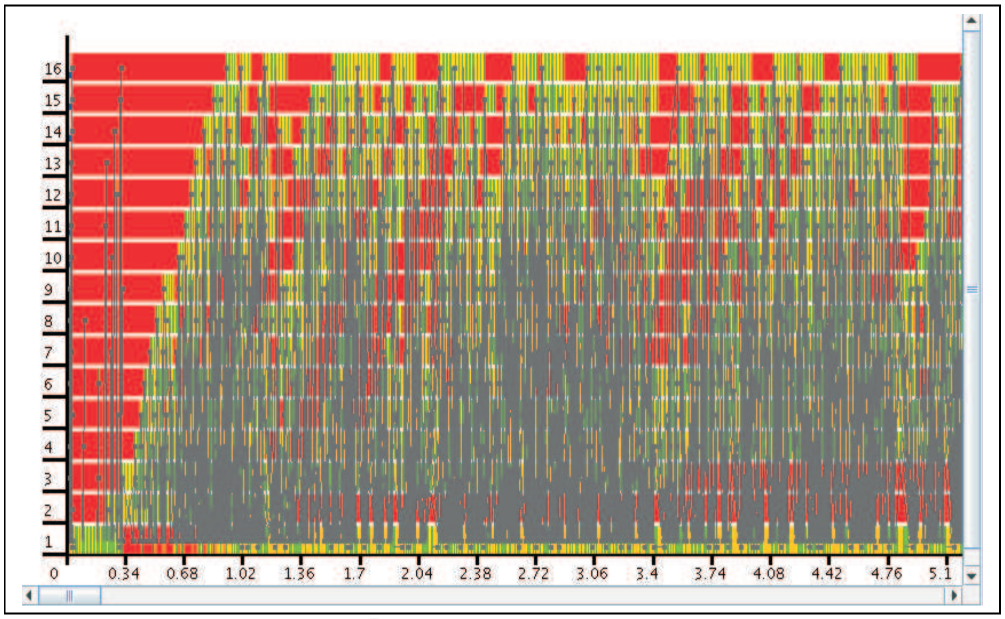
Runtime: 77.88 sec.

Figure 6: Warshall-Algorithm (500 node graph) using **static connections** for ring
(Beowulf Cluster, Heriot Watt University, Edinburgh, 16 machines)



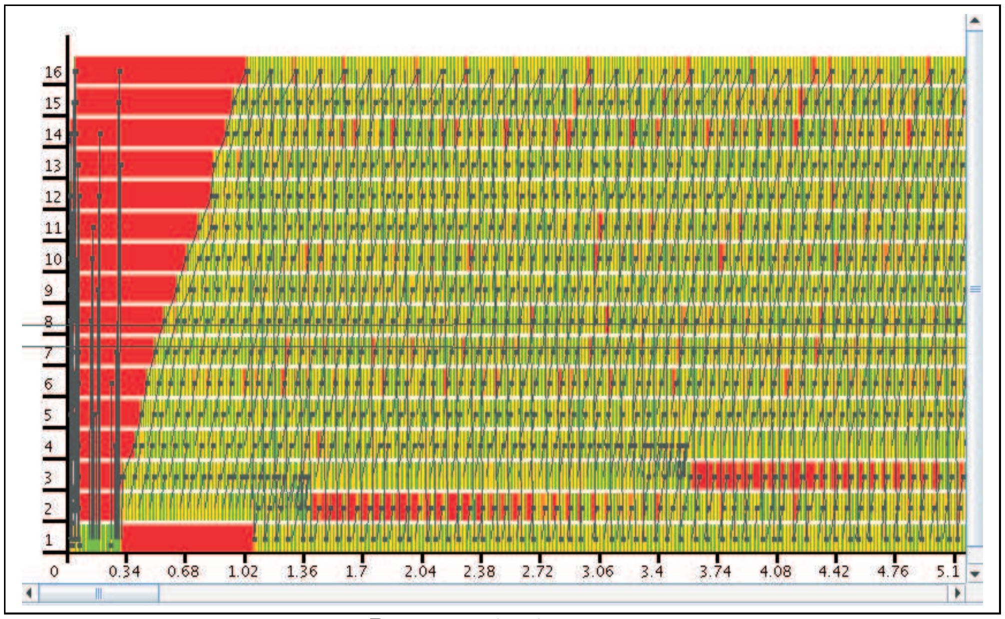
Runtime: 40.37 sec.

Figure 7: Warshall-Algorithm (500 node graph) using **dynamic channels** for ring
(Beowulf Cluster, Heriot Watt University, Edinburgh, 16 machines)



Runtime: 77.88 sec.

Figure 8: Message traffic during the initial 5 seconds of the Warshall-Algorithm (500 node graph) – using **static connections** for ring



Runtime: 40.37 sec.

Figure 9: Message traffic during the initial 5 seconds of the Warshall-Algorithm (500 node graph) – using **dynamic channels** for ring

4.2. Matrix multiplication in a torus

The toroid structure can be applied for a parallel matrix multiplication algorithm by Gentleman [18]. The result matrix is split into a square of submatrix blocks which are computed by parallel processes. The needed data to compute a result block are whole rows of the first and columns of the second matrix. These matrices are split into blocks of the same shape and the blocks passed through a torus process topology to avoid data duplication. The torus processes receive suitable input blocks after an initial rotation, and successively pass them to torus neighbours (in both dimensions). Every input block does one round through the torus, thus all processes of a block row eventually receive it. Each process accumulates a sum of products of the input blocks as the final result. The analyses in [10] have shown that this program, using dynamic reply channels, delivers good speedups on up to 36 processors, predictable by a suitable skeleton cost model.

The traces clearly show that the processes tend to communicate earlier than they start their computation. This is due to Eden's eager communication since every process can give away its block and all blocks it receives from its neighbours without any evaluation.

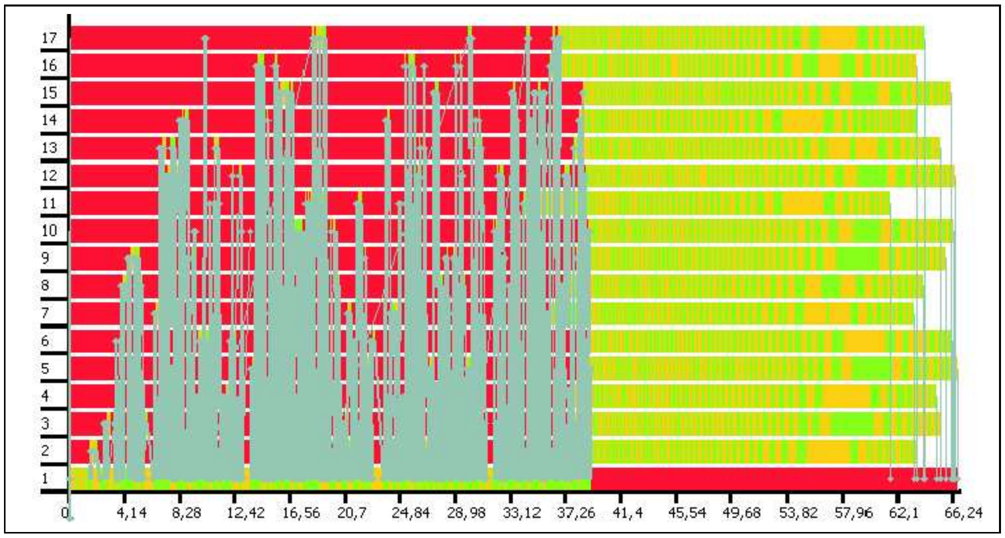
For the 4×4 torus used here, the data passed through the torus connections is a list of 3 matrices, $v_{k,l} = h_{k,l} = 4$. As the formula in Section 3.2 shows, this exactly outweighs the message reduction. Different numbers of messages result from the fact that smaller messages (channel names instead of matrices) are exchanged in the dynamic channel version. The number of messages drops from 1049 messages in the static torus multiplication of two matrices of size 600 to 761 messages, i.e. by about 30 %.

The runtimes of the version with dynamic channels (trace shown in Figure 11) is 40% less than for the version with static connections. Again, the improvement in runtime does not only result from saved messages, but from eliminating the bottleneck in the parent process. Without the direct torus connections, the algorithm must communicate the matrix blocks twice with a serious bottleneck in the parent process. As can be seen in Figure 10, the pure computation time is about the same (27 sec.) in both versions, but in the original version, it is preceded by an immense communication phase (almost 40 instead of 10 sec.).

4.3. Sorting in a Hypercube

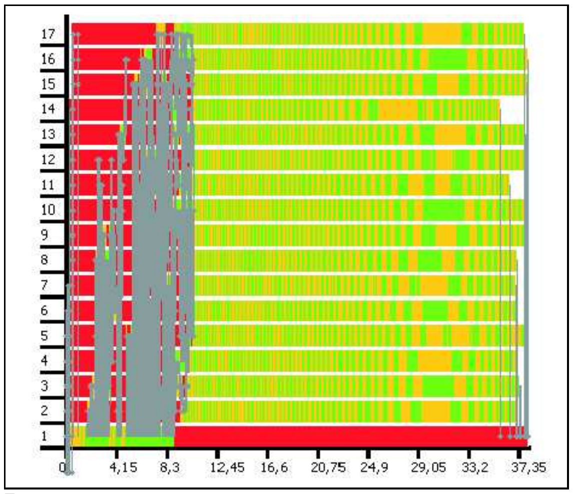
A lot of hypercube algorithms can be found in classical literature on parallel programming. As explained in Section 3.3, a comparison of the hypercube skeleton with and without dynamic channels is only possible for algorithms in which the hypercube nodes do not communicate simultanously in different hypercube dimensions. Test programs for this special case expose the anticipated bottleneck in the parent process, leading to dramatically increased runtimes (factor 8) for the static version.

We instead show the trace of a recursive parallel quicksort in a hypercube with dynamic channels (Figure 12) with message traffic, exposing the typical hypercube communication pattern. The random input list is locally created by the hypercube nodes (first phase in the trace, ca. 14 sec.), before the algorithm starts. The node



Runtime: 66 sec.

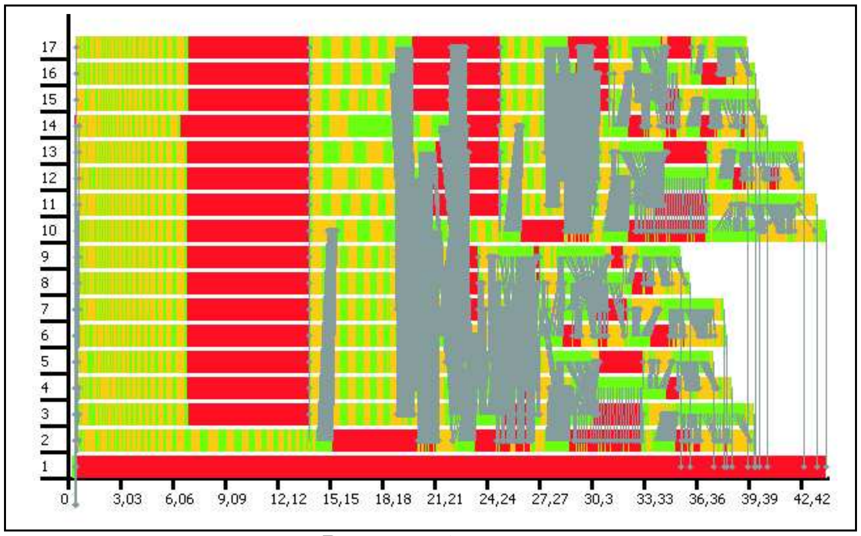
Figure 10: Matrix multiplication (600 rows), toroid **without dynamic channels**
(Beowulf Cluster, Heriot Watt University, Edinburgh, 17 machines)



Runtime: 38 sec.

Figure 11: Matrix multiplication (600 rows), toroid **with dynamic channels**
(Beowulf Cluster, Heriot Watt University, Edinburgh, 17 machines)

with the lowest address chooses a pivot element, which is broadcasted in the entire hypercube. All partners in the highest dimension exchange sublists, where the higher half keeps elements bigger than the chosen pivot. Then, the hypercube is split in half, and the algorithm is recursively repeated in the two subcubes.



Runtime: 43.8 sec.

Figure 12: Parallel Quicksort in hypercube **with dynamic channels**
(Beowulf Cluster, Heriot Watt University, Edinburgh, 17 machines)

5. Related Work

Dynamic reply channels are a simple but effective concept to support reactive communication in distributed systems. It is related to the 'incomplete message principle' known from concurrent logic languages [23] and the 'channel name passing' principle of the π-calculus [13]. In the context of a functional language it must however be introduced in a far more restricted way in order to preserve referential transparency (at least for a subset of the language).

By allowing completely free communication structures between parallel processes, for instance in the style of MPI [14], one gives up much programming comfort and security. The underlying theories which model communicating processes, namely π-calculus [13] and its predecessors, are as well liberal in terms of communication partners and usually untyped. Due to this inherent need for liberty which does not fit well in the functional model and its general aim of soundness and abstraction, not many parallel functional languages support arbitrary connections between units of computation at all. The concepts for Clean presented in [22] go in this direction and make use of Clean's uniqueness concept to purify some points. In the same way, languages like Facile [7] or Concurrent ML [20] support communication facilities on a lower-level of abstraction than the dynamic channel concept of Eden.

Functional languages like NESL [1], OCamlP3l [4], or PMLS [12] where the parallelism is introduced by pre-defined data-parallel operations or skeletons have the advantage to provide optimal parallel implementations of their parallel skeletons, but suffer from a lack of flexibility, as the programmer has no chance to invent new problem-specific skeletons or operations.

6. Conclusions

Our evaluation of topology skeletons shows that using dynamic channel connections substantially decreases the number of messages and eliminates bottlenecks. Dynamic channels can be usefully applied to speed up parallel computations substantially, as exemplified by typical case studies for the different topology skeletons discussed in this paper. As explained for the hypercube skeleton, dynamic channels may also be used to introduce more concurrency, and therefore offer new possibilities for skeletons. Using the trace visualisation for Eden, process behaviour at runtime and inter-process communication can be analysed more thoroughly than by simple runtime comparisons, which allows further optimisations for Eden skeletons.

Besides skeleton runtime analysis and optimisations, an area for future work is to investigate the potential performance gain and pragmatics of explicit process placement (possible in the Eden runtime system, but not exposed to language level yet) in conjunction with the presented and other topology skeletons.

Acknowledgements

The authors thank Phil Trinder from Heriot-Watt-University, Edinburgh and Hans-Wolfgang Loidl from LMU, Munich, for fruitful discussions and in particular for the opportunity to work with their Beowulf clusters.

References

[1] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[2] S. Breitinger and R. Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Funct. Prg.*, 1997. Available online.

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

[4] M. Danelutto, R. DiCosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *Proceedings of the ACM workshop on ML and its applications*, page 31ff. Cornell University, 1998.

[5] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. While. Parallel Programming Using Skeleton Functions. In *PARLE'93 — Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, page 146ff. Springer, 1993.

[6] L. A. Galán, C. Pareja, and R. Peña. Functional skeletons generate process topologies in Eden. In *PLILP'96 – Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *LNCS*, page 289ff, Aachen, Germany, Sep 1996. Springer.

[7] A. Giacalone, P. Mishra, and S. Prasad. Facile: a Symmetric Integration of Concurrent and Functional Programming. In *Tapsoft'89 – Int. Joint Conf. on Theory and Practice of Software Development*, volume 352 of *LNCS*, page 181ff. Springer, 1989.

[8] P. Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, Aug. 1986.

[9] Leighton, F. T. *Introduction to Parallel Architectures : Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

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

[11] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming*, 15(4):1–45, 2005.

[12] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.*, 16:181–206, 2001.

[13] R. Milner. *Communicating and Mobile Systems: The π Calculus.* Cambridge University Press, Cambridge, England, 1999.

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

[15] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01)*, volume 3 of *Trends in Functional Programming*, page 51ff. Intellect, 2001.

[16] M. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Reading, Massachusetts, USA, 1993.

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

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

[19] D. A. Reed and R. A. Aydt et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. of the Scalable Parallel Libraries Conference*, page 104ff. IEEE Computer Society, 1993.

[20] J. H. Reppy. *Concurrent Programming in ML.* CUP, Aug. 1999.

[21] P. Roldán-Gómez. Eden Trace Viewer: A Tool to Visualize Parallel Functional Program Executions. Master's thesis, Universidad Complutense de Madrid, Spain, 2004. (in German).

[22] P. R. Serrarens and R. Plasmeijer. Explicit message passing for concurrent clean. In *IFL'98 — Intl. Workshop on the Implementation of Functional Languages*, volume 1595 of *LNCS*, London, GB, 1999. Springer.

[23] E. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[24] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable implementation of Haskell. In *IFL'95 — Intl. Workshop on the Implementation of Functional Languages*, Bastad, Sweden, 1995. Available online.