JFP **26**, e11, 40 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000083

Skeleton composition versus stable process systems in Eden

M. DIETERLE, T. HORSTMEYER and R. LOOGEN
Fachbereich Mathematik und Informatik, Philipps-Universität, Marburg, Germany
(*e-mail:* dieterle@informatik.uni-marburg.de, horstmey@informatik.uni-marburg.de, loogen@informatik.uni-marburg.de)

J. BERTHOLD
Commonwealth Bank of Australia, Sydney, Australia
(*e-mail:* jberthold@acm.org)

Abstract

We compare two inherently different approaches to implement complex process systems in Eden: *stable process systems* and a *compositional* approach. A *stable process system* is characterised by handling several computation stages in each of the participating processes. Often, processes communicate using streams of data, change behaviour with the different computation phases, and more often than not, exactly one process is allocated to each processor element. In contrast, a complex process system can also be achieved by *skeleton composition* of a number of elementary skeletons, such as parallel transformation, reduction, or special communication patterns. In a compositional implementation, each computation phase leads to a new set of interacting processes. When implementing complex parallel algorithms, skeleton composition is usually easier and more flexible, but has a larger overhead from additional process creation and communication. We present case studies of different parallel application kernels implemented as stable systems and using composition in Eden, including a comprehensive description of Eden's features. Our results show that the compositional performance loss can be alleviated by co-locating processes which directly communicate, and by using Eden's remote data concept to enable such direct communication. Moreover, Eden's parallel runtime system handles communication between co-located processes in an optimised way. EdenTV visualisations of execution traces are invaluable to analyse program characteristics and for targeted optimisations towards better process placement and communication avoidance.

1 Introduction

With the advent of massively parallel hardware like GPUs and specialised manycore CPUs, functional approaches to parallel programming have received increased attention. Efficient parallel data processing is a crucial foundation for many modern application areas, and parallel functional programming is increasingly accepted as one of the most promising approaches to productive parallel programming.

Because of their mathematical nature, functional languages like Haskell (Haskell, 2010) are excellent tools for reasoning about patterns and skeletons of parallel processing. Purely functional languages are explicit about all possible side-effects

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

and specify the intended computation at an abstraction level suitable for algorithmic reasoning and optimisations. The approach of *algorithmic skeletons* (Cole, 1989) has largely similar goals, and adds a generalising algorithmic aspect: An algorithmic skeleton captures an algorithmic structure and its inherent parallelism as a higher order function realising its parallelism (semi-) automatically. Algorithmic skeletons are therefore particularly suited for a functional implementation. Skeletons can be easily provided in a library (of higher order functions) when implemented in a functional language.

The Haskell dialect Eden (Loogen *et al*., 2005) can both express a skeleton implementation (lower library level) and be used directly for application programming, which makes it easy for programmers to switch role. In many cases, parallel applications can be built simply by choosing an appropriate skeleton from Eden's skeleton library, or possibly by composing several skeletons for different subcomputations of more complex algorithms. Haskell's purity encourages modularity and composition of functionality from small well-tailored building blocks. However, these possibilities raise the question of which abstraction level is appropriate for skeleton-based parallel programming. When several parallel skeletons are combined into a larger algorithmic pattern, the granularity of each process will decrease, and compared to a monolithic implementation, a compositional one will incur an overhead of additional communication and synchronisation between them.

Up to now, Eden's philosophy has always been to create stable process topologies with a one-to-one mapping of processes to processor elements (PEs). Already in 1998, Breitinger stated in her Ph.D thesis (Breitinger, 1998) that *"For the implementation of an iteration algorithm, it is particularly important that the processes involved are created only once and not for every iteration anew. Eden differs from implicitly parallel functional languages in that it makes the specification of such stable process systems possible"*.

Consequently, typical algorithmic skeletons have been developed in Eden as stable process systems, such as an iteration skeleton `iterUntil` in Loogen *et al.* (2003). This skeleton uses a stable master-worker system to implement an iterated parallel `map` skeleton: A set of worker processes evaluates the iteration body in parallel, and a controlling master process supplies the input and collects the workers' results. The communication between the master and the workers takes place via stream channels, enabling a reuse of the worker processes across all iteration steps. As soon as the termination condition is fulfiled, the master process finishes the evaluation and closes the stream channels, terminating the worker processes. More recently, we developed a more general skeleton iteration framework (Dieterle *et al*., 2013) where arbitrary skeletons can be iterated and the master process is replaced with more general iteration controls, all of this on the basis of static process networks.

On the other hand, a remote data concept for Eden had been developed to support skeleton composition (Dieterle *et al*., 2010) and shortcut communication between process siblings in composed skeleton setups. Remote data handles are passed between composed skeletons to establish direct data communication between corresponding processes. This is especially important in a distributed setting because the distributed result of one skeleton does not need to be collected and re-distributed

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

for the subsequent skeleton but can simply remain distributed and/or passed directly to the locations where it is needed. Such local communication has since been further optimised in the Eden runtime system (RTS), enabling co-located processes to actually *share* data instead of communicating it. It turns out that this change has important consequences on the underlying cost models which guide the monolithic optimisations like the `iterUntil` described above.

In Dieterle *et al.* (2010), the idea of composing skeletons using the remote data concept was introduced and some elementary building blocks like parallel reduction and all-to-all communication were defined as skeletons with remote data interface. However, the paper did not contain any more elaborate case study, and was not supported by the underlying RTS optimisation. Later, during development of our iteration framework, a comparison of the framework and a recursive compositional implementation using remote data had a surprising result: despite the fact that the recursive version would instantiate a new set of worker processes for each iteration step, its performance was competitive — sometimes better — than the stable monolithic framework version. This contradicts the optimality assumption about monolithic process networks, and motivated the systematic investigation, comparing recursive and monolithic skeleton instantiations through systematic case studies in the paper at hand.

The focus of this paper is on programming methodology. It continues and extends work presented in the previous publications (Dieterle *et al.*, 2010; Dieterle *et al.*, 2013) , comparing the two different approaches that were previously advocated, in several case studies:

1. We discuss the parallel sorting algorithm PSRS (Parallel Sorting by Regular Sampling) (Li *et al.*, 1993), implemented using a composition of simple `parMap` skeletons or using a larger monolithic `allToAll` skeleton from the Eden skeleton library;
2. we contrast a compositional and a monolithic implementation of the conjugate gradient method; and
3. we analyse the different implementations of the n-body problem, which was the original motivator for this paper.

Using these case studies, we show that complex parallel algorithms can elegantly be implemented in Eden using composition of skeletons with remote data interface, and show that the performance of compositional implementations can match that of sophisticated monolithic skeletons with stable process systems.

The following section presents all features of Eden necessary to understand the case studies and skeletons we present. Subsequently, Sections 3.1 through 3.3 present the case studies. The PSRS algorithm composes transposition and parallel `map`, while the other two case studies iteratively execute an inner parallel skeleton to approximate a solution. Section 3.4 summarises the results of the case studies. Section 4 discusses related work, Section 5 concludes.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

2 Eden: parallel Haskell using explicit processes

Eden (Loogen *et al.*, 2005) extends Haskell (2010) with a small set of syntactic constructs for explicit process specification and creation. While providing enough control to implement parallel algorithms efficiently, it frees the programmer from managing tedious low-level details by introducing automatic communication (via head-strict lazy lists), synchronisation, and process handling.

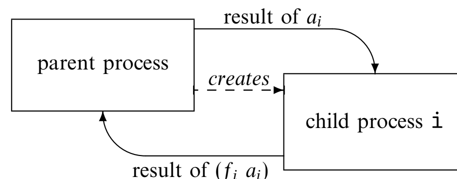
In this section, we explain the main Eden constructs relevant for skeleton programming and composition: eager creation of parallel processes using function `spawnF`, Eden's type-based communication mechanisms and philosophy, and the *remote data concept*, which forms the basis for passing distributed data between different skeleton instances. Also, we give a short overview of Eden's skeleton library and explain the analysis of Eden programs using the Eden Trace Viewer EdenTV. All Eden programs have to import the module `Control.Parallel.Eden` which provides definitions of all Eden constructs as well as providing the type class `Trans` of *transmissible data types*.

2.1 *Eager process creation with* `spawnF`

```
spawnF :: (Trans a, Trans b) ⇒ [a → b] → [a] → [b]
```

The `spawnF` function, when applied to a list of functions $[f_1, \dots, f_n]$ and a list of arguments $[a_1, \dots, a_m]$. creates a series of $k = (min\ n\ m)$ processes, where the ith process, $1 \leqslant i \leqslant k$ receives the value of a_i as input, evaluates the application $(f_i\ a_i)$ and outputs its result. `spawnF` returns the list of all process outputs. Either of the argument lists may be infinite if the other one is finite.

On process creation, implicit one-to-one communication channels between the parent process (evaluating the `spawnF` application) and the newly created child processes are installed. For each child process, its argument expression a_i is evaluated in the parent process by a newly created concurrent thread, and then communicated via a channel of type `a` to the corresponding child processes. Vice versa, each child process will evaluate the application of its function f_i to the received argument a_i and send it back to the parent process via a corresponding channel of type `b` (details will be explained in Subsection 2.2).



Eden's `spawnF` must change Haskell's lazy evaluation principle in order to parallelise a computation. Lazy evaluation is inherently sequential and would otherwise lead to distributed sequentiality instead of true parallelism. Therefore, `spawnF` eagerly creates all child processes irrespectively of the actual demand for their outputs. Furthermore, all inputs and outputs of Eden processes are evaluated to normal

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

form before being transferred via communication channels. Both design decisions overrule laziness in support of parallelism and must be used carefully in order to avoid unnecessary computations and overhead.

All communication happens automatically using appropriate communication functions provided via the type class `Trans`. The type of any data that is exchanged between processes must therefore be an instance of the type class `Trans`. The creation of processes and of communication channels, data transfer between parent and child processes, as well as the suspension of processes when their input is needed but not yet available will be done automatically by the Eden parallel RTS.[1]

2.1.1 *Process placement*

In Eden's RTS, PEs are numbered from 1 to the number of PEs. This number is available to the program as constant `noPe :: Int`. Furthermore, the number of the PE on which a process is executed is available as constant `selfPe :: Int`. In the simple `spawnF` version presented above, new processes are placed evenly in the system by a local round-robin scheme on each PE, starting with `selfPe`+1. While this is a convenient default, and sufficient for simple parallel applications, more advanced parallel programs with many processes often benefit massively from explicitly placing processes on specific PEs, to achieve an even work-load on all PEs and to optimise communication by co-locating processes that will communicate. For this purpose, Eden provides a version `spawnFAt` which includes explicit placement.

```
spawnFAt :: (Trans a, Trans b) ⇒
            [Int] → [a → b] → [a] → [b]
```

This variant takes an additional parameter of type `[Int]` which specifies the PE numbers on which the newly created processes should be placed. The list of PE numbers is used modulo `noPe` plus 1, and with wraparound if it is shorter than the two other argument lists. Values of `0` and an empty list act as defaults to fall back to the RTS-supported round-robin placement (PE 0 does not exist), whereby: `spawnF = spawnFAt [0] = spawnFAt []`.

2.2 *Type-driven communication: The type class* `Trans`

Arguments and results of an Eden process are generally evaluated to normal form before they are sent to their destination. Therefore, process instantiation is roughly equivalent to hyper-strict function application from a denotational perspective. However, two special communication modes exist, which introduce implicit concurrency for top-level tuples and stream communication for lists:

1. *Tuples* are evaluated by concurrent threads, one per component, which allows processes to concurrently produce several independent outputs.
2. *Lists* are transmitted as streams, one (fully evaluated) element at a time.

[1] Eden has been implemented by extending the Glasgow Haskell Compiler (GHC)(GHC, 1991–2015), see (Berthold & Loogen, 2007).

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

The type class Trans implements the Eden process communication by implicitly used communication functions which are overloaded for these two special communication modes.

Implicit concurrency and stream communication, together with Haskell's lazy evaluation, allow for the definition of recursive process networks and processes connected by infinite data streams, such as a process ring.

Example: A simple process ring could be defined as follows:

```
ring0        :: (Trans i, Trans o, Trans r) ⇒
                [(i,r) → (o,r)] → [i] → [o]
ring0 fs is =  let (os,outrs) = unzip $ spawnF fs (zip is inrs)
                   inrs       = last outrs : init outrs
               in  os
```

Each ring process takes a pair of inputs: an input from the parent process and a ring input from its predecessor in the ring. It produces a pair of outputs: an output for the parent and a ring output for the successor in the ring. All ring processes are eagerly created using spawnF. The ring communication is achieved by using the right-rotated list of ring outputs as ring inputs.[2] In most applications, the type of data passed on the ring will carry a stream, i.e. r will be a list type. ◁

The send function defined in Trans will evaluate the data to be communicated to normal form before packing and sending it. Therefore, a data type can only be instance of Trans if the data can be evaluated to normal form, i.e. if the type is an instance of NFData. Trans instances for many standard data types are pre-defined in the Eden module. Note that there will be exactly one thread per process outport, where an outport connects a sender process to a communication channel. Eden works with *push communication*, i.e. values are communicated as soon as available.

2.2.1 Chunking

Although streaming of lists is important for Eden's expressivity, in most cases, it is too expensive to send every list element in a single message. A well-known approach to throttle the number of messages is stream chunking, i.e. the stream is transformed into a stream of subsequent sublists using the following function chunk:

```
chunk           :: Int → [a] → [[a]]
chunk size [] =  []
chunk size xs =  ys : chunk size zs
                 where (ys,zs) = splitAt size xs
```

This works, because only the outermost list is sent as a stream. The corresponding dechunk function is simply concat :: [[a]] → [a] from the Haskell prelude, which flattens a list of lists into a single list by concatenating all sublists. The most appropriate chunk size is problem-dependent; therefore, list chunking cannot be automated without any further analysis.

[2] The alert reader might have noticed that with the above definition, the "ring communication" will happen indirectly via the parent process which passes the ring output of each process to its successor process. We will later introduce the remote data concept which can be used to install direct channel connections between ring processes.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

2.2.2 Optimised communication between processes

By default, data sent over an Eden channel are serialised and communicated between PEs, and then deserialised by the receiver, which creates a copy. However, communication between processes which are placed on the same PE is optimised, the two processes will share the same copy instead of exchanging messages and serialising data. This optimisation is essential for an effective composition and iteration of skeletons.

2.3 Remote data concept

During process creation, channels are only installed between parent and child processes. This implies that only process trees can be built. In order to enable the generation of general process topologies like rings or grids, Eden supports a special way of installing direct communication channels between sibling processes, or generally between processes which are not parent and child: the remote data concept.

Remote data in Eden (Dieterle *et al.*, 2010) is provided with the following API:

```
type RD a   -- remote data

-- convert local data into remote data
release :: Trans a  ⇒ a → RD a

-- convert remote data into local data
fetch :: Trans a  ⇒ RD a → a
```

The main idea of remote data is to replace data communication between processes with an exchange of handles to data (called remote data). These handles can be used to *fetch* the real data directly to the desired target. Remote data of type `a` is represented by a handle of type `RD a` with interface functions `release` and `fetch`. Function `release` creates a remote data handle that can be passed to other processes, which will in turn use the function `fetch` to access the remote data. When data are fetched, the releasing process will transmit the data completely automatically, in a separate implicit Haskell thread.

The following simple example illustrates how the remote data concept is used to establish a direct channel connection between sibling processes.

Example: Given functions `f` and `g`, the expression `((g ∘ f) inp)` can be calculated in parallel[3] by creating a process for each function. However, simply replacing the function calls by process instantiations:

```
spawnF [g] ∘ spawnF [f] $ [inp]
```

leads to the process network in Figure 1(a). Process `main` (which is assumed to evaluate the above expression) instantiates child processes calculating `f` and `g`. It sends the input `inp` to the process calculating `f`. When the result of `f inp` has been received by `main`, it is passed on to the process calculating `g`, which will likewise

[3] In fact, the two function applications will of course be evaluated in distributed sequentiality; unless `(f inp)` has a list type, in which case one gets a two-stage *pipeline*.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

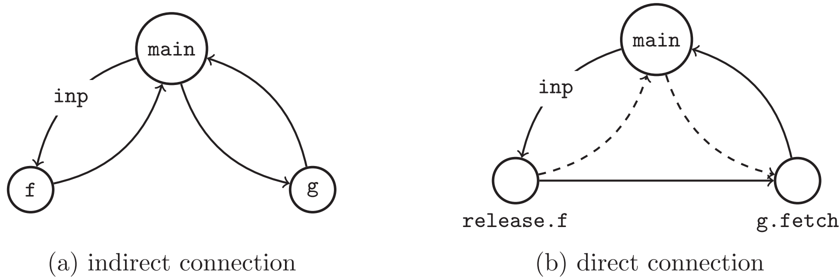


Fig. 1. Process topology (a) without and (b) with remote data.

send its result back to `main`. Obviously, the result of the process calculating `f` is not sent directly to the process calculating g, thus causing unnecessary communication costs.

To achieve direct communication between the sibling processes, we can use remote data (see Figure 1(b)):

```
spawnF [g ∘ fetch] ∘ spawnF [release ∘ f] $ [inp]
```

The output produced by the process calculating `f` is now encapsulated in a remote handle that is passed to the process calculating g, and fetched there. Note that the remote data handle is treated like the original data in the first version, i.e. it is passed via the main process from the process computing `f` to the one computing g. ◁

Example: Using the remote data concept, we can modify the simple process ring defined above to establish direct communication channels for the ring communication:

```
ring :: (Trans i, Trans o, Trans r) ⇒
        [(i,r) → (o,r)] → [i] → [o]
ring fs is
  = let (os,outrs) = unzip $ spawnF fsRD (zip is inrs)
        inrs       = last outrs : init outrs
        fsRD       = map lift fs
    in  os

lift :: ((i,r) → (o,r)) → (i, RD r) → (o, RD r)
lift f (i, h_inr) = let (o, outr) = f (i, fetch h_inr)
                        h_outr    = release outr
                    in  (o, h_outr)
```

The ring functions are lifted to use remote data in the second, i.e. the ring component. The lifted function takes a handle and fetches the input data from the ring. The ring output is released and the handle is returned. ◁

An important restriction is that remote data handles can only be used once to fetch remote data. If data will be fetched by several processes, several different handles have to be released to the different processes. These handles can e.g. be produced by replicating the data and calling `(map release)` on the replicated data. Convenience functions `releaseAll` and `fetchAll` are provided, which are eager versions of `(map release)` and `(map fetch)`, respectively:

```
releaseAll :: [a] → [RD a]
fetchAll   :: [RD a] → [a]
```

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

2.4 Eden skeleton library

Eden's programming methodology is based on skeletons (Cole, 1989) which define general parallel computation schemes like parallel maps and master-worker systems, divide-and-conquer schemes, or communication topologies like pipelines, rings, torus, hypercubes, or grids. In Eden, skeletons can simply be defined as parallel higher order functions. The `ring` skeleton above is an example of a topology skeleton. A well-known elementary skeleton is the parallel `map` implementation `parMap` which creates a process for each application of the parameter function to an element of the input list. The `parMap` function can be defined easily using `spawnF`:

```
parMap   :: (Trans a, Trans b) ⇒
            (a → b) → [a] → [b]
parMap f =  spawnF (repeat f)
```

Note that `parMap` is more eager than the standard `map`. The input list should be finite to guarantee a finite number of processes, and the input and the output list will both be evaluated to normal form.

Eden's skeleton library `edenskel`[4] consists of several modules:

Control.Parallel.Eden.Map contains several parallel implementations of `map :: (a→b) → [a] → [b]` as e.g. the simple `parMap` defined above.

Control.Parallel.Eden.MapReduce provides simple parallel versions of the composition of a `foldr` or `foldl` and a `map`.

Control.Parallel.Eden.Workpool considers workpool processing, i.e. replicated-worker skeletons which implement a dynamic task distribution in contrast to the parallel map implementations which distribute input values or tasks statically to the worker processes.

Control.Parallel.Eden.DivConq provides several divide-and-conquer skeletons, which differ in the way parallelism is established. Mainly, two different parallel divide-and-conquer versions are distinguished. The *distributed expansion scheme* creates processes for recursive calls during the recursive unfolding of the divide-and-conquer scheme while the *flat expansion scheme* does the recursive unfolding up to a given depth and then uses a parallel map skeleton for the parallel evaluation of all recursive calls of the divide-and-conquer scheme on this depth.

Control.Parallel.Eden.Topology defines several skeletons that implement a network of processes interconnected by a characteristic communication topology. Among others, there are pipeline and ring skeletons, a torus skeleton, an all-to-all skeleton and an all-reduce skeleton.

Control.Parallel.Eden.Iteration provides the skeleton `iterUntil` which implements iterative algorithms where the iteration body is evaluated in parallel by a number of worker processes and a master process decides whether to finish the evaluation or to start another round.

[4] available on hackage: `http://hackage.haskell.org/package/edenskel`

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

2.5 Skeleton composition

The remote data concept is the key concept for an efficient composition of skeletons in Eden. Skeletons with a remote data interface receive handles which they can use to fetch their input data immediately from the processes producing the data. In the same way, their results are not returned as data, but just as handles which can be used to fetch the output data. Thus, with a remote data interface between composed skeletons, data can directly be transferred between communicating processes of subsequent skeleton instantiations.

2.5.1 Composing `parMap` and `parRed`

The library `Control.Parallel.Eden.MapReduce` contains simple parallel map-reduce skeletons, which typically split the input list into as many sublists as PEs available (constant `noPe`). For each sublist, a parallel process is created which does a sequential map-reduce on this sublist. Finally, the results of all processes are collected in the parent process where a final reduction is done. This simple two-stage parallel reduction is sufficient for small numbers of PEs. In large systems, the sequential reduction in the parent process may quickly lead to a bottleneck.

The parallel reduction skeleton `parRed` has been designed to fold data that is distributed among a set of processes using a set of processes in a tree scheme, i.e. it starts combining the data from pairs of processes and continues pairing and combining until the final reduction result has been computed by the root process. The skeleton's interface uses remote data, taking a list of remote data handles as arguments and delivering the result again as a remote data handle.

```
parRed :: (Trans a) ⇒
          (a → a → a) →       -- Reduction function
          a →                 -- neutral element
          [RD a] → RD a       -- Input → Output
```

The `parMap` skeleton can simply be lifted to return remote data handles instead of the data itself:

```
parMapRD    :: (Trans a, Trans b) ⇒
               (a → b) → [a] → [Rd b]
parMapRD f =   parMap (release ∘ f)
```

Using normal function composition, these skeletons can be composed to define a parallel map-reduce skeleton which does a tree-like parallel reduction:

```
parMapRed         :: (Trans a, Trans b) ⇒
                       (b → b → b) → b →
                       (a → b) →
                       [a] → b
parMapRed g e f =   fetch ∘ (parRed g e) ∘ (parMapRD f)
```

Each `parMap` process releases its data (i.e. returns a handle), which will be fetched by one of the `parRed` processes. The `parRed` skeleton returns a handle to the result which must be fetched by the main process.

Example: Consider the simple recursive divide-and-conquer definition of the well-known mergesort algorithm given in Figure 2. Empty or singleton lists are al-

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
mergeSort         :: (Ord a, Show a) ⇒ [a] → [a]
mergeSort []      =  []
mergeSort [x]     =  [x]
mergeSort xs      =  sortMerge (mergeSort xs1)  (mergeSort xs2)
                            where   [xs1,xs2] = unshuffle 2 x

-- unshuffle :: Int → [a] → [[a]]
-- imported from Control.Parallel.Eden.Auxiliary

sortMerge :: (Ord a, Show a) ⇒ [a] → [a] → [a]
sortMerge xs [] = xs
sortMerge [] ys = ys
sortMerge l1@(x:xs) l2@(y:ys) | x ≤ y     = x : sortMerge xs l2
                              | otherwise = y : sortMerge l1 ys
```

Fig. 2. Haskell Mergesort program.

ready sorted. Lists with more than two elements will be split into two halves by `unshuffle 2`. The sublists are sorted by recursive calls of `mergeSort`. Finally, the sorted result lists are merged into the sorted result list using the auxiliary function `sortMerge`:

A parallel version of mergesort can be defined using the `parMapRed` skeleton defined above:

```
parms   :: (Ord a, Show a, Trans a) ⇒
              Int →       -- number of sort processes
              [a] → [a]   -- sorting function
parms  np xs
  =  parMapRed sortMerge [] mergeSort $ unshuffle np xs
```

The input list is divided into `np` sublists in a round robin manner. The parallel map-reduce skeleton takes the function `mergeSort` for the sublist sorting by the parallel map processes and the function `sortMerge` for the merging of the sorted sublists by the parallel reduction processes. For simplicity, we have omitted chunking and dechunking of process inputs and outputs. Figure 3 illustrates the resulting parallel process network for `np`= 8. The dotted lines indicate the passing of remote data handles via the main process, to establish the direct data connections from the map processes to the reduction processes. ◁

2.6 *EdenTV*

When developing parallel programs, tool support for tracing parallel program behaviour is essential for program analysis and performance tuning. Very often, the parallel execution can yield unexpected interleavings and one can easily create single hotspots in a parallel system. To support tracing for optimisation and debugging, Eden's parallel RTS has been instrumented to produce execution trace files containing sequences of important events during program execution, such as process or thread creation, exchange of messages, and garbage collection (Berthold & Loogen, 2008). Modern GHC versions since version 6.12.3 support tracing of multithreaded program executions in a similar way, which can be analysed with the tool ThreadScope (Jones *et al.*, 2009). Eden's and GHC's tracing solutions share the

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

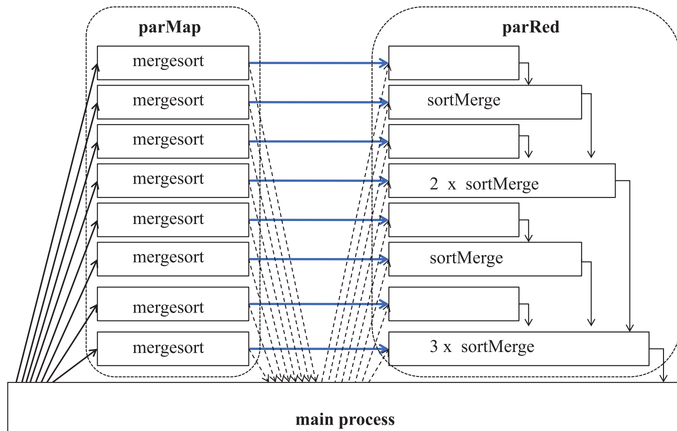


Fig. 3. MergeSort process system using compositional `parMapRed` skeleton.

same trace data format and infrastructure today. GHC's trace files can be visualised by EdenTV to analyse thread activities.

Eden's parallel RTS creates `noPe` instantiations of the sequential GHC RTS, called logical PEs or machines, which will be mapped on the physical processing elements by the operating system or by the used middleware (MPI or PVM). In most cases, the programmer will create as many logical PEs as physical cores or processing nodes are available, although this is not mandatory. An Eden program will generate processes which are allocated to the different (logical) PEs and which comprise several threads to compute the process outputs. Thus, there are three levels of execution units: logical machines — processes — threads.

The Eden Trace Viewer (EdenTV)[5] can visualise and analyse trace files produced by Eden's parallel RTS. EdenTV generates activity profiles (space-time diagrams with time on the x-axis) with horizontal bars representing the respective displayed execution units (machines (i.e. PEs), processes or Haskell threads). The machines diagram (see e.g. Figure 4(a)) shows PE utilisation over time. The diagrams for processes and threads (see e.g. Figure 4(c)) show the activity of the different processes and threads (including internal system threads which run in their own 'virtual' process). An additional 'processes per machine' diagram (see e.g. Figure 4(b)) shows the same bars as the processes diagram but groups them according to their placement on the machines.

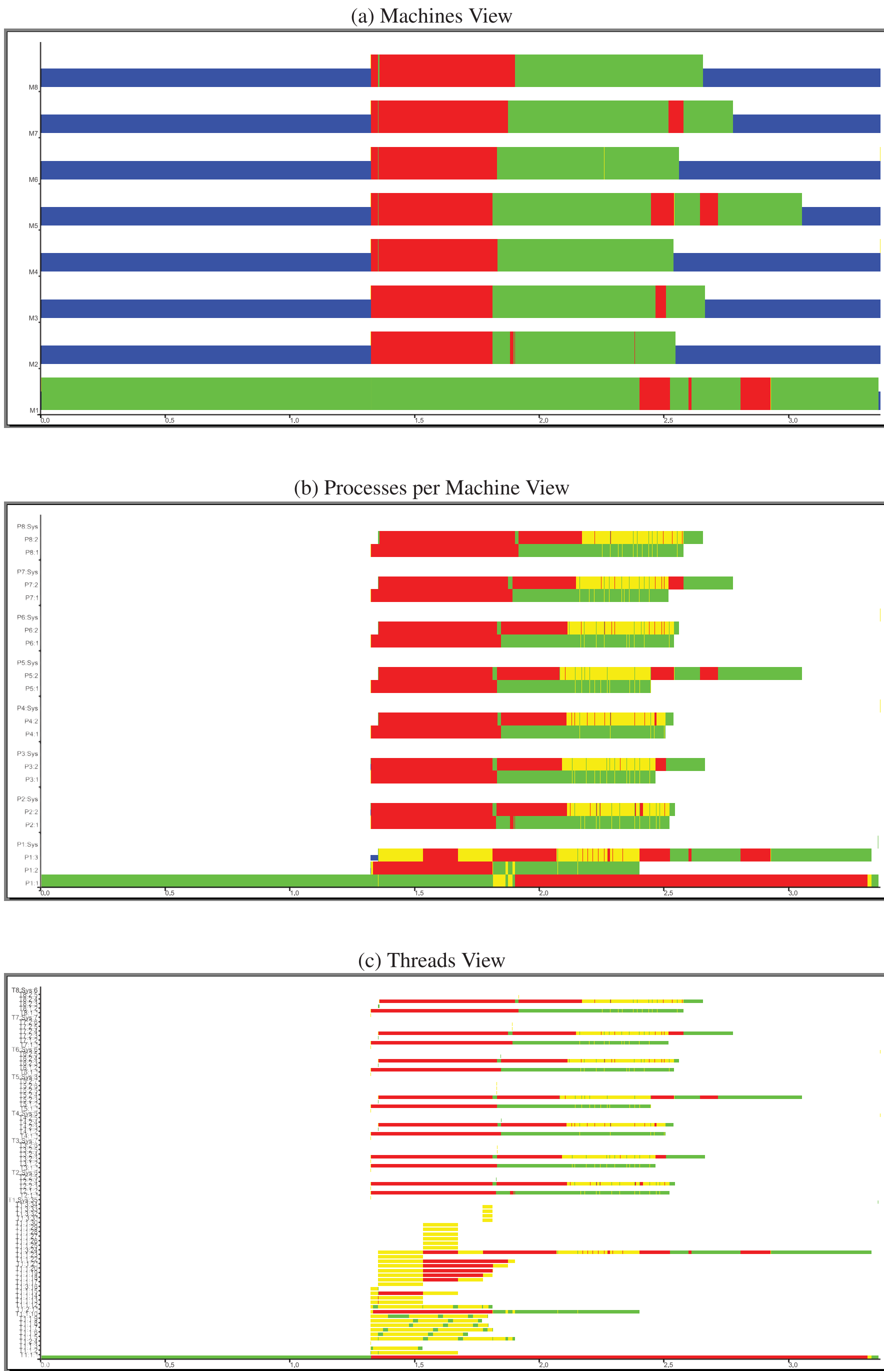
The diagram bars have segments in different colours, which indicate the activities of the respective execution unit (machine, process, or thread) over time during execution. Bars for threads and processes are:

- green (gray), when the logical unit is running;
- yellow (light gray), when it is runnable but currently not running; and
- red (dark grey), when the unit is blocked.

In addition, a machine can be idle which means that no processes are allocated to the machine. Idleness of machines is indicated by a small blue bar. The thread states

[5] available on hackage: `http://hackage.haskell.org/package/edentv`

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press



(d) Additional Statistical Information

Runtime: 3,368630s, 8 Machines, 17 Processes, 81 Threads, 96 Conversations, 2475 Messages

Fig. 4. EdenTV activity profiles and statistics.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

are immediately determined from the thread state events in the traces of processes. The states of processes and machines are derived from the information about thread states.

Figure 4 shows three different EdenTV activity profiles of a run of the above map-reduce mergesort program (plus chunking). One million random numbers have been sorted, chunks of size 1,000 have been used in streams. In addition to the different activity profiles, EdenTV provides some statistical information as shown in Figure 4(d). Apart from the overall runtime, this tells the number of machines, processes, threads, conversations, and messages. The difference between conversations and messages is that message streams are counted as a single conversation but as many messages as have been sent in the stream. If there are no message streams, the number of conversations and messages will be identical. In our example program, the number of conversations (96) is much less than the total number of messages (2,475). This shows immediately that communicated lists in the program have been streamed.

The machines view (Figure 4(a)) shows that the program has been executed on eight PEs. The lowest bar corresponds to the main PE, numbered 1, which executes the main process. The computation starts with a sequential phase where only PE 1 has been active. PEs 2 to 8 initially run idle (small blue bars). In this initial phase, the input list of random numbers is computed by the main process on PE 1. After about 1.3 seconds PEs 2 to 8 start working, but initially, all processes on these PEs are blocked. The processes-per-machine diagram (Figure 4(b)) reveals that, in addition to system processes on each PE and the main process on PE 1, there are two processes on each PE. From the program code it is clear that one map process (lower bar) and one reduce process (upper bar) have been allocated to each PE. The map and reduce processes are created almost at the same time which is due to Haskell's outermost evaluation strategy. Map processes are created first because they deliver the input for the reduce processes. Accordingly, they start running before the reduce processes which are initially blocked waiting for data from the map processes. The threads diagram (Figure 4(c)) shows most details. There are 81 threads, 35 of them on PE 1. Seventeen long-lasting threads compute the main output of the 17 processes. The other threads evaluate the inputs of all child processes. In this paper, we work mainly with the processes-per-machine view, because this view reflects the programmer's process-oriented view of Eden programs. Sometimes it is also instructive to inspect the thread view. In Figure 4(c), there are e.g. the eight threads just above the lowest (main) thread which compute and send the inputs for the eight map processes. Most of the time they are runnable (yellow) and there are small running phases (green) which alternate between the eight threads according to the thread scheduling. This detail shows that there is a bottleneck in the delivering of input for the map processes which explains the initial blocking phase of the map processes.

In order to analyse the communication between processes and machines, it is possible to overlay messages/streams on the corresponding processes and machines activity files. Figure 5 shows streams and messages overlays in the machines and processes-per-machine views, zoomed in the parallel program phase. Streams and

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

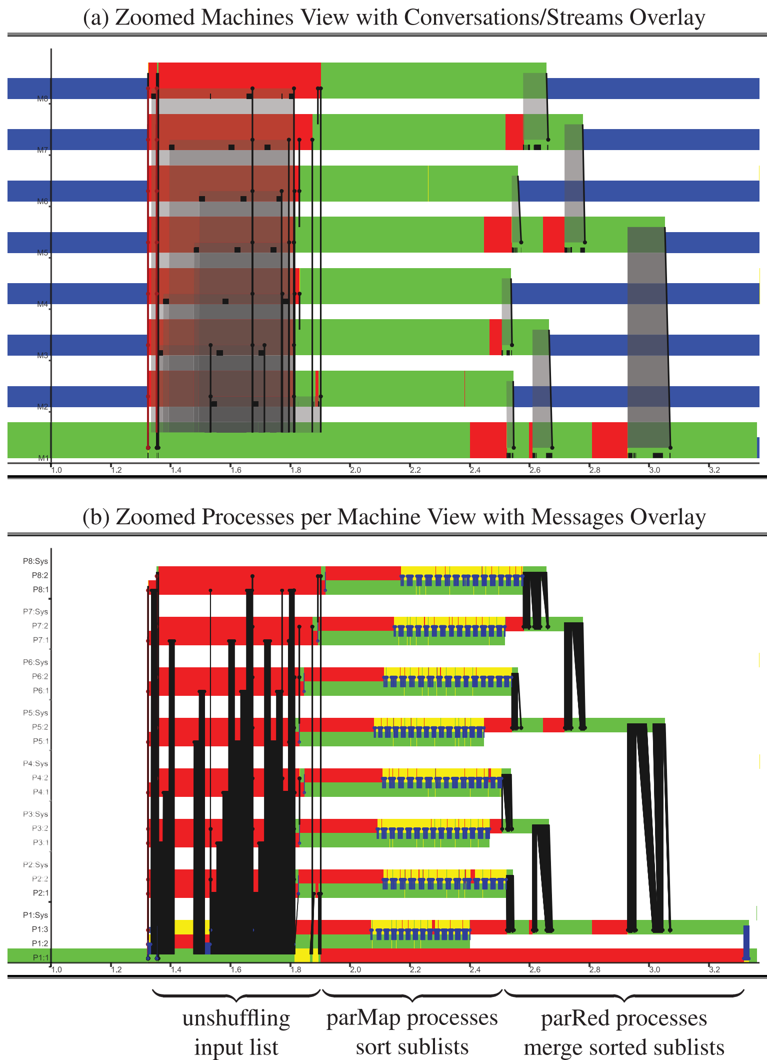


Fig. 5. Zoomed EdenTV views with streams and message overlays.

messages overlays can only be shown for machines and processes, and not for threads, because only processes and not threads exchange messages in Eden. In the conversations overlay, message streams are indicated as shaded areas while in the messages overlay, there is a vector (black line with end marked with a dot) from the sender to the receiver process or machine for each message that has been sent. Note that process creation messages are shown as red vectors while data messages between processes located on the same machine are shown in blue and all other data messages are shown in black.

The messages overlays show more clearly that there are three program phases apart from the sequential start phase (see the underbracing in Figure 5): when the child processes have been started, the main process, or more precisely the eight threads within the main process computing the input for the map processes, is busy

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

unshuffling the list of random numbers into eight sublists and passing chunks of these sublists to all PEs (including itself, as PE 1 is not reserved for the main process in this version). Then, all map processes sort the received sublists and pass the sorted sublist to the reduce processes allocated to the same PE, using optimised communication shortcuts (indicated by vertical short blue vectors). Finally, the reduce processes perform a parallel reduction following the common recursive doubling scheme. Please note the close correspondence between the process scheme of the mergesort program in Figure 3 and the processes-per-machine activity profile with message overlays in Figure 5.

3 Case studies

In this section, we consider three case studies with which we investigate the performance hit caused by using skeleton composition instead of stable process systems. The first case study shows that complex parallel algorithms and process networks can easily be defined in Eden using skeleton composition with remote data interfaces. The further case studies compare the implementation of parallel iterative algorithms with recursive skeleton instantiations and stable process systems. The iteration case studies are a conjugate gradient computation and, again, the n-body simulation already considered in Dieterle *et al.* (2013). All program runs analysed in this paper have been executed on the following platform:

> 4 × AMD Opteron[tm] Processors 6378 with 16 Cores, 16 MB L3-Cache at 2.4 GHz, and 64 GB DDR3 SDRAM, 1,600 MHz

3.1 Finite composition: Parallel sorting by regular sampling (PSRS)

In the area of distributed parallel sorting, PSRS: Parallel Sorting by Regular Sampling (Li *et al.*, 1993) is a specialised version of mergesort aimed at good scaling properties. Its complexity is optimal, $O(\frac{n \cdot \log(n)}{p})$, if the number n of values to be sorted is greater than p^3, where p is the number of available PEs. The PSRS algorithm takes a distributed unsorted list and produces a distributed sorted list. Therefore, the distribution of the input list from one source and the collection of the result lists to one destination is not a necessary part of the sorting algorithm — in contrast to parallel mergesort, which performs a non-trivial reduction with `sortMerge` when collecting the workers' results. This property ensures that the PSRS algorithm can be efficiently composed with other skeletons for distributed data processing.

Assuming that input is provided in p segments of equal size distributed on p PEs, PSRS consists of four phases:

1. In parallel: Each process sorts one segment and selects a sample of p elements;
2. The main process collects and sorts all p^2 samples (p samples from each process), selects $(p-1)$ pivot elements and broadcasts them to all processes;
3. In parallel: Each process decomposes its segments into p partitions (according to the pivot elements) and sends partition j to process j $(1 \leqslant j \leqslant p)$, keeping one partition;

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
psrs :: forall a. (Trans a, Ord a)
                  ⇒ Int → [RD [a]] → [RD [a]]
psrs p =
   --(4) merge partitioned, presorted lists
   parMapAt [2..p+1] (release ∘ mergeAll ∘ fetchAll)

    ∘ transpose  -- assign partitions to corresponding processes

   --(3) partition presorted lists by global samples
    ∘ parMapAt [2..p+1] (releaseAll ∘ partition)

   --(2) gather local-, derive & distrib. global samples
    ∘ uncurry zip ∘ first processSamples ∘ unzip

   --(1) sort segments and get local samples
    ∘ parMapAt [2..p+1] (sortNSample ∘ fetch)
 where
  sortNSample :: [a] → ([a], RD [a])
  sortNSample cs = let ys = sort cs
                   in (getSamples p ys, release ys)

  processSamples :: [[a]] → [[a]]
  processSamples = replicate p ∘ getGlobalSamples p ∘ mergeAll

  partition :: ([a], RD [a]) → [[a]]
  partition (pivots, handle)
    = decompose pivots ∘ fetch $ handle
```

Fig. 6. PSRS in Eden.

4. In parallel: Each process merges $p-1$ partitions received from siblings with its own.

3.1.1 Compositional definition of PSRS in Eden

The PSRS algorithm can be implemented straightforwardly in Eden, using the remote data based composition technique and the `parMap` skeleton. Figure 6 shows the essentials of the implementation. The four phases of PSRS correspond exactly to the functions composed in the top-level definition.

In Phase 1, the `parMapAt` processes `fetch` their remote input data, sort it locally, and return to the parent process both a list of samples and a handle to the sorted data. The parent process gathers all samples and extracts the global pivots (a sample of all samples), which are then distributed back to the other processes for Phase 3 (as the first tuple component). The data handles are left untouched in-between the two `parMap` instances (second tuple component); they only forward the data between the different co-located mapper processes. In Phase 3, each parallel process partitions its sorted local data according to the global pivots, and creates a list of p remote handles, one for each of the p partitions.[6] These handles are returned to the parent process, which transposes the matrix of $p \times p$ handles and returns them

[6] This includes a handle for the partition that should be kept, making extra code for this special case unnecessary. Due to the optimised local communication, this incurs no overhead.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

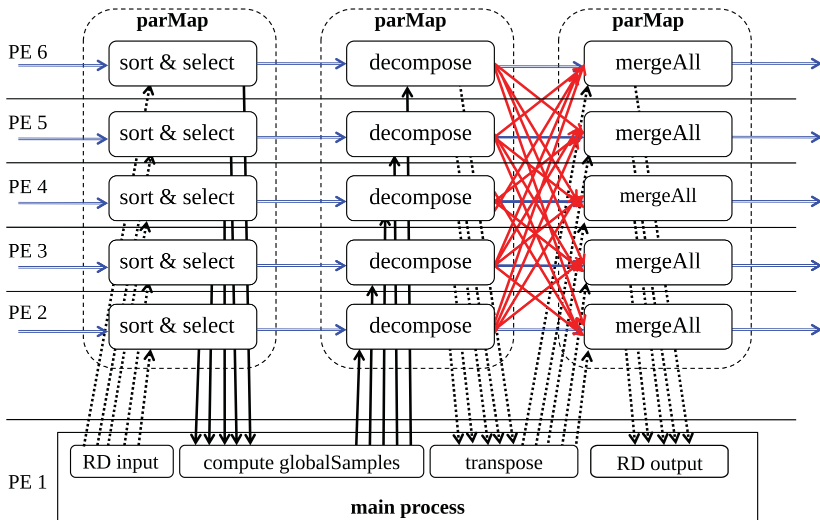


Fig. 7. PSRS process network.

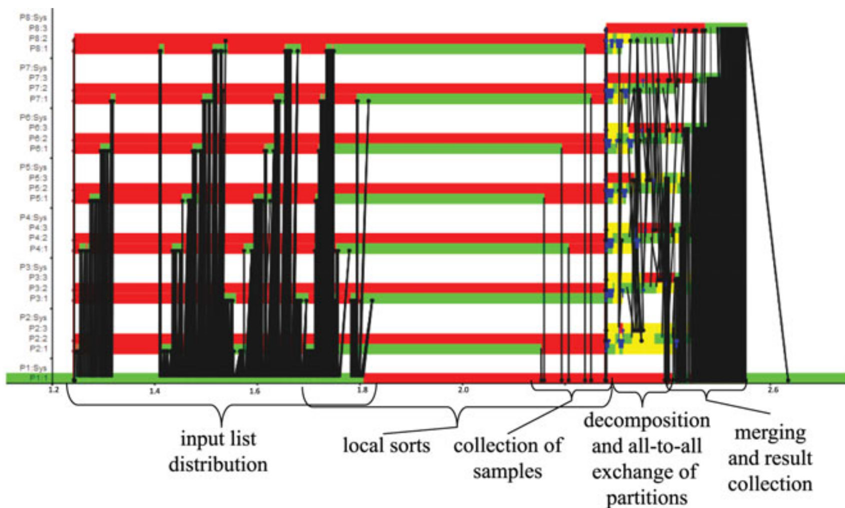


Fig. 8. Activity profile of PSRS: final phase and communication.

to the child processes. Hence, every process of Phase 4 is responsible for one of the partitions: It `fetches` the respective partitions from all other processes, merges these (sorted) partitions, and `releases` its segment of the globally sorted distributed data.

The process network is depicted in Figure 7. Figure 8 shows a runtime trace on eight PEs with input size n = 1,000,000 and $p = 7$ worker PEs (PE 1 was dedicated to the main process). The different phases can be clearly recognised from the activity shown in the processes-per-machine view. We see different processes in the worker PEs for Phases 1, 3, and 4. Worker PEs are most active in Phase 1 (local sorting).

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
allToAllRD :: forall a b i. (Trans a, Trans b, Trans i)
             ⇒ (Int → a → [i])
             → (a → [i] → b)
             → [RD a] → [RD b]
allToAllRD t1 t2 xs = res where
  n = length xs    -- no of processes
  (res,iss) = n `pseq` unzip $ parMap p inp
  inp       = zip xs $ lazy $ transpose iss
  p :: (RD a, [RD i]) → (RD b, [RD i])
  p = (release *** releaseAll)
      ∘ ((uncurry t2) &&& (t1 n ∘ fst))
      ∘ (fetch *** fetchAll)
```

Fig. 9. The allToAllRD skeleton.

Phases 3 and 4, after the parent has processed the samples, are marked by communication with other siblings. The input is distributed by `releaseAll ∘ unshuffle` beforehand, and collected by `concat ∘ fetchAll` after the computation. Runtime is dominated by this distribution (message blocks in the beginning) and result list collection (black area in the end) phases, which overlap with the distributed processing phases.

3.1.2 Definition of PSRS using the `AlltoAllRD` Skeleton

The *parMap* - *transpose* pattern in phases 3–4 of our PSRS implementation:

```
parMap(t2 ∘ fetchAll) ∘ transpose ∘ parMap(releaseAll ∘ t1)
```

is quite common in parallel algorithms. The transposition combined with `fetchAll` and `releaseAll` establishes an all-to-all topology to transpose the distributed matrix. We find this pattern e.g. in our definition of the googleMapReduce skeleton (Berthold *et al.*, 2009a), our parallel FFT implementation (Berthold *et al.*, 2009b) or the n-body simulation presented in Section 3.3 of this paper. Therefore, this pattern is provided as a general topology skeleton `allToAllRD` (Dieterle *et al.*, 2010), which performs both `map` computations in the same set of processes, using Eden's tuple concurrency to provide the two inputs lazily when available (see Figure 9, definition of `inp` and two-component worker function `p` in arrow notation).

Also note a necessary invariant which cannot be expressed in the (standard) Haskell type system: As the transposed input is supplied to n child processes (second input component), each process should compute n intermediate values from the argument to function `t1` (first input component). `t1` is supplied by the application programmer. Therefore, the skeleton includes a first parameter n to `t1` which can determine the number of list elements that the function will yield (of course, this number will be inherently known in many applications).

A version of PSRS using the `allToAllRD` skeleton is listed in Figure 10. Most code is taken from the compositional version presented earlier. The first phase of PSRS (local pre-sorting and sampling) is separately done in a `parMap` call with remote data handles. The subsequent parallel phases 3 and 4 are implemented together by

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
psrsA2A :: (Trans a, Ord a) ⇒ Int → [RD [a]] → [RD [a]]
psrsA2A p
  = allToAllRDAt [2..p+1] (const partition) (const mergeAll)
    ∘ releaseAll ∘ uncurry zip ∘ first processSamples ∘ unzip
    ∘ parMapAt [2..p+1] (sortNSample ∘ fetch)
 where ...
  -- sortNSample, processSamples, partition are as before
```

Fig. 10. PSRS using all-to-all in Eden.

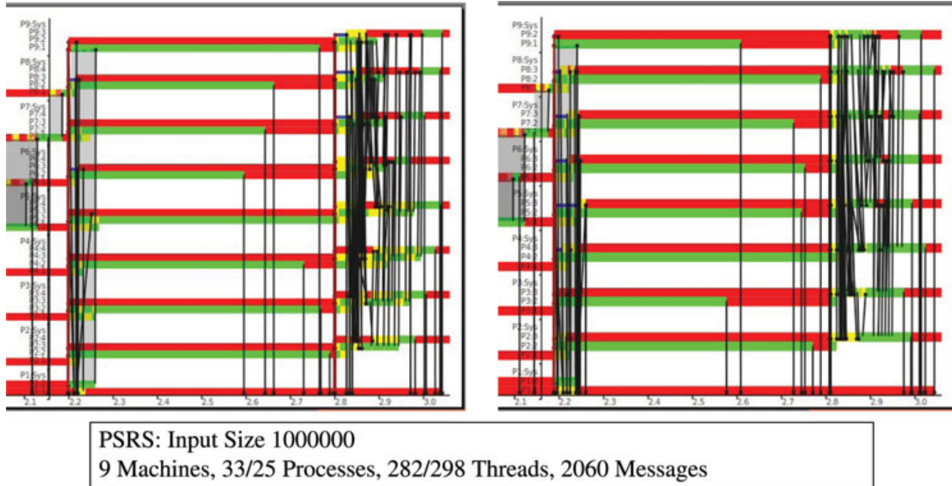


Fig. 11. PSRS compositional (left) and using all-to-all (right) with nine PEs (eight workers) input size 1,000,000, zoom into distributed phase.

a call to `allToAllRDAt` with `const partition` and `const mergeAll`, hence both ignoring their first argument.[7]

The code does not require many `fetch` and `release` calls, as most of them are now implicitly done inside the `allToAllRD` skeleton. One extra `releaseAll` is required to allow for skeleton input of type `[RD ([a],RD [a])]`.

3.1.3 Experimental evaluation

To compare the compositional implementation and the one using the `allToAll` skeleton, we focus on the distributed computation phase of the PSRS algorithm. Figure 11 shows the runtime trace of the two PSRS versions (compositional implementation on the left), with input size 1,000,000 and $p = 8$ worker PEs. The `allToAll` version (right trace) works with one process less per PE, but runtimes and process activities of both versions are largely similar, with slightly better runtimes here for the compositional version (average difference: −5.85%). This is confirmed by the runtimes and speedups of PSRS on an increasing number of processors with

[7] For `partition`, the result list length is determined by the first argument (`pivots`), which is the same for all child processes (produced by `replicate` in `processSamples`). `mergeAll` does not require the pivots, as it only merges sorted lists.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

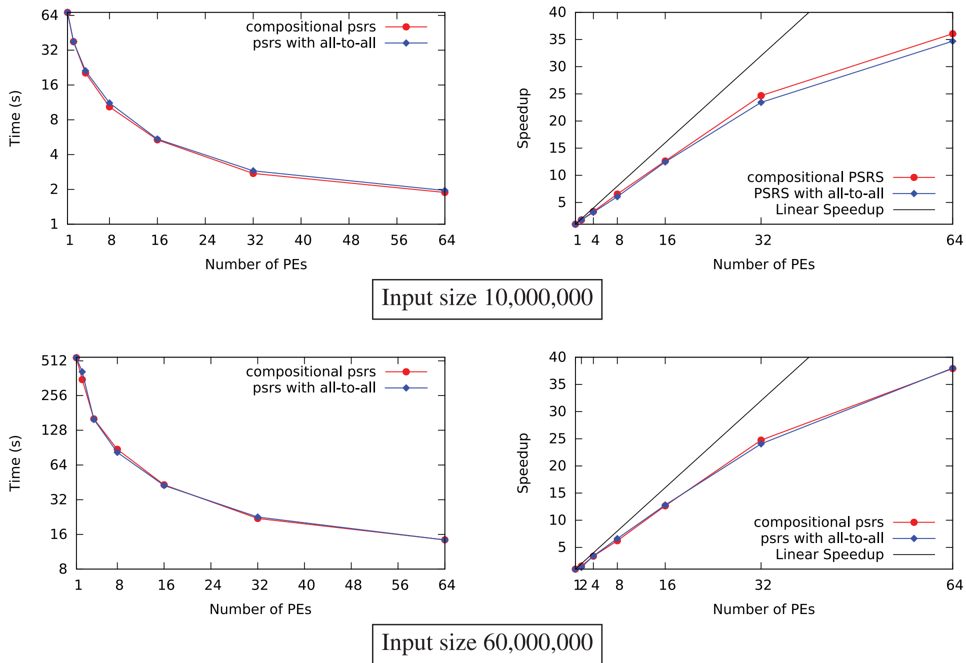


Fig. 12. PSRS runtimes and speedups, only distributed phase.

input sizes 10,000,000 and 60,000,000 (see Figure 12). The close correspondence between both versions is independent from the input size. Well-understood, the algorithm scales well when input and output stay distributed.

3.2 Iteration: Conjugate gradient (CG)

Conjugate gradient is an efficient iterative algorithm for solving large linear systems whose matrix is symmetric and positive definite (Saad, 1996). It generates vector sequences of iterates which are successive approximations to the solution, residual vectors corresponding to the iterates and search directions used in updating the iterates and the residuals.

The Haskell code given in Figure 13 follows the presentation of Breitinger (1998). A data type `ISol` is used for representing an iterative solution comprising an iterate `x`, a residual `r`, a search direction `p`, and an iteration counter. The `cg` code uses common basic matrix and vector operations (scalar and dot product, vector arithmetics, and matrix-vector multiplication `matVec`), whose definitions are omitted here. We use the library `Data.Vector.Unboxed` for the representation of vectors. A `Matrix` is a list of vectors. The use of unboxed vectors instead of lists accelerated our programs by the factor 10.

3.2.1 Recursive process instantiations

This iterative program is parallelised by decomposing the matrix-vector multiplication into as many tasks as PEs are available. The matrix is split into chunks of row

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
data ISol = IterSol Vector Vector Vector Int
--    ISol x r p i means iterate x, residual vector r,
--                       search direction p, iteration counter i

cg   :: Matrix   → Vector →   ISol
cg   a v = until converge (nextIter a) (initSol v)
  where
    converge :: ISol → Bool
    converge (IterSol _ r _ _) = dotProd r r < epsilon
      where  epsilon = 0.01

    nextIter     ::   Matrix   →   ISol                →   ISol
    nextIter          a            (IterSol x r p k)
      =  (IterSol x' r' p' (k+1))
      where
        -- matrix vector multiplication
        q      = matVec a p

        pq     = dotProd p q
        rr     = dotProd r r
        qq     = dotProd q q
        alpha = rr / pq
        beta  = alpha * qq / (pq - 1)
        r'     = vecSub r  $ scalMult alpha q
        x'     = vecAdd x  $ scalMult alpha p
        p'     = vecAdd r' $ scalMult beta  p

    initSol vector  = IterSol (zero vector) vector vector 0
```

Fig. 13. Haskell implementation of conjugate gradient.

vectors and the vector is broadcast to all worker processes, such that each worker process can compute a chunk of the result vector. The parallel version differs from the sequential one by only two small changes:

1. `q = matVec a p` is replaced with
   `q = concat $ parMap (uncurry matVec) (zip aSplit (repeat p)))`
2. and a respective decomposition `aSplit` of the matrix in the outer `where`-block
   `aSplit = splitIntoN np a :: [[Matrix]]`.

Each process receives a matrix chunk and the whole vector. The number of PEs `np` becomes an additional parameter of the function `cg`. The drawback of this parallelisation is that the matrix chunks would repeatedly be distributed by the main process to all child processes. A better approach is to forward the matrix chunks locally between the corresponding processes in the iteration, using remote data and co-allocation to reduce communication overhead. Moreover, it is advantageous to suppress streaming by boxing the matrix chunks into a special *lazy box* `LBox`:

```
newtype LBox a = LBox {unLbox :: a}
instance NFData a ⇒ NFData (LBox a)
 where rnf (LBox x) = () -- rnf x is avoided
instance Trans a  ⇒ Trans (LBox a)
```

Data in a lazy box will not be evaluated to normal form before sending. This is especially useful for data that is known to be in normal form. The repeated traversal

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
cgpar  :: Int → Matrix  → Vector →  ISol
cgpar np a v
  = snd $ until converge nextIter (aSplit, (initSol v))
  where
    converge :: ([RD (LBox Matrix)],  ISol) → Bool
    converge (_, (IterSol _ r _ _)) = dotProd r r < epsilon
      where epsilon = 0.01

    aSplit  :: [RD (LBox Matrix)]
    aSplit  =  map (release.LBox) $ splitIntoN np a

    nextIter :: ([RD (LBox Matrix)],  ISol)
                 → ([RD (LBox Matrix)], ISol)
    nextIter (hbms, IterSol x r p k)
      = (map fst ress, IterSol x' r' p' (k+1))
        where
          ress  = parMapAt [1..]
                    (λ (hbm, vec) →
                       let bm = fetch hbm
                           hbm = release bm -- new handle
                           mat = unLbox  bm -- matrix chunk
                       in  (hbm, matVec mat vec)
                    (zip hbms (repeat p))
          q     = concat $ map snd ress
          -- following code as before
          pq = ...
```

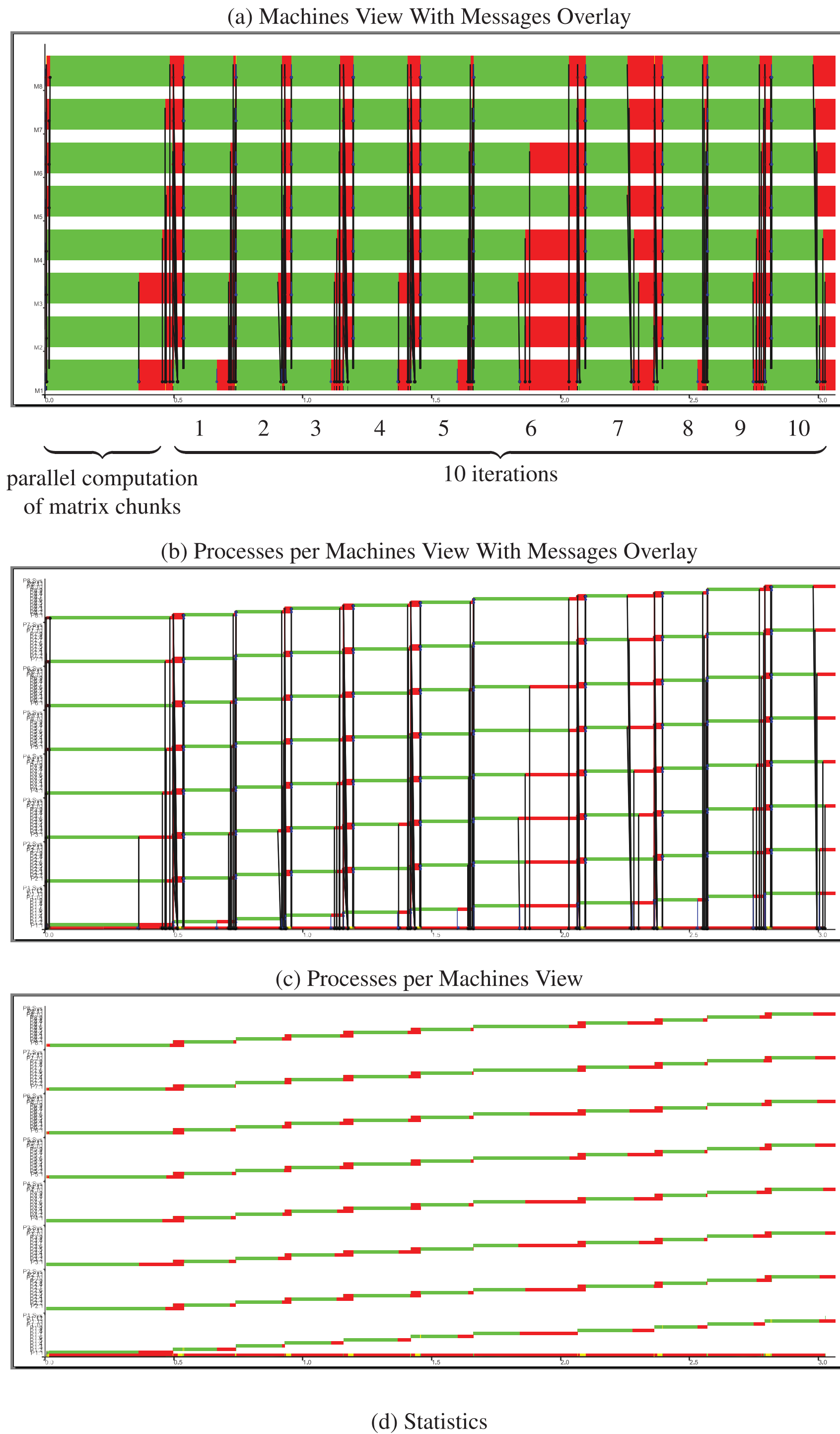
Fig. 14. Parallel conjugate gradient with recursive process instantiations.

of the data by `rnf` is suppressed. Lazy boxes will be sent in a single message, i.e. streaming or concurrent sending will be suppressed. The resulting program code is given in Figure 14.

The iteration function `nextIter` now receives the list of handles to the boxed matrix chunks as an additional parameter of type `[RD (LBox Matrix)]`. It is important that each process produces a new handle to be passed to its successor (via the main process) because a handle can only be used once. The worker processes are created using the `parMapAt` skeleton with the desired placement on successive PEs starting with PE 2 (indicated by the first parameter `[2..]`).

Figure 15 shows three views of a trace of a program run for a matrix and vectors of size 20,000 on eight PEs. Ten iterations are performed. The machines view in Figure 15(a) and the processes-per-machine view in Figure 15(b) have messages overlayed. In Figure 15(c), the processes-per-machine view is shown without messages. The computation starts with the distributed evaluation of the input matrix chunks by auxiliary processes. This evaluation takes about half a second. Then, the parallel iteration phase starts. In Figure 15(a) and (b), the iterations are clearly separated by the message exchanges between the main PE and main process, respectively, and all other PEs and processes. The message traffic consists of the following communications: First, the main process sends process instantiation messages to all PEs. As a reply, it receives a message with the input channels from each process. Then, it sends the processes' input via the received channels. The input consists of the matrix chunk handle and the vector. The processes perform a matrix vector multiplication with their matrix chunk and the input vector and return their

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press



(d) Statistics

Input Parameters: Matrix Dimension 20000, 10 Iterations, Time for Parallel Iterations: 2,529
Total Runtime: 3,066135s, 8 Machines, 89 Processes, 513 Threads, 473 Messages

Fig. 15. Activity profile of parallel CG program with recursive process instantiations.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
iterUntil :: (Trans sw, Trans iw, Trans rw) ⇒
  (inp → ([sw],[iw],sm))    -- ^input transformation function
  → (sw → iw → (rw,sw))     -- ^worker function
  → (sm → [rw] → Either r ([iw],sm)) -- ^combine function
  → inp                     -- ^input
  → r                       -- ^result
```

Fig. 16. Interface of `iterUntil` iteration skeleton.

result vector which is a chunk of the overall result vector to the main process. The main process decides whether to terminate or to start another iteration step. This synchronisation by the master induces that slow worker processes may slow down the whole computation, as can be observed in iteration steps 6 and 7, where the worker processes on PEs 5 and 7 need more time which leads to blocking phases in the other PEs. In iteration step 7, the worker on PE 2 is the slowest. Although the computation is very regular and each worker has to perform computations with the same complexity, such imbalances in computation time can be caused by concurrent computations performed on the physical machine which could not be used exclusively.

In the processes-per-machine view in Figure 15(b) and (c), it can be observed that the main process starts eight new `parMap` processes per iteration. The short process bars overlap slightly which can more clearly be seen without the message overlay (Figure 15(c)). The overlap is due to the passing of the matrix chunk from one process to its successor process. Each process terminates as soon as it has sent its matrix chunk. Because subsequent processes are co-allocated and share their PE's heap, passing the matrix chunk reduces to a simple pointer exchange.

The statistics in Figure 15(d) shows that 89 processes have been created, the main processes, eight auxiliary processes to compute the matrix chunks, and finally 10 times eight processes for the 10 parallel iteration steps on eight PEs. Five hundred thirteen threads are created: the main thread, eight system threads (one per PE), one input and two output threads per auxiliary process and, in particular, three input and three output threads for each of the 80 worker processes — two inputs and two outputs from and to the main process and one input and output from the predecessor process and to the successor process, i.e. in total there are $1 + 8 + 3 \times 8 + 6 \times 80 = 513$ threads.

3.2.2 *Parallel implementation using the `iterUntil` skeleton*

The Eden skeleton library provides the `iterUntil` skeleton for the evaluation of simple parallel iterative computations. A set of worker processes is created, which perform a parallel `map` as the iteration body, while a controlling main (or master) process decides about termination and provides the input for each iteration. Both the workers and the master process keep a local state. Internally, data are exchanged between workers and master as *streams* of tasks (from the master to all workers) and results (from all workers to the master), one element per iteration step and worker, respectively.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
cg_par_skel  :: Int → Matrix → Vector →  ISol
cg_par_skel  np   mat  vec
   =  iterUntil transform workerfct combine (mat,vec)
   where
     transform :: (Matrix,Vector) → ([Matrix],[Vector],ISol)
     transform    (matrix,vector)
       = (splitIntoN np matrix, replicate np vector,
          IterSol (zero vector) vector vector 0)

     workerfct :: Matrix →  Vector → (Vector, Matrix)
     workerfct a v =  (matVec a v, a)

     combine :: ISol → [Vector] → Either ISol ([Vector],ISol)
     combine itersol@(IterSol x r p k)  vs
         | converge itersol' = Left  itersol'
         | otherwise         = Right (replicate n p', itersol')
         where
           itersol'@(IterSol _ _ p' _)
              = nextIter (concat vs) itersol

     nextIter :: Vector → ISol  →  ISol
     nextIter    q  (IterSol x r p k)
         = (IterSol x' r' p' (k+1))
         where
           -- following code as before
           pq    = ...
```

Fig. 17. Parallel implementation of conjugate gradient using `iterUntil` skeleton.

The `iterUntil` skeleton's interface is shown in Figure 16. The behaviour of the skeleton is controlled by three functions:

1. The *input transformation function* transforms the input into lists of worker states and of worker inputs and a master state.
2. The *worker function* takes a state and an input and produces a result and a state.
3. The *combine function* is used by the master to decide about termination. It takes the master state and the list of worker results and produces either a final result or a list of worker inputs and a master state.

The conjugate gradient algorithm can easily be parallelised using the `iterUntil` skeleton. Figure 17 shows the necessary definitions of the three function parameters. In each iteration, the current vector is `replicated` as input for all workers. The matrix chunks are the worker states, which actually do not change during the entire computation. The master state is the iterative solution of type `ISol`. The worker function computes its chunk of the matrix vector multiplication and keeps its matrix part as the local worker state. The master concatenates all vector chunks received from the workers, and checks for termination (using `converge` as defined before). The `nextIter` function is adapted to take the current vector, which is the result of the matrix vector multiplication, instead of the matrix as parameter, as the matrix vector multiplication is computed in parallel by the worker processes.

Figure 18 shows two views of an activity profile of this `iterUntil`-based version, run with the same program parameters as in the activity profile shown in Figure 15

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

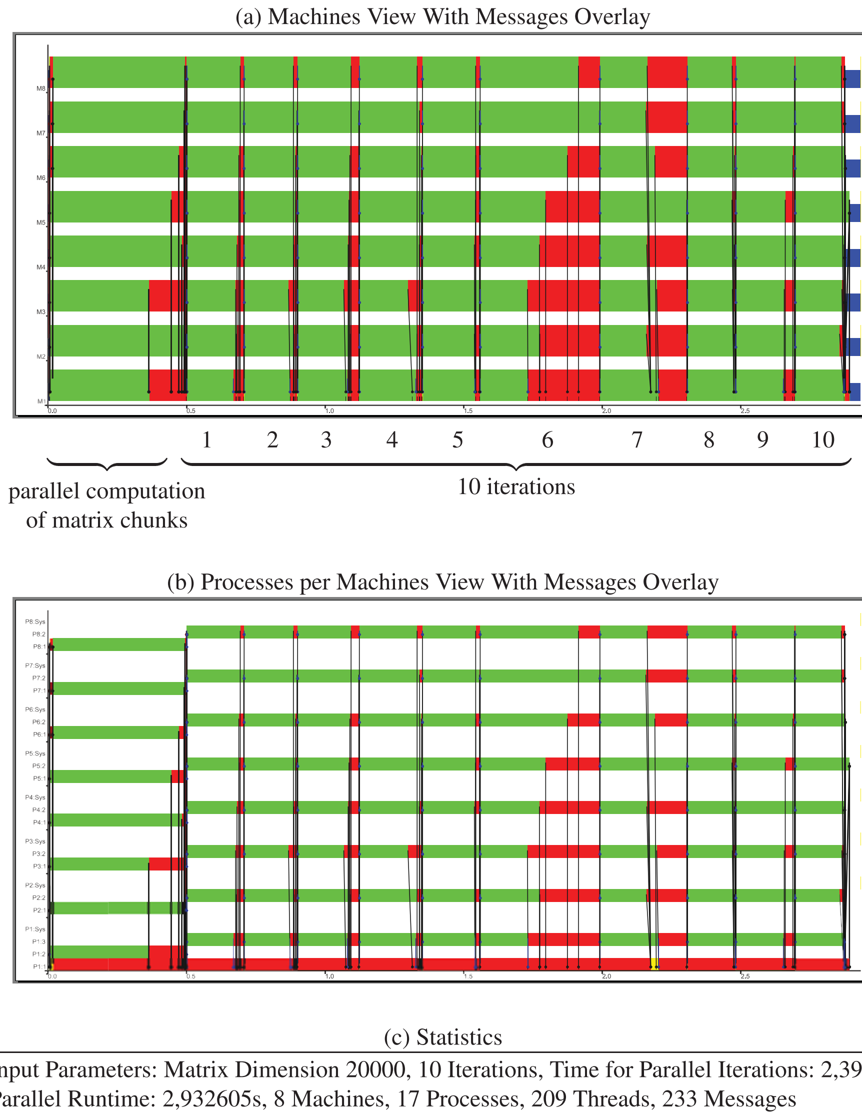


Fig. 18. Activity profile of parallel `iterUntil` CG program.

for the compositional (recursive) program version. For this program version, the machines view in Figure 18(a) is very similar to the machines view in Figure 15(a). The processes-per-machine views, however, reveal the different approaches. With the `iterUntil` skeleton exactly one process is generated per machine and these communicate with the main process between the different iterations, see Figure 18(b). The message streams overlay would cover most of the activity bars. As in Figure 15, there is an imbalance, by chance in the same iterations 6 and 7 as in Figure 15, but due to different PEs causing the delays. The machines views in Figures 18 and 15(a) do not reveal whether processes are started for each iteration or whether a process system is stable during the whole computation. Only by looking at the processes-

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

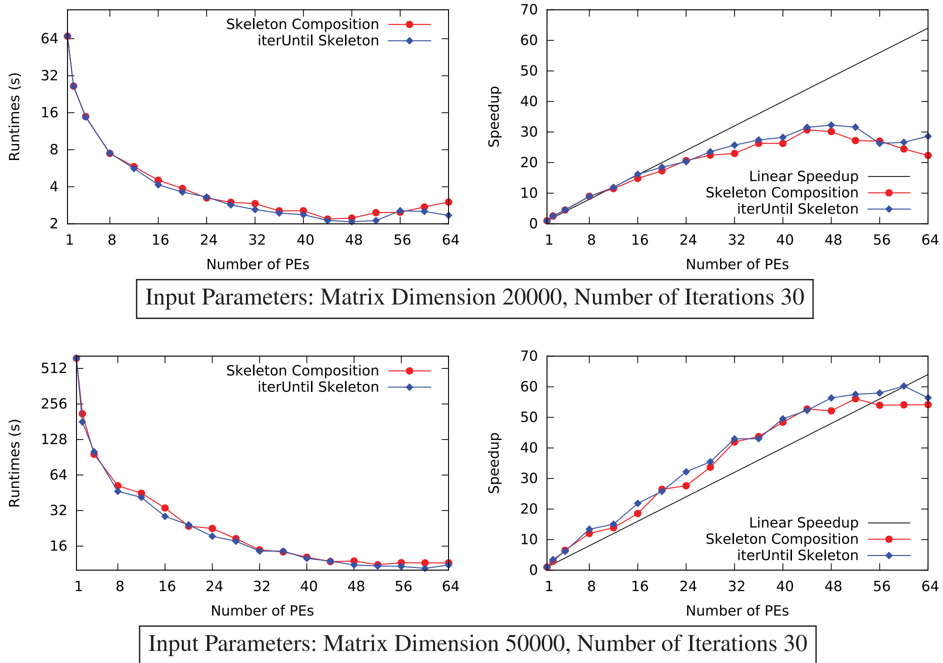


Fig. 19. Runtimes (in seconds) and speedup of parallel conjugate gradient programs for input sizes 20,000 and 50,000 and 30 iterations.

per-machine view, one detects the different approaches. The statistics in Figure 18(c) show that only 17 processes have been created, namely the main process, eight auxiliary processes and eight `iterUntil` processes. Accordingly, less threads have been created and less messages have been exchanged. The compositional version is a tenth of a second slower than the `iterUntil` skeleton version in this experiment, a very small difference given the total runtimes are close to 3 seconds.

3.2.3 Experimental evaluation

The advantage of the compositional parallel CG program is that it is very close to the original sequential program. Only local changes to the sequential program led to a parallel program which shows the same overall behaviour as the `iterUntil` instantiation when looking at the machine level only. Figure 19 shows the runtimes of the two program versions for input vectors of 20,000 and 50,000 elements when performing 30 iterations on 2 to 64 PEs as well as the absolute speedups achieved in comparison with the sequential `cg` program (see Figure 13) running on a single PE. Both programs scale very well and the runtimes and speedup curves are very close. More often than not, the compositional version is slightly slower than the `iterUntil` version, but the differences are small (around 10% on average). For the matrix dimension 50,000 both program versions achieve even super-linear speedups up to 52 PEs which will be due to cache effects. The parallel programs take benefit from the multiple cache stores due to the inherent locality in the unboxed vector data type. When the runtimes become very small, i.e. when the amount of work per

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

PE falls below a certain threshold, additional PEs will no longer improve speedups. This effect can be observed for more than 44 PEs with input size 20,000 and for more than 52 PEs with input size 50,000.

3.3 Using the iteration framework: N-Body, CG

The conjugate gradient computation shown the last section is a true iterated parallel `map`, the worker processes in each iteration are completely decoupled from each other. In many applications, worker processes need information from their peers, which is not implemented in the `iterUntil` skeleton. As an example, we discuss an all-pairs n-body simulation. N-body simulations approximate the movement of n-celestial bodies in three-dimensional space by discrete time steps, taking into account their mutual gravitational attraction. Each body has a known mass and a current position and velocity at any given time step. In each time step t, position, and velocity of each body are updated as follows:

1. Velocity changes depending on all other bodies' masses and positions, $\dot{\vec{p}}_t = \dot{\vec{p}}_{t-1} - G \cdot \sum m_i \frac{\vec{p}_{t-1} - \vec{p}_{i_{t-1}}}{|\vec{p_{t-1}} - \vec{p}_{i_{t-1}}|^3}$;
2. position changes according to the body's new current velocity, $\vec{p}_t = \vec{p}_{t-1} + \dot{\vec{p}}_t$.

The multiplicative gravitational constant G can be omitted, by assuming that the unit of the coordinate system is Gm.

The sequential code (which we do not show here in full) is based on a data type `Body` that encapsulates a body's mass, position, and velocity and three functions: A function `getMasspoint` to extract a body's mass and position, a function `changeVel` that changes a body's velocity by considering a set of other masspoints (other bodies, Equation (1) above), and a function `move` that moves a body by changing its position according to its velocity and one time unit (Equation (2) above).

```
getMasspoint :: Body → Masspoint
changeVel    :: [Masspoint] → Body → Body
move         :: Body → Body
```

3.3.1 Compositional parallelisation of N-body

A simple parallel n-body implementation will distribute all bodies over the available PEs. In a single iteration, each PE will

1. communicate masses and positions for its own bodies (*masspoints*) to all other PEs;
2. consider the received masspoints to update its own bodies' velocities;
3. and finally update its bodies' positions using new velocities.

The communication pattern used here is, again, described by the `allToAllRD`-skeleton from Section 3.1.2, but in this case, every process sends the same data to all peers, which is commonly called *allGather*, implemented by the following `allGatherRD` skeleton.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

```
allGatherRD :: (a → i) → (a →  [i] → b) → [RD a] → [RD b]
allGatherRD f g = allToAllRD f' g
  where
    f' n xs = replicate n $ f xs
```

Using `allGatherRD`, a single iteration step is implemented by the following `step` function:

```
step :: [RD [Body]] → [RD [Body]]
step = allGatherRD (map getMassPoint) updateAll

updateAll :: [Body] → [[MassPoint]] → [Body]
updateAll bs mss = map move $ map (updateVel ms) bs
  where
    ms = concat mss
```

Again, the use of remote data handles in the interface allows for iterating this step function easily by function composition. Since we are going to use all available PEs per step, the implicit round-robin placement of processes done by the RTS will co-allocate them nicely. (Otherwise, we would have used placement parameters.) The list of bodies needs to be partitioned into the available amount of workers (`noPe`) beforehand, and data are handed over as remote data handles created by `releaseAll`. The actual distribution of the lists happens when the parallel processes of the first iteration `fetch` the data. After the n steps of the iteration, data are fetched back to the main PE and `shuffled` back into the original order.

```
nbodysim :: Int → [Body] → [Body]
nbodysim n = (shuffle ∘ fetchAll)                  -- combine
             ∘ (foldr1 (∘) (replicate n step))     -- iterate
             ∘ (releaseAll ∘ unshuffle noPe)       -- partition
```

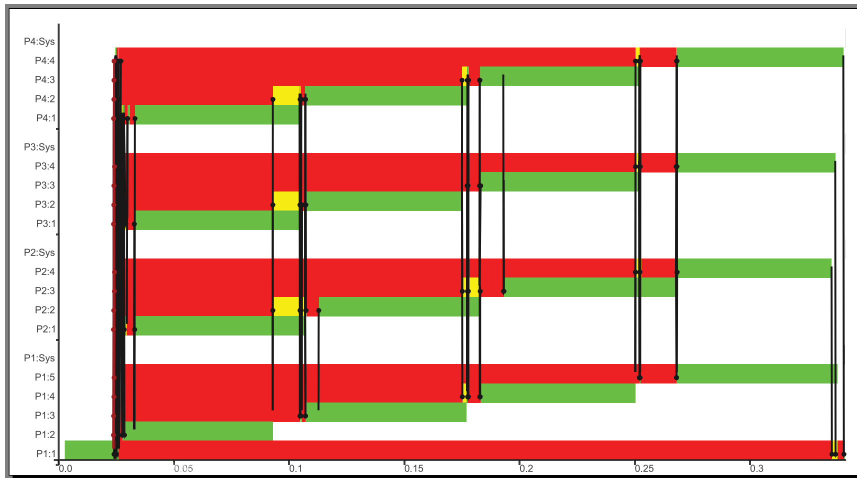
Figure 20 illustrates the runtime behaviour of the computation, in a small example run with 2,000 bodies and four iterations on four PEs. In contrast to the conjugate gradient example, the number of iterations is known from the start; therefore, processes for all four iterations are created as soon as input data is available to the main process. On each PE and in each iteration, one process becomes active when it receives the bodies from its predecessor. The local mass points are communicated to the siblings, then the local bodies velocities and positions are updated, and those are finally handed over to the successor process.

3.3.2 Comparison with iteration framework implementation

In Dieterle *et al.* (2013), we presented a framework for skeleton iteration that avoids the repeated process instantiation in favour of long-lived processes connected by streams for the iteration. While some applications benefit in performance from using our framework, the n-body simulation performed better in the compositional variant described here. Figure 21 shows runtimes and speedups for both versions with 20,000 bodies and 10 iterations. On average, the compositional program is about 10% faster than the version based on the iteration framework, with the performance gain increasing with PE numbers.

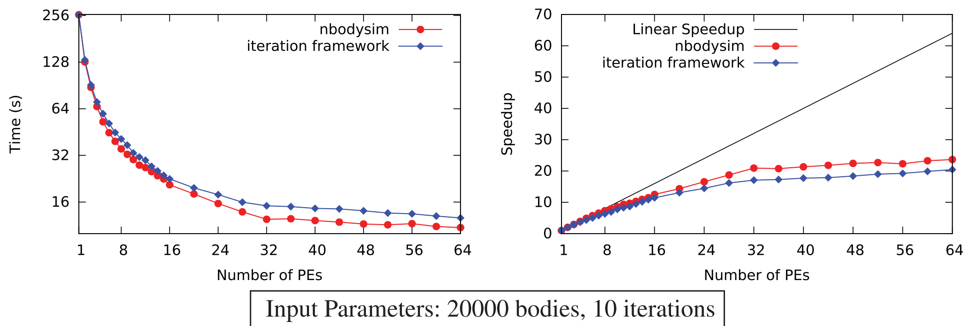
Figure 22 shows an activity profile of our n-body implementation using the iteration framework, for the same parameters as in the other trace (Figure 20).

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press



Input Parameters: 2000 bodies, 4 iterations
Parallel Runtime: 0,342s, 4 Machines, 17 Processes, 237 Threads, 184 Messages

Fig. 20. Activity profile (processes-per-machine view) of parallel n-body program with composed process instantiations, with messages overlay.



Input Parameters: 20000 bodies, 10 iterations

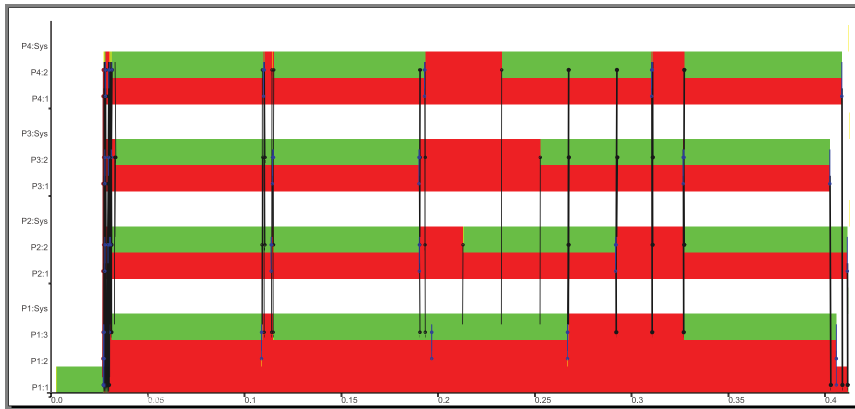
Fig. 21. Runtimes and relative Speedups of n-body-simulations on different numbers of PEs.

This version uses two processes per PE (apart from the main process): a *controller* process which determines the termination of the iteration (i.e. simply counting to four in this case), and the *worker* process which computes the current bodies and queries the controller after each step.

The third iteration in Figure 22 shows an interesting anomaly. Process P1:3 on machine 1 is the last to begin the third iteration. It has already received the masspoints for iteration 3 from its peers, which are blocked waiting for masspoints from P1:3. However, P1:3 now first computes its updates instead of sending its data to the peers,[8] thereby delaying the iteration step globally. This is an artefact

[8] Each Eden process communication happens in a different thread. Therefore, each process internally has five threads (one to communicate with peers, including itself, one to compute and send the final iteration result).

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press



Input Parameters: 2000 bodies, 4 iterations
Parallel Runtime: 0,413s, 4 Machines, 9 Processes, 101 Threads, 134 Messages

Fig. 22. Activity profile (processes-per-machine view) of parallel n-body program with iteration framework, with messages overlay.

of GHC's Haskell thread scheduling: A thread that is descheduled due to heap overflow is rescheduled immediately after GC. Threads for communication with peers should have higher priority than the main computation thread in the skeleton, but the GHC runtime does not support thread priorities.[9] However, the anomaly does not explain the performance differences observed. We examined several activity profiles for both program versions, and found that this happens in about the same frequency for both versions.

3.3.3 Re-considering CG

The iteration framework can also be used to implement the conjugate gradient algorithm using a stable process system. We repeated our measurements for the CG algorithm in order to find out how the iteration framework performs in the case of simple iterations without need for inter-worker commmunication. The results are shown in Figure 23. As the iteration framework works with different Eden modules, we repeated also the measurements shown in Figure 19.

Again runtimes of the three different implementations are very close. Differences appear more clearly in the speedup curves. Here, the iteration framework version performs slightly better than the monolithic `iterUntil` skeleton version. With the modular approach of the iteration framework, we can match the problem more specifically and gain more performance, even though we use the same problem specific parameter functions.

[9] There were promising attempts at implementing priority scheduling for Haskell threads, but the implemented solution suffered from performance problems and the feature was controversial. For the full discussion, see `https://ghc.haskell.org/trac/ghc/ticket/7606`.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

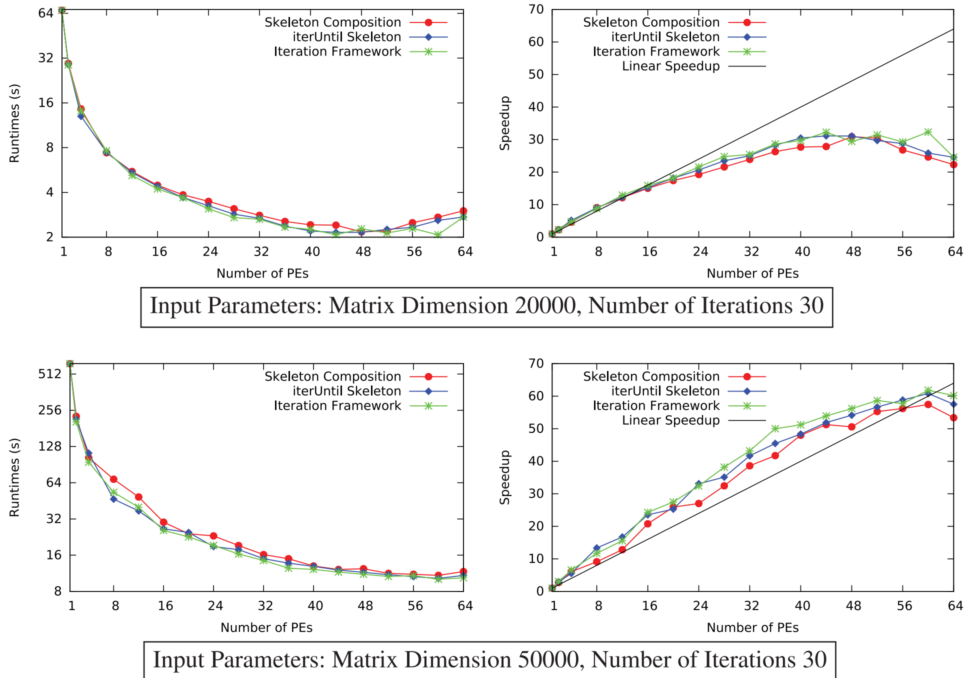


Fig. 23. Runtimes (in seconds) and speedup of parallel conjugate gradient programs for input sizes 20,000 and 50,000 and 30 iterations including iteration framework measurements.

In order to measure only the core phase of the parallel computation, we compute the corresponding matrix chunks in parallel on all PEs and pass remote data handles to the processes of the `iterUntil` skeleton and the iteration framework, respectively. In the `iterUntil` case, we are forced to repeatedly fetch and release the local matrix chunks on all the workers in every iteration step, because the skeleton does not include special treatment for the initial step. This local bypassing is efficiently supported by the RTS, but even though incurs a certain overhead. The more flexible iteration framework allows us to define a version where the initial input (the matrix chunk) is fetched only once and then kept as local state of the worker processes.

3.4 Discussion

We have presented three different case studies. In each case study, we have developed a parallel program using skeleton composition where new parallel processes are created for subsequent computation phases. Such compositional programs are much easier to develop because the program code is much closer to a sequential program or specification. They will be more amenable to optimisations, maintenance, and further developments, but may exhibit slightly worse performance due to repeated process creations and additional need for communications. This could be avoided by the development of stable process systems where a single set of worker processes is created for the whole parallel computation.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

Our goal has been to compare compositional parallel programs with corresponding stable process parallelisations using typical case studies. Due to the different characteristics of the case studies, we had to use different approaches to establish stable process systems, namely pre-defined skeletons `alltoallRD` and `iterUntil`, and our recently developed iteration framework (Dieterle *et al.*, 2013). The following table summarises our experiments and shows the average runtime gain by developing a stable-process implementation:

Case study	**Kind of computation**	**Stable approach**	**Average runtime gain**
PSRS	Fixed number of stages	`alltoallRD` skeleton	−2.85
CG	Iteration, independent workers	`iterUntil` skeleton	5.5%
N-Body	Iteration, interconnected workers	Iteration framework	−14%
CG	Iteration with independent steps	Iteration framework	8.85%

Only for the Conjugate Gradient algorithm, we were able to improve the performance by implementing stable process systems. The performance gain is less than 10%. In general, the development costs of stable systems might not be proportional to the performance gain. If a pre-defined stable-process skeleton or framework is not available, it might even not be worthwhile to invest in its development. The compositional program versions are much easier to develop than the stable-process program versions.

Trace analyses and runtime measurements show in all case studies that both program versions show almost identical parallel program behaviours. When the programmers keep in mind the following simple rules to avoid sources of inefficiency like e.g. a bad process distribution causing load imbalance or unnecessary communications, the performance loss caused by skeleton composition will be minimized:

- Use a remote data interface to support direct communication between producer and consumer processes;
- co-allocate corresponding processes of subsequent computation phases to take profit from the optimised local communication between co-allocated processes in the Eden parallel RTS.

When working with remote data handles, it is important to keep in mind that these handles can be used only once. If data has to be fetched from several processes, several remote data handles must be created. It is not possible to simply copy or share handles and use them to fetch the same data multiple times.

4 Related work

4.1 Parallel functional programming using Haskell

Functional languages have traditionally been recognised for their potential to easily express parallelism and write safe parallel programs. Therefore, functional languages

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

and concepts for parallelism have received much attention when multicore CPUs and later GPUs became common. During the same time, the last decade, Haskell has seen widespread adoption in industry and a rapidly growing community.

Historically, the first parallel Haskells were aimed at exploiting inherent parallelism in the graph reduction, such as the annotation-based approach of Glasgow parallel Haskell (Peyton Jones *et al.*, 1987; Trinder *et al.*, 1996). The concept of evaluation strategies (Trinder *et al.*, 1998; Marlow *et al.*, 2010) allows programmers to decouple computation and coordination, in a somewhat similar way as skeletons. The semi-implicit annotation-based approaches (as well as the completely implicit pH (Aditya *et al.*, 1995)) fit well with the declarative nature of functional languages and with Haskell's laziness and purity. However, it was found to incur overheads and require a certain sophistication when optimising parallel programs (argued in Marlow *et al.* (2011)).

Another research direction in parallel Haskell is to follow a data-parallel approach, providing special container types (arrays) and operations with parallel implementations on them. The full data-parallel Haskell (Chakravarty *et al.*, 2007) inspired by NESL (Blelloch, 1996) has taken a long time to mature, and yielded several by-products: libraries RePA (Lippmeier *et al.*, 2012) and `vector` (Leshchinskiy, 2008), and type system features of general interest (especially indexed types (Chakravarty *et al.*, 2005)). GPU languages like Accelerate (Chakravarty *et al.*, 2011) or Nikola (Mainland & Morrisett, 2010) follow the same data-parallel philosophy, but target GPUs in a two-stage DSL architecture. Parallel array operations in such array-oriented approaches can be considered as data-parallel skeletons, and nesting/flattening is central to DpH (whereas RePA and Accelerate only allow *regular* nesting).

The common denominator to all approaches mentioned so far is that they completely retain the purity of the Haskell computation language. Explicit concurrency and more explicit parallelism deviate from Haskell's purity to achieve better performance and give programmers the ability (and burden) to explicitly control parallelism and coordinate sub-computations.

Explicit concurrency can of course be used to parallelise programs — but only on a fairly small scale of a few cores, and in shared memory. To some extent, the same holds for the Par Monad library (Marlow *et al.*, 2011); where explicitly forked tasks are connected in a data-flow style, and for our working language Eden, (especially its monadic lower level implementation language EdI (Berthold & Loogen, 2007) with explicit communication, resembling the Par Monad closely). To capitalise on the functional setting, the explicit and liberal programming style must be replaced by a more structured parallelism specification. Besides the differences due to shared-heap or distributed-heap implementations, both Par Monad and Eden are equally well-suited for the development of skeletons and the work presented here. The Par Monad approach is limited in scaling due to its multicore implementation. The Eden implementation uses distributed heaps on separate RTS instances in a cluster (or separate OS processes) and can therefore scale better. In turn, the optimisations discussed in the main part are specifically targeting a cost model where communication is expensive, whereas communication cost would be very minor in an equivalent Par Monad implementation.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

Finally, there are Cloud Haskell (Epstein *et al.*, 2011) and HdpH (Maier *et al.*, 2014), which both target compute clusters and similar networked systems. Both are relatively new, and their rather tricky way to realise communication between different PEs somewhat complicates skeleton development. Cloud Haskell follows an actor model (Hewitt *et al.*, 1973) in the spirit of Erlang (Armstrong, 2007), and concentrates on portability of network layers and basic control support, rather than providing high-level skeleton libraries. HdpH generalises the Par Monad programming model for compute clusters, and concentrates on realising failover safety, combined with load-balancing work stealing mechanisms similar to the GUM implementation of Glasgow parallel Haskell.

For the topic of the paper at hand, the Par Monad would be the most suitable replacement for Eden to develop composable skeletons for parallel programming. HdpH has similar potential, but sets focus on different aspects, and does not provide the easy communication of Eden and its implementation.

4.2 Nesting and composition in other skeleton libraries

In the skeleton programming community, the problem of nesting, and more generally, composing skeletons, has been identified as crucial from the very beginnings (Cole, 1989). Type-checking and scheduling have been identified as the main complications of nesting and composing skeletons; the former harms programming safety and comfort, while the latter affects execution performance. Many skeleton libraries are based on imperative host languages like C++, C, or Java, and differ in terms of their nesting capabilities. We elaborate on a few examples, following the survey work of González-Vélez and Leyton (2010).

Some libraries distinguish data- and task-parallel skeletons, and only allow to nest a data-parallel skeleton inside a task-parallel one. Examples are Muesli (Kuchen, 2007) and the older P^3L (Bacci *et al.*, 1995). JaSkel (Ferreira *et al.*, 2006), a Java-based library, allows arbitrary nesting, but sacrifices type checking of skeletal programs by very liberal skeleton types (all data are considered as `Objects`). The prominent C++-based library eSkel (Benoit & Cole, 2002) developed in Edinburgh integrates the nesting deeply into its concepts. Nesting in *transient* mode leads to repeated instantiations of the nested skeleton, while *persistent* mode reuses the created instance. Our work described here essentially investigates this very distinction; our case studies compare transient compositional solutions to bespoke ones with persistent worker processes.

In general, when a non-functional skeleton library has matured beyond prototype stage, skeleton nesting is usually supported, sometimes by making design compromises, sometimes as an integral design aspect.

5 Conclusions and future work

The parallel Haskell dialect Eden features control constructs for the explicit parallel execution of communicating processes with independent heaps. Its original target architecture were cluster systems with distributed memory, but it also performs

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

competitive on modern multicore platforms in NUMA architecture. The Eden compiler extends GHC with a parallel RTS. Eden can be deployed on distributed systems, currently on top of middleware like MPI and PVM, as well as on multicores, where a special implementation using copying instead of message passing is provided. The language is well-suited for the implementation of algorithmic skeletons, making them directly available in the Eden skeleton library. In the best case, parallel programming in Eden consists mainly of choosing and composing suitable skeletons from the library, and of appropriate instantiation. EdenTV is a powerful tool to analyse the runtime behaviour of Eden programs.

The focus of this paper has been on programming methodology for skeleton-based parallel Eden programs. We have contrasted two different approaches to implement complex parallel algorithms: stable process systems and skeleton composition. Up to now, Eden's philosophy has always been to work with stable process systems in order to save process creation and communication costs. On the other hand, Eden skeletons can easily be composed to create complex process systems using remote data and specific co-allocation of processes. Eden's parallel RTS optimises communication between co-allocated processes, replacing communication with copying and thus saving the overhead induced by serialisation and deserialisation of data to be communicated.

We have considered three different case studies, a fixed composition (PSRS) and two iteration problems where the iteration steps are performed in parallel by a set of workers (CG, n-Body). In all case studies, our trace analyses with EdenTV show very similar overall program behaviour of both versions when only looking at the machines view. The more detailed processes-per-machine views as well as the statistics provided by EdenTV confirm that many more processes are created and many more messages are exchanged with the compositional approaches than with the stable process program versions. Nevertheless, the runtime measurements of both program versions are always very close: for PSRS and n-Body, the compositional version performs even better than the stable versions, whereas for CG two different stable versions are better than the compositional version. In total, the differences in runtimes and speedups are very low, in average between -15% and $+10\%$.

In general, composing complex process topologies from elementary skeletons is easier than the design and implementation of a sophisticated stable process skeleton for the same purpose. The program code is much closer to the sequential program version and easier to understand. This simplifies optimisations and is a good basis for further developments and code maintenance. Due to Eden's remote data concept, the explicit co-allocation of processes, and in particular the optimised communication between co-allocated processes within Eden's parallel RTS, the process creation and communication overhead can be minimised, and thus, the performance of the compositional approach is competitive with the more involved development of stable process systems.

In our case studies, the processes of the compositional program versions have mostly been created using `parMap` instantiations. Our approach is far more general. It is e.g. no problem to recursively instantiate other skeletons like e.g. a `torus` skeleton. Powers of matrices could e.g. be computed by iterating a parallel matrix

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

multiplication in a torus network. Thus, the technique leaves space for additional investigations. In addition to our work in this direction, we are currently working on a redesign and generalisation of the implicit communication mechanism of Eden to abstract from and generalise input/output transformations on communicated data.

Acknowledgments

We thank the other members of the Marburg Eden group, namely Florian Fey, Bastian Reitemeier, Lukas Schiller, and Andreas Voeth, as well as Hans-Wolfgang Loidl, Phil Trinder, and Prabhat Totoo for many fruitful discussions on skeleton programming in Eden and on parallel functional programming in general. We are grateful to the anonymous referees for their constructive criticism which helped us a lot to improve the paper.

References

Aditya, Sh., Arvind, Augustsson, L., Maessen, J.-W. & Nikhil, R. S. (1995) Semantics of pH: A parallel dialect of Haskell. In Proceedings of the Haskell Workshop, Hudak, P. (ed), YALE Research Report DCS/RR-1075, Yale University. Available at `https://www.haskell.org/haskell-workshop/1995/HW1995-Proceedings.pdf` [Accessed 27 Apr. 2016] pp. 35–49.

Armstrong, J. (2007) A history of Erlang. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages. HOPL III*. ACM, New York, NY, USA, pp. 6–1–6–26.

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S. & Vanneschi, M. (1995) P^3L: A structured high level programming language and its structured support. *Concurrency — Pract. Exp.* **7**(3), pp. 225–255.

Benoit, A. & Cole, M. (2002) *eSkel — The Edinburgh Skeleton Library*. Available at: `http://homepages.inf.ed.ac.uk/mic/eSkel/` [Accessed 27 Apr. 2016].

Berthold, J., Dieterle, M. & Loogen, R. (2009a) Implementing parallel google map-reduce in Eden. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Proceedings*, Sips, H., Epema, D. & Lin, H. X. (eds), Lecture Notes in Computer Science, vol. 5704. Springer, pp. 990–1002.

Berthold, J., Dieterle, M., Lobachev, O. & Loogen, R. (2009b) Parallel FFT using Eden Skeletons. In *Parallel Computing Technologies, 10th International Conference, PaCT 2009*, Malyshkin, V. (ed), Lecture Notes in Computer Science, vol. 5698. Springer, pp. 73–83.

Berthold, J. & Loogen, R. (2007) Parallel coordination made explicit in a functional setting. In *IFL 2006, 18th Intl. Symposium on the Implementation of Functional Languages*, Horváth, Z. & Zsók, V. (eds), Lecture Notes in Computer Science, vol. 4449, Springer. (**awarded best paper** of IFL'06), pp. 73–90.

Berthold, J. & Loogen, R. (2008) Visualizing parallel functional program Executions: Case studies with the Eden trace viewer. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference Parco 2007*, Bischof, C. H., Bücker, H. M., Gibbon, P., Joubert, G. R., Lippert, Th., Mohr, B. & Peters, F. J. (eds), Advances in Parallel Computing, vol. 15. IOS Press, pp. 121–128.

Blelloch, G. E. (1996) Programming parallel algorithms. *Commun.ACM* **39**(3), 85–97.

Breitinger, S. (1998) *Design and Implementation of the Parallel Functional Language Eden*. Ph.D. thesis, Philipps-University of Marburg, Germany. Available at `http://archiv.ub.uni-marburg.de/diss/z1999/0142/` [Accessed 27 Apr. 2016].

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

Chakravarty, M. M. T., Keller, G., Peyton Jones, S. & Marlow, S. (2005) Associated types with class. In *POPL 2005, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Palsberg, J. & Abadi, M. (eds), ACM, pp. 1–13.

Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011) Accelerating Haskell array codes with multicore GPUs. In *DAMP 2011, Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*, Carro, M. & Reppy, J. H. (eds), ACM, pp. 3–14.

Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G. & Marlow, S. (2007) Data parallel Haskell: A status report. In *DAMP 2007, Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming*, Glew, N. & Blelloch, G. E. (eds), ACM, pp. 10–18.

Cole, M. I. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. MIT Press.

Dieterle, M., Horstmeyer, Th., Berthold, J. & Loogen, R. (2013) Iterating Skeletons – structured parallelism by composition. In *IFL 2012, Implementation and Application of Functional Languages, Revised selected papers*, Hinze, R. & Gill, A. (eds), Lecture Notes in Computer Science, vol. 8241. Springer, pp. 18–36.

Dieterle, M., Horstmeyer, Th. & Loogen, R. (2010) Skeleton composition using remote data. In *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010*, Carro, M. & Peña, R. (eds), Lecture Notes in Computer Science, vol. 5937. Springer, pp. 73–87.

Epstein, J., Black, A. P. & Peyton-Jones, S. (2011) Towards Haskell in the Cloud. In *Haskell 2011: Proceedings of the 4th ACM Symposium on Haskell*, Claessen, K. (ed), ACM, pp. 118–129.

Ferreira, J. F., Sobral, J. L. & Proença, A. J. (2006) JaSkel: A Java skeleton-based framework for structured cluster and grid computing. CCGrid 2006, 6th IEEE International Symposium on Cluster Computing and the Grid. IEEE Computer Society, pp. 301–304.

GHC. (1991–2015) *The Glasgow Haskell Compiler*. Available at: `http://www.haskell.org/ghc` [Accessed 27 Apr. 2016].

González-Vélez, H. & Leyton, M. (2010) A survey of algorithmic Skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.* **40**(12), 1135–1160.

Haskell. (2010) *Haskell 2010 Language Report*. edited by S. Marlow. Available at: `http://www.haskell.org/` [Accessed 27 Apr. 2016].

Hewitt, C., Bishop, P. & Steiger, R. (1973) A universal modular ACTOR formalism for artificial intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence. IJCAI'73. Morgan Kaufmann, pp. 235–245.

Jones, Jr., D., Marlow, S. & Singh, S. (2009) Parallel performance tuning for Haskell. In Haskell 2009, Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. ACM, pp. 81–92.

Kuchen, H. (2007) *The Münster Skeleton Library Muesli*. Universität Münster, Available at: `http://www.wi1.uni-muenster.de/pi/forschung/Skeletons/` [Accessed 27 Apr. 2016].

Leshchinskiy, R. (2008) *Vector library*. Available at: `http://hackage.haskell.org/package/vector` [Accessed 27 Apr. 2016].

Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P. S. & Shi, H. (1993) On the versatility of parallel sorting by regular sampling. *Parallel Comput.* **19**(10), 1079–1103.

Lippmeier, B., Chakravarty, M., Keller, G. & Peyton Jones, S. (2012) Guiding parallel array fusion with indexed types. In Haskell 2012, Proceedings of the 2012 Haskell Symposium. ACM, pp. 25–36.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press

Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, St. & Rubio, F. (2003) Parallelism Abstractions in Eden. In Rabhi, F. A. & Gorlatch, S. (eds), *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, pp. 95–128.

Loogen, R., Ortega-Mallén, Y. & Peña-Marí, R. (2005) Parallel functional programming in Eden. *J. Funct. Program.* **15**(3), 431–475.

Maier, P., Stewart, R. & Trinder, P. (2014) The HdpH DSLs for scalable reliable computation. In Haskell 2014: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. ACM, pp. 65–76.

Mainland, G. & Morrisett, G. (2010) Nikola: Embedding compiled GPU functions in Haskell. In Haskell 2010, Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell. ACM, pp. 67–78.

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K. & Trinder, P. (2010) Seq no more: Better strategies for parallel Haskell. In Haskell 2010, Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell. ACM, pp. 91–102.

Marlow, S., Newton, R. & Peyton Jones, S. (2011) A monad for deterministic parallelism. In Haskell 2011, Proceedings of the 4th ACM Symposium on Haskell. ACM, pp. 71–82.

Peyton Jones, S. L., Clack, C. D., Salkild, J. & Hardie, M. (1987) GRIP - a high-performance architecture for parallel graph reduction. In *FPCA 1987, Intl. Conf. on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 274. Springer, pp. 98–112.

Saad, Y. (1996) *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company.

Trinder, P. W., Hammond, K., Jr., Mattson, J. S., Partridge, A. S. & Peyton Jones, S. L. (1996) GUM: A portable parallel implementation of Haskell. In *PLDI 1996, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Fischer, Ch. N. (ed), ACM, pp. 78–88.

Trinder, P. W., Hammond, K., Loidl, H.-W. & Peyton Jones, S. L. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Program.* **8**(1), 23–60.

https://doi.org/10.1017/S0956796816000083 Published online by Cambridge University Press