Iterating Skeletons
Structured Parallelism by Composition

Mischa Dieterle[1](✉), Thomas Horstmeyer[1], Jost Berthold[2], and Rita Loogen[1]

[1] FB Mathematik und Informatik, Philipps-Universität Marburg, Marburg, Germany
{dieterle, horstmey, loogen}@informatik.uni-marburg.de
[2] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
berthold@diku.dk

Abstract. Algorithmic skeletons are higher-order functions which provide tools for parallel programming at a higher abstraction level, hiding the technical details of parallel execution inside the skeleton implementation. However, this encapsulation becomes an obstacle when the actual algorithm is one that involves iterative application of the same skeleton to successively improve or approximate the result. Striving for a general and portable solution, we propose a skeleton iteration framework in which arbitrary skeletons can be embedded with only minor modifications. The framework is flexible and allows for various parallel iteration control and parallel iteration body variants. We have implemented it in the parallel Haskell dialect Eden using dedicated stream communication types for the iteration. Two non-trivial case studies show the practicality of our approach. The performance of our compositional iteration framework is competitive with customised iteration skeletons.

1 Introduction

Modern hardware shows an increasing degree of parallelism at multiple levels. Graphics processing units (GPUs) and modern multicore CPUs offer numerous processing elements on one chip; cloud computing solutions promise to scale compute clusters up to previously inconceivable node counts with ease. It therefore becomes more and more difficult to effectively program these complex large-scale platforms at a convenient level of abstraction, especially when the programmer is not a parallelism expert. Research in parallel programming has developed a range of concepts and models for *skeleton-based parallel programming* to facilitate parallel programming and separate algorithm and parallelism concerns in this increasingly parallel computer landscape.

Algorithmic skeletons implement the parallel behaviour for applications of an algorithm class [4], represented directly as higher-order functions in functional languages. A concrete algorithm can be parallelised simply by applying the appropriate skeleton to function parameters which define the details of this algorithm, entirely hiding parallelism aspects in the skeleton implementation.

R. Hinze (Ed.): IFL 2012, LNCS 8241, pp. 18–36, 2013.
DOI: 10.1007/978-3-642-41582-1_2, © Springer-Verlag Berlin Heidelberg 2013

This approach of "parallel building blocks" constitutes a problem when the parallel algorithm involves iterations – applying the same skeleton repeatedly to successively improving data. Each skeleton incurs a certain overhead of thread and process creation, termination detection and communication/synchronisation. Repeatedly using one and the same skeleton leads to a repetition of this parallel overhead for every skeleton instantiation.

Example. Consider a simple *genetic algorithm* which computes the development of a population of individuals under some mutation until a termination criterion is met. The flowchart in Fig. 1 shows the iterated steps of the algorithm.

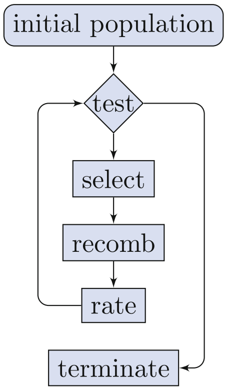


Fig. 1. Flowchart

```
type Individual = (Genome, Rating)
test   :: Individual → Bool       -- terminate?
select :: [Individual] → [(Genome, Genome)]
                          -- parents for next gen.
recomb :: (Genome, Genome) → [Genome]
                          -- generate offspring
rate   :: Genome → Individual     -- evaluation
```

A straightforward parallel version of the algorithm using recursion is listed beneath. It tests whether at least one individual of a given population fulfills the termination criteria. If not, genomes are `selected` based on their fitness (i.e. their relative rating) and paired as parents for the next generation. A parallel `map` implementation (`parMap`) is used to `recombine` the parents (already distributed into n sublists, one for each processing element (PE)) and `rate` the offspring – working on each sublist of the population in an own parallel process. The results of all processes are gathered and passed to a recursive call of the main function `ga`. The algorithm terminates when one of the new individuals passes the test.

```
ga :: [[Individual]] → Individual
ga pop = case (test_select pop) of
           Left parentss → ga $ parMap recomb_rate parentss
           Right solution → solution
test_select :: [[Individual]]
                → Either [[(Genome, Genome)]] Individual
recomb_rate :: [(Genome, Genome)] → [Individual]
```

In this parallel implementation, new `parMap` processes are created for each recursive call of `ga`. However, it would be much better to reuse processes, initialisation data, and communication channels across the different `parMap` invocations, especially when running the parallel program in a distributed environment. Also, if processes were reused, they would work on localised data and could even share a local state across the entire computation.

As the parallel behaviour is encapsulated inside a skeleton's implementation, it is generally very hard to optimise the repeated use of a skeleton without modifying the skeleton itself. On the other hand, a solution that involves rewriting parallel skeletons for every concrete sequence of applications is not favourable; we seek a more general method to compose skeletons for iterative computations, which we call *skeleton iteration*[1] subsequently.

Our Approach. We propose a general functional iteration scheme `iter` which is a meta-skeleton (combinator) using an iteration control and an iteration body function as parameters, and streams for exchanging data between both. Specific control and body functionality can be freely combined to express a wide range of iterative algorithmic patterns. We show how to lift ordinary skeletons in a systematic way to work on communication streams such that they can be used as iteration bodies in our iteration scheme. The central idea is to replace the repeated instantiations of the same body skeleton in an iteration with the single instantiation of a lifted body skeleton, the *iteration body*, which works on a stream of input values instead and produces a stream of output values. The control function transforms the output stream into the input stream which thus depends on the output stream, yielding a circular program, i.e. a program which uses such a self-referential data structure [3]. Each value on the input stream corresponds to an instantiation of the original skeleton, i.e. to an iteration step.

To improve programming comfort and safety we introduce special types for the communication streams as these replace iterative processing. Special support is provided for iteratively processing distributed data structures.

We have implemented our iteration framework in the parallel Haskell dialect Eden [9,10]. The functional approach makes it easy to precisely state interfaces and to identify conceptional requirements from our implementation. Using two non-trivial case studies, K-means and N-body, we compare the performance of implementations using our framework to that of straightforward recursion-based implementations, and, for K-means, to a monolithic customised `parMap` iteration skeleton [13]. The K-means case study shows that our framework performs much better than a straightforward recursion-based version with repeated process instantiation, and that it is competitive with the specialised monolithic skeleton. In the N-body case study, the framework-based implementation scales better and reduces overhead compared to the recursive version. However, when run on a small number of processors, the latter has slightly better overall runtimes.

In total, our skeleton iteration framework allows for targeted optimisations of iterative algorithms, with respect to minimising data transfers and controlling dependencies. It drastically improves code structure and readability and provides an acceptable performance with low effort.

Plan of Paper. In the next section, we introduce the proposed skeleton iteration framework gradually, starting with the Haskell prelude function for iteration. The performance evaluation follows in Sect. 3. Sections 4 and 5 provide a discussion of related work and conclusions.

[1] *Skeleton iteration* should not be confused with *parallel for-loops* or *parallel map*, where a sequential block is executed in parallel by multiple threads, instead of several times. We focus on computations defined by algorithmic skeletons which are by themselves already parallel and will be executed several times in sequence.

2 Iterating Skeletons

The Haskell prelude function `iterate` defines the iteration of a parameter function `f`, producing an infinite list (or: stream) of all intermediate results of the iteration: `[x, (f x), (f (f x)),...]`. The same stream can be defined in a self-referential way, using the `map` function and a feedback of the result stream instead of a recursive function call (this technique of *circular programs* has been used by Bird [3] to improve data structure traversals).

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)

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

We are especially interested in the case where the parameter `f` of `map`, which we call *iteration body* in the following, is a parallel skeleton, the *body skeleton*, i.e. when evaluation of `f` involves the creation of threads or processes and communication of data between them. Both the `iterate` function and the variant `streamIterate` above would in this case repeatedly construct and destroy the parallel process system evaluating `f` in every iteration step. As an illustrative example, consider the case where the body skeleton is a parallel `map` (`parMap`), i.e. creates one parallel process per input list element to apply a parameter function to it. The following specialised version of `streamIterate` implements this:

```
iterateParMap0 :: (a → a) → [a] → [[a]]
iterateParMap0 g xs = xss
  where xss = xs : map (parMap g) xss
```

Note that, in the result type of `iterateParMap0`, type `[[a]]` denotes a *stream of lists*, i.e. the outer list is infinite, while the inner lists are finite and computed in parallel (by the iteration body `parMap g` of type `[a] → [a]`).

As the iteration body `(parMap g)` is always the same, it would be desirable to use just one set of processes for all iterations, instead of creating a new set of processes in each step. This can be achieved by first transposing the input into a list of streams and then applying `parMap (map g)` to it; and finally restoring the original order with a second transposition. The transposition function `transposeS` fixes the length of its result list to the length of the first inner list of its input. This guarantees that `parMap` is applied to a finite list.

```
iterateParMap1 :: (a → a) → [a] → [[a]]
iterateParMap1 g xs = xss
  where xss = xs : transposeS (parMap (map g) (transposeS xss))
```

Now the iteration via `map` takes place within the processes created by `parMap` only once, saving the process creation overhead. In this simple example, it is sufficient to replace the iteration `map (parMap g)` with the composition

```
transposeS ∘ (parMap (map g)) ∘ transposeS.
```

It is by virtue of streaming and the use of `map` to express the iteration that we can lift the body skeleton `parMap` to work on streams and push the iteration (expressed by `map`) inside the processes. Just swapping `map` and `parMap` in the definition (leading to `parMap (map g) xss`) would instead lead to a pseudo-parallelisation over the *stream* instead of over the *lists*. In the absence of a distinction between lists (for parallelism) and streams (for iteration), types do not indicate this mistake. In the following, we will propose special types and mechanisms to generalise this approach and make a clear distinction between the iteration stream and the list of inputs to the parallel processes. We will also add special control functions for the iteration to improve locality and performance.

2.1 Iteration Type and Body

In this subsection, we introduce the iteration type used to distinguish between streams and lists and we show how to lift body skeletons, which can then be embedded in the iteration scheme discussed in the subsequent subsection.

Implementation Language and Process Types. We use the parallel Haskell dialect Eden to present our language-independent concept. Eden is geared towards distributed memory settings, but works equally well on shared memory system [10]. In Eden, the `parMap` skeleton

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

creates a parallel process for every element of the input list, which eagerly evaluates the application of the parameter function (mapping input of type `b` to output of type `c`). Processes are distributed among the available processing elements (PEs) (i.e. cores of a multicore or nodes of a compute cluster); and their inputs (the list elements) and process outputs (elements of the result list) are sent implicitly to and from these processes.

Communication-related properties of Eden processes are determined by types, using overloaded communication functions in the type class `Trans` for transmissible data. As a principle, data transmitted between processes will be evaluated to normal form prior to sending, which introduces additional strictness into Haskell in favour of parallelism. Furthermore, instances for `Trans` determine different send modes: while the default mode is to fully evaluate and send data as a single item, product types (tuples) can be decomposed and sent concurrently, and recursive types (such as lists) can be transmitted as streams, element by element. The important aspect here is that the type of a process determines the communication mode for its input and output data.

Special Stream Type for Iteration. In our `iterateParMap` definitions above, streams were modeled as lists, leading to a potential pseudo-parallelisation of the algorithm when parallelisation is applied at the outer level. In order to have a clear distinction of the (sequential) iteration stream and the (parallel) input to the iteration body, we introduce a special iteration type `Iter` (see Fig. 2), which is isomorphic to lists but different with respect to the communication mode. This

```
newtype  Iter a = Iter {fromIter :: [a]}
instance Functor  Iter where
    fmap f = Iter ∘ map f ∘ fromIter

distribWith :: (a → [b]^k) → Iter a → [Iter b]^k
distribWith f = map Iter ∘ transposeS ∘ map f ∘ fromIter

combineWith :: ([b]^k → a) → [Iter b]^k → Iter a
combineWith f = Iter ∘ map f ∘ transposeS ∘ map fromIter
```

Fig. 2. Iter type and auxiliary functions

enables the programmer and the type checker to identify iteration inputs and outputs in type signatures and thereby increases readability and type safety. Furthermore, the intended streaming behaviour can be defined in a targeted manner by an appropriate `Trans` instance for `Iter`, while other lists can be communicated as single items.[2] The functor instance of `Iter` provides `fmap`, lifting a function of type `a` $\longrightarrow$ `b` to iteration streams, `Iter a` $\longrightarrow$ `Iter b`.

Aside from the new data type, Fig. 2 shows auxiliary functions for common uses of `Iter` data when defining efficiently iterable skeletons. Function `distribWith` splits a single iteration stream into many iteration streams, where the ith element of each output stream is generated from the ith element of the input stream. The function parameter `f` produces a list of output elements for each element of the input iteration stream; these lists are then distributed into a list of output streams using `map Iter` $\circ$ `transposeS`. Consider the special case of `f = id`, which implies `a = [b]` and merely interchanges an outer `Iter` and an inner list. One subtle detail here is that `f` must produce lists of identical length k for all its arguments (elements of the iteration stream) as indicated by the superscript k of the list result type of `f`.[3] The number of output streams, which defines the parallelism degree, is determined by the first incoming stream element and thus equal to the size of the result lists of `f`, again indicated by superscript k in the list type. Finally, the function `combineWith` defines the inverse transformation.

Lifting Body Skeleton `parMap`. With these tools at hand, it is easy to define the efficient iterable version of `parMap` in a more readable and type-safe way (see Fig. 3). The lifted skeleton `simpleParMapIter` transforms inputs of type `Iter [b]`^{k}, i.e. streams of fixed-length lists, element by element into outputs of type `Iter [c]`^{k}. It creates k `map` processes, each transforming a stream of values of type `b` into a stream of values of type `c`. The auxiliary functions `distribWith` and `combineWith` are applied to the identity function `id` and thus reduce to type conversions and transpositions. Consequently, the behaviour of

[2] The original Eden definition specifies that top-level lists are communicated as streams. In this work, we use a modified `Trans` class which gives programmers more control of streaming through separate stream types.

[3] The superscripts in our types are merely annotations to indicate implicit constraints on the list lengths. Fixed sized lists could however be implemented e.g. using the recent Haskell library `Vec`, see http://hackage.haskell.org/package/Vec

```
simpleParMapIter :: forall b c. (Trans b, Trans c)
                    ⇒ (b → c) → Iter [b]^k → Iter [c]^k
simpleParMapIter f bss = css where
  bss' = distribWith id bss    :: [Iter b]^k
  css' = parMap (fmap f) bss'  :: [Iter c]^k
  css  = combineWith id css'
```

Fig. 3. Parallel map as an iteration body

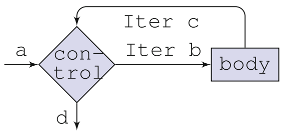
`simpleParMapIter` corresponds to the iteration body of `iterateParMap1`: `transposeS ∘ (parMap (map g)) ∘ transposeS`.

In `iterateParMap1`, the output stream was simply fed back into the iteration body. Instead, an iteration control function should be used to decide about termination. In the following, we propose an iteration scheme which combines an iteration body, i.e. a lifted body skeleton, with such an iteration control.

2.2 Iteration Scheme and Iteration Control

A Generic Iteration Scheme. *Iteration control* links together the output and input iteration streams of the body skeletons, to produce new input and decide termination. The body skeleton's input stream must be started with initial data, and the result stream must be conditionally fed back to the body skeleton, or terminated by closing the input stream and returning a final result. This can be defined in terms of the following generic iteration scheme:

```
simpleIter :: (a → Iter c → (Iter b,d)) --control
           → (Iter b → Iter c)   --body
           → a → d               --in/out
simpleIter control body a = d where
  (iterB,d) = control a iterC
  iterC     = body iterB
```



The meta-skeleton `simpleIter` takes two function parameters: an *iteration control* function which produces initial input and handles the two loose ends of the iteration stream, also determining the final result, and an *iteration body* function. While not restricted to it, the iteration body is typically an iterable skeleton like `simpleParMapIter`. All parallelism is encapsulated in these two parameter functions, `simpleIter` only deals with the interconnection, and thereby provides a very liberal interface to combine iteration control and body functions.

Iteration Control Functions. The iteration body is allowed to transform input of type `Iter b` to a different type `Iter c`. Thus, output cannot be fed back directly by the control function, but needs to be transformed back from `Iter c` to `Iter b`, in an element-wise fashion. The *iteration control function* must be carefully defined to ensure progress in the circular iteration scheme. It has to provide the initial input for the iteration body, it needs to check a termination condition, and to produce the final output from the iteration body's output upon termination. Two common examples for iteration control functions are `loopControl`, which performs exactly *n iterations* by forwarding *n* inputs without any transformation, and whileControl, which takes a function parameter checkNext to transform the initial input and iteration output of type a to a new iteration input of type b (Left alternative). It stops the iteration with a result of type d (Right alternative). The lazy patterns ~(...:_) in both control functions are necessary because the corresponding pattern matching can only be performed after the final iteration step. Note that the rest stream matching the underscore pattern _ is empty. Both control functions ensure progress because they provide their second argument a as initial input and essentially pass the elements of the output stream (or at least parts of them) to the input stream until the number of iterations is reached or the termination condition is fulfilled.

```
loopControl :: Int → a → Iter a → (Iter a, a)
loopControl n a (Iter as) = (Iter as', a') where
  (as',~(a':_)) = splitAt n (a:as)

whileControl :: (a → Either b d) → a → Iter a → (Iter  b,d)
whileControl checkNext a (Iter as) = (Iter $ lefts bs, d) where
  (bs,~(Right d:_)) = (break isRight ∘ map checkNext) (a:as)
```

In whileControl, the parameter function checkNext only considers the output of a single iteration step to decide termination or to compute the input for the next step. The general control function type in simpleIter is much more liberal, in fact it is not even required that the control function generates exactly one iteration body input for each iteration body output. Often, it appears more suitable to use a *state-based* control function like the one shown here:

```
whileControlS :: (a → State s (Either b d)) → s
                → a → Iter a → (Iter b, d)
```

Its first parameter function is a state transformation for a single iteration step, thereby combining safety (i.e. guaranteed progress) and flexibility. We have implemented generic stateful control functions and used them in our measurements, but present our work in terms of the stateless interface due to space constraints.

Running Example. The genetic algorithm presented earlier is an example of a parallel map iterated with a conditional control function:

```
gaBody    :: Iter [[(Genome, Genome)]]^k → Iter [[Individual]]^k
gaBody    = simpleParMapIter recomb_rate

gaControl :: [[Individual]]^k → Iter [[Individual]]^k
               → (Iter [[(Genome, Genome)]]^k, Individual)
gaControl =  whileControl test_select

gaIter    :: [[Individual]]^k → Individual
gaIter    = simpleIter gaControl gaBody
```

The iteration body is constructed from `recomb_rate` by `simpleParMapIter`, and iteration control uses the `test_select` function inside `whileControl`. Function `simpleIter` combines `gaControl` and `gaBody` to implement the genetic algorithm with parallel recombination and rating.

2.3 Performance Tweaking

The main potential for optimisation of iteration steps lies in reducing communication overhead. One obvious bottleneck is that data is gathered in the control function and then redistributed to the iteration body in each step. One approach to optimise communication is to *keep all data distributed* between the iterations. In Eden, this can be done using remote data [6]. We can create a remote data handle from local data and fetch the data remotely using functions:

```
release :: Trans a ⇒ a → RD a
fetch   :: Trans a ⇒ RD a → a
```

When data is `release`d, an intermediate data handle of type `RD a` is created, which can be forwarded between several processes at negligible communication cost, until the destination process `fetch`es the real data. `release` and `fetch` establish a direct connection between a producer and a consumer process.

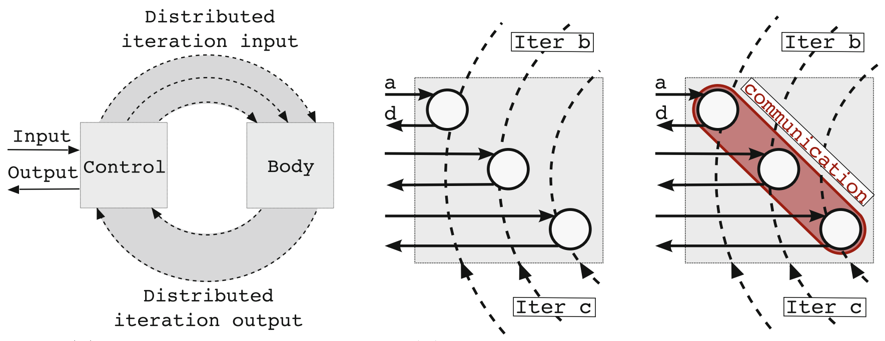
In our scenario of iterative algorithms, termination can often be decided from only a small fraction of data, while most of the data remains unmodified across several iteration steps. When the iteration body's inputs and outputs are lifted to remote data, data will be passed directly from the output of a process to its input for the following iteration step. It is straightforward to define a variant of the `simpleParMapIter` skeleton for remote data, by lifting its parameter function to the remote data interface:

```
parMapIterRD :: (Trans b, Trans c)
                ⇒ (b → c) → Iter [RD b]^k → Iter [RD c]^k
parMapIterRD f = simpleParMapIter (release ∘ f ∘ fetch)
```

This variant can now be combined with control function `loopControl n` to iterate a computation n times on input (already supplied as remote data), and data will never be gathered and re-distributed in-between the iteration steps. In every iteration step, input for each process will be `fetch`ed for local processing using function `f`, and `release`d afterwards, only to be fetched within one and the same process in the next iteration step. Other control functions, like e.g. `whileControl`, need to gather data in-between iteration steps to decide termination and provide input for further iteration steps. Therefore, a *parallel iteration control* skeleton should be used to achieve locality and save communication without compromising abstraction by a manual decomposition of iteration data.

2.4 Parallel Iteration Control Skeletons

In many cases where the iteration body uses a skeleton to work on distributed data, a corresponding *control skeleton* with parallel processes can be used to



(a) Iteration scheme (b) local control (c) global control

Fig. 4. Parallel iteration control

```
localControl :: (Trans a, Trans b, Trans c, Trans d)
  ⇒ (a → Iter c → (Iter b, d)) -- process-local control
  → [RD a]^k                    -- initial Input
  → Iter [RD c]^k               -- output of loops
  → (Iter [RD b]^k, [RD d]^k)   -- input for loops, final result
```

Fig. 5. Local iteration control skeleton

inspect the distributed data, exchanging only the parts of it that are needed globally (see Fig. 4a). In addition, corresponding processes of control and body can be placed on the same processing element to avoid communication.[4] This concept can be used with arbitrary distributed data structures, in our implementation we focus on the special case of iterations over distributed lists (lists of remote data). Two different types of parallel iteration control can be distinguished: *local* and *global* iteration control, with respect to the data dependencies within the control processes.

Local Iteration Control means that no data exchange with other control processes is necessary – data dependency is *local*, as depicted in Fig. 4b. Otherwise, a *global* data exchange is necessary, as depicted in Fig. 4c. The type of a local iteration control skeleton for lists of remote data is given in Fig. 5. The implementation is similar to the implementation of `parMapIterRD`, but takes the two input values and the tuple output into account. The control processes will connect both to their predecessor processes that produce the distributed list beforehand and to the processes of the iteration body, fetching data on-demand, or else passing on the RD handles. Functionality in each process is described by the process-local control function which transforms the initial input and the output of a process in the iteration body (stream-wise) in the respective control process. This skeleton can implement several common iteration control variants simply by partially apply-

[4] Eden supports explicit placement of computations in a multi-node parallel system. We have omitted placement aspects from our code for simplicity throughout.

ing the control skeleton to a suitable control function. For example, a variant of `whileControl` where termination can be decided from local data would be:

```
localWhileCtrl :: (a → Either b d) →
                  [RD a]^k → Iter [RD a]^k → (Iter [RD b]^k,[RD d]^k)
localWhileCtrl checkNext = localControl (whileControl checkNext)
```

The control function `checkNext` works on the local part of a distributed list (of type `[RD a]`), and either produces input for the next iteration or the final output (again a distributed list).

Global Iteration Control. If the control function needs information from multiple processes to calculate the next input for the body or to determine termination, the processes of the control skeleton need to exchange these data. As an example of this kind of control skeleton, consider an *all-gather* pattern where all processes gather selected data from all other processes in a distributed manner (see Fig. 6). We only discuss the signature of the skeleton here:

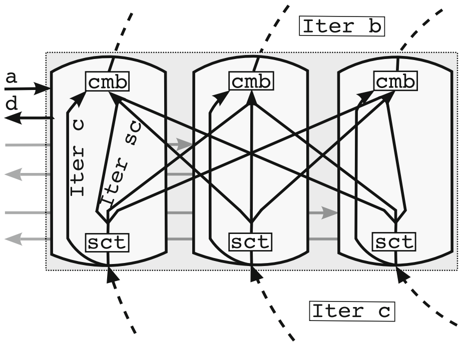


Fig. 6. all-gather control

```
allGatherControl::(Trans a, Trans b, Trans c, Trans d, Trans sc)
  ⇒ (a → Iter c → Iter sc)                               --select
  → (Int → a → Iter c → Iter [sc]^k → (Iter b, d))       --combine
  → [RD a]^k → Iter [RD c]^k → (Iter [RD b]^k, [RD d]^k) --controlType
```

Aside from the iteration body output (distributed list of type `[RD c]`, iterated), the input for the next iteration and the final result (distributed lists `[RD b]` and `[RD d]`) depend on additional synchronisation data (of type `sc`, iterated). Combine function `cmb` produces the local next input and result, but considers the entire list of synchronisation data (iterated) and the own position in the list of processes (`Int`). Select function `sct` yields the local synchronisation data which will be communicated to all other control processes.
A skeleton `allGatherWhileCtrl` can be defined as a specialisation of skeleton `allGatherControl` with simpler interface, where type `a`=`c`.

```
allGatherWhileCtrl :: (Trans a, Trans b, Trans d, Trans sc)
  ⇒ (a → sc)                                             --select
  → (Int → a → [sc]^k → Either b d)                       --combine
  → [RD a]^k → Iter [RD a]^k → (Iter [RD b]^k, [RD d]^k) --controlType

allGatherWhileCtrl sct cmb = allGatherControl sct' cmb' where
  sct' a (Iter as) = Iter $ map sct (a:as)
  cmb' self a (Iter as) (Iter scss) = (Iter $ lefts bs,d) where
    (bs,~((Right d):_)) = break isRight $
                            zipWith (cmb self) (a:as) scss
```

The select and combine function of this skeleton work on single elements of the iteration stream. The encoding of the termination condition in `cmb` is similar to the simple `whileControl` function presented in Sect. 2.2.

Running Example. The genetic algorithm described earlier needs to consider the entire population to decide about termination (`test`) and produce input for the next iteration step (`select`). Therefore, it uses a global control variant when implemented with parallel iteration control.

```
gaBodyRD :: Iter [RD[(Genome,Genome)]]^k → Iter [RD[Individual]]^k
gaBodyRD = parMapIterRD recomb_rate

gaControlRD :: [RD [Individual]]^k → Iter [RD [Individual]]^k
            → (Iter [RD [(Genome,Genome)]]^k, [RD Individual]^k)
gaControlRD = allGatherWhileCtrl id cmb where
  cmb self _ pop = case test_select pop of
                     Left  next → Left $ next !! self
                     Right res → Right res

gaIterRD :: [RD [Individual]]^k → Individual
gaIterRD = head ∘ fetchAll ∘ simpleIter gaControlRD gaBodyRD
```

Iteration control is constructed from `allGatherWhileCtrl`, broadcasting the *local* population (`sct=id`) to all sibling processes, such that every process can use the whole *global* population in function `cmb`. The latter calls `test_select` to either terminate (yielding `Right res`) or produce the next input (`Left next`) for *all* body processes. Each process then selects (by `!! self`) its own next input from the list.

2.5 Inlining the Iteration Streams

Up to now, we derived the type `Iter` and with `iterSimple` the signature of iterated skeletons. We introduced remote data to achieve direct communication among processes and used streams of parallel inputs (`Iter [RD x]`) to connect the processes of iteration control and body. This has two drawbacks: (1) In the skeleton definitions, we have to drag the iteration stream from the outside of the iterated list to its elements. (2) The channel connections between the processes of the body and the control skeleton have to be rebuilt in every iteration step. Instead of having a stream of parallel inputs, we will use parallel input streams, leading to type `[RD (Iter x)]`. The transpositions implied by `distribWith` and `combineWith` are now obsolete. Further, streams of data will be communicated over remote data connections established only once. The following `parMap` variant with modified interface implements these static remote data connections:

```
parMapIter :: (Trans b, Trans c)
              ⇒ (b → c) → [RD (Iter b)]^k → [RD (Iter c)]^k
parMapIter f = parMap (release ∘ fmap f ∘ fetch)
```

Notice that we can express the iterable skeleton simply by transforming the function parameter. We observed that the transformation of more complex topology skeletons, such as `allToAllRD` and `allReduceRD` (both developed in the context

of remote data [6]), are similarly easy, only involving the respective function parameters (all transformations done by the nodes are function parameters to these skeletons).
The iteration streams to and from all processes have to be processed by a control function or skeleton which exactly matches the particular distributed data shape. This constraint can be fulfilled by adjusting the type signature of `simpleIter` to reflect the change from a stream of parallel inputs to a parallel input stream:

```
iterD :: (a → [RD (Iter c)]^k → ([RD (Iter b)]^m,d))        --control
         → ([RD (Iter b)]^m → [RD (Iter c)]^k)              --body
         → a → d                                            --in/out
iterD = ...                                    -- code from simpleIter
```

We need to define specialised versions of local and global iteration control correspondingly, which again are simplifications of the existing implementations.

2.6 Unifying the Interface

The adjusted signature of `iterD` of the last section is not compatible with the `simpleIter` function, even though their implementations are identical. It is easy to specify a more general type for the iteration combinator,

```
type generalIter = (a → iterC → (iterB,d))
                   → (iterB → iterC)
                   → a → d
```

but we lose type safety when dropping the type of the `Iter` streams. This problem can be addressed using a type family which describes iteration types used to interconnect iteration control and body. We want to have special instances for distributed data types. As an example we define a special type for distributed finite lists.

```
type family Iterated a :: *

newtype DList a = DList [RD a]  --Distributed List
type instance Iterated (DList  a) = DList (Iter a)
```

The distributed list type `DList a` is defined, containing a list of remote data which represent the distributed elements of type `a`. Exchanging the iteration stream and the distribution by `[RD _]` is now done automatically in the type instance for `DList` of the `Iterated` type family, which yields `DList (Iter a)` – isomorphic to type `[RD (Iter a)]`. Other distributed data types and `Iterated` instances can be defined in the same way, e.g. distributed trees or matrices.

We use the simple type mapping `type instance Iterated a = Iter a` to define the types of iterations for ordinary types. It is not possible to allow overlapping instances for type families, so we have to define these instances for every base-type

```
iter :: (b → c                           --b/c to typecheck Iterated b/c
           → a → Iterated c → (Iterated b,d))            --control
         → (Iterated b → Iterated c)                      --body
         → a → d                                          --in/out
iter iterControl iterBody a = d where
  (iterB,d) = control undefined undefined a iterC
  iterC     = body iterB
```

Fig. 7. General iteration skeleton

separately. Quite advisedly, we have defined `DList a` as `newtype`, so an instance for lists can be defined without overlapping `Iterated (DList a)`:

```
type instance Iterated [a] = Iter [a]
```

The type family approach enables us to finally define a generic but type-safe iteration skeleton `iter` (see Fig. 7) which subsumes all previously presented definitions. It works for both `DLists` and for any other reasonable type instance of `Iterated`. A small caveat is that two dummy parameters `b` and `c` need to be used in the control function, in order for the typechecker to check the types `Iterated b` and `Iterated c`. This is needed to determine the types, because the type family mapping might not be injective.

3 Evaluation

We measured the performance of our iteration framework on a 32 node Beowulf cluster at the Heriot-Watt University Edinburgh, each node with 2 x 4-core@2.00 GHz Intel Xeon E5504 processors, connected by gigabit Ethernet. Eden runtime system instances were co-located on the nodes to make use of all processor cores (which we further refer to as processors). The cluster provides a total of 256 processors. However, as it could not be used exclusively, measurements are limited to a maximum of 128 processors. All program versions where tested on 2^i processors with i ranging from 0 to 7. The reported runtimes are mean values of 5 program runs. They are presented in diagrams with logarithmically scaled axes, with runtimes corresponding to a linear speedup indicated by dotted lines. In the following we present measurement results for two non-trivial case studies: k-means and n-body.

K-means clustering is a heuristic method to partition a given data set of n d-dimensional vectors into k clusters. In an iterative approximation, the method identifies clusters such that the average distance (a metric such as the euclidian or Manhattan distance) between each vector and its nearest cluster centroid is minimal [12]. The algorithm proceeds as follows: (1) randomly choose k vectors from the data set as starting centroids, (2) define clusters by assigning each vector to the nearest centroid, (3) compute the centroids of these clusters, and (4) repeat the last two steps until the clusters do not change anymore. The iteration body takes a list of cluster centroids as input and computes the list of new centroids as output. The iteration continues until two subsequent iteration results are equal or their differences fall below a threshold. The cluster assignment and part of the centroid computation can be parallelised using `parMap`. Each parallel process receives a subset of the vectors, and the whole list of centroids. Every process then computes a list of weighted sub-centroids which are combined to the list of new centroids by the iteration control.

We measured the runtimes of this parallel k-means algorithm with a data set of 600000 vectors and $k = 25$ cluster centroids. The whole computation comprised 142 iterations. Three different implementations were compared:

– *recursive parMap* is a naïve implementation which creates new processes and re-distributes not only the centroids but also the (unchanged) list of vectors in each iteration step. As the parallel processes are newly created for each step, there is no way to share the vector list across iterations.
– *untilControl/simpleParMapIter* uses our iteration scheme with *stateful* versions of `untilControl`[5] and `simpleParMapIter`. Only the centroids are gathered and distributed for each iteration step, while the data vectors are once distributed and then kept in the worker states during the iteration.
– *monolithic iterUntil* uses a *special monolithic* iteration skeleton `iterUntil` presented in [13]. Like the composed version above, it uses a stable process system and holds the data set in local states. While a perfect match for the parallel k-means, other iterative algorithms would require a complete re-design and re-write of the skeleton.

Figure 8 shows the mean runtimes plotted against the number of processors. The modular skeleton *untilControl/simpleParMapIter* performs as well as the specialised *monolithic iterUntil* version. Both scale well, showing an almost linear speedup up to 8 processors. On more than 32 processors, initialising and distributing the vectors increasingly influences runtime, leading to lower speedup.

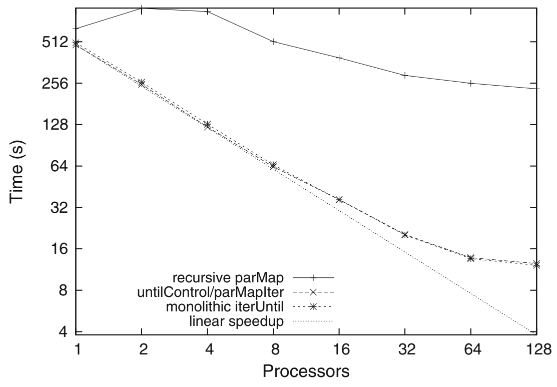


Fig. 8. Runtimes for k-means with 600000 vectors, 25 clusters, 142 iterations

The naïve *recursive parMap* version performs dramatically worse. The overhead of distributing the vectors for every iteration enormously slows down the computation.

N-body. The n-body problem is to simulate the movement of n particles in a 3-dimensional space, taking into account their mutual gravitational forces. In a

[5] Similar to `whileControl`, but forwards the initial input directly to the iteration body, thus doing at least one iteration before termination.

straightforward parallel N-body algorithm, particles are distributed to processes and each process computes the new velocity and position for its own particles. To update its particles' velocities, each process needs position and mass (but not velocity) of all other particles. This information needs to be exchanged in-between the iterations, leading to considerable communication between the parallel processes, in contrast to the parallel k-means algorithm described earlier.

We have used variants of the skeleton `allToAll` to parallelise the iteration body. Each process holds a subset of the particles and processes exchange particle information in every iteration step in a distributed manner using the all-to-all topology. We implemented the following versions:

– *recursive allToAllRD* recursively instantiates the skeleton `allToAllRD`. As the corresponding processes are allocated on the same processor in each iteration, all data transfers occur between processes on the same processors. The Eden runtime system optimises this processor-local communication by passing references to existing data instead of serialising and sending it. That is, processor-local communications do not incur any overhead. Thus, the only remaining overhead consists of the repeated process creations.
– *loopControl/allToAllIter* instantiates our iteration scheme with the skeletons `loopControl` for the iteration control and `allToAllIter` (`allToAllRD` lifted to the `Iter` type) for the body.

In the first setting, we ran the n-body simulation for 10 iterations with 15000 bodies. This constitutes a relatively high workload and large amounts of data have to be exchanged in every iteration. Runtimes against number of processes are plotted in Fig. 9.

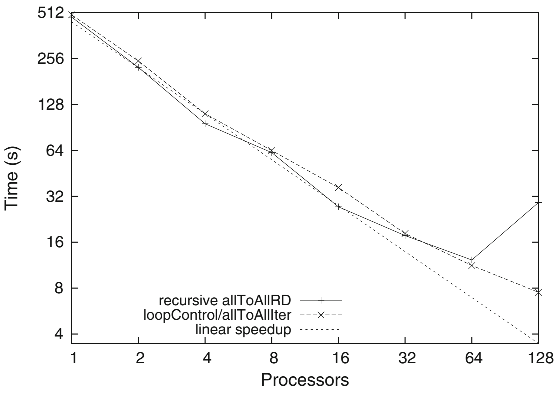


Fig. 9. Runtimes for n-body with 15000 bodies, 10 iterations

Surprisingly, the *recursive allToAllRD* version performs slightly better on up to 32 processors, showing even a super-linear speedup on 2 and 4 processors. Only on 64 and 128 processors, the *loopControl/allToAllIter* version is faster than the recursive version. An analysis of runtime behaviors revealed that the *recursive allToAllRD* has no disadvantage in the communication steps due to the optimised local communications, but the computation phases seem to be shorter, although sharing the same sequential code base with the iteration scheme version. Pending further investigation, we assume that the differences originate from the runtime system, maybe the garbage collection does not work as effectively for the data streams in our iteration scheme version. In any case, our measurements show that the *loopControl/allToAllIter* version scales better than the recursive version.

In the second setting (Fig. 10), we reduced the workload and amount of data to be communicated for every iteration step, in order to measure the parallelism overhead. We used only 1500 bodies but increased the number of iteration steps to 100. This time, version *loopControl/allToAllIter* clearly outperforms the recursive version independent of the number of processors.

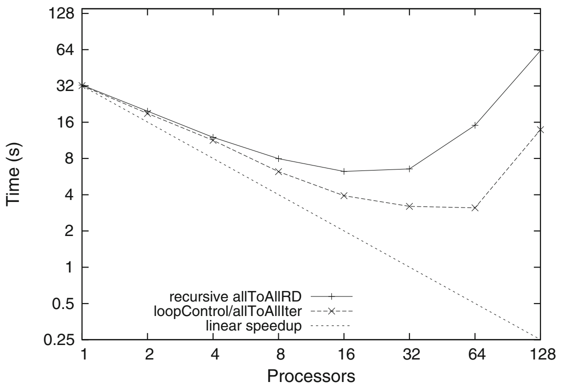


Fig. 10. Runtimes for n-body with 1500 bodies, 100 iterations (overhead measurement)

4 Related Work

The original skeleton work by Murray Cole [4] contains a chapter on an iterative completion, parallelised on a grid of processes, but does not generalise iteration as we do. Slightly more general is the iteration skeleton proposed in earlier Eden skeleton work [13], realising an iteration of a stateful parallel `map`. This work lays the grounds for our investigation, but does not generalise iteration bodies and types, nor does it consider parallel control skeletons.

Many skeleton libraries, especially those based on imperative programming languages, provide the constructs `while` for conditional iteration or `for` for fixed iteration and support skeleton nesting, see e.g. the Scandium library [11], which uses Java as computation language. However, no indications are made about whether iterated body skeletons will be optimised with respect to process creation overhead. A slightly larger corpus of related work can be found in the cloud computing community but usually restricted to map-reduce [1,5] computations, like e.g. [7,14]. HaLoop [2] is another Map-Reduce extension, which mainly capitalises on caching mechanisms for unmodified data and reduction results across several iterations of one map-reduce computation over the same dataset. A small API extension is provided to specify how existing map-reduce (Hadoop) computations should be iterated.

None of these publications addresses parallel iteration as a general concept or distills out algorithmic patterns as we do. This generalising conceptual angle is present in very recent work in the data-flow framework Stratosphere [8]. The authors propose the concept of "incremental" iteration and "microsteps" to exploit sparseness of data dependencies and optimise read-only data accesses, but thereby break up the iterative nature of the computation.

5 Conclusions and Future Work

Iteration is one of the main building blocks of programming. In this work, we developed a general approach to describing iteration that works not only in the common sequential setting but also in the case where the iterated computation is highly parallel and executed in a distributed setting. We allow for arbitrary parallel body skeletons and supply some parameterised control functions including step counting and termination conditions on local and global data. We have shown how body skeletons can be transformed in such a way that the body processes will be re-used for all iterations, how to handle streams of input and output data, and how to optimise communication between distributed processes in a parallel execution. Runtime measurements for two non-trivial example applications, k-means and n-body, clearly show that our framework performs similar to monolithic iteration skeletons and better than directly programmed iterations where the iterated skeletons are repeatedly instantiated.

We will further investigate the field of skeleton composition in the future. In particular, we plan to extend the work at hand by adding other distributed data structures, to augment programming comfort for such distributed data by suitable indexed types and type classes.

Acknowledgements. The authors thank the anonymous referees for their helpful comments on a previous version of this paper. Jost Berthold was partially supported by DSF under contract number 10-092299 (Hiperfit).

References

1. Berthold, J., Dieterle, M., Loogen, R.: Implementing parallel google map-reduce in Eden. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 990–1002. Springer, Heidelberg (2009)
2. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB J. **21**(2), 169–190 (2012)
3. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Inform. **21**, 239–250 (1984)
4. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. CACM **51**(1), 107–113 (2008)
6. Dieterle, M., Horstmeyer, T., Loogen, R.: Skeleton composition using remote data. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 73–87. Springer, Heidelberg (2010)
7. Ekanayake, J., Li, H., Zhang, B., Gunarathne, Th., Bae, S., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: HPDC '10. ACM (2010)
8. Ewen, St, Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. PVLDB **5**(11), 1268–1279 (2012)
9. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. J. Funct. Program. **15**(3), 431–475 (2005)

10. Loogen, R.: Eden – parallel functional programming with Haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) CEFP. LNCS, vol. 7241, pp. 142–206. Springer, Heidelberg (2012)
11. Leyton, M., Piquer, J.M.: Skandium: multi-core programming with algorithmic skeletons. In: PDP. IEEE Computer Society (2010)
12. MacKay, D.: Information Theory, Inference, and Learning Algorithms. Cambridge University Press, Cambridge (2003). See chapter 20, p. 284ff
13. Peña, R., Rubio, F.: Parallel functional programming at two levels of abstraction. In: PPDP'01, pp. 187–198. ACM (2001)
14. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. JOGC **10**, 47–68 (2012)