Skeleton Composition using Remote Data

Mischa Dieterle, Thomas Horstmeyer, and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{dieterle,horstmey,loogen}@informatik.uni-marburg.de

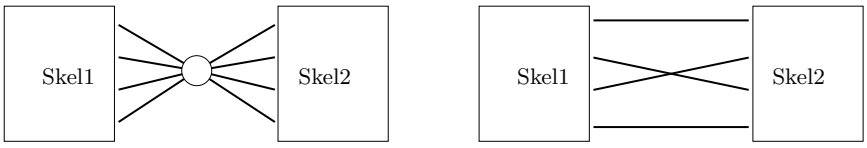
Abstract. Skeletons simplify parallel programming by providing general patterns of parallel computations. When several skeletons are used inside the same program, skeleton composition usually leads to aggregation and redistribution of the intermediate data on a single process. Though the programmer can overcome the performance loss at a lower level of abstraction by altering the existing skeletons or not using them at all. A high-level concept like skeleton-based programming, however, calls for a more general solution.
Remote data provides runtime mechanisms that allow declaratively specified processes to access other processes' data via remote handles. This enables the programmer to easily build complex skeletons by combining simpler ones. Skeletons can be composed without the drawback of collecting and then redistributing the data in between two skeleton instances. Another advantage is that skeletons which *inherently* depend on their inner communication patterns are easily implemented using remote data. We present the implementation of remote data in the parallel functional language Eden and show the definition of some example skeletons with a remote data interface.

Keywords: skeletons, composition, parallel, functional

1 Introduction

Algorithmic skeletons [5] capture common patterns of parallel evaluations like task farms, pipelines, divide-and-conquer schemes etc. The application programmer only needs to instantiate a skeleton appropriately, thereby concentrating on the problem-specific matters and trusting on the skeleton with respect to all parallel details. Skeletons should be small and simple to instantiate to increase the ease and flexibility of their use. In particular, it should be possible to compose and nest skeleton instantiations arbitrarily. This means for the case of a distributed memory setup and structured data that must be passed from one skeleton to the next that the result of the first skeleton is gathered in a single process and redistributed for the following skeleton execution. This causes unnecessary communication and holds the danger of a communication bottleneck in the caller process (see Fig. 1 (a)). A typical example is the composition of two parallel maps (parallel task farms) producing a two dimensional matrix with an intermediate transpose.

In Manuel Carro and Ricardo Pea, editors, Practical Aspects of Declarative Languages 12th International Symposium, PADL 2010, LNCS 5937, pages 73-87, Madrid, Spain, January 2010.



(a) collection and redistribution (b) direct redistribution

Fig. 1. Data transfer between composed skeleton instances

There exist several proposals to avoid the gathering and redistribution of distributed data. One could introduce a new distributed data type as common in languages with a data-parallel concept [7, 6] where data can be passed in a distributed manner. In this case, one needs special transformation and conversion functions to redistribute the distributed data or to switch between distributed and common data types. Another simple alternative would be to design a new integrated skeleton for the composition by merging the two skeleton instantiations and organising the redistribution explicitly within the new skeleton context. This approach has the disadvantage that the programmer has to go into the internals of skeleton design and that the clarity of the original composition is lost.

In this paper, we present an alternative approach that allows the direct passing of distributed result data from one skeleton instance to the next one (see Fig. 1 (b)). The main idea is to replace the data by handles to it, called remote data, which are gathered and redistributed instead. The handles can then be used to pull the real data directly to the target. This concept which has independently been suggested by Alt and Gorlatch [2, 3, 1] can be easily used: normal data is replaced by the corresponding remote data handles and skeletons that operate on the new remote data can be composed as before. Only that now the gathering and redistribution of complex data is replaced by the gathering and exchange of small remote data handles which are used for the direct data exchange between processes within different skeleton instances. Thus, remote data handles for data which may be located elsewhere can be used like the original data but cause only low communication costs. They can occur everywhere where ordinary data may occur, e.g. in lists or trees to model distributed data structures. As we will show, this concept is flexible to use and still type-safe.

We develop the concept of remote data in the context of our parallel functional language Eden, although the concept itself is language-independent. It could equally well be added to other parallel languages, see [2, 3, 1] for a realisation in Java. The realisation in a declarative language has the advantage that the beauty and elegance of declarative programming is maintained for parallel skeleton-based programming. In functional languages, skeletons are realised as higher order functions. Skeleton instantiation reduces to function application and skeleton composition is nothing else than function composition.

We will introduce a new data type `RD a` representing a handle for remote data of type `a` and provide interface functions `release :: a → RD a` and `fetch :: RD a → a`. The function `release` yields a remote data handle that can be passed to other processes, which will in turn use the function `fetch`

to access the remote data. The data transmission occurs automatically from the processes that released the data to the process which uses the handle to fetch the remote data. Skeleton composition `skel2 ∘ skel1` of type `a → c` where `skel2` of type `b → c` and `skel1` of type `a → b` will now be replaced by `skel2' ∘ skel1'` of type `a → c` where `skel2'` is of type `RD b → c` and `skel1'` is of type `a → RD b`. The modified skeleton definitions differ from the original ones only in additional applications of `release` in `skel1'` and `fetch` in `skel2'`. These small modifications solve our problem while preserving the original program structure. We will show that complex communication structures like an all-to-all scheme can easily and elegantly be defined using remote data.

Plan of the paper In Section 2 we give a introduction to the language Eden. Section 3 presents the implementation of the new data type constructor `RD` with interface functions `fetch` and `release` in Eden. Section 4 shows how to use remote data for skeleton composition and the definition of complex communication patterns. Section 5 compares with related work while Section 6 finally concludes.

2 Eden in a Nutshell

The parallel Haskell dialect Eden [9] extends Haskell with an explicit notion of processes (function applications evaluated remotely in parallel). The programmer has direct control over evaluation site, process granularity, data distribution and communication topology, but does not have to manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer.

The essential two coordination constructs of Eden are process abstraction and instantiation:

```
process :: (Trans a, Trans b) ⇒ (a → b) → Process a b
( # )   :: (Trans a, Trans b) ⇒ Process a b → a → b
```

The function `process` embeds functions of type `a → b` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the `Trans` class of transmissible values. Evaluation of an expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`.

For immediately instantiating a list of process abstractions with appropriate inputs, Eden provides a (predefined) function `spawn`, and a variant `spawnAt` which additionally locates the created processes on given processor elements. Neglecting demand control, `spawn` is denotationally specified, and could be defined, by the following equation.

```
spawn :: (Trans a, Trans b)⇒ [Process a b] → [a] → [b]
spawn =  zipWith (#)
```

Eden further provides functions to create and use explicit connections between arbitrary processes.

```
new     :: Trans a ⇒ (ChanName a → a → b) → b
parfill :: Trans a ⇒ ChanName a → a → b → b
```

They can be used to shortcut the tree-shaped topologies created by the basic functions. The function `new` is used at the receiver side to created a receiver port of a unidirectional channel connection. It works in continuation passing style, `new`'s parameter function's first parameter is the "name" of the channel (type `ChanName a`) whose incoming port is created as a side effect. The second parameter is the value that will be received via the channel. The parameter function's output is the result of the function `new`. The sender of the connection is still not determined. The channel's "name" can be passed to another process. The connection gets established when `parfill` is used at the sender side using the "name" of the channel (containing the receivers process ID and port). The function `parfill` takes the value to be written in the channel, `parfill`'s third argument is returned unchanged after forking a thread that sends the data through the channel.

These two function are quite complicated to use for people new to Eden. Their signatures interplay well in some circumstances, but they are not intuitive at all. The main problem when using dynamic channels is the change of direction in the communication: when Process 1 wants to send data directly to Process 2 using a dynamic channel, this channel must first be generated by Process 2 and sent from Process 2 to Process 1 before the proper data transfer from Process 1 to Process 2 can take place. Thus, the dynamic channel must be communicated in the opposite direction in which the data is to be transferred. This complicates the use of dynamic channels. The remote data approach keeps the direction of the communication by introducing another channel transfer from Process 1 to Process 2. This transfer sends a channel via which Process 2 can send its data channel to Process 1. Thus, an exchange of dynamic channels takes place between Process 1 and Process 2 which automatically establishes a data channel connection from Process 1 to Process 2. In the following, we implement the remote data concept using Eden's dynamic channels. Note that this concept provides the same expressive power as dynamic channels, but in a more natural and easier-to-use way.

3 Eden Implementation of Remote Data

The implementation of remote data in Eden (Figure 2) is simple and elegant. To `release` a local data `x` of type `a` we create – using the function `new` – a channel name `cc` of type `ChanName (ChanName a)` via which a channel `c` of type `ChanName a` will be received. Using `parfill` a thread is forked that subsequently sends the local data `x` via the channel `c`. The result of the `release` function is the newly created channel `cc :: ChanName (ChanName a)`. Note that the remote data type `RD a` is a synonym of `cc`'s type. Data of type `RD a`

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

-- convert local data into corresponding remote data
release :: Trans a ⇒ a → RD a
release x = new (λcc c → parfill c x cc)

-- convert remote data into corresponding local data
fetch   :: Trans a ⇒ RD a → a
fetch cc = new (λc x → parfill cc c x)
```

Fig. 2. Remote data definition

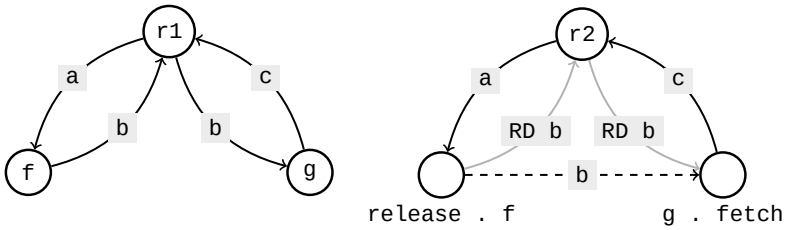


Fig. 3. Using remote data

is merely a channel name and thus very lightweight with low communication costs. To access remote data we need to `fetch` it by again creating a channel `c :: ChanName a` using the function `new`. This channel is sent via the remote data handle, i.e. the channel `cc` of type `RD a`. The proper data is then received via channel `c` and returned as the result of the `fetch` function.

A problem arises when remote data needs to be duplicated. Channel names (of type `ChanName a`) cannot be used more than once to retain referential transparency [9]. As remote data is implemented as a specialized channel name, it must not be duplicated and fetched several times in parallel. A manual workaround to duplicate remote data on a node would be to fetch the data and release it again repeatedly. We considered more sophisticated versions which make the use of remote data more comfortable, but they expose nondeterminism and should therefore not be implemented in the actual version of Eden.

Our new way of communication creates a slight overhead. In comparison to the common way of defining explicit communication we have an additional channel per direct connection that is used only before the transmission of the actual data begins. However, as this channel only transports a value of type `ChanName a` which is quite small the increase in communication cost should not be noticeable in most cases.

Example. We show a small example where the remote data concept is used to establish a direct channel connection between sibling processes. Given functions `f` and `g`, one can calculate `(g ∘ f) a` in parallel creating a process for each

function. Figure 3 shows two different ways to implement this. Simply replacing the function calls by process instantiations

```
r1 a = process g # (process f # a)
```

leads to the following behaviour (visualised in the left part of Fig. 3): Function `r1` instantiates the first process calculating f, passes its input to this process and receives the remotely calculated result. It instantiates a second process calculating `g` and passes the result of process `f` to this new process. The output of the second process is also sent back to the caller. The drawback of this approach is that the result of the first process will not be sent directly to the second process. This causes unnecessary communication costs.

We use remote data `RD a` in the second implementation

```
r2 a = process (g ∘ fetch) # (process (release ∘ f) # a).
```

It uses function `release` to produce a handle of type `RD a` for data of type `a`. Calling `fetch` with remote data returns the value released before. Function `r2` is identical to `r1` except for the conversion of the result type of `f`'s process and the input type of `g`'s process to remote data. The use of remote data leads to a direct communication of the actual data between the processes of `f` and `g` (see the right part of Fig. 3). The remote data handles are treated like the original data in the first version and the basic structure of the program, i.e. the composition of two process instantiations, remains the same.

4 Composing Predefined Skeletons

Before handling the composition of skeletons using the remote data concept, we show the lifting of a simple parallel map skeleton to a remote data interface. Then we define a parallel all-to-all skeleton which generates a number of processes each of which exchanges data with any of the others. Using these skeletons with their remote data interfaces enables us to define a sequence consisting of a parallel map, a parallel transpose (realised using the all-to-all skeleton) and a second parallel map. This can be useful in an implementation of a parallel FFT skeleton [8] or a Google Map-Reduce skeleton [4]. In [4, 8], corresponding parallel map-transpose skeletons have been defined as monolithic skeletons without composing simpler skeletons. With the remote data interface, we can define the same skeleton as a composition of the three component skeletons. This leads to a much better understandable definition while achieving the same performance. Finally, we present another elegant and concise definition of an even more complex communication pattern: a butterfly scheme which is used to define an all-reduce-skeleton.

4.1 The `parmapDC` skeleton

A parallel map creates a process for each element of the input list. In Eden, it can easily be defined using the function `spawn` (see Fig. 4). Note that this definition implies that the process evaluating `parmap` creates as many processes as there are elements in the input list and sends each of theses elements to the corresponding process. Using a remote data interface, each process only gets a handle to

```
parmap :: (Trans a,Trans b) ⇒ (a→b) → [a] → [b]
parmap f xs = spawn pfs xs
    where pfs = repeat (process f)

parmapDC :: (Trans a,Trans b) ⇒ (a→b) → [RD a] → [RD b]
parmapDC f xs = spawn pfs xs
    where pfs = repeat (process (liftRD f))

liftRD   :: (Trans a, Trans b) ⇒ (a→b) → RD a → RD b
liftRD f =  release ∘ f ∘ fetch
```

Fig. 4. The `parmap` and `parmapDC` skeletons

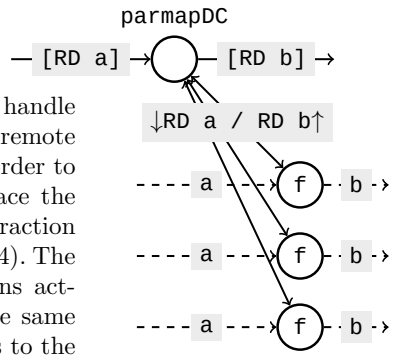


Fig. 5. Visualization of the `parmapDC` skeleton

its list element. It can then use this handle to fetch the element directly from the remote place where this element is located. In order to achieve this behaviour, we simply replace the parameter function `f` in the process abstraction by its lifted pendant `liftRD f` (see Fig. 4). The function `liftRD` is used to lift functions acting on data to functions performing the same computation on remote data. This leads to the skeleton `parmapDC` where the ending `DC` stands for **D**irectly **C**omposable due to the remote data interface. This interface makes it possible for skeletons to receive distributed input and to produce distributed output which is crucial for an efficient composition of skeletons. Fig. 5 visualises the behaviour and communication paths of the `parmapDC` skeleton. The upper circle represents the process evaluating the `parmapDC` instantiation. It generates the other processes whose task is to apply the parameter function `f` to input of type `a` and produce output of type `b`. Note that only remote data handles for the input and the output values are communicated between the generator process and its child processes. The proper data is communicated via dynamic channel connections indicated by dashed lines.

4.2 The `allToAllDC` skeleton

In Figure 6 we present an all-to-all skeleton `allToAllDC`. This skeleton depends inherently on its inner communication pattern which we will implement using remote data. We need the following variants of the remote data interface functions in order to fetch or release a list of remote data:

– `releaseAll :: [a] → [RD a]` is defined as `map release`.

– `fetchAll :: [RD a] → [a]` is semantically equivalent to `map fetch`, but needs a special eager implementation which initiates to fetch each input list element without waiting for the result of this action.

The input of the `allToAllDC` skeleton is a list of remote data with, say, n elements and two transformation functions `t1` and `t2` to allow the processes to transform the input data before sending data to all other processes and after

```
allToAllDC :: forall a b i. (Trans a, Trans b, Trans i) ⇒
          --(#Elements, data in, data out)
            (Int→a→[i]) →       -- transform before transpose
            ([i]→b) →           -- transform after transpose
            [RD a] → [RD b]
allToAllDC t1 t2 xs = res where
  t1' = t1 (length xs)          --same amount of procs as #xs
  (res,iss) = unzip $ spawn procs inp
  inp       = lazy2Zip xs (transpose iss)

  procs      = repeat $ process $ uncurry p
  p :: (Trans a,Trans b,Trans i)⇒ RD a→ [RD i]→ (RD b,[RD i])
  p x theirIs = (res, myIs) where
    res   = (release ∘ t2 ∘ fetchAll) theirIs
    myIs  = (releaseAll ∘ t1' ∘ fetch) x

--lazy in second argument
lazy2Zip (x:xs) ~(y:ys) = (x,y): lazy2Zip xs ys
lazy2Zip []     _     = []
```

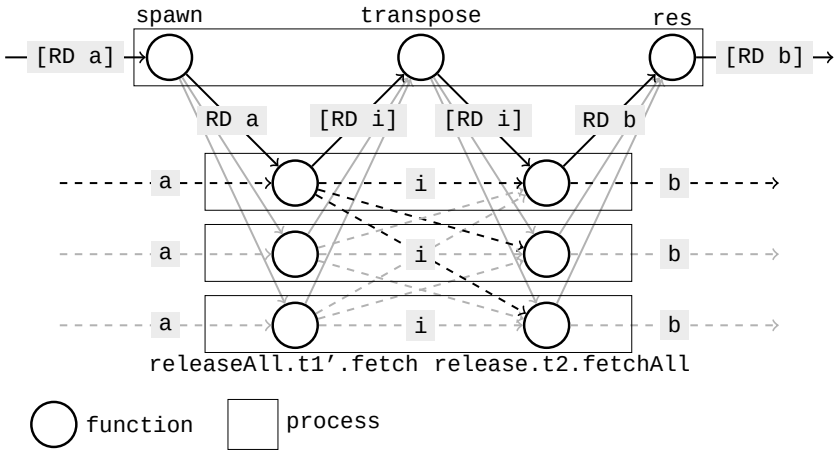


Fig. 6. The `allToAllDC` skeleton: code and visualisation. (The darker shading of the arrows from the uppermost child process emphasizes the connectivity of a single process.)

```
mtmDC :: (Trans a, Trans b, Trans c)
         ⇒ (a→[[b]]) → ([[b]]→c) → [RD a] → [RD c]
mtmDC f g = parmapDC g ∘ parTransposeDC ∘ parmapDC f

parTransposeDC :: Trans b ⇒  [RD [[b]]]→[RD[[b]]]
parTransposeDC = allToAllDC (λ n → unshuffleN n ∘ transpose)
                            (map shuffle ∘ transpose)

-- round robin / segmented distribution
unshuffleN     , splitEvery :: Int → [a] → [[a]]
unshuffleN n xs = transpose $ splitEvery n xs
shuffle :: [[a]] → [a]  -- inverse function
shuffle = concat ∘ transpose
```

Fig. 7. Composition of parmap and transpose skeletons

receiving data from all other processes, respectively. The length of the input list determines the number of processes to be created by `spawn`. Every process will fetch its remote input `x` and transform it with the transformation function `t1`. This yields a list of intermediate data for each child process which is released element-wise by `releaseAll`, giving the list `myIs :: [RD i]` with remote data handles. Note that this list must have the same number n of elements as the input list. This list of remote data handles is returned to the root process in the second component of each process's result tuple. The root process receives one such list from each of its child processes resulting in the $n \times n$ matrix `iss :: [[RD i]]`. It transposes this matrix and sends the result back to the processes as its second, lazily supplied parameter `theirIs`. Each process gets thus one remote intermediate value of type `RD i` of each sibling process and of itself. The values are gathered using `fetchAll`, transformed by the second parameter function `t2` to the output type `b` and released. The visualisation in Fig. 6 again shows the exchange of remote data handles between the root process the child processes and using dashed arrow the direct communication of data between the processes.

4.3 Composing Skeletons with Remote Data Interface

The `allToAllDC` skeleton can be used to express arbitrary data exchange that requires an all-to-all network. A common special case is the transposition of a matrix which is distributed over several processes. The way the matrix is distributed over the processes can be manifold. Each process might be assigned e.g. to one row or — more general — to several rows of the matrix. In the example skeleton `parTransposeDC` of Fig. 7, we implement the more general case. Thus, we are not restricted to 1:1 relations between rows and processes. We assume that rows are distributed round robin over the processes. The advantage against a block distribution is that the matrix can be assigned partially to the processes without knowledge of the overall number of rows. Hence, the transposition skeleton has to assign the columns of the overall matrix (rows of

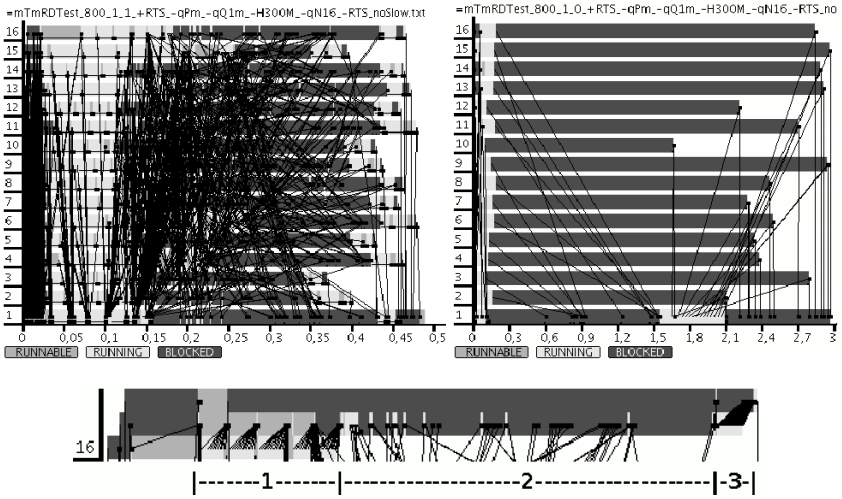


Fig. 8. Runtime behaviour of the skeleton `mtmDC` in the global view(left), the zoomed process on Machine 16 (bottom) vs. the local transposition version (right).
(Note the different scaling of the x-axes in the upper traces and that the zoomed view has been taken from a processes-per-machine view, here showing the activity bars of the three processes on Machine 16.)

the transposed matrix) round robin to the processes. The first transformation function of type `Int → [[b]] → [[[b]]]` first transposes a list of rows to get the list of the former columns. In a second step, these are round robin distributed to sublists, one for each process. Process i will consequently receive one row-sliced and column-sliced partial matrix from each process. The second transformation of type `[[[b]]] → [[b]]` will shuffle the row-slices (transposed column-slices) into each other to recover the rows of the overall transposed matrix. This is done by flipping the outer dimension (the list of partial matrices) with the row-dimension using `transpose`. Thus every outer list element contains all partial rows belonging to the same row of the overall matrix. The transformation `map shuffle` re-establishes each row.

Now, we can combine the `parmapDC` skeleton of Fig. 4 and the parallel transpose skeleton `parTransposeDC` in the function `mtmDC` (cf. Fig. 7), a parallel version of the function composition `map g ∘ transpose ∘ map f`. Without remote data a naive parallel implementation would be

```
parmap g ∘ unshuffleN n ∘ transpose ∘ shuffle ∘ parmap f
```

This version gathers the data for the intermediate transposition step in the caller process.

We compared runtime activity profiles of the `mtmDC` skeleton with the naive version. In our example executions, the parameter functions `f` and `g` have been set to the dummy function `map (scanl1 (+))` which creates rows of prefix sums. The input matrix contained the number 1 in each position.

In order to focus on communications in the middle part of the composed skeletons, input and output communications have been suppressed in the runtime traces underlying the activity profiles. Moreover, the default streaming mode of the communication has been replaced by a single message mode to reduce the number of messages exchanged between the processes.

Each skeleton was instantiated with an input matrix of size 800×800 and evaluated on 8 Intel Core 2 Duo machines with a Fast Ethernet connection, where each processor core hosted two virtual machines of the Eden runtime system. In Fig. 8, we present the activity profiles of the corresponding runtime traces for the two skeletons. The trace visualisations show the activity of each machine on a horizontal bar. The different activity phases of the virtual machines (runnable, running, blocked) are indicated by different colours explained in the traces legend. Messages are depicted by black lines with an arrow (black dot) on the receiver side. The x-axis shows the time in seconds.

The upper left trace in Fig. 8 clearly reveals the distributed transposition by the multitude of messages exchanged right after the initial data generation phase and the first `map`-phase, which is depicted "running" in the trace. The exchange of remote data starts very early overlapping the `map`-phase and forming dense bundles of messages. The second `map`-phase at the end of the program execution is rather short. Note that the overall runtime was less than 0.5 seconds.

We have placed the ith process of every skeleton on the same machine, such that communication costs are low. The lower zoomed view of the figure shows the activity bars of the three processes located on the virtual machine 16. The lowest bar belongs to a child of the first `parmapDC`-instantiation. The upper two bars show the processes of the parallel transpose skeleton and the second `parmap` instantiation. With this information, we can easily identify the different types of messages. During phase 1 the process of the first `parmapDC` skeleton sends its results to the `parTransposeDC` process. In the second phase the intermediate data is exchanged with the processes on the other machines. Finally, in phase 3, the result of the transposition is passed on to the second `parmapDC` process.

The upper right trace in Fig. 8 belongs to the naive version which performs a local transposition in the root process. As expected, this version is much slower with an overall runtime of approximately 3 seconds. The conspicuously fast communication between machine 1 and machine 10 is because the two virtual machines share the same physical machine. Further tests with varying input sizes (not shown) confirmed the enormous runtime advantages of the distributed version.

4.4 The `allReduceDC` skeleton

The all-reduce skeleton combines distributed data using a binary reduction function. It leaves the result duplicated on all processes involved in the reduction. Usually, it is implemented using the classical butterfly scheme which is also a common way to efficiently synchronise data between parallel processes. As for the `allToAllDC` skeleton, it is crucial for the all-reduce skeleton that data is

transferred to and from the skeleton in a distributed way. The butterfly reduction for n processes is done in $logn$ parallel communication and local reduction steps. In each step, the communication partner of process k is usually calculated with the boolean function k `xor` 2^{step-1}.
Fig. 9 shows the definition of the function `bitFlipF` which applies a transformational way to determine the communication partner for the current `step`. The input list `xs` contains at position j the value of process j. `xs` is distributed round robin to `d=(2^step)` sublists. The values to be exchanged are in the same columns of the transformed matrix. Their indexes differ by 2^{step-1} which equals `d 'div' 2` or half the number of inner lists. We flip the first half of inner lists with the second half and achieve the desired value exchange. A function call to `shuffle` re-establishes the original list structure.

The `allReduceDC` (see Figure 10) skeleton uses the function `bitFlipF` to rearrange lists of remote data in the caller process which represent the results of the intermediate reduction steps of the skeleton's processes. The rearranged lists are sent back to the processes. Thus, each process gets the remote values released by one partner in every step. Fetching these values establishes the butterfly communication topology.

The skeleton's input is a list with 2^{steps} remote data handles[1]. For each handle a process will be instantiated. The skeleton takes two parameter functions: function `initF :: a → b` is used to transform the initial remote value of each process after it is fetched. This transformation allows to work with different types for the input values and the reduction function inputs. The reduction function `redF :: b → b → b` which should be associative and commutative is applied in each step to the results of the previous step of a process and of its partner. This behaviour can concisely be expressed with `scanl1 redF` applied to the stream `toReduce` of values to be reduced. The stream `toReduce` is composed of the initial value and the stream input `theirReds`. The latter contains the partners' values for all steps. Note that the complete list structure of `theirReds` is already built in `theirReds'` even before its first element is received. Thus the request for all remote values can be eagerly initiated by the function `fetchAll` which would otherwise block on an incomplete list structure. The result of the `scanl1` application is element-wise released in every process, resulting in a list of remote data which is also generated in advance. This happens because the evaluation of `releaseAll` equally depends only on its parameter list's structure.

[1] The `allToAllDC` skeleton only works for input lists where the length is a power of two. Other lists are cut to the next smaller power of two.

```
bitFlipF :: Int → [a] → [a]
bitFlipF step xs = (shuffle ∘ flipAtHalfF ∘ unshuffleN d) xs where
  d = (2 ^ step)
  flipAtHalfF xs = let (xs1, xs2) = splitAt (d `div` 2) xs
                   in xs2 ++ xs1
```

Fig. 9. Flip of values at bit `ldi`

```
allReduceDC :: forall a b. (Trans a, Trans b) ⇒
            (a → b) →              --initial transform function
            (b → b → b) →          --reduce function
            [RD a] → [RD b]
allReduceDC initF redF rdAs = rdBss !! steps where
  steps = (floor ∘ logBase 2 ∘ fromIntegral ∘ length) rdAs
  rdAs' = take (2^steps) rdAs          --cut input to power of 2

 -- topology, inputs and instantiation
  rdBss = (transpose ∘ spawn procs) inp         --steps in rows
  bufly = zipWith bitFlipF [1..steps] rdBss     --only init rdBss
  inp   = lazy2Zip rdAs' (transpose bufly)      --steps in cols

 -- process functionality and abstraction
  procs = repeat $ process $ uncurry p
  p :: (Trans a, Trans b) ⇒ RD a → [RD b] → [RD b]
  p rdA theirReds = (releaseAll ∘ scanl1 redF) toReduce where
    toReduce = (initF ∘  fetch) rdA : fetchAll theirReds'
    theirReds'= lazy2ZipWith (curry snd) [0..steps] theirReds
```

Fig. 10. The `allReduceDC` skeleton

Thus the exchange of remote data handles via the root process can happen in advance, independently of the parallel reduction steps.

The caller process gathers the result streams of all processes in a nested list. We transpose this list to have all remote values of a step in each inner list of `rdBss`. Applying the function `bitFlipF` to the first `steps` lists permutes these according to the butterfly scheme. We transpose this permutation `bufly` such that each process's input is located in one inner list. This transposed list is lazily zipped with the initially supplied input list `rdAs` using `lazy2Zip` and passed back to the processes. The final result consists of the results of the last reduction step, i.e. the last element of the list `rdBss`.

We have tested the `allReduceDC` skeleton with a dummy example which we executed on an 8 core Intel Xeon machine. The initial transformation function `initF` serves as generator and generates the list `[1..nElems]`, where `nElems` is a parameter of the program and in our example set to 200000. The trace visualisation in Fig. 11 reveals interchanging computation and communication phases. The butterfly interconnection scheme can clearly be recognised in the messages exchanged between the processes. The generation of elements is depicted as the first "running" phase. The reduction network has been set up before, by exchanging the remote data messages via the root process on Machine 1 (initial messages). Three reduction phases follow. First the direct neighbours exchange their lists leading to the typical butterfly pattern of messages. The processes reduce their lists using the reduction function `redF` which is set to `zipWith (+)`. For the next steps, the distance to the partner process is doubled every time. Finally, a `parmapDC` skeleton is called to consume the data and return an empty list to the root process.

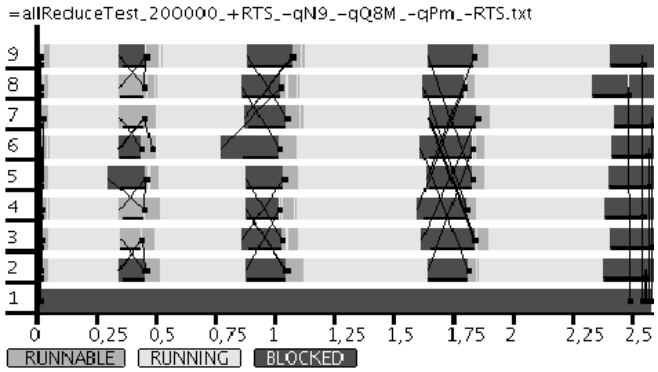


Fig. 11. Runtime behaviour of the `allReduceDC` skeleton

5 Related Work

Alt and Gorlatch [1–3] introduced a concept similar to remote data called remote references in the context of optimisations of Java RMI. They concentrated on what they called lazy RMI, localised RMI and future-based RMI. Lazy RMI describes the basic functionality. Future-based RMI allows to create and pass remote references before the corresponding values are computed. We get this in Eden for free because of Haskell's laziness. An optimization for data passed locally on the same machine, like localised RMI, would be a good optimisation of the Eden runtime system, but is currently not implemented.

Alternative approaches to skeleton composition are based on the use of distributed data structures. Kuchen and Cole [7] describe a skeleton library based on C++ and MPI which integrates task and data parallel skeletons. Darlington [6] uses an imperative base as well but describes the composition of (predefined) skeletons itself functionally in the structured coordination language (SCL).

Although programming with distributed data structures is comfortable and efficient, the number of predifined data structures is limited and their use is thus not as flexible as working with remote data. Remote data can be nested in arbitrary algebraic data structures and manipulated by standard functions on those structures.

6 Conclusions and Future Work

The remote data concept uses an existing communication topology to build direct connections between different processes. Existing bottlenecks are thereby circumvented and the total communication amount is reduced. Although being language-independent the concept enrolls its power and expressive elegance especially in the context of a declarative host language like Eden, where the concept

itself is implemented with small effort and only minor changes to existing code are needed to lift functional results to the new data type.

Algorithmic skeletons that define process networks with complex communication patterns can be defined in an elegant and concise way. Composition of skeletons with a remote data interface enables direct communication between processes within the different skeleton instances. Communication overhead is substantially reduced and skeleton compositions do not suffer anymore from the performance penalty caused by the collection and redistribution of distributed data in ordinary settings. Thus, the remote data concept enhances modularity of skeleton-based parallel programming, especially by promoting easily composable skeletons.

With remote data the explicit channel handling using `new` and `parfill` can in most cases be abandoned. This improves the the elegance and usability of Eden even more. We think that there is room for improvements in other parts of the language as well. One of the topics that has been brought up many times is the question whether the Eden functions should get an IO-interface or remain unchanged. We plan an intensive study of the benefits and drawbacks of an language specification that makes the side effects explicit.

References

1. M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, July 2007.
2. M. Alt and S. Gorlatch. Future-Based RMI: Optimizing compositions of remote method calls on the Grid. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 682–693. Springer-Verlag, Aug. 2003.
3. M. Alt and S. Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2004.
4. J. Berthold, M. Dieterle, and R. Loogen. Implementing parallel google map-reduce in eden. In H. Sips, D. Epema, and H. Lin, editors, *Euro-Par 2009*, volume 5704 of *Lecture Notes in Computer Science*, pages 990–1002. Springer-Verlag, 2009.
5. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
6. J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 19–28, New York, NY, USA, 1995. ACM.
7. H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.
8. O. Lobachev, J. Berthold, M. Dieterle, and R. Loogen. Parallel fft using eden skeletons. In *10th Intl. Conference on Parallel Computing Technologies (PaCT)'09*, volume 5698 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.
9. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.