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 decrease the number of messages by up to 50% and substantially reduce runtime. Detailed analyses of Eden TV (trace viewer) execution profiles reveal a bottleneck in the root process when only hierarchical channel connections are used and a better overlap of communications with dynamic channels.

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 [5,10]. 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. It can even be *implemented* in such a language, provided that appropriate constructs for expressing parallelism are available. 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 the programmer can introduce new skeletons if necessary.

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 [16] rely on such underlying communication topologies. As any skeleton, topology skeletons can easily be expressed in a functional language. A

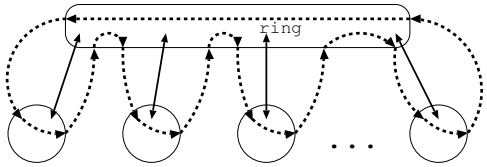


Figure 1: Hierarchical Ring Skeleton

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 size 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 defining `ringIns = rightRotate ringOuts`; the ring outputs, rotated by one position, are reused as ring inputs. 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) [20], Concurrent Clean [15], or Eden [9] use primitives to spawn subexpressions for parallel evaluation by a separate thread or process. Necessary arguments and the results are automatically 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 passed. The dashed lines show ring connections that are also established between the nodes and the parent, which forwards the ring data as indicated. This unnec-

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

essarily increases the number of messages and causes a bottleneck in the parent process.

The parallel functional language Eden [9] 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 [13], 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. A new trace viewer tool [18] 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 [9], 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 (with static connections)
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 convert between arbitrary input/output types `i/o` and the inputs `ri` / outputs `ro` of the ring processes.

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

```
ring, ringDC :: (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 [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. ◁

Eden provides the dynamic creation of reply channels, *dynamic channels* for short, to establish direct connections between arbitrary processes[c]. A unary type constructor `ChanName` is used to represent the name of a *dynamic channel*, which can be created and passed to another process to receive data from it. *Dynamic channels* are installed using the following two operators:

```
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`.

Example 2 Ring (with dynamic channels)
In the version of Figure 3, the static ring connections are replaced by dynamic 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

[c]The only difference between static and dynamic channels is that the former are generated during process creation while the latter are dynamically installed.

```
-- 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))
```

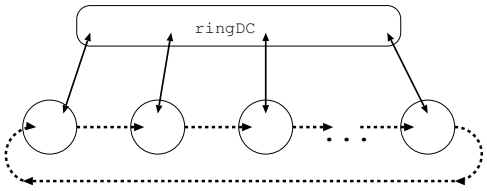


Figure 3: Ring Skeleton Using Dynamic Channels

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 `ri` from the parent) needs $tsize(\texttt{ri}) + 1$ *(closing message)* administrative messages from the child, where

tsize(`a`) is the number of top level tuple components for a tuple type `a`, and 1 otherwise. For simplicity, we only compute the amount of messages 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 $3 = tsize(ri, [r])+1$ administrative 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 [13]. 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 messages is

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

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

Figure 4: Static definition of toroid skeleton

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 in 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 `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*

```
hypercube :: (Trans i, Trans o, Trans r) =>
             Int -> ((i,[[r]]) -> (o,[[r]])) -- dimension / 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 -> ((i,[[r]]) -> (o,[[r]])) -- dimension / 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

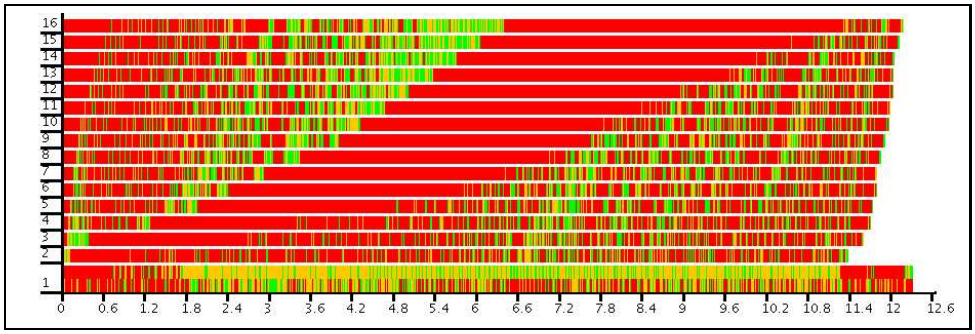
(unless using *Channel Structures* [2], which are currently not implemented). By using dynamic reply channels, communication between neighbours is handled in separate streams and independent threads. When communicated via the parent, the *list of streams* between hypercube neighbours would be communicated 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. In order to compare the dynamic channel version with a static version, we have defined specialized static versions for hypercubes up to dimension 4.

4. Case Studies

In this section, we will illustrate the skeleton improvements by runtime trace visualisations of applications which use the previously presented skeletons. We use the Eden Trace Viewer [18], a new tool for the post-mortem analysis of Eden program executions, which consists of an instrumented runtime system that produces execution traces and a stand-alone tool to analyse and visualise trace information. The execution as well as the communication between units is shown for all machines, processes and threads in different zoomable views.

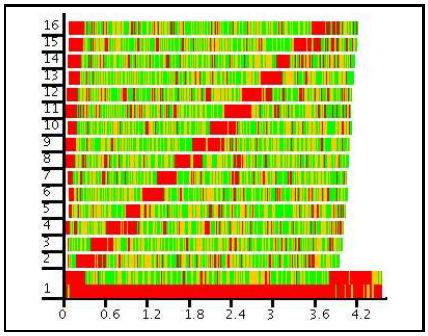
4.1. Warshall-Algorithm Using a Ring

A common use of ring structures is to circulate global data in parts between nodes of a parallel computation. As an example, we analyse a parallel implementation



Runtime: 12.33 sec.

Figure 6: Warshall-Algorithm (500 node graph) using **static connections** in ring



Runtime: 4.56 sec.

Figure 7: Warshall-Algorithm (500 node graph) using **dynamic channels** in ring

Common platform:
Beowulf Cluster, Heriot-Watt University, Edinburgh, 16 machines

of Warshall's algorithm to compute minimum distances between nodes of a graph (adapted from [14]). Each process evaluates rows of minimum distances to all other nodes for a subset of the graph nodes. Starting with the row for the first graph node, intermediate results are communicated to other processes through the ring. Each updated row flows through the whole ring for one round. On receiving a row from the predecessor, a process updates its own rows by checking whether paths via the respective node are shorter than the known minimum, 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 (Intel P4-SMP-Prcessors at 3GHz, 512MB RAM, Fast Ethernet), 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 16629) – network traffic is considerably reduced. Figure 8 shows zooms of the initial second of both traces, with messages between processes drawn as lines between the hori-

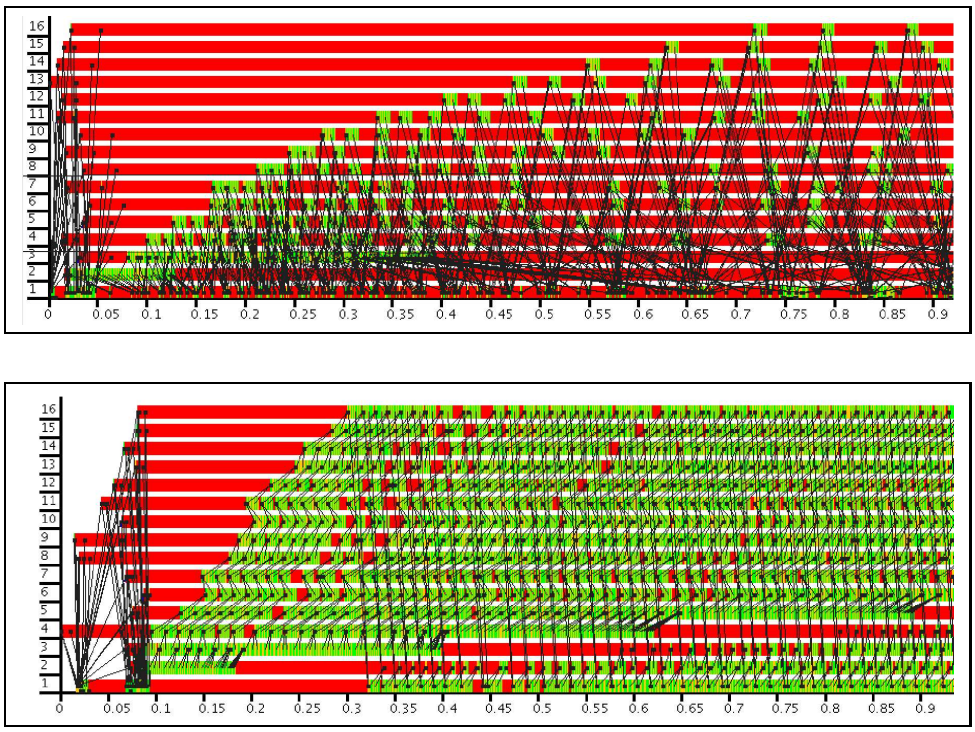


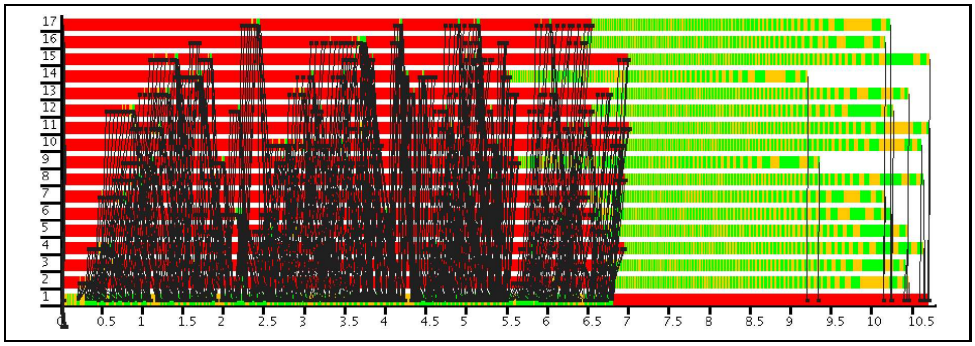
Figure 8: Zooms of the initial second with message traffic (black arrows) of the Warshall traces in Figures 6 and 7
– above using **static connections** – below using **dynamic channels** in ring

zontal bars. The static version shows the massive bottleneck in the parent process; worker processes often block waiting for data. The trace of the dynamic version nicely shows the intended ring structure and much less blocked phases.

The number of messages drops to about 50% and the runtime even drops to approximately 37%. The substantial runtime improvement is due to the algorithm's inherent data dependency: each process must wait for updated results of its predecessor. This dependency leads to a gap between the two phases passing through the ring. In the static version, the time each ring process waits for data from the heavily-loaded parent is *accumulated* through the whole ring, leading to a successively increasing wait phase while data flows through the ring. Although a small gap is also observable in the dynamic version, the directly connected ring processes overlap computation and communication and thus show a better workload distribution with only short blocked or idle phases.

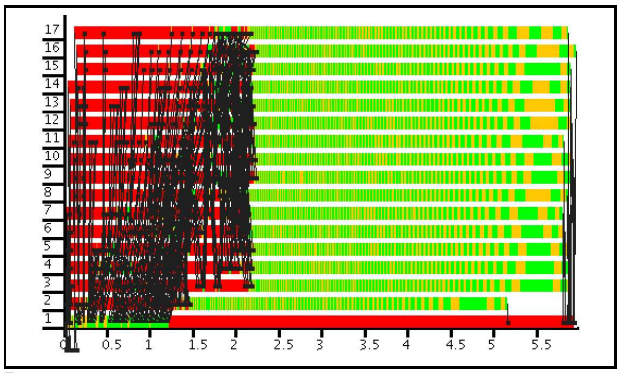
4.2. Matrix multiplication in a torus

The presented toroid structure can be used to implement the parallel matrix multiplication algorithm by Gentleman [16]. The result matrix is split into a square of



Runtime: 10.76 sec.

Figure 9: Matrix multiplication (600 rows), toroid **with static connections**



Runtime: 5.96 sec.

Figure 10: Matrix multiplication (600 rows), toroid **with dynamic channels**

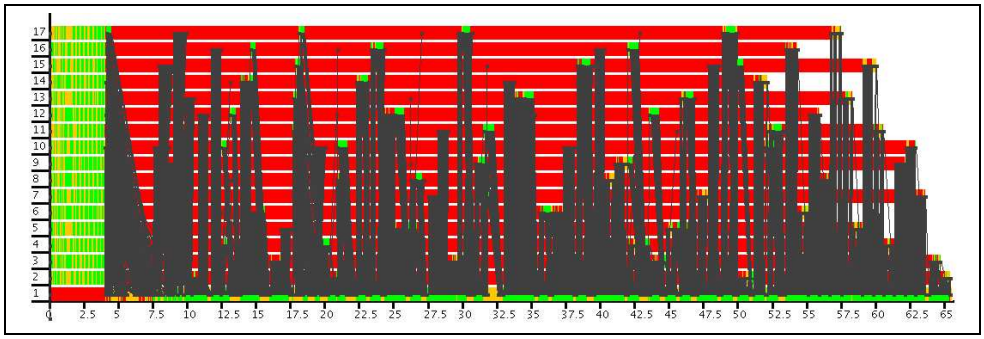
Common platform:
Beowulf Cluster,
Heriot-Watt University,
Edinburgh, 17 machines

submatrix blocks which are computed by parallel processes. The data needed to compute a result block are corresponding 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). Input blocks do one round through the torus, so that every process eventually receives every input block it needs. Each process accumulates a sum of products of the input blocks as the final result. The analyses in [8] 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.

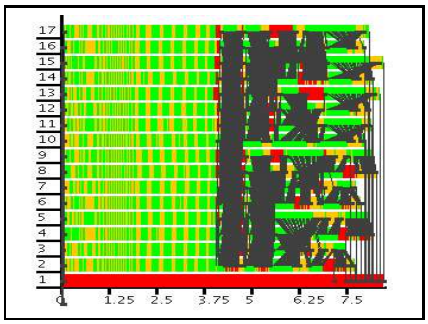
For the 4×4 torus used here, the data passed through the torus connections is a list of 3 matrices[e] $v_{k,l} = h_{k,l} = 4$. As the formula in Section 3.2 shows, this exactly

[e] number of blocks (=4) per row/column minus local block (=1)



Runtime: 65.57 sec.

Figure 11: Parallel Quicksort in hypercube **with static connections**



Runtime: 8.47 sec.

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

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 10) is 44% 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 Figures 9 and 10, the pure computation times (final green/middle grey parts) are about the same (approx. 4 sec.) in both versions, but in the original version it is preceded by an immense communication phase (almost 7 instead of 2 sec.).

4.3. Sorting in a Hypercube

As explained in Section 3.3, a comparison of the hypercube skeleton with and without dynamic channels is only possible using fixed size static hypercube skeletons. We show the traces of a recursive parallel quicksort in a hypercube with static

connections (for the fixed dimension 4) (Figure 11) and a hypercube with dynamic channels (Figure 12) which takes the dimension as a parameter. The input list of $5m$ random integers is locally created by the hypercube nodes (ca. 4 sec.), before the sorting algorithm starts. The node 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. The version with dynamic channels performs well and exposes the typical hypercube communication pattern. The runtime for the static version dramatically increases (factor 7.7) due to the obvious bottleneck.

5. Related Work

Dynamic reply channels are a simple but effective concept to support reactive communication in distributed systems. By allowing completely free communication structures between parallel processes, for instance in the style of MPI [12], one gives up much programming comfort and security. The underlying theories which model communicating processes, namely π-calculus [11] 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, only few parallel functional languages support arbitrary connections between units of computation at all. The channel concept of Clean [19] as well as the communication features in Facile [7] or Concurrent ML [17] are more general and powerful, yet on a lower level of abstraction than the dynamic channel concept of Eden.

Functional languages like NESL [1], OCamlP3l [4], or PMLS [10] 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, 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. We thank Phil Trinder from Heriot-Watt University, Edinburgh and Hans-Wolfgang Loidl from Ludwig-Maximilians-Universität, Munich, for fruitful discussions and 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*. MIT Press, 1989.

[4] M. Danelutto, R. DiCosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *ACM workshop on ML and its applications*, page 31ff, 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*, LNCS 694, 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*, LNCS 1140, page 289ff. Springer, 1996.

[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*, LNCS 352, page 181ff. Springer, 1989.

[8] 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.

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

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

[11] R. Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge Univ. Press, 1999.

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

[13] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. Trends in Functional Programming, Vol. 3, page 51ff. Intellect, 2001.

[14] M. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[15] 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.

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

[17] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ. Press, 1999.

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

[19] P. R. Serrarens and R. Plasmeijer. Explicit message passing for concurrent clean. In *IFL'98 — Implementation of Functional Languages*, LNCS 1595. Springer, 1999.

[20] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Proc. PLDI '96 — Progr. Language Design and Impl.*, pages 78–88. ACM, 1996.