Structured Parallelism by Composition

Design and implementation of a framework supporting skeleton compositionality

Dissertation

zur
Erlangung des Doktorgrades (Dr. rer. nat.)
des
Fachbereichs Mathematik und Informatik
der Philipps-Universität Marburg
Hochschulkennziffer 1180

vorgelegt von
Mischa Dieterle
geboren in Heidelberg

Marburg/Lahn, 2016

Einreichung: 21. Dezember 2015
Disputation: 15. Februar 2016

Gutachter: Prof. Dr. Rita Loogen & Prof. Dr. Fernando Rubio Diez

Angefertigt mit Genehmigung des Fachbereichs Mathematik und
Informatik der Philipps-Universität Marburg

Dekanin: Prof. Dr. habil. Ilka Agricola

1. Gutachterin: Prof. Dr. Rita Loogen, Philipps-Universität Marburg
2. Gutachter: Prof. Dr. Fernando Rubio Diez, Universidad Complutense de Madrid

Für meine Oma

Brigitte Böss
12.7.1936 - 20.3.2016

Acknowledgements

I thank Rita Loogen for the supervision of this work, Fernando Rubio for his consent to review this work, Thomas Horstmeyer and Jost Berthold for the cooperation, Lukas Schiller, Marco Grebe and Felix Rieger for their helpful comments and the Heriot-Watt University of Edinburgh for giving me access to their Beowulf cluster.

Special thanks go to my wife and children for their support and patience when I was writing this thesis and to my parents who always supported my studies.

Zusammenfassung

Diese Arbeit widmet sich der effizienten Komponierbarkeit von algorithmischen Skeletten, einer Abstraktion von gängigen parallelen Programmierschemen, die sich in der funktionalen parallelen Programmiersprache Eden in einfacher Weise als Funktionen höherer Ordnung darstellen lassen. Durch algorithmische Skelette lässt sich paralleles Programmieren extrem erleichtern, da sie die kniffligen Details paralleler Abläufe bereits beinhalten und sich durch bloße Bereitstellung problemspezifischer Funktionen auf konkrete Anwendungen spezialisieren lassen. Dabei kommt der effizienten Komponierbarkeit von parallelen Skeletten eine besondere Bedeutung zu, da sich hierdurch komplexe, spezialisierte Skelette aus einfacheren Basisskeletten zusammensetzen lassen. Die so gewonnene Modularität ist gerade für funktionales Programmieren wichtig und sollte insbesondere in einer funktionalen parallelen Sprache nicht fehlen. Komposition wird hier in drei Kategorien unterteilt:

Verschachtelung: Ein Skelett wird aus einem anderen Skelett heraus instanziiert. Kommunikation findet baumartig entlang der Aufrufhierarchie statt. Dies wird in Eden direkt unterstützt.

Verkettung oder Hintereinanderausführung: Das Ergebnis einer Skelettberechnung wird zur Eingabe eines Folgeskeletts. Komposition von Funktionen wird in Haskell mit dem Kompositionsoperator `( ∘ )` ausgedrückt. Aus Performanzgründen sollen die Prozesse beider Skelette Ergebnisse direkt austauschen können, ohne diese über den Aufrufprozess schicken zu müssen. Hierfür wird das *Remote-Data*-Konzept eingeführt.

Iteration: Ein Skelett wird variabel oft hintereinander ausgeführt. Dies kann durch Rekursion und Hintereinanderausführung definiert werden. Optimiert werden die Anzahl der Skelett-Instanzen, die Kommunikation zwischen den Iterationsschritten und die Kontrolle der Schleifendurchläufe. Dazu dient ein eigens entwickeltes Iterationsframework, in dem Iterationsskelette aus Kontroll- und Rumpfskeletten zusammengefügt werden.

In dieser Arbeit haben wir neue Konzepte zur verteilten Skelettkomposition erforscht. Wir wollten weder von einer speziellen Kompilerunterstützung zur Realisierung der Kompo-

sition abhängig sein, noch wollten wir einfach einen Satz aus vordefinierten verteilten Datenstrukturen bereitstellen, die nach ihrer Implementierung eine feste API mit einer begrenzten Anzahl an unterstützten Datenstrukturen haben. Dennoch sollte eine performante Implementierung der Skelettkomposition auf Basis unserer Konzepte umgesetzt werden. Die Konzepte sollten sich reibungslos in den Kontext funktionaler Programmierung einbetten lassen, also Merkmale funktionaler Sprachen wie Funktionen als Werte, Rekursion und nicht strikte Datenstrukturen unterstützen. Anders ausgedrückt behandeln wir die Frage:

> Was sind konzeptionelle Bausteine, die *performante* Skelettkomposition erlauben, *einfach zu benutzen* sind und *hohe Flexibilität* in Bezug auf Konnektivität, Erweiterbarkeit und Transformierbarkeit gewährleisten?

Eine Schlüsselrolle bei unserem Kompositionskonzept kommt *Remote Data* zu. An Stelle der eigentlichen Daten kann ein *Remote-Data*-Handle verschickt werden, das an seinem Zielort benutzt wird, um die referenzierten Daten anzufordern. *Remote Data* kann in beliebigen Containertypen ähnlich wie verteilte Datenstrukturen zur effizienten Skelettkomposition benutzt werden. Die freie Zusammensetzung von *Remote Data* mit beliebigen Containertypen sorgt dabei für einen sehr hohen Grad an Flexibilität. Der Programmierer ist nicht auf vordefinierte verteilte Schnittstellen oder (Um-)Verteilungsfunktionen festgelegt und kann damit auch in eleganter Weise Prozesstopologien erzeugen.

Für den Spezialfall der Iteration algorithmischer Skelette versuchen wir den sukzessiven Auf- und Abbau eines Skeletts in jedem Iterationsschritt zu verhindern, der bei der rekursiven Benutzung von Skeletten üblich ist. Dies minimiert den parallelen Overhead für Prozess- und Kanalerzeugung und ermöglicht es Daten lokal auf persistenten Prozessen zu belassen. Dazu stellen wir ein *Iterations Framework* bereit. Dieses Konzept ist unabhängig von der Benutzung von *Remote Data*, lässt sich aber durch die Benutzung von *Remote Data* flexibel erweitern.

Beide oben genannte Ansätze zeigen für unsere Fallbeispiele vergleichbare Laufzeiten zu Programmen mit identischem parallelem Aufbau, bei denen das zugrunde liegende Skelett aber nicht aus einfacheren Basisskeletten komponiert, sondern als monolithisches Skelett implementiert wurde.

Des weiteren präsentieren wir Erweiterungen der Programmiersprache Eden, mit denen wir Komposition besser unterstützen können: Verallgemeinerung der überladenen Kommunikation, verallgemeinerte Prozessinstanziierung, kompositionelle Prozessplatzierung und Erweiterung von Box-Typen zum Anpassen von Kommunikationsverhalten.

Abstract

This thesis is dedicated to the efficient compositionality of algorithmic skeletons, which are abstractions of common parallel programming patterns. Skeletons can be implemented in the functional parallel language Eden as mere parallel higher order functions. The use of algorithmic skeletons facilitates parallel programming massively. This is because they already implement the tedious details of parallel programming and can be specialised for concrete applications by providing problem specific functions and parameters. Efficient skeleton compositionality is of particular importance because complex, specialised skeletons can be compound of simpler base skeletons. The resulting modularity is especially important for the context of functional programming and should not be missing in a functional language. We subdivide composition into three categories:

Nesting A skeleton is instantiated from another skeleton instance. Communication is tree shaped, along the call hierarchy. This is directly supported by Eden.

Composition in sequence The result of a skeleton is the input for a succeeding skeleton. Function composition is expressed in Eden by the `( ∘ )` operator. For performance reasons the processes of both skeletons should be able to exchange results directly instead of using the indirection via the caller process. We therefore introduce the *remote data* concept.

Iteration A skeleton is called in sequence a variable number of times. This can be defined using recursion and composition in sequence. We optimise the number of skeleton instances, the communication in between the iteration steps and the control of the loop. To this end, we developed an iteration framework where iteration skeletons are composed from control and body skeletons.

Central to our composition concept is *remote data*. We send a *remote data handle* instead of ordinary data, the data handle is used at its destination to request the referenced data. *Remote data* can be used inside arbitrary container types for efficient skeleton composition similar to ordinary distributed data types. The free combinability of *remote data* with arbitrary container types leads to a high degree of flexibility. The programmer is not restricted by using a predefined set of distributed data types and (re-)distribution

functions. Moreover, he can use *remote data* with arbitrary container types to elegantly create process topologies.

For the special case of skeleton iteration we prevent the repeated construction and deconstruction of skeleton instances for each single iteration step, which is common for the recursive use of skeletons. This minimises the parallel overhead for process and channel creation and allows to keep data local on persistent processes. To this end we provide a skeleton framework. This concept is independent of *remote data*, however the use of remote data in combination with the iteration framework makes the framework more flexible.

For our case studies, both approaches perform competitively compared to programs with identical parallel structure but which are implemented using monolithic skeletons – i.e. skeleton not composed from simpler ones.

Further, we present extensions of Eden which enhance composition support: generalisation of overloaded communication, generalisation of process instantiation, compositional process placement and extensions of `Box` types used to adapt communication behaviour.

Contents

CHAPTER 1

Introduction

Parallel computing has gained more and more importance with the emergence of manifold modern parallel hardware like multicore-CPUs, GPGPUs, parallel co-processors (like Intels manycore co-processor Xeon Phi) or computer clusters, typically already containing parallel hardware. Such heterogeneous, potentially hierarchical parallel systems are hard to program, especially if the code ought to be independent of the concrete hardware in place. In this setting, functional languages with their high level of abstraction and features like *referential transparency*, *compositionality*, *laziness* or *type classes* offer a great deal of advantages. Abstraction often has a price in terms of performance, but modern compilers increasingly manage to reduce the gap to low-level approaches. Moreover, adopting low-level approaches to modernised hardware is costly whereas high-level approaches are often easily portable and – at the same time – may profit from compiler development.

These features of functional languages have a great impact on parallel functional programming:

Referential transparency: Purely functional programs are deterministic. In the parallel setting, this guarantees freedom from race conditions through the absence of side effects.

Compositionality: Higher-order functions allow functions as input and output parameters. Functions are first class citizens and can be flexibly composed, combined or nested.

Laziness: Values are evaluated in a demand-driven way. This allows for process networks interconnected through lazy lists (streams) and thus the easy implementation e.g. of simple pipelines, but also cyclic process networks using self referential streams.

Type classes: Overloaded functions allow for a high degree of flexibility. The definition of overloaded functions depends on their input or output types. In parallel, we can control e.g. communication modes of channels alone through their input type.

In the following we concentrate on the definition of skeletons.

1.1 Algorithmic Skeletons

A major concept for abstraction in parallel programs are *algorithmic skeletons* [Col89]. These are parallel computation patterns including specification and implementation for whole algorithm classes like divide and conquer, map, reduce, etc., abstracting from the concrete algorithm in question. Skeletons are supplied with the concrete, typically sequential details of the algorithm in question to realise a parallel implementation of the algorithm. Typically, several different parallel implementations are provided for each parallel skeleton. Each implementation is further associated with a cost model. The cost model for a skeleton is applied to parameters of a parallel architecture and an algorithm to yield the cost of the different skeleton implementations. Thus, cost models help to decide for the best suited parallel implementation for a given situation.
However, we choose to not use cost models for the work at hand. We focus rather on the conceptual and technical details of skeleton composition. Therefore we use a slightly different definition of algorithmic skeletons. There may be manifold skeletons for a single algorithm class, where each skeleton exploits parallelism differently. For example skeletons for the well known divide and conquer scheme may create parallelism (processes or threads) recursively along the divide and conquer recursion scheme (distributed expansion scheme), or they may alternatively do some initial recursion steps sequentially, create parallelism to compute the sequential divide and conquer algorithm for all the expanded tasks in parallel and combine the upper most divide and conquer levels again sequentially (flat expansion scheme) [BDLL09].
The application programmer still has to choose the parallel behaviour by choosing a suitable skeleton from the skeleton library, but he or she is liberated from the gory details of parallel programming. This concept is independent from the functional setting, but integrates nicely in functional languages as algorithmic skeletons are implemented as mere parallel higher-order functions.

Typically, two kinds of skeletons are distinguished, namely *task* parallel and *data* parallel skeletons [Pel03, KC02, Alt07]. In task-parallel skeletons, different tasks are processed in parallel, like in pipeline, master worker or divide and conquer skeletons. In data-parallel skeletons, input data like e.g. matrices are split and processed in parallel. We use the further category *topology* skeletons, like pipeline, ring or torus skeletons which are named according to the process topology in use. This classification in categories is orthogonal to the others! *Topology* skeletons are often data parallel, distributing the input data over the processes, but not necessarily as e.g. the task parallel pipeline skeleton.

Language vs. library approach

Skeleton frameworks may be either implemented in own languages or language extensions with appropriate compiler support or they may be supplied in a library on top of an existing language [KC02]. The language approach gives more flexibility to the skeleton framework's design and allows for extensive compiler optimisations, where the library approach is easier to maintain and poses less barriers to the programmer, who may already be familiar with the base language and the corresponding tools (compiler, IDE-support, etc.).

In the following section, we will discuss several skeleton frameworks focusing on their treatment of skeleton composition. All the presented frameworks use a distributed memory model, where optimised communication is crucial for efficient skeleton composition. For shared memory systems, this aspect is not relevant.

1.2 Skeleton Composition in Literature

When parallelism is encapsulated in algorithmic skeletons, the problem of efficient skeleton composition arises. Cole argued in his seminal thesis [Col89], that skeletons should not be nestable:

> "If we are to avoid falling into the trap of trying to implement a universal language automatically, it seems that we must restrict ourselves to exploiting only the parallelism inherent in the basic structure of each skeleton. The lower level problem specific functions may or may not be parallelizable, but we should ignore them, since to attempt deeper analysis leads us into the original trap of "universality". The implementation task is to parallelize the distribution and manipulation of data implied by the highest level structure, leaving other functions to be executed entirely sequentially on individual processors as required, just as user code would slot directly into the structure of an entirely sequential implementation."[1]

Hence, the original skeleton set of 4 skeletons proposed by Cole[2] and as well as the first version of his skeleton library ESkel [BC02] implemented in C and MPI do not allow composition, they are flat skeletal systems. Pelegatti commented on flat skeletal systems:

> "In principle, flat systems could be extended with new skeletons if needed, however this makes systems very large and impractical. It also forces some of

1 [Col89, page 20]

2 The original set comprises a divide and conquer skeleton, an iterative combination skeleton, a cluster skeleton and a task queue (pipeline) skeleton.

> the skeletons to be quite complex, providing combinations of much simpler patterns to match the structure of a specific application."[3]

The second version of Cole's skeleton library ESkel(2) [BC02] adds support for skeleton nesting. In [BC05], Benoit and Cole describe fundamental concepts in skeletal programming. They use the terms *activities* and *interactions*. Activities correspond to processes or threads, and may internally be parallel. Interactions correspond to messages or communication via shared memory. Relevant for skeleton composition is the concept *nesting mode* with the variants:

Transient Nesting: Activities can invoke skeletons (sub-skeletons in the hierarchy) to solve sub tasks. If the invoking activity processes a stream of inputs, then the decision to invoke a sub-skeleton must be made for every single stream element, the sub-skeletons are terminated after processing a stream element. In the work at hand, we call this alternating skeleton invocations and terminations the *heartbeat effect* [DHBL13].

Persistent Nesting: The decision to invoke a sub skeleton must be taken for the entirety of an activity's input. Input streams are passed to the sub-skeletons, which do not terminate before the input stream, the parent process or the application terminates. The activities and interactions of the sub skeleton interact directly with the activities and interactions of the enclosing skeleton and replace the calling activity effectively. In this thesis, we call persistent nestings *stable process systems*.

We can use this concept to characterise previous work on skeleton composition.

An interesting early approach to skeleton composition is the Pisa parallel programming language (*P3L*) [Pel03]. P3L is implemented with C and MPI, skeletons are provided in a library as implementation templates. A cross compiler converts the user defined composition into an optimised implementation. There are data- task- and control parallel skeletons. Data parallel composition works without distributed data types [DPP97]. The data parallel skeletons (parallel `map`, `reduce`, `scan` and `comp` *(composition)*) define the data distribution as annotations in the body in-list of a skeleton call. The cross compiler optimises the communication patterns in between the skeleton calls, which are implemented by collective MPI operations. This optimisation works e.g. "by eliminating gather and scatter of the same data with the same distribution strategy [...] [or] substituting pairs of operations with more efficient combined ones"[4] (e.g. *reduce broadcast* with *reduce-broadcast*). The task parallel skeletons (pipe and farm[5]) process a stream of independent input data. At the root level a program is a pipe with an initial stage, which provides the input stream and a final stage to consume the result. The detection of the end of the

3 [Pel03, page 156]
4 [DPP97, page 626]
5 A dynamically load balancing farm

input stream by every process initiates termination. Control skeletons are `seq f x` for sequential code which is used to instantiate parallel skeletons and skeleton `loop c f x`, which iterates an arbitrary skeleton `f` without adding additional processes. P3L programs have a two-tier structure: "At a finer level, parallelism is expressed using DP tasks [data parallel skeletons]. Then, coarse interaction between DP tasks is expressed according to task parallel skeletons. To achieve this, P3L forbids the nesting of task parallel skeletons with data parallel ones. Control skeletons have no restrictions as they do not modify the parallel structure of a computation"[6]. The skeleton nesting of a program is persistent.

Darlington et al. [DGTY95b] present an alternative solution to skeleton composition, the structured coordination language (*SCL*) which integrates task and data parallelism. They define a two-tier structure, where the upper level handles the parallel aspects of programming and the lower level contains the sequential computation written in any sequential base language. Parallel programs are defined by composing procedures in the base language, using skeletons which work on distributed data structures. Darlington distinguishes between configuration, elementary and computational skeletons. Configuration skeletons define data partitioning, distribution and alignment **explicitly**, following the model of High Performance Fortrans [Lov93] data distribution directives closely. Elementary skeletons are `map`, `fold`, `scan` and skeletons for data communication `rotate` and `brdcast`, which work as expected. The computational skeletons abstract control flow. There is e.g. a `farm` skeleton, (unlike in P3L) realising a static task distribution and an `iterUntil` skeleton, which abstracts until iterations, where an iteration is executed until a termination condition is matched. The presentation of SCL in [DGTY95b, DGTY95a] does not mention stream processing of skeleton input or output. We suspect therefore that Darlington et al. use transient skeleton composition an do not reuse skeleton instances when implementing skeleton iteration.

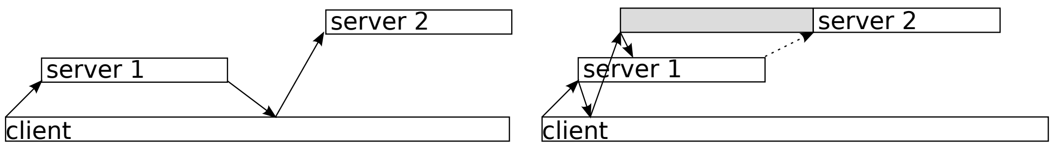
Kuchen and Cole [KC02] present a skeleton library for C++ and MPI which borrows the two-tier model of P3L, where data parallel skeletons may be nested inside task parallel ones, but not vice versa. Data parallelism and composition are based on distributed data structures. Communication is always explicit using *communication skeletons* which correspond mostly to the collective operations of MPI (`allToAll`, `gather`, `broadcastRow`...). Data parallel skeletons are e.g. `mapIndexInPlace` (the map function takes the index position as additional input), `fold`, `scan`, `zip` (combining the elements of two distributed lists). Data parallel operations are object functions of the distributed data types and partial application is supported, which allows a concise and powerful notation of skeleton calls using partially applied function parameters.
"A task parallel skeleton consumes a stream of input values and produces a stream of output values. This behavior corresponds to the behavior of an atomic process. Due to

6 [Pel03, page 159]

this property, a skeleton, like e.g. a pipeline, which coordinates several processes, may as well coordinate skeletons. Thus, skeletons can be arbitrarily nested."[7] Task parallel skeletons are e.g. `pipeline`, `farm`[8] and parallel composition, where several workers process the same duplicated input stream in parallel, their output is merged nondeterministically. Further, there is a control parallel `loop` skeleton [Kuc02], which creates a persistent loop with a cyclic process topology. A `loop` is applied to a parameter skeleton for the loop body and additionally to two control functions `propagate` and `feedback`. If function `propagate` applied to a loop result yields true, then the result is propagated to the successor skeleton of the `loop`. If function `feedback` applied to a loop result yields true, then the result is fed back to the loop. Both, one or none of the functions may yield true. In fact, the whole upper tier, the task parallel skeleton composition is persistent after construction. The lower data parallel tier allows transient nesting via direct function calls[BC05].

Alt[Alt07] presents a grid programming framework with skeletons written in Java using Java's **R**emote **M**ethode **I**nvocation (*RMI*) mechanism for skeleton invocation. He distinguishes skeleton composition of multi-server or single-server skeletons. For multi server skeleton composition, he uses `RArray`s, a distributed array structure. Single server skeleton composition is realised by two succeeding RMI's of the skeletons. This bears a performance problem, as the result of the first server is send back to the calling client before it is passed to the second server (see Figure 1.1a). Alt and Gorlatch present three optimisation steps, called *lazy RMI*, *localised RMI* and *future based RMI*[AG03, AG04, Alt07]. Lazy RMI is the first optimisation step, it replaces the data passed via the client by a remote reference, which is used to reference the result directly by the second server. Localised RMI, the second optimisation step is based on lazy RMI and adds optimised communication for composition on the same server, avoiding message passing in this case. Future based RMI is the final optimisation step which allows to pass the reference to the second server before the result is actually computed on the first server (see Figure 1.1b).

The language in which we perform our studies and implement our composition concepts is Eden [LOMP05], a parallel Haskell dialect. Though studied and implemented in a single



(a) RMI **(b)** Future based RMI

Figure 1.1: Single server skeleton composition using 2 Java RMI variants

7 [Kuc02, page 23]
8 load balancing farm, like in P3L

language, the concepts are of general nature.

For the functional language Eden, Peña and Rubio [PR01] discussed the issue of skeleton composition in earlier work: “From the language point of view, composition is trivial as skeletons are just functions. The real problem is how to assign processors to the skeletons in order to achieve good performance. We do not see this problem as an implementation issue but as a program design one. In our opinion, the programmer should be involved in this assignment.”[9] Parallel composition in general can be distinguished as three categories [Fos95, chapter 4]: *sequential composition*, where instances are executed one after the other and use identical machines, *parallel composition*, where instances are executed in parallel using distinct machines and exchanging data and *concurrent composition*, where composed instances may use arbitrary machines, are interconnected via channels and execute in a data driven manner. “Current Eden features do not support an easy way of splitting the processors into subsets and assigning them to skeletons.”[9] However, recent versions of Eden support explicit process placement, which can be used to express a partitioning of the available machines, e.g. by using a ticket mechanism as presented in Chapter 3.1.
Besides process placement issues, skeleton composition in Eden has a performance issue. Composing two skeletons in sequence with the predefined function composition operator `skel2 o skel1`, all data will be gathered and redistributed in between the skeleton calls. As Eden is designed to be used in a distributed setting, this gathering and redistribution of data may be very time-consuming. Rubio [Rub01] designed compiler optimisations at core Haskell level to optimise such indirections automatically. Unfortunately, these are not included in the current Eden compiler. Eden is developing from a language approach with an own compiler to a library approach, working with any standard Haskell compiler. The current stable implementation [Ber08] uses parallel runtime system support built into GHC [GHC15], leaving the originary compile process untouched. A full library approach is currently under development[10] and in prototype status. Hence, the above described compiler optimisations can not be realised in the future Eden system any more, but must be handled on library level if possible. All the novel Eden specific features presented in this thesis are implemented at library level and would work equally well with a full library implementation. Such a library implementation may also include optimisation of local messages, as has been described in the context of *localised RMI* above. Such an optimisation has been added to the Edens parallel runtime system recently.

9 [PR01, page 194]
10 Our goal is to drastically reduce the maintenance effort for the Eden system and to reach a broader audience, as interested programmers need only a standard Haskell-compiler and Eden libraries.

Conclusions

For the subject of efficient skeleton composition in distributed memory settings, language approaches have a certain advantage, as they may optimise the composition topology by the compiler. The second common approach is to use distributed data structures to efficiently pass data between skeleton calls, without the need for intermediate gathering and redistribution of the data. As a functional language, Eden has certain advantages when coping with skeleton composition, as composition is natural in functional languages. However, it neither has compiler support to optimise indirections in communication (library approach) nor supports distributed data structures. We will work with remote data (Chapter 2.3) to realise an alternative approach to skeleton composition similar to future based RMI, which was used only to connect single server skeletons.

Skeleton iteration is a subject which has been specially treated by all the composition frameworks discussed above. Here, with a persistent composition of skeletons, interconnected by streams of data like in P3L or Kuchen and Cole's skeleton library, skeleton iteration can be treated by a simple control construct.

1.3 A Framework Supporting Skeleton Compositionality

In this thesis we want to investigate new concepts for distributed skeleton composition. We want neither to depend on specialised compiler support for efficient composition, nor do we seek for a pool of pre-defined distributed data types which, once implemented has a fixed API and provides a limited amount of data structures. However, we want an implementation based on the concepts to provide good performance. The concepts shall fit smoothly in the functional programming context, thus naturally support functional features like functions as values, recursion or lazy data structures. In other words:

> What are alternative conceptual building blocks, which enable *performant* skeleton compositionality, are *easy to use* and provide *high flexibility* in terms of connectivity, extensibility and transformability?

The central publications underlying this thesis are:

- Mischa Dieterle, Thomas Horstmeyer and Rita Loogen
  Skeleton Composition using Remote Data.
  In Carro, Manuel; Peña, Ricardo, editors, *Practical Aspects of Declarative Languages 12th International Symposium, PADL 2010*, LNCS 5937, pages 73-87, Madrid, Spain, January 2010. Springer, 2010. (**awarded most practical paper** of PADL'10)

- Mischa Dieterle, Thomas Horstmeyer, Jost Berthold and Rita Loogen
  Iterating Skeletons - Structured Parallelism by Composition.
  In Ralf Hinze and Andy Gill (Eds.), IFL 2012, *24th Symposium on Implementation and Application of Functional Languages*, Revised Selected Papers, LNCS 8241, pages 18-36. Springer, 2013.
- Mischa Dieterle, Thomas Horstmeyer, Rita Loogen, and Jost Berthold.
  Skeleton composition vs. stable process systems in Eden.
  Journal of functional Programming, Vol. 26, to appear. Cambridge University Press, 2016.

1.3.1 Contributions

Remote data: We present remote data, providing lightweight handles for physical data. Remote data handles can be passed cheaply among processes in place of the referenced data. The referenced data is fetched directly at the target machine using the remote data handles.

Skeleton composition: We introduce a scheme for efficient skeleton composition using lists of remote data as skeleton interfaces. More general, we can use arbitrary `Traversable` data containers with remote data elements.

Process topology creation: We show how to create process topologies and topology skeletons by transforming arbitrary sequential data containers containing remote data.

Data redistribution: Similar to the topology creation using remote data, we can use arbitrary sequential data containers with remote data to redistribute data in between skeleton calls. In this step, we can also convert the container type, enabling skeleton composition among skeletons with incompatible container types.

Iteration framework: We define a framework for persistent skeleton iteration. The framework defines an *iteration scheme*, which is compound of an *iteration control* function and *iteration body* skeleton. The body skeleton uses a persistent process topology. Iteration body and iteration control are interconnected by a stream of data, where each stream element contains the input of an iteration step. In an optimisation step, we switch from a single iteration stream to parallel iteration streams connecting the processes using persistent communication channels. We introduce parallel iteration control skeletons to allow for efficient iteration control of the parallel iteration streams.

Lifting to iterable skeletons: Iteration body skeletons need a stream interface to be used in the iteration framework. We call such skeletons iterable. We show how many skeletons can be lifted to iterable versions by merely lifting their parameter functions. This is possible, if the skeletons input to output transformation can be sufficiently controlled by parameter functions of the skeleton.

Communication: Enhanced typeclass approach for custom communication We allow to define stream or concurrent communication for arbitrary data types. The user has merely to define a transformation of the data type to lists, or tuples respectively, and the inverse transformation. In General, we allow a user defined encoding of data for the communication process, enabling e.g. automatic chunking, compression and encryption.

Overloaded parallel input containers: We define a generic `spawn` function to allow for eager process instantiation of processes and data in arbitrary `Traversable` container types. Function `spawn` only instantiates the processes, establishing input and output channels for each process, which are connected to the caller. Various process topologies can be created by moving remote data handles along the structure of a container type.

Overloaded process placement: We propose a type class based solution which allows for process placement determined by the parallel skeleton input. Thus, the placement information is propagated implicitly when composing skeletons and does not have to be indicated explicitly for each skeleton call. Further we define location aware remote data that can be used for the same purpose.

Distributed Data: We define type support for arbitrary distributed data types, compound of remote data elements, `Traversable` container types and placement information.

Dynamic boxes to override default communication: Communication cost is crucial for the performance of parallel programs. Boxes [DHLB16] are a way of changing the communication behaviour of arbitrary data. We introduce dynamic boxes and a type class for boxes.

PA-Monad: We introduce a monad for **p**arallel **a**ctions in Eden. We allow to run parallel actions and to use their result in the functional Eden code.

1.3.2 Outline

This thesis is structured as follows:

Chapter 2: Eden

In the following chapter, we give a short introduction to Eden, also introducing two extensions to Eden which originate in this thesis:

- The PA-Monad which allows to define custom sequences of **p**arallel **a**ctions which will be executed eagerly. These custom sequences can be freed from the monad, wrapped in a function and used in Eden programs in a functional style.
- Remote data (`RD`) allows creating a handle of data by function `release` and to use the handle on a different machine to fetch the referenced data with function `fetch`. The `RD` handle is light weight and can be passed among processes with low overhead. *Remote data was originally published in* `PADL10DHL`*, where the introduction to remote data is taken from.*

We will introduce some Eden skeletons and show that *topology* skeletons are easily defined using remote data.

In the subsequent chapters, we focus on the principal part of skeleton composition. For this, we found the following distinction of composition approaches appropriate:

Skeleton nesting: A skeleton is called by the leaf processes of another skeleton, leading to tree-shaped process topologies. This is typically a form of parallel composition (using distinct processors for sub skeletons), but may also be concurrent composition – in Eden especially if no mind is put to process placement issues.

Skeleton composition in sequence: Skeletons are lined up, e.g. using function composition. This is typically sequential composition, but may be also parallel composition, e.g. a skeleton pipeline where skeletons are interconnected by streams.

Skeleton iteration: Is a special case of composition in sequence, where the same [sequence of] skeleton[s] is called many times. Skeleton iteration should not be mixed up with parallel for loops like in OpenMP, where (partially) independent sequential iteration steps are calculated in parallel. We rather evaluate a sequence of already parallel iteration steps.

Chapter 3: Skeleton nesting

We distinguish between skeleton nesting and skeleton composition in sequence, because of the data flow among the processes. We use the term *skeleton nesting* for tree shaped topologies which are constructed by nesting skeletons, e.g. when constructing the divide and conquer tree or hierarchical master worker skeletons. Here, the call tree corresponds to the desired channel topology.

Communication in Eden is set up implicitly along the hierarchy of process creation which is optimal for skeleton nesting. If a one-to-one process to processor placement is favoured, process placement has to be defined manually by the programmer to achieve a parallel composition of child skeletons. In Chapter 3.1, we show how a simple ticket mechanism can be used to organise this.
This ticket mechanism in the context of divide and conquer skeletons was originally published in [BDLL09]. The nested master worker skeletons were originally published by Priebe in [Pri06], our revised version is taken from [BDLP08].

Chapter 4: Skeleton composition in sequence

In imperative languages, composition is sometimes realised by a sequence of skeletons which is nested in a pipeline skeleton, e.g. [Pel03, KC02]. In the functional setting, we can use the function composition operator directly to compose a sequence of skeletons. In both cases, the data flow via the caller (pipeline or function composition) is not desired and we seek to optimise it. For our terminology, it does not matter if the skeleton sequence is achieved by composition or nesting in a pipeline, we refer to it as *composition in sequence.* We present three example skeletons, which where composed by hand. The *distributed workpool skeleton (based on [DBL10] and [Die07]), the Eden implementation of Google map-reduce(based on [BDL09]) and the Eden implementation of the distributed homomorphism skeleton (based on [LBDL09]).* We extracted a monolithic version of the `allToAll` skeleton which we used to implement revised versions of the Google map-reduce skeleton and the distributed homomorphism skeleton presented in Chapter 4.1.3 and Chapter 4.1.4.

In Eden, optimised communication does not come implicitly when using function composition on skeletons. As presented earlier, this is due to the lack of distributed data structures and the missing compiler optimisations to eliminate communication indirections. Our approach to optimise communication among processes is based on remote data, which we put in arbitrary container structures such as lists or trees. A list of remote data handles can be used similar to a distributed data structure to interconnect succeeding skeleton calls in a distributed way, but it is much more flexible. Skeletons with a remote data interface (e.g. `[RD a]`) are thus efficiently composable, if they use `fetch` and `release` on the processes in an appropriate way. *This was first described in* `PADL10DHL`*, which is the base version of Chapter 4.2 and Chapter 4.3.2.* Common languages offer a limited set of distributed data structures (e.g. array and matrix) each associated with a fixed set of redistribution functions. We can further manipulate the data container and the order of the remote data elements by recursion, pattern matching and predefined library functions. Thus, we can redistribute the data or convert it to another container type. This can be used to make skeletons with incompatible container signatures compatible. We also use this technique skeleton internally to define a core set of communication skeletons similar

to MPI's collective operations like `allToAll`, `reduce` or `allReduce`.
We further compare monolithic skeletons to skeletons composed by component skeletons with remote data interface and evaluate them by some example case studies.

Chapter 5: Skeleton iteration

For skeleton iteration, we present an iteration framework for persistent skeleton iteration in Eden. Eden skeletons are usually transient, they terminate after evaluating their results. Eden allows for persistent skeletons by using streams as process input and output. The iteration framework consists of control- and body skeletons, both with stream interfaces. The control skeleton may be a single process or a parallel skeleton connected to the body in parallel via a remote data interface. A control skeleton transforms the initial input to a skeleton body input, transforms the result from the body skeleton to the loop input for the body skeleton and decides about termination and the final result. Body skeletons are mostly standard Eden skeletons, lifted to a stream interface simply by lifting the parameter functions to a stream.
Chapter 5 - Chapter 5.2 are a revised version of [DHBL13], Chapter 5.4.2 is taken from [DHLB16].

Chapter 6: Types and Type Class Support for Efficient Composition

This chapter describes original, unpublished work.
In Chapter 6, we propose some extensions for Eden to improve composition support. We introduce new versions of the type class for overloaded communication, namely class `Trans` for *transformable* data (high level communication semantics, reduced by transformation to ...) and class `Transmissible` for transmissible data (... the low level communication semantics). The `Trans` class is used to define communication semantics of a type by reducing it to the communication semantics of a type already defined in the `Transmissible` class. Thus, e.g. a tree can be streamed by default by transforming it to a stream type. The inverse transformation has to be added as well and has to be unambiguous, such that the transformation can be undone at the receiver side. Interesting use cases include automatic chunking of data to reduce streaming overhead or automatic data compression in order to minimise communication cost. Our main purpose is to allow nested streams inside iterations.

Eden's instantiation function `spawn` is defined for lists. In order to allow more general parallel input we generalise `spawn` to work with arbitrary `Spawnable` containers, which are essentially `Traversable` containers plus process placement information. We also generalise function `fetchAll`, which is an eager version of `map fetch` to work on arbitrary `Traversable` container types with remote data elements. We allow for `Targetted` container types (enriched with process placement information) and standard container types to be instantiated by a single function `spawn`, which allows us to generalise skeleton definitions with or without process placement information. Both enables us to create skeletons with arbitrary container types, which can be composed with other skeletons with concrete or arbitrary interface types when instantiated with a matching input type. We define further a type class for distributed data types (container with remote data) and introduce location-aware remote data, which can be used e.g. to achieve corresponding placement for two composed skeletons which are interconnected using remote data.

We present boxes, simple wrapper types to escape the communication behaviour of a contained type. Boxes may define special evaluation forms when evaluated with `rnf`. This allows e.g. to send values unevaluated in a lazy box or to send streams as single elements in a strict box. *Boxes were first introduced in [DHLB16].* Furthermore, we present a type class for boxes and dynamic boxes, with parametric evaluation strategy which may change the evaluation degree.

Chapter 8 concludes and **Chapter 7** presents related work.

The following chapter contains a small introduction to Eden.

CHAPTER 2

Eden

Eden is a semi-explicit parallel functional language, originally designated for distributed memory settings, which works equally well on shared memory systems. Eden works with a distributed memory model, thus we use the categories machines, processes and threads for the different parallel entities.
An Eden program is started with n virtual machines on m physical machines. Each virtual machine contains a sequential runtime system instance with separate scheduler and heap. We will use the term **physical machines** explicitly if we want to refer to them and use the term **machines** for such **virtual machines**. The machines are executed by OS Threads, which works well in parallel if there are sufficiently many cores available. The machines communicate by a middleware like MPI or PVM. When working on a single physical machine, we additionally offer the “copy way”, a special way for the Eden runtime system which works by directly copying data among the different heaps exploiting the shared memory and without involving a separate middleware. Such connected runtime systems together form the parallel runtime system of the Eden compiler.

Eden has an explicit process concept with explicit process creation, but mostly implicit communication. Processes are executed on the available machines. Processes which share the same machine are executed concurrently, but don’t explore further parallelism as they are scheduled by a sequential runtime system.[11]
Eden extends Haskell by a small set of functions and a type class for overloaded communication. This type class `Trans` for transmissible data determines how data is communicated. Before sending, data is evaluated to normal form, thus class `Trans` has an `NFData` context: `NFData a ⇒ Trans a`. There are three sending modes:

Lists are evaluated and sent element wise as a (possibly infinite) stream. Nested lists are

11 This may change, we plan to combine Eden’s parallel runtime system with GHC’s [GHC15] threaded runtime system.

only streamed at top-level.

Tuples are evaluated and send concurrently for all tuple elements. Tuples of lists are streamed concurrently, lists of tuples are streamed with single messages. Tuples of tuples are only send concurrently at top-level.

Other values are evaluated and sent in a single chunk.

Eden has a push communication semantics, it puts demand on values to be sent. This differs from Haskell's lazy evaluation semantics.

Eden uses Haskells light weight threads for concurrency, which incur only minor overhead. Like processes, threads in Eden are scheduled by the sequential runtime system.

2.1 Language Definition

Basic Operations

Eden's basic operations are process creation and instantiation:

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

The `process` function applied to a function `f ::⇒ a → b` creates a process abstraction. The `(#)` operator instantiates a process abstraction `Process a b` with a corresponding input `x` of type `a`. This leads to the creation of a process, the evaluation of the input `x` to normal form and its sending to the newly created process. The process evaluates `f x` and returns the result to the caller process. Inputs and outputs are sent via implicit channels. More convenient is the combination of `process` and `#`, which creates directly a remote process from a function:

```
($#) :: (Trans a, Trans b) ⇒ (a → b) → a → b
($#) f x = process f # x
```

A problem arises when multiple processes shall be instantiated, e.g. by:

```
zipWith ($#) [f1,f2] [x1,x2]
```

Here, Haskells lazy evaluation causes that the instantiation `f2 $# x2` is delayed until the weak head normal form (*whnf*) of `f1 $# x1` has been evaluated in the caller process, which is when the caller process receives (some part of) the first processes result. We call this effect distributed sequentiality. Thus, Eden has been extended by functions:

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

to eagerly instantiate lists of processes and lists of functions respectively, where `spawn` is equal to `zipWith (#)` and `spawnF` is equal to `zipWith ($#)` except of demand control.

Usually, process placement takes place implicitly – with a machine local round robin scheme among the available machines. There are variants `spawnAt` and `spawnFAt` of `spawn` which take additionally a list of id's to explicitly determine the machines for placing the different processes.

```
type Places = [Int]
spawnAt  :: (Trans a, Trans b) ⇒ Places → [Process a b] → [a] → [b]
spawnFAt :: (Trans a, Trans b) ⇒ Places → [a → b]       → [a] → [b]
```

Machines are cyclically reused, thus `spawnAt [1..n+1]` uses machine `1` twice if there are only `n` machines available. Places are also cyclically reused if they are not sufficiently available, e.g. `spawnAt [1] [f,g] [1,2]` will instantiate `f` and `g` on machine 1. Place `0` is specially treated by using the automatic placement. Thus `spawn = spawnAt [0]`.

Nondeterminism

Eden programs are basically deterministic, but for efficient programming it is crucial to have a notion of parallelism degree and process position. Here Eden provides the constants `noPe :: Int` giving the total number of connected machines and `selfPe :: Int`, which is the machine id where the current process resides. Of course these constructs may lead to different results dependent on the number of machines in the current runtime environment. This is most obvious for the program:

```
main = return noPe
```

However, programs using `noPe` and `selfPe` run deterministically on the same number of machines.

The second source of nondeterminism is function `merge :: [[a]] → [a]` which merges nondeterministically a list of lists (or incoming streams) by the order of arrivals. This is necessary to define e.g. systems for dynamic load balancing. A program using `merge` is not purely functional. However, nondeterministic results can be avoided when the programmer takes care e.g. to manually re-establish the original order of the resulting elements.

Dynamic Channels

A limitation of the basic set of functions is that we can only produce implicit channels along the caller hierarchy which is always tree shaped. Therefore Eden provides also explicit communication channels of type `ChanName a` for the creation of arbitrary process topologies with functions:

```
new :: Trans a
       ⇒ (ChanName a  --parameter function that takes a channel name
           → a → b)   --and a substitute for the lazily received value.
       → b            --forwarded result
parfill :: Trans a
           ⇒ ChanName a   --ChanName to connect with
           → a            --data that will be sent
           → b            --forwarded to result
           → b            --result
```

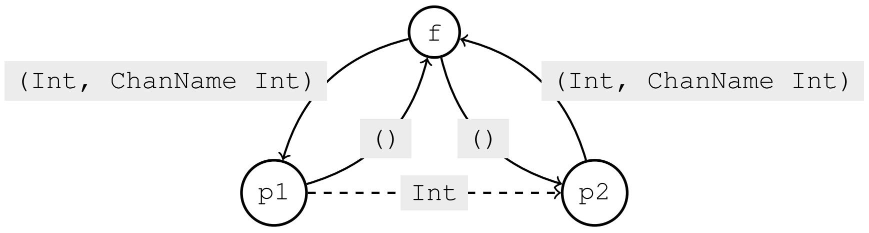
Function `new` produces a new channel consisting of an open inport where data will be received by the local process and a channel name (handle) for sending the data from an arbitrary process. `new` works in a continuation passing style taking a function as parameter which describes how the result depends on the name and the contents of the newly generated channel.
Function `parfill` is the counterpart of `new` used on the sender side to fill a channel. It takes a channel handle of type `ChanName a`, a value of type `a` and a continuation of type `b` which is directly returned as result. `parfill` forks a thread for the evaluation of the value and the execution of the sending operation. Note that channel creation and use are non functional features. `new`'s and `parfill`'s purpose is to trigger a side effect, the channel creation and channel use, respectively, not to calculate a result. The forwarded results of both functions are artificially added, to allow for these functions to be integrated in the demand string. This may easily lead to problems, e.g. deadlocks, as we show in the subsequent example. We also trigger side effects by using the instantiation operator `($#)`, but we supply `($#)` with an input and we are interested in the output. Apart from strictness issues, it makes no difference in denotational semantics if we use `($#)` or `($)`, the difference is basically where and to which degree we calculate the result.

A dynamic channel created with `new` can only be used once with parfill. After the use, the inport at the receiver side is closed and further messages will not be received. Multiple use of a single channel leads to a runtime error in the current implementation.

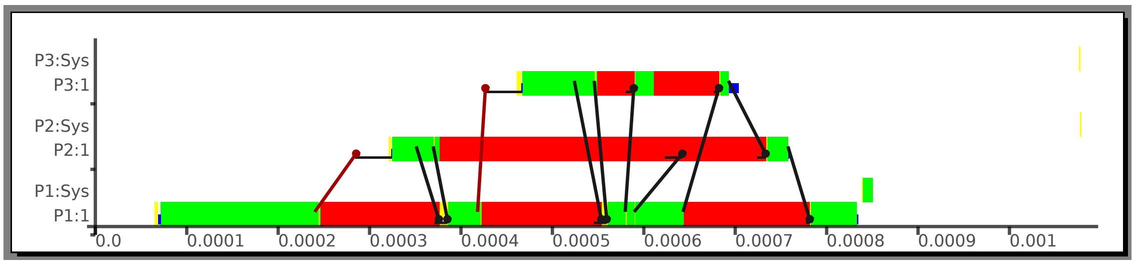
We can connect two processes directly using `new` and `parfill`. A small example is depicted in Figure 2.1. `f1` and `f2` define the functionality of two processes. `f1` takes `x` of type `Int` and, lazily supplied in the second tuple component, a channel handle `ch'` of type `ChanName Int`. It forks a thread to evaluate `x+1` and to send the result via `ch'` and returns the dummy value `()`. `f2` ignores its unit input and creates a channel `ch''` of type `ChanName Int` to receive a value `y` of type `Int`. It returns `y+1` and the channel handle `ch'`. We instantiate `f1` with the input `inp` and lazily, with the channel which will be created by `f2`. By instantiating `f2` with `f1`'s dummy result `z`, we enforce the evaluation of `z` due to Eden's eager communications semantics. If there would be no demand for `z` in `f` (here by supplying `f2` directly with `()` instead of `z`), then `f1` would never be instantiated due to Haskell's lazy evaluation and `f` would deadlock.

```
f :: Int → Int                                                        --
f inp = res where

  f1 :: (Int, ChanName Int) → ()
  f1 (x,ch') = parfill ch' (x+1) ()

  f2 :: () → (Int, ChanName Int)
  f2          = const $ new (λch'' y → (y+1,ch''))

  z           = f1 $# (inp, ch)
  (res,ch)    = f2 $# z
```

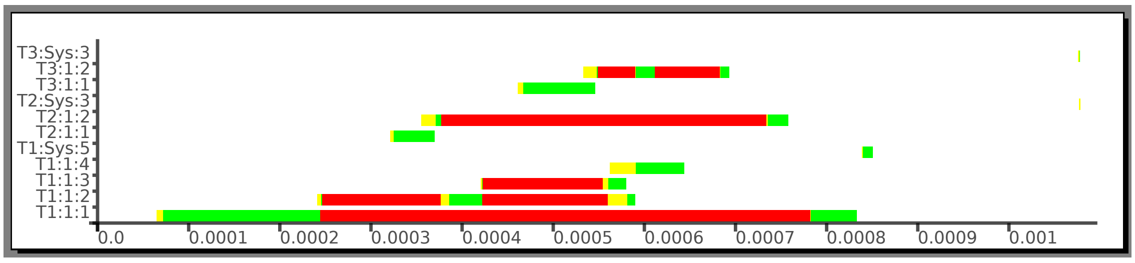
(a) code



(b) process scheme



(c) runtime trace: processes per machines view



(d) runtime trace: threads view

Figure 2.1: A simple example for dynamic channel usage

2.2 Analysis of Runtime Behaviour

The runtime traces in Figure 2.1c and 2.1d visualised by **Eden**'s **T**race **V**iewer EdenTV [BL07b] reveal the runtime behaviour of function `f`. EdenTV is a post mortem trace visualisation tool which depicts runtime behaviour of machines, processes or threads, depending on the view, in horizontal bars[12] and overlays message communication by black and red arrows (lines with a dot at the end), black arrows for normal communication, red arrows for process instantiation. Time is depicted on the x-axis. The bars are coloured in green if the computation unit (machine/process/thread) was running, yellow if it was runnable but not running (e.g. because other processes were using the physical resources or garbage collection takes place) and red if it was blocked (e.g. waiting for data) at this moment.

The trace visualisation in Figure 2.1c shows the processes per machines view, we can see that the main process (P1:1)[13] instantiates first process P2:1 (red arrow), which corresponds to `f2` in the code. `f2` is instantiated prior to `f1`, because there is demand on its result `res`. After process creation, the process evaluating `f2` sends a channel handle to receive its unit input (this happens after each process creation) and the dynamic channel handle `ch` to the main process. Now, there is demand to evaluate `f2`'s unit input `z`, which happens always concurrently by a separate thread (T1:1:2[14] in the thread view, Figure 2.1d). This causes the instantiation of the process P3:1 evaluating `f1` with result `z`. `f1` directly returns a channel handle for its input and the unit result `z`, after forking a thread (T3:1:2) to evaluate `x+1` and sending it over the dynamic channel. Thread T3:1:2 is blocked on the missing input `x` and the channel handle `ch`; after receiving both, `x+1` is send to `f2`, which returns `y+1` (`(x+1)+1`) to the main process. On each machine, there is an additional system process performing finaliser tasks, which has nothing to do with the program code.

12 one bar per computation unit (machine/process/thread, depending on the current view)
13 Notation in Process view: P*machine*:*process*
14 Notation in Thread view: P*machine*:*process*:*thread*

2.2.1 Test Environments

Our runtime experiments were executed on the following parallel machines:

name	type	details
igor:	multi-core processor	Intel Core i7-3770K@3.50GHz with 4 cores (hyperthreaded) and 16GB RAM at the Philipps-Universität Marburg.
hex:	multi-core multiprocessor	4 × AMD Opteron Processor 6378@2.40GHz with each 16 cores (64 cores total) and total 64GB RAM at the Philipps-Universität Marburg.
Beowulf:	cluster of multi-core multiprocessors	32 machines with each 12GB RAM and 2 × Intel Xeon E5504@2.00GHz with each 4 cores (256 cores total) at the Heriot-Watt University, Edinburgh.

2.3 Remote Data

The idea is to use lightweight handles for data – called *remote data* – which can be passed cheaply between machines and used at the destination machine to retrieve the data directly from the point of origin.

This section follows and reuses the corresponding presentation in [DHL10]:
We develop the concept of remote data in the context of our parallel functional language Eden, although the concept itself is language-independent. It could equally well be added to other parallel languages, see [AG03, AG04, Alt07] for a realisation in Java.

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

Eden Implementation of remote data

The implementation of remote data in Eden (Figure 2.2) is simple and elegant. To `release` local data `x` of type `a` we create – using the function `new` – a channel name `cc` of type `ChanName (ChanName a)` via which a channel `c` of type `ChanName a` will be received. Using `parfill` a thread is forked that subsequently sends the local data `x` via the channel `c`. The result of the `release` function is the newly created channel `cc :: ChanName (ChanName a)`. Note that the remote data type `RD a` is a synonym of `cc`'s type. Data of type `RD a` is merely a channel name and thus very lightweight with low communication costs. To access remote data we need to `fetch` it by again creating a channel `c :: ChanName a` using the function `new`. This channel is sent via the remote data handle, i.e. the channel `cc` of type `RD a`. The proper data is then received via channel `c` and returned as the result of the `fetch` function.

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

Example: We show a small example where the remote data concept is used to establish a direct channel connection between the sibling processes. Given functions `f` and `g`, one can calculate `(g ∘ f) a` in parallel creating a process for each function. Figure 2.3 shows two different ways to implement this. Simply replacing the function calls by process instantiations

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

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

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

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

Figure 2.2: Remote data definition

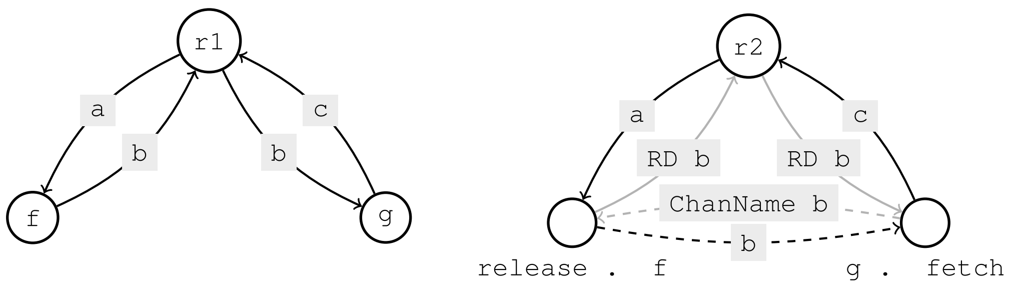


Figure 2.3: Using remote data

leads to the following behaviour (visualised in the left part of Fig. 2.3): Function `r1` instantiates first the process calculating `g`. Then `r1` instanciates the process remotely calculating `f` in order to evaluate `g`'s input. `r1` passes its input to this process and receives the remotely calculated result and passes the result to process `g`. The output of the process `g` is also sent back to the caller. The drawback of this approach is that the result of the process `f` will not be sent directly to process `g`. This causes unnecessary communication costs.

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

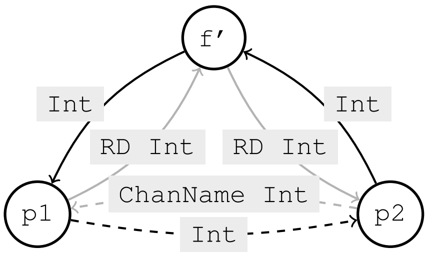
```
r2 a = (g ∘ fetch) $# ((release ∘ f) $# a).
```

It uses function `release` to produce a handle of type `RD a` for data of type `a`. Calling `fetch` with remote data returns the value released before. Function `r2` is identical to `r1` except for the conversion of the result type of `f`'s process and the input type of `g`'s process to remote data. The use of remote data leads to a direct communication of the actual data between the processes of `f` and `g` (see the right part of Fig. 2.3). The remote data handles are treated like the original data in the first version and the basic structure of the program, i.e. the composition of two process instantiations, remains the same.

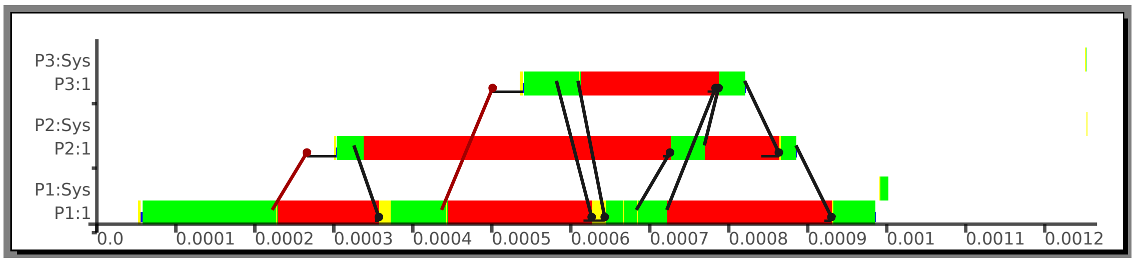
This is similar to the previous process composition example using dynamic channels in Figure 2.1, where the structure of the process composition had to be changed because of the receiver initiated channel creation. We had to be very careful to put demand on the instantiation of process 1 in order to avoid a deadlock situation. A revised version of the process composition example using remote data is depicted in Figure 2.4a and Figure 2.4b. With remote data, the demand problem vanishes, as we use remote data structurally like the original data, also the demand structure of the original program is preserved. The runtime trace visualisations in Figure 2.4c and Figure 2.4d reveal that the runtime behaviour is similar. The remote data example uses 1 thread and 1 message less than the example using dynamic channels. This is due to the tuple usage in the previous example, causing the use of an additional thread and a additional message per tuple in or output. Even though the use of remote data also involves the creation of an additional

```
f' :: Int → Int                                                    --
f' inp = res where

  f1 :: Int → RD Int
  f1 x     = release $ x+1

  f2 :: RD Int → Int
  f2 rd_y = fetch rd_y + 1

  z        = f1 $# inp
  res      = f2 $# z
```

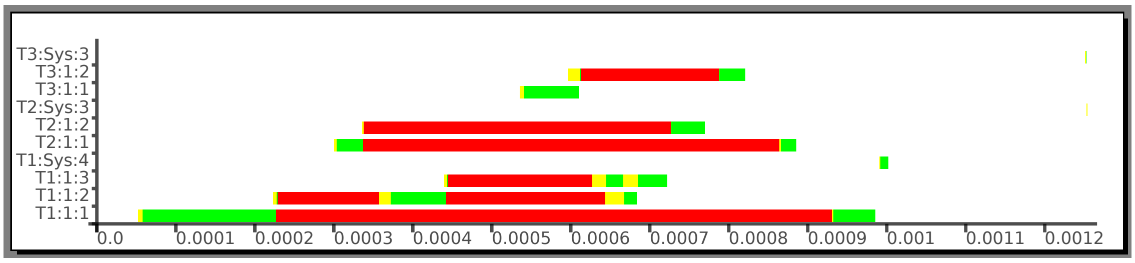
(a) code



(b) process scheme



(c) runtime trace: processes per machines view



(d) runtime trace: threads view

Figure 2.4: A simple example for remote data usage

thread and an additional message, the overhead caused by two tuples outweighs the use of one remote data handle. Comparing the effective runtime, we observe that the remote data example is about 0.0001 seconds slower. This is because the extra channels for the tuple in and outputs are used concurrently while the extra channel for the remote data handle is used in sequence, delaying the execution. ◁

A problem arises when remote data needs to be duplicated. Channel names (of type `ChanName a`) cannot be used more than once to retain referential transparency [LOMP05]. As remote data is implemented as a specialised channel name, it must not be duplicated and fetched several times in parallel. To duplicate Remote Data on a node it is necessary to fetch the data and release it again repeatedly.

Runtime support has been implemented for optimised communication on the same machine[15]. Communication is avoided if the released data of the sending process is referenced by the fetched data on the receiving process which both share the same heap (machine). This optimisation applies to all Eden channels, independent of their construction (remote data channels, dynamic channels or implicit channels).

2.4 Eden Implementation

The Eden implementation is split into parallel runtime system support and primitive parallel operations, which are built into the Eden version of GHC, a branch of the upstream GHC [GHC15], and wraped on library level in the `ParPrim` module. The Eden module defines the previously introduced Eden functions based on the `ParPrim` module. Both are part of the `edenmodules` package available on hackage:

`http://hackage.haskell.org/package/edenmodules`.

Parallel Primitives

The Eden module is based on the primitive parallel operations of the Eden compiler, which are wrapped at Haskell level in the module `Control.Parallel.Eden.ParPrim`. The `parPrim` module exposes functions

```
noPe   :: IO Int
selfPe :: IO Int
```

which are versions of the previously presented `selfPe` and `noPe` in the `IO`-Monad. Eden's channel handles of type `ChanName` include overloaded send functions and are based on primitive channels

15 This was implemented by Jost Berthold, thanks.

```
data ChanName' a = Chan Int# Int# Int# deriving Show
```

which are mere triples of primitive integers to identify machine, process and inport, corresponding to a primitive channel created by the IO action:

```
createC :: IO ( ChanName' a, a )
```

Before sending data, we have to connect to a channel

```
connectToPort :: ChanName' a → IO ()
```

Then we can send data to the connected receiver in a custom mode.

```
sendData :: Mode → a → IO ()

data Mode = Data    -- data to send is single value
          | Stream  -- data to send is element of a list/stream
          | Instantiate Int -- data is IO(), receiver to create a thread for it
          | Connect -- announce sender at receiver side
```

where messages in

- `Data` mode are sent in a single chunk.
- `Stream` mode are sent element wise (only for lists).
- `Instantiate` mode cause a process creation on the receiver side, the contained message is here the function which will be evaluated by the process.
- `Connect` mode announce the sender at the receiver machine. This allows to close the channel and inform the sender about the breakup by the receiver if a message or remainder of a stream is no longer needed.

For further implementation details see [BL07a].

PA-Monad

The **P**arallel **A**ction Monad (PA-Monad) is used to wrap the IO-Monad for parallel actions defined in the Eden module:

```
newtype PA a = PA { fromPA :: IO a } deriving (Monad,MonadFix,
                                               Functor,Applicative)
runPA :: PA a → a
runPA = unsafePerformIO ∘ fromPA
```

The PA constructor is not exported by the Eden Module, thus it is not possible to put an arbitrary `IO` action into the `PA`-Monad[16]. Inside the Eden Module, there are some actions defined in the PA-Monad[17] which can be freed from the Monad using `runPA`, a wrapper for `unsafePerformIO`. One can define a custom sequence of parallel actions eagerly using `runPA`, without the need for `unsafePerformIO` to get rid of the Monad. Of course, `runPA` is basically the same, but it can only be used to free selected actions from the Monad. This way we basically declare "it is fine to use `runPA`". We do not need to encourage programmers to use `unsafePerformIO`.

Type class `Trans`

Central to the Eden implementation is the type class `Trans` for transmissible data.

```
class NFData a ⇒ Trans a where
    write :: a → IO ()
    write =...
    createComm :: IO (ChanName a, a)
    createComm =...

newtype ChanName a = Comm (Trans a ⇒ a → IO())
```

It defines the function `createComm`, which produces a Channel with an open in-port at the local machine. `createComm` returns a handle of type `ChanName a` to the channel and a variable of type `a` which will contain the value of the channel after receiving it. The handle itself is merely an IO-monadic send function, which will be used at the sender side and is based on the overloaded function `write`. `write` is only used to implement `createComm`, it is not used outside of the `Trans` class. Details of the instance declarations of `Trans` will be discussed in Chapter 6.1.2.

Re-Implementing Remote Data

We implemented PA-Monadic versions of `fetch` and `release`, to allow for custom sequences of parallel actions generating and using remote data.

To `release` a value `val`, we use `createComm` to create a *channel handle, value* pair, where the channel handle `cc` is synonym for the remote data handle, which will be used to send a second channel handle `Comm sendValC` to the process currently evaluating `releasePA`. We

16 Of course it is possible to define e.g. `return(putStrLn "unsafeIO") :: PA (IO ())`, but `PA (putStrLn "unsafeIO") :: PA ()`, and thus `runPA (PA (putStrLn "unsafeIO")) :: ()` is not possible.

17 like instantiate or fetchPA

fork a thread to evaluate the send function `sendValC` applied to `val` after receiving the channel handle[18] and return the remote data handle `cc` immediately:

```
releasePA :: Trans a
             ⇒ a            -- ^ The original data
             → PA (RD a)    -- ^ The remote data handle
releasePA val = PA $ do
  (cc , Comm sendValC) ← createComm
  fork (sendValC val)
  return cc
```

To `fetch` a value, we create a *channel handle, value* pair, where the channel handle `c` will be sent to the releasing process using the send function `sendValCC` wrapped inside the remote data handle. Thus we can receive and return the actual data `val`:

```
fetchPA   :: Trans a
             ⇒ RD a         -- ^ The remote data handle
             → PA a         -- ^ The original data
fetchPA (Comm sendValCC) = PA $ do
  (c,val) ← createComm
  fork (sendValCC c)
  return val
```

Now we can re-implement `fetch` and `release` by `runPA ∘ fetchPA` and `runPA ∘ releasePA`. We also define the eager version `fetchAll` of `fetch` for lists based on `fetchPA`:

```
fetchAll :: Trans a
            ⇒ [RD a] -- ^ The Remote Data handles
            → [a]    -- ^ The original data
fetchAll ras = runPA $ mapM fetchPA ras
```

For convenience, we define `releaseAll :: [RD a] → [a]` similarly using `releasePA`, even though `map release` has no demand problem because the released handles are generated immediately. Note that `fetchAll` and `releaseAll` will block on partially defined input lists structures, as the sequence of actions has to be triggered as a whole in order to yield a result.

18 Note: This implies that the evaluation of the value `val` is also stalled until we receive the channel handle (after `fetchPA` is called on the receiver side), because this evaluation is defined by the send function `sendValC` which is part of the channel handle. This applies equally for the previous implementation, but the cause is more transparent in the new implementation.

Process Creation

Function `createComm` is also used to define a process abstraction:

```
data Process a b
    = Proc (ChanName b →              -- send back result, overloaded
            ChanName' (ChanName a) → -- send input Comm., not overloaded
            IO ()
           )

process :: (Trans a, Trans b) ⇒ (a → b) → Process a b
process f = Proc f_remote
    where f_remote (Comm sendResult) inCC
              = do (sendInput, input) ← createComm
                   connectToPort inCC
                   sendData Data sendInput
                   sendResult (f input)
```

Function `process` converts a function of type `(a→b)` to a process abstraction of type `Process a b`, which describes the processes behaviour on the remote machine. The first parameter of the process abstraction (`Comm sendResult`) is a channel handle which will be used to send the processes result back to the caller process. The second parameter `inCC`, a nested primitive channel is similar to a remote data handle to fetch the processes input. Here, the use of the primitive channel avoids the creation of an additional thread which would be the case by using an overloaded channel of type `ChanName a`. The remote process creates a channel which will be used to receive its input. The channel handle is passed to the caller process as well as the result of `f` applied to the `input`. The whole `do` block of the process definition would be equivalent to `sendResult ∘ f ∘ fetch $ inCC` (except the additional thread) if it would be defined with remote data instead of the nested primitive channel.

A process abstraction can be instantiated by function `instantiate`:

```
instantiate :: (Trans a, Trans b) ⇒ Int  -- ^Machine number
                 → Process a b           -- ^Process abstraction
                 → a                     -- ^Process input
                 → PA b                  -- ^Process output
instantiate p (Proc f_remote) procInput
  = PA $ do (sendResult, r)        ← createComm   -- result communicator
            (inCC, Comm sendInput) ← createC      -- reply: input communicator
            sendData (Instantiate p)
                     (f_remote sendResult inCC)
            fork (sendInput procInput)
            return r
```

Firstly, instantiate creates a channel to receive the processes result `r`. Below, the use of `sendData` with mode `Instantiate` instantiates `p` on a remote machine. The rest of the code makes the processes input `procInput` remotely available. Here is the corresponding code using remote data instead of the nested primitive channel:

```
  = do (sendResult, r) ← PA createComm        -- result communicator
       inCC            ← releasePA procInput  -- remote input
       PA (sendData (Instantiate p)
                    (f_remote sendResult inCC))
       return r
```

We release the `procInput` and send the released input handle `inCC` with the instantiate action.

With instantiate we can define the `(#)` operator simply as:

```
(#) :: (Trans a, Trans b) ⇒ Process a b → a → b
p # x = runPA $ instantiateAt 0 p x
```

Function `spawnAt` for eager process instantiation of lists can be defined as a simple sequence of instantiate actions, which are escaped to the functional world by `runPA`:

```
spawn :: (Trans a, Trans b)
      ⇒ [Process a b]  -- ^Process abstractions
      → [a]            -- ^Process inputs
      → [b]            -- ^Process outputs
spawn ps is = runPA $ zipWithM (instantiateAt 0) ps is
```

Similarly, the skeleton programmer can define custom instantiation actions, e.g. to eagerly instantiate tuples with different types:

```
spawnT2 :: (Trans a, Trans b, Trans c, Trans d)
        ⇒ (Process a b, Process c d)  -- ^Process abstractions
        → (a,c)                       -- ^Process inputs
        → (b,d)                       -- ^Process outputs
spawnT2 (p1,p2) (i1,i2)
  = runPA $ do r1 ← instantiateAt 0 p1 i1
               r2 ← instantiateAt 0 p2 i2
               return (r1,r2)
```

These monadic-style functions conflict with the functional programming notation of Eden programs, but when defined in separate function definitions with a pure interface, they are a feasible and easy way to overcome laziness which in turn conflicts with the necessity of eager execution of parallel actions like process instantiations.

Dynamic Channels

With `createComm` we can also define dynamic channel creation and use:

```
new :: Trans a ⇒ (ChanName a → a→ b) → b
new chanValCont = unsafePerformIO $ do
        (chan , val) ← createComm
        return (chanValCont chan val)
```

The definition of `new` is straight forward. It takes function parameter `chanValCont` and uses `createComm` to create a channel. `new` provides `chanValCont` with channel handle `chan` and the subsequently received value `val`.

We use a channel handle `(Comm sendVal)` to send a value `val` with function `parfill`:

```
parfill :: Trans a ⇒ ChanName a → a → b → b
parfill (Comm sendVal) val cont
    = unsafePerformIO (fork (sendVal val) » return cont)
```

`parfill` forks a thread to concurrently send value `val` with the send function `sendVal`. The third parameter `cont` is returned as the functions result.

Both functions, `new` and `parfill` use `unsefePerformIO` to escape the `IO`-monad.

2.5 Algorithmic Skeletons in Eden

Eden provides a skeleton library `edenskel` which is available on hackage:
`http://hackage.haskell.org/package/edenskel`.
In the following, we present a small set of skeletons from the skeleton library which we will use later in this thesis.

The Parallel Map Skeleton

Skeletons can be easily expressed in Eden as parallel higher order functions. A simple example is `parMap`, a parallel `map` variant which produces one process per list element. It can be easily expressed using `spawnF`:

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

The Ranch Skeleton

The `ranch` is a parallel map variant which takes arbitrary input of type `a`, a `transform` function which generates the parallel `map` input list and a function to `reduce` the parallel result. Each process evaluates `f` on its input:

```
ranch :: (Trans b, Trans c)
         ⇒ (a → [b])   -- ^input transformation function
         → ([c] → d)   -- ^result reduction function
         → (b → c)     -- ^worker function
         → a           -- ^input
         → d           -- ^output
ranch transform reduce f xs = reduce ∘ parMap f ∘ transform $ xs
```

The Farm Skeleton

The `farm` is a parallel map variant which increases work granularity per process. It can be implemented using the `ranch` skeleton. It is controlled by function parameters `distr` and `combine` for input distribution and result combination, which implement a static task distribution[19]. Each process evaluates `f` on a stream of data:

```
farm :: (Trans a, Trans b)
        ⇒ ([a] → [[a]])  -- ^input distribution function
        → ([[b]] → [b])  -- ^result combination function
        → (a →  b)       -- ^map function
        → [a]            -- ^input
        → [b]            -- ^output
farm distr combine f xs = ranch distr combine (map f) xs
```

The Ring Skeleton

A typical topology skeleton is the `ring`. In Listing 2.1 we present a simple version. The ring takes a list of ring-worker functions (cycled) and a list of inputs, one for each process respectively. The processes take further a ring input of type `i` from their left neighbor. If the type of the ring input is a list (`i ~ [i']`), then the ring processes are connected by a stream of data, thus the ring is something like a cycled pipeline. The ring connections are exchanged via the caller process, where we right-rotate the ring inputs and re-feed them lazily to the processes. This indirection of messages is inefficient, especially if there are ring inputs of bigger size or if the ring processes are connected by streams.

In an optimised version we use dynamic channels to shortcut the indirection (see Listing 2.2).

Listing 2.1: A simple ring skeleton

```
ringFlSimpleAt :: (Trans a,Trans b,Trans r)
        ⇒ Places                 -- ^where to put workers
        → [(a → r → (b,r))]      -- ^ring process functions
        → [a]                    -- ^ring input
        → [b]                    -- ^ring output
ringFlSimpleAt pls fs as = bs where
  (bs, ringOuts) = unzip $ spawnFAt pls (map uncurry (cycle fs))
                                        (zip as $ lazy ringIns)
  ringIns        = rightRotate ringOuts

rightRotate :: [a] → [a]
rightRotate [] =  []
rightRotate xs = last xs : init xs
```

19 Unlike the `farm` in [BDO+95] and [Kuc02], which implement dynamic task distribution

Listing 2.2: A ring skeleton interconnected by dynamic channels

```
ringFlChanAt :: (Trans a, Trans b, Trans r)
        ⇒ Places                   -- ^where to put workers
        → [(a → r → (b,r))]      -- ^ring process functions
        → [a]                      -- ^ring input
        → [b]                      -- ^ring output
ringFlChanAt pls fs as = bs where
  (bs, ringOuts) = unzip $ spawnFAt pls (map (link ∘ uncurry) (cycle fs))
                                        (zip as $ lazy ringIns)
  ringIns        = leftRotate ringOuts

link :: (Trans i, Trans o, Trans r)
        ⇒ ((i,r) → (o,r))                         -- ^ ring process function f
        → ((i, ChanName r) → (o, ChanName r)) -- ^ f with dynamic channels
link  f (i, ch)  = new (λch' ringIn → let (o, ringOut) = f (i, ringIn)
                                       in  parfill ch ringOut (o,ch') )
leftRotate :: [a] → [a]
leftRotate [] =  []
leftRotate (x:xs) = xs ++ [x]
```

Listing 2.3: A ring skeleton interconnected by remote data

```
ringFlAt :: (Trans a,Trans b,Trans r)
        ⇒ Places                   -- ^where to put workers
        → [(a → r → (b,r))]      -- ^ring process functions
        → [a]                      -- ^ring input
        → [b]                      -- ^ring output
ringFlAt pls fs as = bs where
  (bs, ringOuts) = unzip $ spawnFAt pls (map (toRD ∘ uncurry) (cycle fs))
                                        (zip as $ lazy ringIns)
  ringIns        = rightRotate ringOuts

toRD :: (Trans i, Trans o, Trans r) ⇒
        ((i,r) → (o,r))            -- ^ ring process function f
        → ((i, RD r) → (o, RD r)) -- ^ f with remote data
toRD  f (i, ringIn)  = (o, release ringOut)
  where (o, ringOut) = f (i, fetch ringIn)
```

Instead of returning the ring outputs, the processes use function `link` to create dynamic channels and pass the channel handles to the caller, in order to receive their ring inputs directly from their left neighbours. Inversely to the simple version, the caller has to left-rotate the channel handles before returning them to the processes.

The same optimisation can be made using remote data (see Listing 2.3). Analogously to function `link` we use function `toRD` to lift the ring functions to remote data in their second component. Due to the use of remote data, this function lifting is the only change to the original skeleton. We don't need to change the code structurally, in this case the direction of the rotation.

Listing 2.4: A simple master worker skeleton

```
mw :: (Trans t, Trans r)
    ⇒ Int → Int      -- #workers, prefetch
    → (t → r)        -- worker function
    → [t] → [r]      -- what to do
mw np prefetch wf tasks = ress
  where
   (reqs, ress) = unzip ∘ merge ∘ tagF $ outs
   tagF         = zipWith zip [[i,i..] | i ← [0..np-1]]
   outs         = parMap (map wf) inputs
   inputs       = distribute np (initReqs ++ reqs) tasks
   initReqs     = concat $ replicate prefetch [0..np-1]
```

The Master Worker Skeleton

Master Worker skeletons implement dynamic load balancing among workers. The load balancing is actively generated by a single master process. The skeleton takes the number of worker processes `np`, a `prefetch` parameter, the worker function `wf` which will be mapped on the tasks by the worker processes and the `tasks` as input. The tasks are dynamically split into `np` sublists `inputs`. This split is defined by auxiliary function `distribute` (from `Control.Parallel.Eden.Auxiliary`) and controlled by a list of requests `reqs`. The workers are instantiated using `parMap` with the `inputs`. Each process receives one task per request. Initially, each process shall get a buffer of `prefetch` tasks and accordingly, `prefetch` initial requests `initReqs` are generated. The worker outputs are tagged with the process ids and nondeterministically `merge`d by the order of arrival. After merging, these tags are used as additional work request `reqs`. Each worker gets an additional task for each result. This simple workpool does not sort the results by the initial task order. See `Control.Parallel.Eden.Workpool` for sorting versions.

CHAPTER 3

Skeleton Nesting

In functional languages, nesting of skeletons works typically out of the box. This is because they are implemented as higher order functions. One can nest arbitrary skeletons using a skeleton as function parameter of another skeleton. The process tree of the parameter skeleton will be appended to the process tree of the caller skeleton. Skeleton nesting is hence the *natural way* of composing skeletons in functional languages.

3.1 Divide and Conquer Skeletons

A simple example for the latter case of skeleton nesting is a parallel divide and conquer skeleton which can be defined exploiting a distributed expansion scheme by nesting itself:

```
parDC :: (Trans a, Trans b)
         (a → Bool)        -- trivial?
         → (a → b)         -- solve
         → (a → [a])       -- split
         → (a → [b] → b)   -- combine
         → a               -- input
         → b               -- result
parDC trivial solve split combine x
  | trivial x = solve x
  | otherwise = combine x $ parMap dc $ split x
  where dc = parDC trivial solve split combine
```

The skeleton takes function parameters to

- check if a task is *trivial*,
- to *solve* a trivial task,
- to *split* a task into sub-tasks and
- to combine solved sub-tasks.

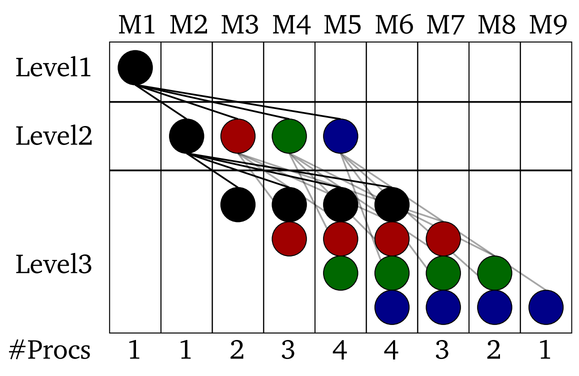
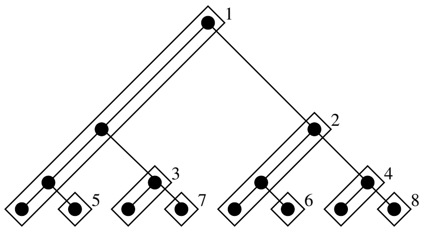


Figure 3.1: 4-ary tree, depth 2; implicit process placement

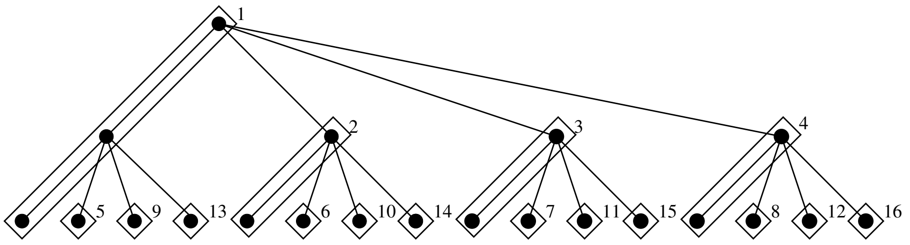
The `parDC` skeleton splits tasks recursively into sub-tasks until a task is trivial. Skeleton `parMap` is used with function parameter `parDC` to realise the recursion in parallel. Skeleton `parDC` will call `parMap` again until a trivial task is met. The problem of such recursive process instantiations in Eden is the mapping of processes to machines. Edens current implementation uses by default an implicit mechanism to assign processes to machines in turn, meaning round robin among the machines. Eden does not use a global counter, instead each machine has its own counter. This leads to an uneven mapping of processes to machines for the case of implicit process placement when instantiating process trees (see Figure 3.1).

An even mapping of process to machines is desirable. This was realised for regular divide and conquer problems with a *fixed branching degree* on every level (see Figure 3.2) and a ticket mechanism, which can be used in combination with explicit process placement.

> “Figure 3.3 shows a distributed expansion divide and conquer skeleton for k-ary task trees. Besides the standard parameter functions, the skeleton takes the branching degree, and a ticket list with PE numbers to place newly created processes. The left-most branch of the task tree is solved locally, other branches are instantiated using the Eden function `spawnAt`, which instantiates a collection of processes (given as a list) with respective input, on explicitly specified PEs. Results are combined by the `combine` function.
>
> ***Explicit Placement via Tickets.*** The ticket list is used to control the placement of newly created processes. First, the PE numbers for placing the immediate child processes are taken from the ticket list. Then, the remaining tickets are distributed to the children in a round-robin manner using the function `unshuffle :: Int → [a] → [[a]]` which unshuffles a given list into as many lists as the first parameter tells. Child computations will be performed locally when no more tickets are available. The explicit process placement via



(a) Binary tree, depth 3



(b) 4-ary tree, depth 2

Figure 3.2: Divide and conquer; explicit process placement

```
dcN :: (Trans a, Trans b) ⇒
       Int → [Int] →      -- branch degree / tickets
       DivideConquer a b
dcN k tickets trivial solve split combine x
   | null tickets = seqDC x
   | trivial x    = solve x
   | otherwise
        = childRes `seq`  -- demand control: ...
          rnf myRes `seq`  ... first start children
          rnf localIns `seq` -- then evaluate locally
          combine ( myRes:childRes ++ localRess )
  where
     -- sequential computation
     seqDC x = if trivial x then solve x
               else combine (map seqDC (split x))
     -- child process generation
     childRes   = spawnAt childTickets childProcs procIns
     childProcs = map (process ∘ rec_dcN) theirTs
     rec_dcN ts = dcN k ts trivial solve split combine
     -- ticket distribution
     (childTickets, restTickets) = splitAt (k-1) tickets
     (myTs: theirTs)             = unshuffle k restTickets
     -- input splitting
     (myIn:theirIn)      = split x
     (procIns, localIns)
        = splitAt (length childTickets) theirIn
     -- local computations
     myRes     = rec_dcN myTs myIn
     localRess = map seqDC localIns
```

Figure 3.3: Distributed expansion divide and conquer skeleton for k-ary task trees.

ticket lists is a simple and flexible way to control the distribution of processes as well as the recursive unfolding of the task tree. If too few tickets are available, computations are performed locally. Duplicate tickets can be used to allocate several child processes on the same PE. The numbers in Figures 3.2a and 3.2b give the PE numbers when the ticket lists `[2..8]` and `[2..16]` are used for placement, respectively.

Demand Control. As Haskell's lazy evaluation would suppress any parallel evaluation, we need to add explicit demand for starting parallel child processes and to ensure early evaluation of local subresults. In the `dcN` skeleton (lines 8 and 9 in Figure 3.3), we first force the creation of the child processes (using `seq`), and then fully evaluate all local computations (using the `rnf` strategy) , before combining all subresults to the overall result."[20]

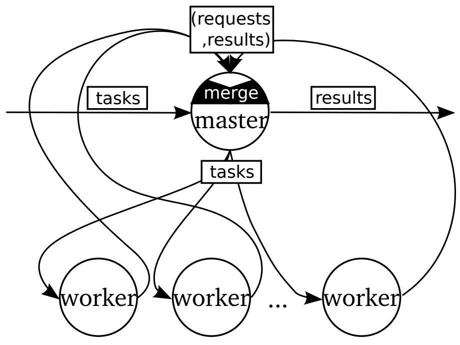
For the presented divide and conquer skeletons, skeleton nesting is the principle of their construction. We will next look at master worker skeletons. Here the basic skeleton version is unnested, but it can be nested to create a hierachical master worker skeleton.

3.2 Hierarchical Master Worker Skeletons

The basic master worker skeleton implements dynamic load balancing among worker process via a single master process (see Figure 3.4). The skeleton input is a list of tasks. Every worker gets an initial amount (*prefetch*) of tasks. Each time when a task is solved by a worker, it returns the result and a request to the master, which will respond with a new task until the task-pool is exhausted. The skeleton is non recursive and not nested.

```
mw :: (Trans task, Trans result)
   ⇒ Int                  --number of workers
   → Int                  --prefetch of tasks
   → (task → result)      --worker function
   → [task] → [result]    --what to do
```

(a) Signature



(b) Visualisation

Figure 3.4: Master worker skeleton

20 [BDLL09, pages 49-50]

The bottleneck of this architecture is the master process, which can slow down the whole system when it is overloaded. To unload the master, one can introduce a hierarchy of submasters, which are served by a master and serve on their part further submaster process or worker processes. This can be defined by nesting the skeleton.

> "To simplify the nesting, the basic skeleton `mw` is modified in such a way that it has the same type as its worker function. We therefore assume a worker function `wf :: [t] → [r]`, and replace the expression `(map wf)` in the worker process definition with `wf`. This leads to a slightly modified version of `mw`, denoted by `mw'` in the following. An elegant nesting scheme (taken from [Pri06]) is defined in Figure 3.5. The parameters specify the branching degrees and prefetch values per level, starting with the root parameters. The length of the parameter lists determines the depth of the generated hierarchical system.
>
> The nesting is achieved by folding the zipped branching degree and prefetches lists, using the proper worker function, of type `[t] → [r]`, as the starting value. The folding function corresponds to the `mw'` skeleton applied to the branching degree and prefetch value parameters taken from the folded list and the worker function produced by folding up to this point.
>
> The parameters in the nesting scheme above allow to freely define tree shape and prefetch values for all levels. As the `mw` skeleton assumes the same worker function for all workers in a group, it generates a regular hierarchy, one cannot define different branching or prefetch within the same level. It is possible to define a version of the static nestable work pool which is even more flexible (not considered here), yet more simple skeleton interfaces are desirable, to provide access to the hierarchical master worker at different levels of abstraction."[21]

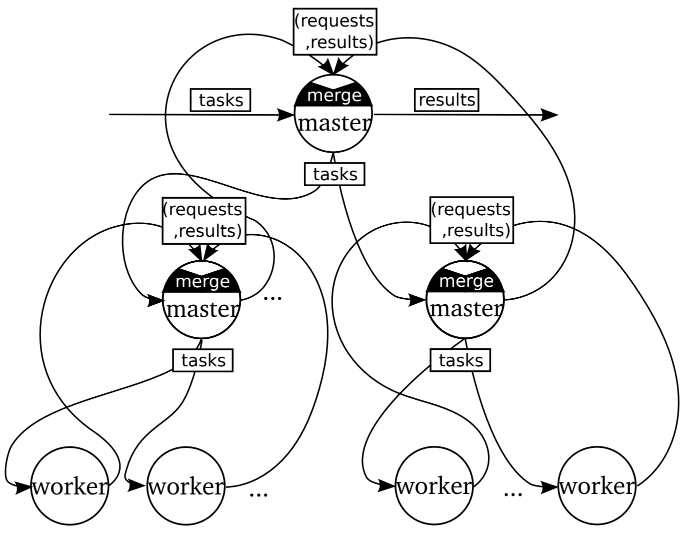
Once again, Eden's default instantiation mechanism leads to an uneven mapping of processes to machines. The ticket mechanism described in the previous section can also be used here, the nesting is a bit more complicated with the ticket mechanism and we implemented it with recursion (see Chapter A.1 for an implementation).

So far, we have investigated skeleton nesting where each nested skeleton was called by (hence connected to) a single process of the caller skeleton. Now we change the subject slightly, we compose skeletons in sequence and we want to connect arbitrary processes of the involved skeletons.

21 [BDLP08, page 251]

```
mwNested :: (Trans t, Trans r) ⇒
            [Int] → [Int] →   -- branching/prefetches per level
            ([t] → [r]) →     -- worker function
            [t] → [r]         -- tasks, results
mwNested ns pfs wf = foldr (uncurry mw') wf (zip ns pfs)
```

(a) Code



(b) Visualisation

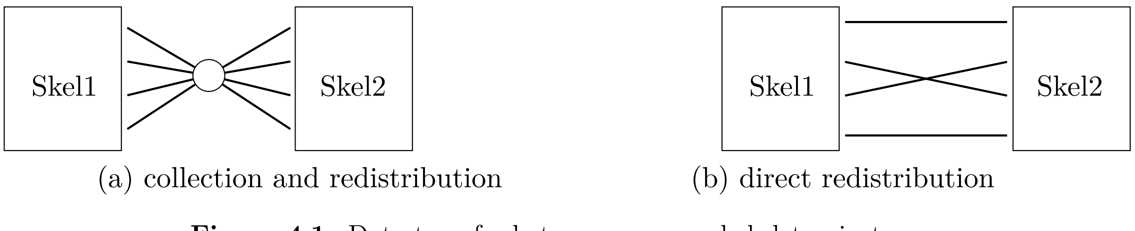
Figure 3.5: Static nesting with equal level-wise branching

CHAPTER 4

Skeleton Composition: Sequences

"Skeletons should be small and simple to instantiate to increase the ease and flexibility of their use. In particular, it should be possible to compose and nest skeleton instantiations arbitrarily. This means for the case of a distributed memory setup and structured data that must be passed from one skeleton to the next that the result of the first skeleton is gathered in a single process and redistributed for the following skeleton execution. This causes unnecessary communication and holds the danger of a communication bottleneck in the caller process (see Figure 4.1 (a)). A typical example is the composition of two parallel maps (parallel task farms) producing a two dimensional matrix with an intermediate transpose.

There exist several proposals to avoid the gathering and redistribution of distributed data. One could introduce a new distributed data type as common in languages with a data-parallel concept [KC02, DGTY95b] where data can be passed in a distributed manner. In this case, one needs special transformation and conversion functions to redistribute the distributed data or to switch between distributed and common data types. Another simple alternative would be to design a new integrated skeleton for the composition by merging the two skeleton instantiations and organising the redistribution explicitly



(a) collection and redistribution (b) direct redistribution

Figure 4.1: Data transfer between composed skeleton instances

> within the new skeleton context. This approach has the disadvantage that the programmer has to go into the internals of skeleton design and that the clarity of the original composition is lost."[22]

On the other hand, this approach of manually merging skeletons is the only possibility of optimisation when lacking a concept for efficient skeleton composition. In the following section, we discuss some examples for manually merged skeleton compositions which initially motivated this thesis.

4.1 Skeleton Sequencing by Hand: Monolithic Skeletons

To overcome the gathering and redistribution of data in between different skeleton instances, it has often been necessary to integrate two skeletons in a single, optimised skeleton. We will exemplify this by three case studies, a Distributed Workpool Skeleton which is logically composed from a reduce skeleton built from a hierarchy of collector processes and a ring skeleton of worker processes. In the second example, we present the implementation of the Google map-reduce framework as Eden skeleton [BDL09] and as third case study we present a parallel map-and-transpose skeleton, which we used to calculate FFT [LBDL09]. The latter two skeletons can be defined as an application of a more general `all2All` skeleton, introduced in Chapter 4.1.2.

4.1.1 Distributed Workpool Skeletons

Distributed workpool skeletons – *like Master-Worker skeletons* – implement dynamic load balancing among worker processes, but – *unlike Master-Worker skeletons* – the load balancing is performed without a central master process in a distributed way. In [DBL10] (based on [Die07]), an implementation of such a distributed workpool skeleton has been described:

> "The distributed work pool skeleton uses a set of workers to solve a list of initial tasks received from the caller. Each worker holds a local task pool, and maybe a local state. New tasks may be created and added while solving the initial task set. Load balancing is achieved by a demand-driven exchange of surplus tasks.
>
> *Skeleton Interface and Application*
> Fig. 4.2 shows the interface of the skeleton, which allows to customise its functionality by a large set of parameter functions. While the last two parameters

22 [DHL10, pages 73-74]

```
distribWP :: (Trans t, Trans r, Trans s, NFData r') ⇒
  Int    →                            -- no of processes
  -- task processing and result post processing
  ([(t,s)] → [(Maybe (r',s),[t])]) → -- worker function wf
  ([Maybe (r',s)] → s → [r])       → -- result transform function resTf
  ([[r]] → [r])                    → -- result merge function
  -- work pool transformation
  ([t] → [t] → s → [t])            → -- attach function ttAf
  ([t] → s → ([t],[t]))            → -- split  function ttSplitf
  ([t] → s → ([t],Maybe (t,s)))    → -- detach function ttDf
  -- state comparison function
  (s → s → Bool)                   → -- compare function cpSf
  -- initialisation
  s  → [t]                         → -- initial state initS/tasks initTs
  [r]                                 -- results
```

Figure 4.2: Interface of the General Distributed Work Pool Skeleton

> provide the initial state and task list, the first parameter specifies the number of processes to be created. The skeleton creates a ring of worker processes together with a hierarchy of collector processes. The latter is used to speed-up result post-processing. Three functions determine task processing and result post processing, i.e. the proper worker functionality. The work pool is manipulated with the following three parameter functions of the general skeleton: the task pool transformation and attach function `ttAf` is used to extend the work pool with newly created tasks, the function `ttSplitf` is used to split the work pool when an external work request arrives, and the function `ttDf` detaches a single task for local evaluation. Different selection strategies can be used for serving oneself via `ttDf` and other workers via `ttSplitf`. Finally, the state comparison function is used for branch-and-bound algorithms to select the optimal solution (state)."[23]

As the quotation states, the skeleton is highly customisable and is implemented on top of a ring topology and a hierarchy of collector processes is included. We actually need some more parameters of the skeleton to customise the shape of the collector hierarchy, which have been omitted in this presentation for the sake of simplicity. We could leave these parameters out and remove the collector hierarchy from the skeleton if we would compose the skeleton from a collectors/reducer skeleton and a basic distributed workpool skeleton (built from a ring skeleton) which would also reduce the complexity of the implementation, compared to the monolithic implementation originally developed in [Die07]. It may seem that the functionality could be composed from a ring of workers nested within a tree of collectors. But by nesting the ring in a tree-shaped reducer skeleton, we would have to call it from the leaf processes of the reducer skeleton. Then it would get instantiated once

23 [DBL10, pages 339-340]

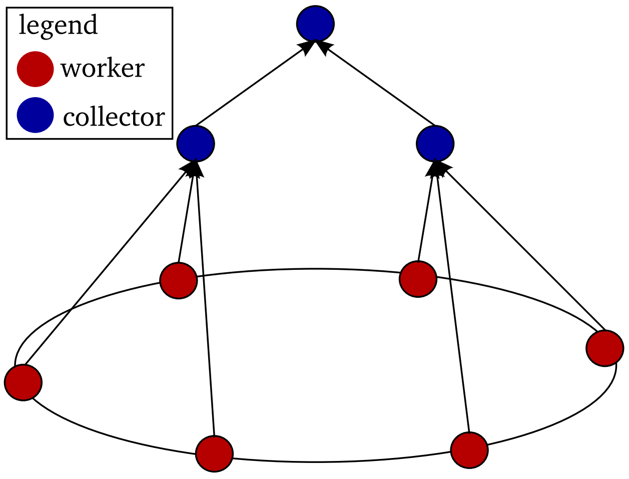


Figure 4.3: Ring of worker and tree of collector processes

for each leaf process, this is not what we seek. The leaf processes should rather correspond to a subset of the ring processes of a single ring instance (see Figure 4.3). We present a compositional implementation at the end of this chapter.

Listing 4.1: The monolithic `allToAll` skeleton

```
allToAll :: (Trans a, Trans b, Trans c) ⇒
              (Int → a → [b])     -- function before allToAll exchange
           → ([b] → c)     -- function after allToAll exchange
           → [a] → [c]     -- input / output
allToAll f1 f2 as = cs where
  myProcs = unzip ∘ (parMap $ uncurry $ allToAllWorker np f1 f2)
  (cs,chanss) = np `pseq` myProcs $ zip as (lazy $ transpose chanss)
  np = length as

allToAllWorker :: Trans b ⇒ Int → (Int → a → [b]) → ([b] → c) →
              a → [ChanName b] → (c,[ChanName b])
allToAllWorker np f1 f2 a theirChanNs
  = let (myChanNs, bs') = createChans np
        bs = f1 np a
        cs = f2 bs'
    in  (multifill theirChanNs bs cs, myChanNs)

--create n (channels,inputs)
createChans :: Trans x ⇒ Int → ([ChanName x], [x])

--send n outputs over n channels and return input b
multifill :: Trans x ⇒ [ChanName x] → [x] → b → b
```

4.1.2 The `allToAll` Skeleton

The `allToAll` skeleton realises an all-to-all data exchange – a distributed transpose phase – in between two parallel computation phases and is used for the implementation of the next two case studies: the Google map-reduce framework as Eden skeleton [BDL09] and a parallel map-and-transpose skeleton. The code of the skeleton is listed in Listing 4.1.

The skeleton is controlled by two function parameters and the input list `as`. By calling `parMap` on the input list[24], the skeleton creates one process per list element, which each use the first function `f1` and the assigned list element `a` to generate a second list `bs`, which will be distributed among all processes, one element per process. Therefore we use `multifill`[25] on a list of dynamic channels to all processes `theirChanNs` and the list of intermediate results `bs`. Beforehand, we need to create and exchange channels among all processes using auxiliary function `createChans`[26]. Data (lazily) received via the channel names can be accessed in the second component of the result tuple of `createChans`. Thus the created channel names `myChans` are returned in the second component of the processes result to the main process, where they will be transposed and again passed to the processes, hence each gets one channel from all the other processes.
After receiving the list of *values* `bs'` from the other processes, the second function parameter `f2` is applied to define the final result `cs`, which is returned to the main process[27].

4.1.3 The Parallel Google Map-Reduce Skeleton

"**The computation scheme** of Google map-reduce is depicted in Figure 4.4. In a nutshell, a Google map-reduce instance first transforms key/value pairs into

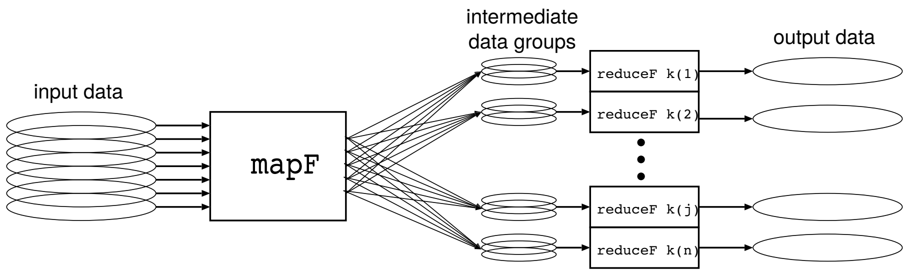


Figure 4.4: Computation scheme of Google map-reduce

24 actually, the initial inputs `as` are `zipped` with the lazily supplied communication channels from all processes `transpose chanss`. Initially only the `as` are available
25 like `parfill`, but for lists of channels and inputs.
26 Creates *np* channels and a corresponding list of received values.
27 The 3rd argument of `multifill` is directly passed as the result

(intermediate) other key/value pairs, using a `mapF` function. After this, each collection of intermediate data *with the same key* is reduced to one resulting key/value pair, using a `reduceF` function. In-between the transformation and the reduction, the intermediate data is grouped by keys, so the whole computation has two logical phases.

[...]

Our optimised implementation uses direct stream communication between mappers and reducers, as depicted in Figure 4.5. Furthermore, instances of mapper and reducer are gathered in one process, which saves some communication. In order to *directly* send the respective parts of each mapper's output to the responsible reducer process via channels, a unidirectional $m : n$ communication must be set up. Each process creates a list of m channels and passes them on to the caller. The latter thus receives a whole matrix of channels (one line received from each worker process) and passes them on to the workers column-wise. Intermediate data can now be partitioned as before, and intermediate grouped pairs directly sent to the worker responsible for the respective part."[28]

We present in Figure 4.6 a revised version of our Google map-reduce skeleton which is implemented as instance of the `allToAll` skeleton of Listing 4.1. Thus, this new version focuses on the mapper and reducer functionality, leaving the communication details to the underlying `allToAll` skeleton. This implementation works with lists of tuples instead of

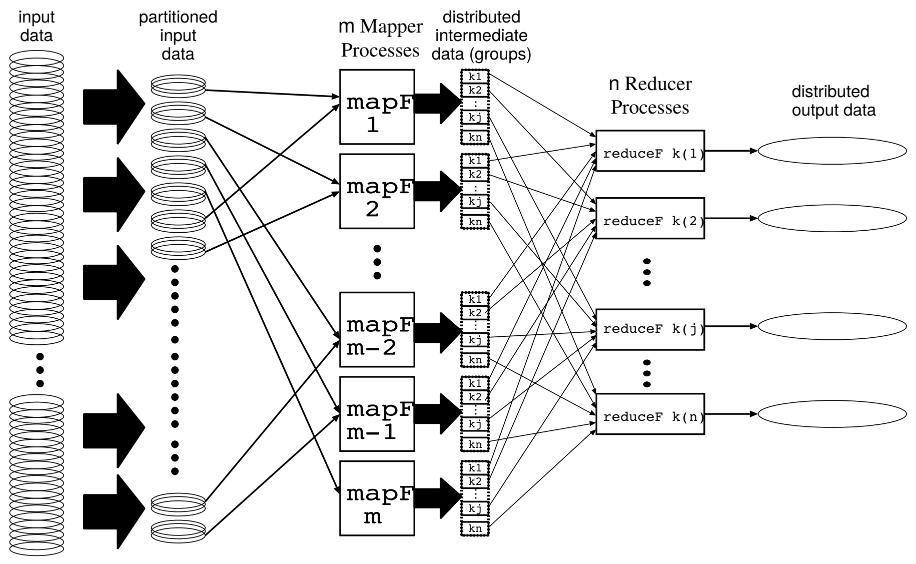


Figure 4.5: Parallel Google map-reduce using distributed transpose functionality

28 [BDL09, pages 993-997, based on [Ber08]]

```
mapReduceList :: forall k1 k2 v1 v2 v3 v4 ∘
  (Trans k1, Trans k2,
   Trans v1, Trans v2, Trans v3, Trans v4,
   Ord k2)
  ⇒ Int             -- Number of partitions
  → (k2 → Int)    -- Partitioning for keys
  → (k1 → v1 → [(k2,v2)])      -- The *'λmap'* function
  → (k2 → [v2] → Maybe v3)     -- The *'λcombiner'* function
  → (k2 → [v3] → Maybe v4)     -- The *'λreduce'* function
  → [[(k1,v1)]]                -- Distributed input data (list)
  → [[(k2,v4)]]                -- Distributed output data
mapReduceList parts keycode mAP cOMBINER rEDUCE input = ress where
  ress = allToAll (const mapper) reducer $ unshuffle parts input
  mapper :: [[(k1,v1)]] → [[(k2,v3)]]
  mapper = map concat ∘ transpose
              ∘ partitionParts ∘ mapCombine where
    mapCombine :: [[(k1,v1)]] → [[(k2,v3)]]
    mapCombine = map (reducePerKeyList cOMBINER)
                 ∘ map groupByKeyList
                 ∘ map (concatMap (uncurry mAP))
    partitionParts :: [[(k2,v3)]] → [[[(k2,v3)]]]
    partitionParts = map (partition parts keycode)

  reducer :: [[(k2,v3)]] → [(k2,v4)]
  reducer = reducePerKeyList rEDUCE
                 ∘ mergeByKeyList ∘ merge
```

Figure 4.6: Implementing GMR as instance of the allToAll skeleton

`Maps`, but a convenient interface using the `Map` type can easily be added. The input list is already chunk-ed into sub-lists, to allow coarse grained stream-processing of the data. This input stream, distributed round robin among the processes by function `unshuffle`, is processed by the `allToAll` skeleton. Each process of the skeleton applies first the mapper, which, per chunk, is composed of mapping the `mAP` function to the sub-lists, then the results are grouped by keys and pre-reduced by the `cOMBINER` function to save communication cost. In preparation of the communication step, the elements are partitioned by a `keycode`, and rearranged such that each keycodes partition is streamed to its corresponding reducer process in the communication step. Thus each process receives map-result streams from all processes for one keycode. The streams are merged nondeterministically by the order of their arrival using function `merge` and reduced per key using function `rEDUCE`. The use of merge improves the performance and is feasible if function `rEDUCE` is commutative, which is a general requirement for function reduce of the Google map-reduce skeleton.

The auxiliary functions are taken from [Ber08]. The implementation based on the `allToAll` skeleton is only possible because it contains function parameters which are applied before and after the transposition in the processes. A real compositional implementation would be composed of three skeletons: `parMap' ∘ parTranspose' ∘ parMap'` where `parMap'` and `parTranspose'` are suitable parallel `map` and `transpose` implementations. In Chapter 4.3.3, we present such a compositional implementation of Google map-reduce.

4.1.4 The Distributed Homomorphism Skeleton

Gorlatch [Gor98] introduced the notion of distributed homomorphisms, which are a subclass of divide and conquer algorithms, where the split step splits the list at its centre[29], the solve step is `id` and the combine step is $zip(\oplus)(u,v) +\!\!+ zip(\otimes)(u,v)$:

Definition 4.1 *For binary operations, $\oplus$ and $\otimes$, a distributable homomorphism is a function $(\oplus \updownarrow \otimes) : [\alpha] \to [\alpha]$, such that for arbitrary lists x and y of equal lengths[30]:*

$$(\oplus \updownarrow \otimes)(x +\!\!+ y) = zip(\oplus)(u,v) +\!\!+ zip(\otimes)(u,v),$$
$$where\ \ u = (\oplus \updownarrow \otimes)x,\ \ v = (\oplus \updownarrow \otimes)y$$

A simple distributed computation scheme can be derived directly from the definition (see Figure 4.7). A distributed homomorphism could be implemented with an butterfly-reduction skeleton (Chapter 4.2.1.4) if it is allowed to use different reduce functions for the left and right part.

Gorlatch presents a more general distributed computation scheme.

Definition 4.2 *Let an input list with length 2^l be nested with $d+1$ dimensions, $1 \leq d < l$ and distributed on d dimensions, such that each PE has a simple list. Let the number of*

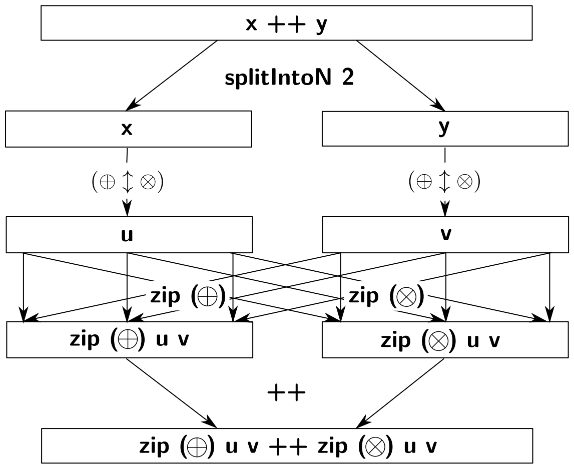


Figure 4.7: Visualisation of the distributed homomorphism computation pattern

29 `splitIntoN 2` from Control.Parallel.Eden.Auxiliary

30 [Gor98], page 8

sub-lists on all dimensions further be equal. The distributed version $\widetilde{h^{(d)}}$ *of a distributed homomorphism h for such an input list can be calculated as follows*[31]*:*

$$\widetilde{h^{(d)}} = \mathop{\circ}_{i=1}^{d} chdim^{(i,i+1)} \circ \mathop{\circ}_{i=1}^{d+1} map^d h \circ chdim^{(i,d+1)}$$

where

- *$map^d h$ maps h to the inner list of the $d+1$-dimensional input,*
- *$chdim^{(i,j)}$ flips dimension i and j of the input and*
- *$\mathop{\circ}\limits_{i=m}^{n}$ denotes function compositions*[32] *for i from m to n.*

This mathematical description of the implementation is elegant and concise. However, it is hard to implement this for arbitrary dimensions in Eden because the number of necessary skeleton compositions is parametric. One can use standard function composition in between the skeleton calls, and thus introduce the overhead of gathering and distributing the data in between all the composition steps. But it is really hard to build a monolithic skeleton, which optimises the communication of the variable number of skeleton compositions by hand. We will pick this example up again at the end of this chapter. For now, let us consider the special case with $d = 1$:

$$\begin{aligned}\widetilde{h^{(1)}} &= \mathop{\circ}_{i=1}^{1} chdim^{(i,i+1)} \circ \mathop{\circ}_{i=1}^{2} map\ h \circ chdim^{(i,2)} \\ &= chdim^{(1,2)} \circ map\ h \circ chdim^{(1,2)} \circ map\ h \circ chdim^{(2,2)} \\ &= transpose \circ map\ h \circ transpose \circ map\ h\end{aligned}$$

Thus we can use a `map ∘ transpose ∘ map` skeleton to implement the distributed homomorphism skeleton for $d = 1$. We want a parallel skeleton for functionality

```
map ∘ transpose ∘ map
```

which agglomerates the input row-wise, say round robin to reduce parallel overhead and thus increase performance.

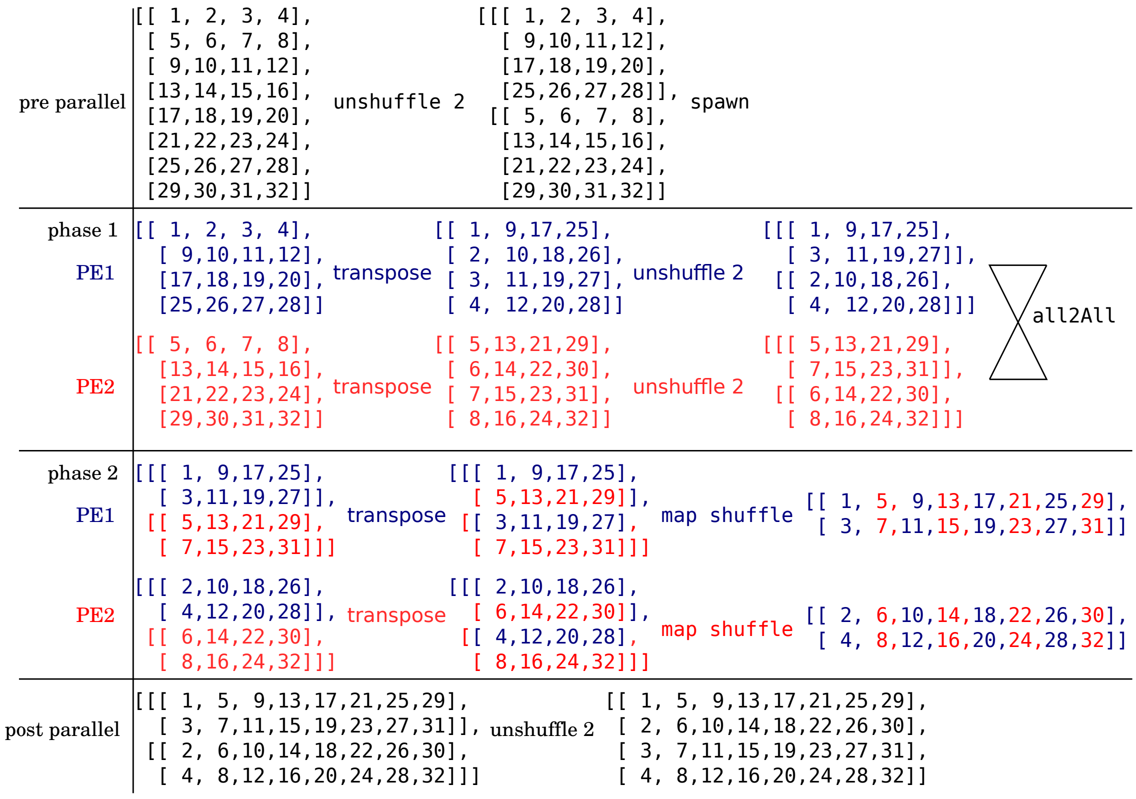
The `parMapTransposeShuffle` skeleton presented in Listing 4.2 realises this. It is a variant of the `allToAll` skeleton (see Listing 4.1) and we can implement it as instance of the `allToAll` skeleton. The `map` phases of the skeleton are implemented as farms, where `map` input is distributed by `unshuffle` round robin among the processes. On each process, prior to the all-to-all communication step, `phase1` is applied to the input. It `maps` `f1` to a subset of the input. We want to transpose the overall matrix, so subsequently, we transpose

31 [Gor98], page 9

32 The order is the typing order, not the evaluation order: $\mathop{\circ}\limits_{i=1}^{2}(+i) = (+1) \circ (+2)$

Listing 4.2: The monolithic parallel map-and-transpose skeleton

```
parMapTransposeShuffle :: (Trans a, Trans b, Trans c) ⇒
                          Int               -- noPe
                          → (a → [b])       -- map function before transpose
                          → ([b] → c)       -- map function after transpose
                          → [a] → [c]       -- input / output
parMapTransposeShuffle np f1 f2 as
  = shuffle $ allToAll phase1 phase2 (unshuffle np as)
  where
    phase1 np =  unshuffle np ∘ transpose ∘ map f1
    phase2    = map (f2 ∘ shuffle) ∘ transpose
```



```
pre parallel  [[ 1, 2, 3, 4],                  [[[ 1, 2, 3, 4],
               [ 5, 6, 7, 8],                    [ 9,10,11,12],
               [ 9,10,11,12],                    [17,18,19,20],
               [13,14,15,16],   unshuffle 2      [25,26,27,28]],   spawn
               [17,18,19,20],                   [[ 5, 6, 7, 8],
               [21,22,23,24],                    [13,14,15,16],
               [25,26,27,28],                    [21,22,23,24],
               [29,30,31,32]]                    [29,30,31,32]]

phase 1       [[ 1, 2, 3, 4],             [[ 1, 9,17,25],              [[[ 1, 9,17,25],
PE1             [ 9,10,11,12],  transpose   [ 2,10,18,26],  unshuffle 2   [ 3,11,19,27]],
                [17,18,19,20],              [ 3,11,19,27],               [[ 2,10,18,26],
                [25,26,27,28]]              [ 4,12,20,28]]                [ 4,12,20,28]]]
                                                                                          all2All
PE2           [[ 5, 6, 7, 8],             [[ 5,13,21,29],              [[[ 5,13,21,29],
                [13,14,15,16],  transpose   [ 6,14,22,30],  unshuffle 2   [ 7,15,23,31]],
                [21,22,23,24],              [ 7,15,23,31],               [[ 6,14,22,30],
                [29,30,31,32]]              [ 8,16,24,32]]                [ 8,16,24,32]]]

phase 2       [[[ 1, 9,17,25],             [[[ 1, 9,17,25],
PE1              [ 3,11,19,27]],  transpose    [ 5,13,21,29]],  map shuffle  [[ 1, 5, 9,13,17,21,25,29],
               [[ 5,13,21,29],               [[ 3,11,19,27],                 [ 3, 7,11,15,19,23,27,31]]
                 [ 7,15,23,31]]]               [ 7,15,23,31]]]

PE2           [[[ 2,10,18,26],             [[[ 2,10,18,26],
                 [ 4,12,20,28]],  transpose    [ 6,14,22,30]],  map shuffle  [[ 2, 6,10,14,18,22,26,30],
               [[ 6,14,22,30],               [[ 4,12,20,28],                 [ 4, 8,12,16,20,24,28,32]]
                 [ 8,16,24,32]]]               [ 8,16,24,32]]]

post parallel [[[ 1, 5, 9,13,17,21,25,29],                  [[ 1, 5, 9,13,17,21,25,29],
                 [ 3, 7,11,15,19,23,27,31]],  unshuffle 2     [ 2, 6,10,14,18,22,26,30],
               [[ 2, 6,10,14,18,22,26,30],                    [ 3, 7,11,15,19,23,27,31],
                 [ 4, 8,12,16,20,24,28,32]]]                  [ 4, 8,12,16,20,24,28,32]]
```

Figure 4.8: Example: Transformation steps of the map-and-transpose skeleton

Listing 4.3: The monolithic 2-dimensional distributed homomorphism skeleton

```
dh2DMono :: Trans a ⇒ Int → ([a] → [a]) → [[a]] → [[a]]
dh2DMono np h = transpose ∘ parMapTransposeShuffle np h h
```

the matrices on each machine. This is exemplified in Figure 4.8 using a 4×8 matrix, but the `map` step is omitted (map function = `id]`) to focus on the transformation steps. Then we distribute the rows to the `np` processes using `unshuffle`. After receiving the sub-results from all partners, each process transposes the list of sub-results to group the i-th inner result lists of each process. Then, we apply `shuffle` to each group of sub-result lists (using `map`) to merge each to a single list and finally apply `f2` to each list (omitted in the example). We see in the example that after the main process receives the results and applies `unshuffle`, the matrix is the transposed original matrix. An implementation of the distributed homomorphism for the 2-dimensional case can be realised by a simple use of the `parMapTransposeShuffle` skeleton (see Listing 4.3).

4.2 Skeleton Composition using Remote Data

This section is a revised and extended version of [DHL10].

Here we present an alternative approach that allows the direct passing of distributed result data from one skeleton instance to the next one (see Figure 4.1 (b)). The main idea is to replace the data by handles to it, called remote data, which are gathered and redistributed instead. The handles can then be used to pull the real data directly to the target. This concept which has independently been suggested by Alt and Gorlatch [AG03, AG04, Alt07] can be easily used: normal data is replaced by the corresponding remote data handles and skeletons that operate on the new remote data can be composed as before. Only that now the gathering and redistribution of complex data is replaced by the gathering and exchange of small remote data handles which are used for the direct data exchange between processes within different skeleton instances. Thus, remote data handles for data which may be located elsewhere can be used like the original data but cause only low communication costs. They can occur everywhere where ordinary data may occur, e.g. in lists or trees to model distributed data structures. As we will show, this concept is flexible to use and still type-safe.

4.2.1 Composable Skeletons: Basic Building Blocks

Before handling the composition of skeletons using the remote data concept, we show the lifting of a simple parallel map skeleton to a remote data interface. Then we define a parallel all-to-all skeleton which generates a number of processes each of which exchanges data with any of the others. Using these skeletons with their remote data interfaces enables us to define a sequence consisting of a parallel map, a parallel transpose and a second parallel map. This can be useful in an implementation of a distributed homomorphism skeleton (see Chapter 4.1.4 and [LBDL09]) or a Google Map-Reduce skeleton (see Chapter 4.1.3 and

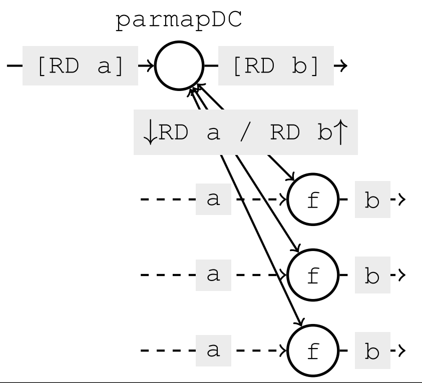
[BDL09]). In [BDL09, LBDL09], corresponding parallel map-transpose skeletons have been defined as monolithic skeletons without composing simpler skeletons. With the remote data interface, we can define the same skeleton as a composition of component skeletons. This leads to a much better understandable definition while achieving competitive performance, as we will show later. Finally, we present other elegant and concise definitions of even more complex communication pattern: a binary tree scheme which is used to define a binary reduction skeleton and a butterfly scheme which is used to define an all-reduce-skeleton.

4.2.1.1 The `parmapRD` Skeleton

A parallel map creates a process for each element of the input list. In Eden, it can easily be defined using the function `spawnF` (see Fig. 4.9a). Using a remote data interface, each process only gets a handle to its list element. It can then use this handle to fetch the element directly from the remote place where this element is located. In order to achieve this behaviour, we simply replace the parameter function `f` in the process abstraction by its lifted pendant `liftRD f` (see Fig. 4.9a). The function `liftRD` is used to lift functions acting on data to functions performing the same computation on remote data. This leads to the skeleton `parmapRD` where the ending `RD` stands for directly composable due to the **R**emote **D**ata interface[33]. This interface makes it possible for skeletons to receive distributed input and to produce distributed output which is crucial for an efficient composition of skeletons. Fig. 4.9b visualises the behaviour and communication paths of the `parmapRD` skeleton. The upper circle represents the process evaluating the `parmapRD` instantiation. It generates the other processes whose task is to apply the parameter function `f` to input of type `a` and produce output of type `b`. Note that only remote data handles for the input and the

```
parmap :: (Trans a, Trans b)
         ⇒ (a→b) → [a] → [b]
parmap f xs = spawnF fs xs
    where fs = repeat f

parmapRD :: (Trans a,Trans b)
             ⇒ (a→b) → [RD a] → [RD b]
parmapRD f xs = spawnF fs xs
    where fs = repeat (liftRD f)

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

(a) Code



(b) Visualisation

Figure 4.9: The `parmap` and `parmapRD` Skeletons

33 Alternatevely: `parMapRD f = parMap (liftRD f)`

output values are communicated between the generator process and its child processes. The proper data is communicated via dynamic channel connections indicated by dashed lines.

4.2.1.2 The `allToAllRD` Skeleton

In Figure 4.10 we present an all-to-all skeleton `allToAllRD`. This skeleton depends inherently on its inner communication pattern which we will implement using remote data. The input of the `allToAllRD` skeleton is a list of remote data with, say, n elements and

```
allToAllRDAt :: forall a b i. (Trans a, Trans b, Trans i)
                ⇒ (Int → a → [i]) -- ^transform before all-to-all
                → (a → [i] →b)    -- ^transform after all-to-all
                → [RD a] → [RD b]
allToAllRDAt t1 t2 xs = res where
  n = length xs           --same amount of procs as #xs
  (res,iss) = n `pseq` unzip $ parMap (uncurry p) inp
  inp       = zip xs $ lazy $ transpose iss

  p :: RD a→ [RD i]→ (RD b,[RD i])
  p xRD theirIs = (resF theirIs, myIsF x) where
    x      = fetch xRD
    myIsF  = releaseAll ∘ t1 n
    resF   = release ∘ t2 x ∘ fetchAll

--lazy list
lazy :: [a] → [a]
lazy ~(x:xs) = x : lazy xs
```

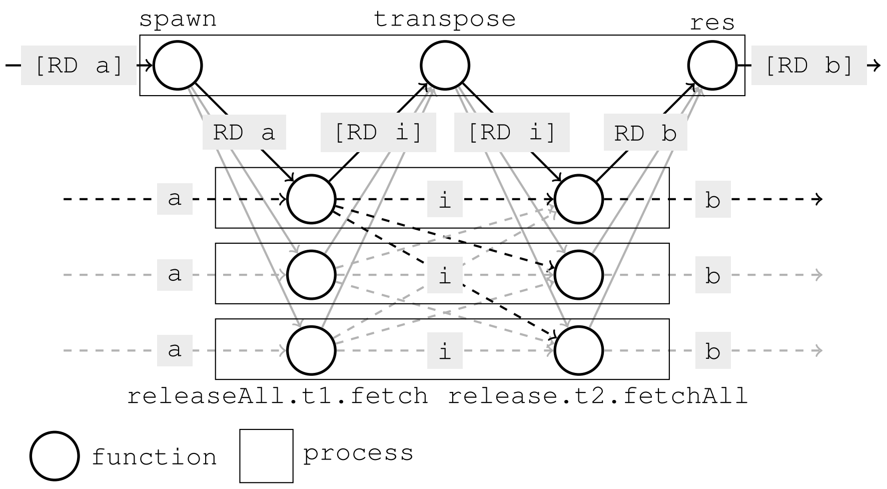


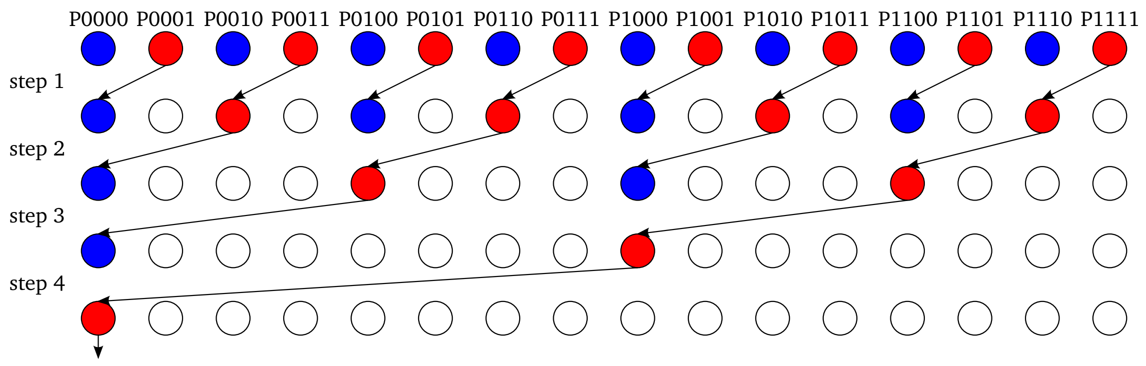
Figure 4.10: The `allToAllRD` skeleton: code and visualisation. (The darker shading of the arrows from the uppermost child process emphasizes the connectivity of a single process.)

two transformation functions `t1` and `t2` to allow the processes to transform the input data before sending data to all other processes and after receiving data from all other processes, respectively. The length of the input list determines the number of processes to be created by `parMap`. Every process will fetch its remote input `x` and transform it with the transformation function `t1`. This yields a list of intermediate data for each child process which is released element-wise by `releaseAll`, giving the list `myIsF x :: [RD i]` with remote data handles. Note that this list must have the same number n of elements as the input list. This list of remote data handles is returned to the root process in the second component of each process's result tuple. The root process receives one such list from each of its child processes resulting in the $n \times n$ matrix `iss :: [[RD i]]`. It transposes this matrix and sends the result back to the processes as its second, lazily supplied parameter `theirIs`. Each process gets thus one remote intermediate value of type `RD i` of each sibling process and of itself. The values are gathered using `fetchAll`, transformed together with the initial input `x` by the second parameter function `t2` to the output type `b` and released. The visualisation in Fig. 4.10 again shows the exchange of remote data handles between the root process the child processes and using dashed arrow the direct communication of data between the processes.

4.2.1.3 The `parRedRD` Skeleton

The binary tree or tournament scheme is a common pattern in parallel programming to calculate reductions or to gather data in $\lceil \log_2 p \rceil$ parallel steps for p processes. The communication pattern is simple to express using a binary-representation of the processes indexes. In step i, those processes communicate which differ only in bit i and which do not have any bit set at all positions smaller than i. The process with a bigger index sends its message to the process with the smaller index. An example using 16 processes is depicted in Figure 4.11a. We define a functional version of this pattern as Eden skeleton using remote data. We do not use any bit arithmetic, instead we use a transformational way to determine the communication partners. An observation of the example in Figure 4.11a reveals that in the first step, every odd process sends data *(depicted red)* and every even process receives data *(depicted blue)*. After sending data, processes get inactive *(white)*. In the subsequent steps again, every second active process sends data *(red)* and the other active processes receive data *(blue)*. In our implementation, we move remote data in a list from the position of the sending processes to the position of the receiving processes.

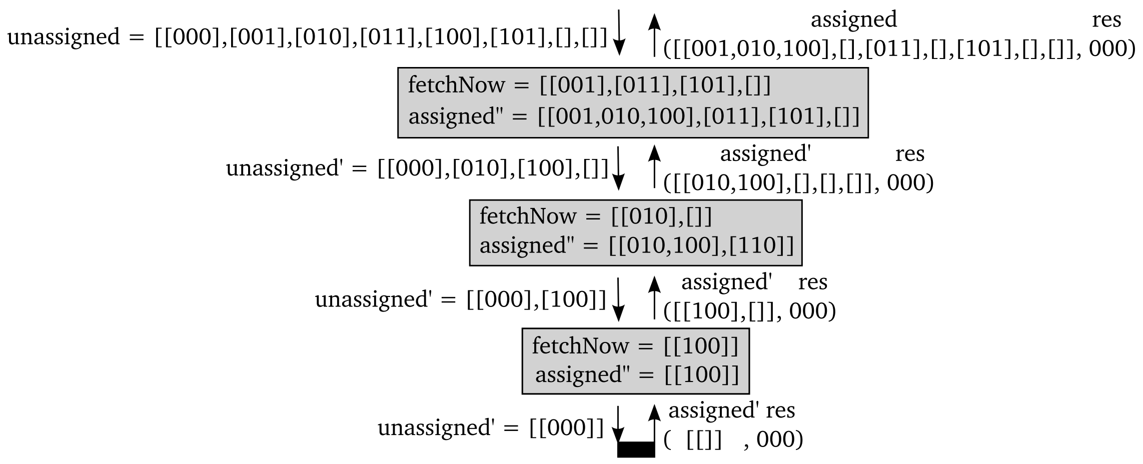
The topology is defined in function `partnering` of Figure 4.11b. The input list of type `a`, where `a` is typically a remote data handle, contains at position i the value of process i. The list length is expected to be a power of 2, the inner lists are expected to be singleton or empty lists (similar to `Maybe a`), empty lists are used to fill the top-level list to the next power of two, if the number of processes is no power of two. The result tuple contains the


P0000 P0001 P0010 P0011 P0100 P0101 P0110 P0111 P1000 P1001 P1010 P1011 P1100 P1101 P1110 P1111
step 1
step 2
step 3
step 4

(a) Tournament scheme with 16 processes

```
partnering :: [[a]] → ([[a]], a)
partnering [x] = case x of [res] → ([[]],res)
                           [] → error "first␣element␣must␣not␣be␣empty"
partnering unassigned = (assigned, res) where
  [unassigned', fetchNow] = unshuffle 2 unassigned
  (assigned', res) = partnering unassigned'
  assigned'' = zipWith (++) fetchNow assigned'
  assigned = take (length unassigned) $ shuffle [assigned'', repeat []]
```

(b) Moving values to positions of partners



(c) Example: `partnering` with 6 processes

Figure 4.11: Definition of the tournament scheme

original values which are moved according to the tournament scheme, where the first part of the result tuple is the list containing at position i (the remote data representation of) the list of values which will be fetched by process i. The second part of the output tuple is the final result (the output of process 0). In the function body, `unshuffle 2` splits the input list of `unassigned` values into even and odd elements. Even elements (`unassigned'`) correspond to remote data which is not fetched in this step (depicted blue). This data will be `assigned'` to processes in one of the subsequent steps, thus we call `partnering` recursively with the `unassigned'` values. The recursive call also returns the final result `res`. The `assigned'` values represent the tournament scheme for the subset of even inputs. The odd elements, which will be fetched in the current step, are in the list `fetchNow`. We concatenate this list element-wise with list of `assigned'` values and interleave the result with empty lists to generate the tournament scheme `assigned` for the whole input.

In Figure 4.11c, an example for the transformations done by the `partnering` function is presented. Instead of remote data values, we use process ids in binary encoding to clarify the data movement. The input list has 8 elements, 6 singleton lists containing process ids and 2 empty lists. The rectangles depict the different recursion steps and include local values of the different variables. In the `assigned` output we can read e.g. that process `000` receives the values from process `001`, `010` and `100` and process `001` receives no data at all.

Listing 4.4: The parRedRD skeleton

```haskell
parRedRDAt :: forall a. Trans a
             ⇒ Places
             → (a → a → a)       -- reduction function
             → [RD a]            -- input
             → RD a              -- ouput
parRedRDAt _ _ [] = error "need␣min␣1␣element␣to␣reduce"
parRedRDAt places rf xs = result where
  p :: RD a → [RD a] → RD a
  p rda rdas = release out where
    inp = fetch rda
    partners = fetchAll rdas
    out = partners `pseq` inp `pseq`
            foldl' rf' inp partners
    rf' xs ys = zs `using` rnf where
      zs = rf xs ys
  toReduce    = parMapAt places (uncurry p) (zip xs (lazy toReduce'''))
  toReduce'   = map (:[]) toReduce ++ repeat []       --fill with empty lists...
  toReduce''  = take (nextPow $ length xs) toReduce' --up to the next power of 2
  (toReduce''', result) = partnering toReduce'' --assign communication partners

--next power of 2
nextPow :: Int → Int
nextPow i = 2 ^ (ceiling $ logBase 2 $ fromIntegral i)
```

The `parRedRD`-skeleton of Listing 4.4 defines a parallel reduction using the tournament topology defined in Figure 4.11. It takes the reduction function `rf`, which applied to the current own value and a received value determines the next own value. The input list `xs zip`ped with a lazily supplied list `toReduce'''` forms the input to the processes `p`, which are instantiated using skeleton `parMap`. Each process fetches the initial input and reduces it and the list `partners` (fetched from the lazily supplied list `rdas`, which contains remote data handles) containing the reduction results from the partner processes with the reduce function `rf`. The result of this sequence of reduce steps `out` is released and returned to the caller. The rest of the process definition introduces demand control to assure that all sub-results are evaluated early without waiting to receive all results from the communication partners. In the caller process, the list `toReduce` containing the results from all processes is lifted to a list of lists and filled to the next power of 2 with empty lists. Then, list `toReduce''` can be processed by function `partnering` which defines the lazy input for the processes and the final output.

The demand control in the definition of `parRedRD` ensures that evaluation of the `fold rf` invocation progresses as far as possible even though the input from the reduce `partners` is only defined for the first elements – the evaluation of early applications of the reduce function does not stop when data for the evaluation of the overall `fold rf` result is missing. This works well for single input elements, but there is also a drawback when processing a stream of input and successively producing a stream of output. Sub-results of each reduce function invocation are not returned until the whole result of a reduce function invocation is evaluated to normal form, thus early streaming of sub-results is suppressed, the skeleton is not incremental. An example for this is depicted in Figure 4.12. The `main` function calls

```
redSkel o mapSkel $ [1..n],
```

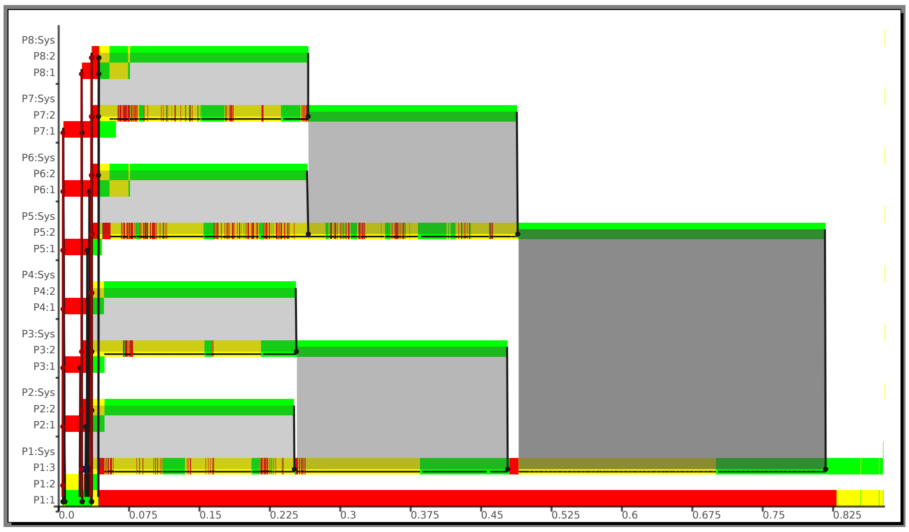
where `mapSkel` and `redSkel` are listed in Figure 4.12a. We use `parMapAt` in combination with `unshuffle` to distribute the input round robin among the machines and `release` it at the processes[34]. The `parRedRDAt` skeleton sorts the distributed, pre-sorted elements by merging them successively. The runtime trace for $p = 8$ processes and $n = 100000$ input elements (see Figure 4.12b) reveals clearly that communication is separated strictly into the $\log_2 p = 3$ reduce phases. This conflict between early demand on the sub-results and early streaming of overall results is hard to resolve, especially with the means of sequential demand control.

The `parRedStreamRD` skeleton is an incremental version of `parRedRD` (see Figure 4.13a). It uses sequences of alternating parallel `fetchPA` and `releasePA` actions in the definition of function `foldlRD` to introduce demand and concurrency. The optimised local communication ensures that no data is actually copied or sent because of the local remote data

34 We use unboxed vectors from `Data.Vector.Unboxed` for the input distribution to minimise communication overhead

```
mapSkel :: (Trans a, V.Unbox a) ⇒ [a] → [RD [a]]
mapSkel xs = parMapAt [1..noPe] (release ∘ V.toList) (map V.fromList $
    unshuffle noPe xs)

redSkel :: (Trans a, Ord a) ⇒ [RD [a]] → [a]
redSkel = fetch ∘ parRedRDAt [1..noPe] sortMerge
```

(a) Code `redSkel ∘ mapSkel $ [1..n]`



(b) Trace for `parRed sortMerge ∘ parMap release` with input size 100000 on 8 machines

Figure 4.12: Streaming Example: `parRed sortMerge`

use. Thus, several threads will eagerly evaluate each invocation of reduce function `rf`. We use the `PA` Monad to force the eager evaluation of the `fetchPA` and `releasePA` sequence. We use `foldlRD` instead of `foldl'` in process `p` and remove the demand control, expect `partners `pseq`...` which ensures the early evaluation of `fetchAll` on the remote input. We repeat the last experiment with `parRedStreamRDAt` skeleton instead of `parRedRDAt`, called by function `redSkelStream` (see Figure 4.13a). The `main` function calls

`redSkelStream ∘ mapSkel $ [1..n]`

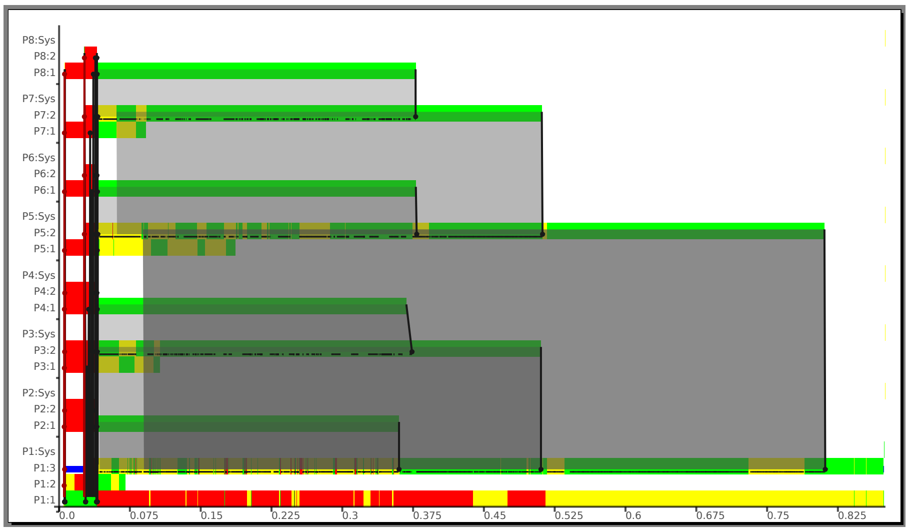
and we use the same parameters as before. A runtime trace is depicted in Figure 4.13c. The 3 reduce phases are clearly interleaved, but the runtime is close to the previous version. Even though we can not measure a better runtime for this example because of the bad ratio of input size to computation complexity, there are problems with higher computation complexity per input element which probably profit from the early streaming of results.

```
parRedStreamRDAt :: forall a. Trans a
                 ⇒ Places
                 → (a → a → a)       -- reduction function
                 → [RD a]            -- input
                 → RD a              -- ouput
parRedStreamRDAt _ _ [] = error "need␣min␣1␣element␣to␣reduce"
parRedStreamRDAt places rf xs = result where
  p :: RD a → [RD a] → RD a
  p rda rdas = runPA out where
    partners = fetchAll rdas
    out = partners `pseq` foldlRD rf rda partners

  toReduce   = parMapAt places (uncurry p) (zip xs (lazy toReduce'''))
  toReduce'  = map (:[]) toReduce ++ repeat []       --fill with empty lists...
  toReduce''= take (nextPow $ length xs) toReduce' --up to the next power of 2
  (toReduce''',result) = partnering toReduce'' --assign communication partners

foldlRD :: Trans a ⇒ (a → b → a) → RD a → [b] → PA (RD a)
foldlRD f x ys = go x ys where
  go x [] = return x
  go x (y:ys) = do x' ← fetchPA x
                   z  ← releasePA $ f x' y
                   go z ys
```

(a) Code `parRedStreamRDAt`

```
redSkelStream :: (Trans a, Ord a) ⇒ [RD [a]] → [a]
redSkelStream = fetch ∘ parRedStreamRDAt [1..noPe] sortMerge
```

(b) Code `redSkelStream` for `redSkelStream ∘ mapSkel $ [1..n]`



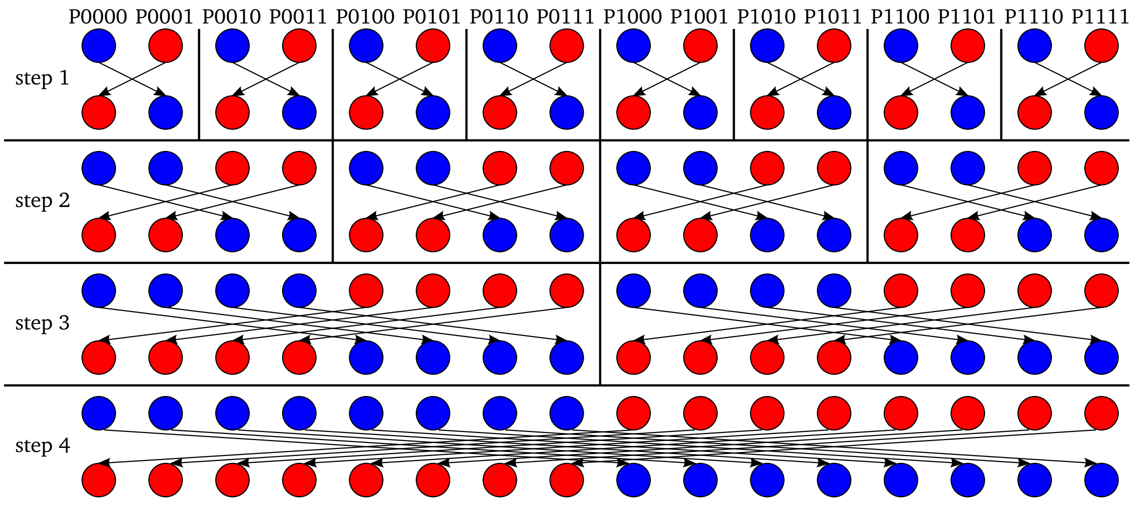
(c) Trace for `parRedStream sortMerge ∘ parMap release` with input size 100000 on 8 machines

Figure 4.13: Streaming Example: `parRedStream sortMerge`

4.2.1.4 The `allReduceRD` Skeleton

The all-reduce skeleton combines distributed data using a binary reduction function. It leaves the result duplicated on all processes involved in the reduction. Usually, it is implemented using the classical butterfly scheme (see [Qui94, p. 57]) which is also a common way to efficiently synchronise data between parallel processes. As for the `allToAllRD` skeleton, it is crucial for the all-reduce skeleton that data is transferred to and from the skeleton in a distributed way. The butterfly reduction for n processes is done in $\log n$ parallel communication and local reduction steps. In each step, the communication partner of process k is calculated with the Boolean function k `xor` 2^{step-1} (see Figure 4.14a). Figure 4.14b shows the definition of the function `partnering2` which applies the following transformation to determine the communication partner for the current `step`. The input list `xs` contains at position j the value of process j. `xs` is distributed round robin to `d=(2^step)` sublists. The values to be exchanged are in the same columns of the transformed matrix. Their indexes differ by 2^{step-1} which equals `d `div` 2` or half the number of inner lists. We flip the first half of inner lists with the second half and achieve the desired value exchange. A function call to `shuffle` re-establishes the original list structure.

Example: **input size:** 2^4, **step:** 2


P0000 P0001 P0010 P0011 P0100 P0101 P0110 P0111 P1000 P1001 P1010 P1011 P1100 P1101 P1110 P1111
step 1
step 2
step 3
step 4

(a) Butterfly scheme with 16 processes

```
partnering2 :: Int → [a] → [a]
partnering2 step xs = (shuffle ∘ flipAtHalfF ∘ unshuffle d) xs where
  d = (2 ^ step)
  flipAtHalfF xs = let (xs1, xs2) = splitAt (d `div` 2) xs
                   in xs2 ++ xs1
```

(b) Move values for the current step

Figure 4.14: Definition of the butterfly scheme

```
[0000,0001,0010,0011,0100,0101,0110,0111,
 1000,1001,1010,1011,1100,1101,1110,1111]
```

unshuffle 4 $\curvearrowright$

```
[[0000,0100,1000,1100],
 [0001,0101,1001,1101],
 [0010,0110,1010,1110],
 [0011,0111,1011,1111]]
```

flipAtHalfF $\curvearrowright$

```
[[0010,0110,1010,1110],
 [0011,0111,1011,1111],
 [0000,0100,1000,1100],
 [0001,0101,1001,1101]]
```

shuffle 4 $\curvearrowright$

```
[0010,0011,0000,0001,0110,0111,0100,0101,
1010,1011,1000,1001,1110,1111,1100,1101]
```

◁

The `allReduceRD` skeleton (see Listing 4.5) uses the function `partnering2` to rearrange lists of remote data in the caller process which represent the results of the intermediate reduction steps of the skeleton's processes. The rearranged lists are sent back to the processes. Thus, each process gets the remote values released by one partner in every step. Fetching these values establishes the butterfly communication topology.

Listing 4.5: The **`allReduceRD`** skeleton

```
allReduceRD :: forall a. Trans a ⇒
             (a → a → a) →             --reduce function
             [RD a] → [RD a]
allReduceRD redF toReduce = intermediates !! steps where
  steps = floor ∘ logBase 2 ∘ fromIntegral ∘ length $ toReduce
  toReduce' = take (2^steps) toReduce     --cut input to power of 2

 -- topology, inputs and instantiation
  intermediates = (transpose ∘ parMap (uncurry p)) inp   --steps in rows
  bufly = zipWith partnering2 [1..steps] intermediates  --only init rdBss
  inp   = zip toReduce' (lazy $ transpose bufly)        --steps in cols

 -- process functionality and abstraction
  p :: Trans a ⇒ RD a → [RD a] → [RD a]
  p rdA intermediates = (releaseAll ∘ scanl1 redF) toReduce where
    toReduce = fetch rdA : fetchAll intermediates'
    intermediates'= zipWith const (lazy intermediates) [0..steps]
```

The skeleton's input is a list with 2^{steps} remote data handles[35]. For each handle a process will be instantiated. The skeleton takes the parameter function `redF :: b → b → b` which should be associative and commutative and is applied in each step to the results of the previous step of a process and of its partner. This behaviour can concisely be expressed with `scanl1 redF` applied to the stream `toReduce` of values to be reduced. The stream `toReduce` is composed of the initial value and the stream input `intermediates`. The latter contains the partners' values for all steps. Note that the complete list structure of `intermediates` is already built in `intermediates'` even before its first element is received. Thus the request for all remote values can be eagerly initiated by the function `fetchAll` which would otherwise block on an incomplete list structure. The result of the `scanl1` application is element-wise released in every process, resulting in a list of remote data which is also generated in advance. This happens because the evaluation of `releaseAll` equally depends only on its parameter list's structure. Thus the exchange of remote data handles via the root process can happen in advance, independently of the parallel reduction steps.

The caller process gathers the result streams of all processes in a nested list. We transpose this list to have all remote values of a step in each inner list of `intermediates`. Applying the function `partnering2` to the first `steps` lists permutes these according to the butterfly scheme. We transpose this permutation `bufly` such that each process's input is located in one inner list. This transposed list is lazily zipped with the initially supplied input list `toReduce` using function `lazy` and passed back to the processes. The final result consists of the results of the last reduction step, i.e. the last element of the list `intermediates`.

We have tested the `allReduceRD` skeleton with a dummy example which we executed on an 8 core Intel Xeon machine. The initial transformation function `initF` serves as generator and generates the list `[1..nElems]`, where `nElems` is a parameter of the program and in our example set to 200000. The trace visualisation in Fig. 4.15 reveals interchanging computation and communication phases. The butterfly interconnection scheme can clearly be recognised in the messages exchanged between the processes. The generation of elements is depicted as the first "running" phase. The reduction network has been set up before, by exchanging the remote data messages via the root process on Machine 1 (initial messages). Three reduction phases follow. First the direct neighbours exchange their lists leading to the typical butterfly pattern of messages. The processes reduce their lists using the reduction function `redF` which is set to `zipWith (+)`. For the next steps, the distance to the partner process is doubled every time. Finally, a `parmapRD` skeleton is called to consume the data and return an empty list to the root process.

35 The `allReduceRD` skeleton only works for input lists where the length is a power of two. Other lists are cut to the next smaller power of two. Further, this version works only for commutative operations. A generalised version which works for all input sizes and non-commutative operations can be found in the Appendix A.2

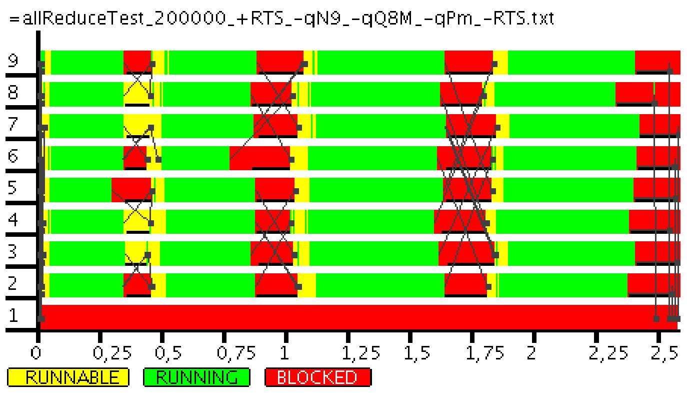


Figure 4.15: Runtime behaviour of the `allReduceRD` skeleton

4.3 Composed Skeletons

In this section we examine some examples of concrete applications and skeletons which are built from sequences of smaller skeletons and we compare example applications using monolithic and compositional implementations of those skeleton sequences. We argue that compositional implementations are more comprehensible, more flexible and competitive in performance. In the following section we create a composed version of the distributed workpool and the `parRed` skeleton. In Chapter 4.3.2, we present some versions of compositional parallel `map - transpose - map` implementations, a basic pattern which we will use in many of the subsequent examples.

Listing 4.6: Adjusted distributed workpool skeleton

```
distribWPAt :: forall t i r s r'. (Trans t,Trans i,Trans r,Trans s,NFData r',
    Show r', Show s)
  ⇒ Places                          -- ^ where to put workers
  → ((t,s) → (Maybe (r',s),[t]))    -- ^ worker function
  → (i → [t])                       -- ^ initial taskpool transform
  → ([Maybe (r',s)] → s → r)        -- ^ result transform
  → ([t]→[t]→s→[t])                 -- ^ taskpool transform attach function
  → ([t]→s→([t],Maybe (t,s)))       -- ^ taskpool transform detach f. (local)
  → ([t]→s→([t],[t]))               -- ^ taskpool transform split f. (remote)
  → (s→s→Bool)                      -- ^ state comparison (take new state?)
  → s                               -- ^ initial state (offline input)
  → [i]                             -- ^ input (numElems → numWorkers)
  → [r]                             -- ^ results of workers
distribWPAt pl wf initF resT ttA ttD ttSplit sUpdate st is
  = ringFlAt pl id id workers is where ...
```

4.3.1 Distributed Workpool - Reduce

We want to reconsider the distributed workpool example of Chapter 4.1.1. We use remote data to realise a distributed reduction of the workpools results without statically including the required tree topology functionality in the workpool skeleton. Before we define the skeleton composition to accomplish this, we adjust the skeleton signature slightly for better composability (see Listing 4.6). We now use explicit placement to allow for co-located sibling processes among composed skeletons. We drop the *number of processes* parameter, instead we produce one process for each input element. After receiving the inputs in the processes, we use the added *initial transform* function to make the input compatible with the type we want to manage in the taskpool. This gives us higher flexibility, particularly let a processes `input` be of type `RD [t]` (`RD [t] ~ i`). Thus the `input` may be fetched using the initial taskpool transform function. We can use the already included result transform function to release remote data after processing the data.

The implementation is based on the ring-skeleton `ringFlAt` (see Listing 2.3), which takes a list of ring-worker functions (cycled) and a list of inputs, one function for each process respectively. The processes take further a *ring input* of type `i` from their left neighbour. If the type of the *ring input* is a list (`i ~ [i']`), then the ring processes are connected by a stream of data, that corresponds in the distributed workpool application to the request stream. See [Die07] for an implementation of the workers.

One use case for a distributed reduction is to restore the order of the worker results, which were produced in non-deterministic order. Here the result transform function can be used to pre-sort the results and we compose the workpool skeleton with the reducer skeleton of Chapter 4.2.1.3 to add a distributed merge phase of the presorted results.

Here a compositional version of the distributed workpool - reduce:

```
distribWPRedCompAt pl wf initF resT redF ttA ttD ttSplit sUpdate st
  = fetch
    ∘ parRedRDAt pl redF
    ∘ distribWPAt pl wf initF resT' ttA ttD ttSplit sUpdate st where
      resT' res s = release $ resT res s
```

The definition is straight forward. The only difficulty is that we need to `release` the result of the distributed workpool processes in the `resT'` function and to fetch the result of the `parRedRDAt` skeleton. This is because the distributed workpool has no remote data in its signature and the `parRedRDAt` has a remote data signature.

We need to redefine the `distribWPAt` skeleton to generate a monolithic version with equivalent computation pattern. In the implementation of the skeleton given in Listing 4.6, we use the `ringFlAt` skeleton to define the basic topology. Thus, we can include the functionality of the `parRed` skeleton by including it in the `ringFlAt` skeleton (see Listing 4.7).

Listing 4.7: The `ringReduceFl` skeleton

```
ringReduceFlAt :: forall a b r i. (Trans a,Trans b,Trans r) ⇒
        Places                      -- ^where to put workers
        → (i → [a])                 -- ^distribute input
        → (b → b → b)               -- ^reduce function
        → [(a → r → (b,r))]         -- ^ring process fcts
        → i                         -- ^ring input
        → b                         -- ^ring output
ringReduceFlAt places distrib rf fs i = fetch result where
  (toReduce, ringOuts)
    = unzip $ spawnFAt places (map uncurry3 (map p $ cycle fs))
                              (zip3 xs (lazy ringIns) (lazy toReduce'''))
-----------------------ring code-------------------------
  xs = distrib i
  ringIns        = rightRotate ringOuts
  p :: (a → r → (b,r)) → a → RD r → [RD b] → (RD b, RD r)
  p f a rIn rdbs = (release bOut, rOut) where
    (ringRes, rOut) = toRD (uncurry f) $ (a,rIn)
----------------------parRed code-------------------------
    partners = fetchAll rdbs
    bOut = partners `pseq` (ringRes `using` rnf) `pseq`
            foldl' rf' ringRes partners
    rf' xs ys = zs `using` rnf where
      zs = rf xs ys
  toReduce'  = map (:[]) toReduce ++ repeat []      --fill with empty lists...
  toReduce'' = take (nextPow $ length xs) toReduce' --up to the next power of 2
  (toReduce''', result) = partnering toReduce'' --assign communication partners
```

We separate the code roughly in three parts. A part originating from the ring skeleton (cf. Listing 2.3), a part originating from the `parRedRD` skeleton (cf. Listing 4.4) and the call of `spawnFAt` to instantiate the processes and provide them with the inputs for the other two parts. The `ring` part is basically untouched while we use the ring result `ringRes` in the `parRed` part directly after forcing its evaluation, instead of fetching the remote data input. The integrated `distribWPRedAt` skeleton (see Listing 4.8) is then defined by the `ringReduceFlAt` skeleton, providing an binary reduce function `redf` as additional input.

4.3.2 Map - Transpose - Map

"The `allToAllRD` skeleton can be used to express arbitrary data exchange that requires an all-to-all network. A common special case is the transposition of a matrix which is distributed over several processes. The way the matrix

Listing 4.8: The `distribWPRedAt` skeleton

```
distribWPRedAt pl wf initF resT redF ttA ttD ttSplit sUpdate st is
  = ringReduceFlAt pl id redF workers is where ...
```

```
mtmRD :: (Trans a, Trans b, Trans c)
         ⇒ (a→[[b]]) → ([[b]]→c) → [RD a] → [RD c]
mtmRD f g = parmapRD g ∘ parTransposeRD ∘ parmapRD f

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

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

Figure 4.16: Composition of parmap and transpose skeletons

is distributed over the processes can be manifold. Each process might be assigned e.g. to one row or — more general — to several rows of the matrix. In the example skeleton `parTransposeRD` of Figure 4.16, we implement the more general case. Thus, we are not restricted to 1:1 relations between rows and processes. We assume that rows are distributed round robin over the processes. The advantage against a block distribution is that the matrix can be assigned partially to the processes without knowledge of the overall number of rows. Hence, the transposition skeleton has to assign the columns of the overall matrix (rows of the transposed matrix) round robin to the processes. The first transformation function of type `Int → [[b]] → [[[b]]]` first transposes a list of rows to get the list of the former columns. In a second step, these are round robin distributed to sublists, one for each process. Process i will consequently receive one row-sliced and column-sliced partial matrix from each process. The second transformation of type `[[[b]]] → [[b]]` will shuffle the row-slices (transposed column-slices) into each other to recover the rows of the overall transposed matrix. This is done by flipping the outer dimension (the list of partial matrices) with the row-dimension using `transpose`. Thus every outer list element contains all partial rows belonging to the same row of the overall matrix. The transformation `map shuffle` re-establishes each row.

Now, we can combine the `parmapRD` skeleton of Fig. 4.9a and the parallel transpose skeleton `parTransposeRD` in the function `mtmRD` (cf. Figure 4.16), a parallel version of the function composition `map g ∘ transpose ∘ map f`. Without remote data a naive parallel implementation would be

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

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

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

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

Each skeleton was instantiated with an input matrix of size 800×800 and evaluated on 8 Intel Core 2 Duo machines with a Fast Ethernet connection, where each processor core hosted two virtual machines of the Eden runtime system. In Fig. 4.17, we present the activity profiles of the corresponding runtime traces for the two skeletons. [...]

The upper left trace in Fig. 4.17 clearly reveals the distributed transposition by the multitude of messages exchanged right after the initial data generation phase and the first `map`-phase, which is depicted "running" in the trace. The exchange of remote data starts very early overlapping the `map`-phase and forming dense bundles of messages. The second `map`-phase at the end of the program execution is rather short. Note that the overall runtime was less than 0.5 seconds.

We have placed the ith process of every skeleton on the same machine, such that communication costs are low[36]. The lower zoomed view of the figure shows the activity bars of the three processes located on the virtual machine 16. The lowest bar belongs to a child of the first `parmapRD`-instantiation. The upper two bars show the processes of the parallel transpose skeleton and the second `parmap` instantiation. With this information, we can easily identify the different types of messages. During phase 1 the process of the first `parmapRD` skeleton sends its results to the `parTransposeRD` process. In the second phase the intermediate data is exchanged with the processes on the other machines. Finally, in phase 3, the result of the transposition is passed on to the second `parmapRD` process.

The upper right trace in Fig. 4.17 belongs to the naive version which performs a local transposition in the root process. As expected, this version is much slower with an overall runtime of approximately 3 seconds. The conspicuously fast communication between machine 1 and machine 10 is because the two

36 This experiment has been parformed before local communication has been optimised

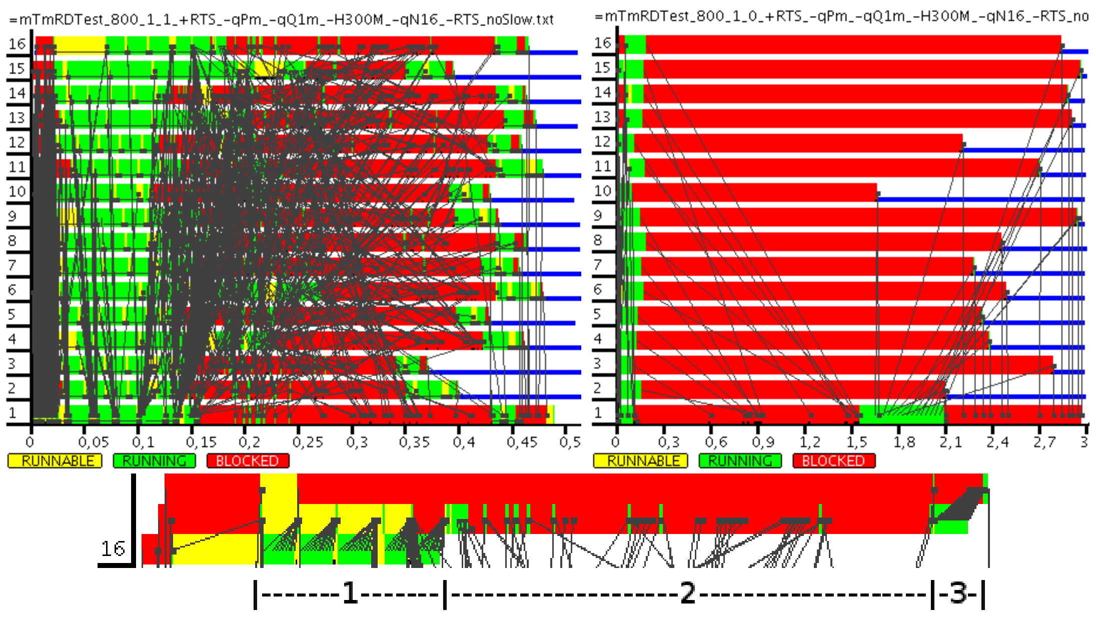


Figure 4.17: Runtime behaviour of the skeleton `mtmRD` in the global view(left), the zoomed process on Machine 16 (bottom) vs. the local transposition version (right). (Note the different scaling of the x-axes in the upper traces and that the zoomed view has been taken from a processes-per-machine view, here showing the activity bars of the three processes on Machine 16.)

> virtual machines share the same physical machine. Further tests with varying input sizes (not shown) confirmed the enormous runtime advantages of the distributed version."[37]

Instead of using `allToAllRD`, we can define the parallel version of composition `map f2 ∘ transpose ∘ map f1` directly using remote data:

```
parMap (f2 ∘ fetchAll) ∘ transpose ∘ parMap (releaseAll ∘ f1)
```

The additional overhead should be minor and the composed code is more comprehensive. We use this pattern to implement a compositional version of the Google map-reduce skeleton, introduced in Chapter 4.1.3.

The functionality of skeleton `mtmRD` of Figure 4.16 is *similar* to the monolithic `parMapTransposeShuffle` skeleton of Listing 4.2. In Figure 4.18, we give an *equivalent* compositional definition `parMapTransposeShuffleComp` without using the `allToAllRD` skeleton. We will compare both versions for the distributed homomorphism FFT implementations in Chapter 4.3.4.

37 [DHL10, pages 81-83]

```
parMapTransposeShuffleComp :: (Trans a, Trans b, Trans c)
      ⇒ Int           -- noPe
      → (a → [b])     -- map function before transpose
      → ([b] → c)     -- map function after transpose
      → [a] → [c]     -- input / output
parMapTransposeShuffleComp np f1 f2 as
 = shuffle
   ∘ parMap (map (f2 ∘ shuffle) ∘ transpose ∘ fetchAll)
   ∘ transpose
   ∘ parMap (releaseAll ∘ unshuffle np ∘ transpose ∘ map f1)
   ∘ unshuffle np $ as
```

Figure 4.18: A compositional implementation of `parMapTransposeShuffle`

4.3.3 The Google map-reduce Skeleton

We present in Figure 4.19 a revised version of our Google map-reduce skeleton which is implemented by means of:

```
parMap(reducer ∘ fetchAll) ∘ transpose ∘ parMap(releaseAll ∘ mapper)
```

The local definitions are equivalent to the monolithic skeleton `mapReduceList` of Chapter 4.1.3 and are thus omitted.

4.3.4 Distributed Homomorphism

In Chapter 4.1.4 we discussed the distributed homomorphism skeletnon and presented a monolithic implementation in Eden for the two dimensional case. Here, we want to adapt

```
mapReduceListComp :: forall k1 k2 v1 v2 v3 v4.
 (Trans k1, Trans k2,
  Trans v1, Trans v2, Trans v3, Trans v4,
  Ord k2)
  ⇒ Int                        -- Number of partitions
  → (k2 → Int)                 -- Partitioning for keys
  → (k1 → v1 → [(k2,v2)])      -- The *'λmap'* function
  → (k2 → [v2] → Maybe v3)     -- The *'λcombiner'* function
  → (k2 → [v3] → Maybe v4)     -- The *'λreduce'* function
  → [[(k1,v1)]]                -- Distributed input data (list)
  → [[(k2,v4)]]                -- Distributed output data
mapReduceListComp parts keycode mAP cOMBINER rEDUCE input = ress where
  ress = parMap (reducer ∘ fetchAll)
        ∘ transpose
        ∘ parMap (releaseAll ∘ mapper)
           $ unshuffle parts input
...
```

Figure 4.19: Implementing GMR as instance of the allToAll skeleton

Listing 4.9: A simple compositional version of the 2-dimensional distributed homomorphism implementation of FFT.

```
dh2DSimpleComp :: Trans a ⇒ ([a] → [a]) → [[a]] → [[a]]
dh2DSimpleComp h = transpose ∘ parMap (h ∘ fetchAll)
                   ∘ transpose ∘ parMap (releaseAll ∘ h)
```

Listing 4.10: The compositional 2-dimensional distributed homomorphism skeleton.

```
dh2DComp :: Trans a ⇒ Int → ([a] → [a]) → [[a]] → [[a]]
dh2DComp np h = transpose ∘ parMapTransposeShuffleComp np h h
```

the two dimensional implementation to a compositional version, compare the behaviour and performance of both implementations.

We use again the *parMap - transpose* pattern to define a first simple compositional version of the 2-dimensional distributed homomorphism implementation presented in Listing 4.3. In the implementation of Listing 4.9, the transposition combined with usage of `fetchAll` and `releaseAll` in the processes establishes an all-to-all topology to transpose the distributed matrix. This straight forward implementation uses 2^k processes for each `parMap` instance, where 2^k is the length of the top level input list. In an optimised version `dh2DComp` (see Listing 4.10), we want to determine the number of processes independently of the input size. We use the compositional `parMapTransposeShuffleComp` skeleton (see Listing 4.13) for row-wise round robin distributed matrices. The usage of skeletons `parMapTransposeShuffle` and `parMapTransposeShuffleComp` is the same, the compositional skeleton is more intuitive at the expense of additional processes.

4.4 Case Studies

4.4.1 Mandelbrot with Distributed Workpool - Reduce

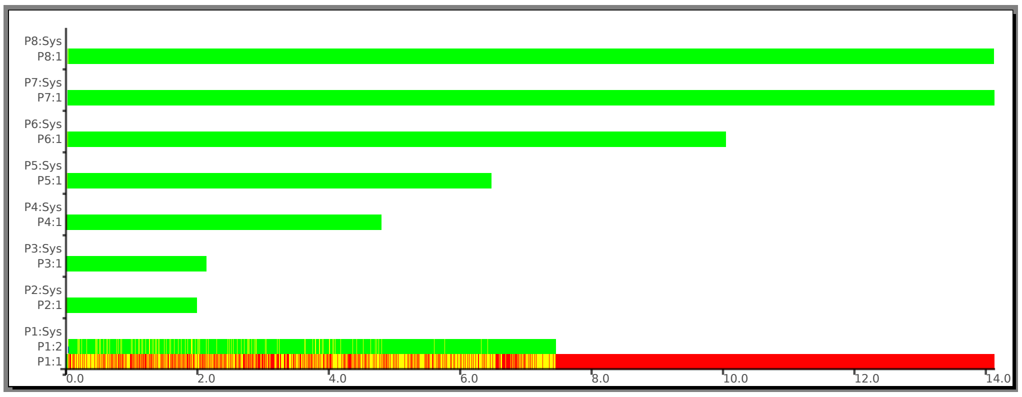
The Mandelbrot set [Man80] is a subset of the complex plain, it contains all numbers c for which the sequence given by $z_0 = 0$ and $z_{n+1} = z_n^2 + c$ remains bounded. The subset can be approximated for a grid `C` representing a finite subset of the complex plain with resolution $n \times m$, with any (parallel) map function. The pixels on the grid which do not belong to the Mandelbrot set are usually coloured, where the colour depends on the number of recursion steps needed to exceed a bound. We use the Mandelbrot implementation from [Ber08], which originally returned `(String,Int,Int)` to represent the result grid. The graph packing needed to send data introduced a lot of overhead because of the `String` representation of the rows of the result grid. We use unboxed vectors of `Chars` instead

(`Vector Char`), which drastically reduces packing overhead when sending the results. We calculate the Mandelbrot set for sub-intervals of a grid `c` in parallel. A static block distribution of `c` into sub-intervals, we used the `farm` skeleton with `splitIntoN`, leads to an uneven load among the processes (see the runtime trace visualisation of Figure 4.20a), as the different areas of the grid have varying computation costs. The uneven system load can be optimised using a different static distribution, e.g. round robin distribution, or using dynamic (re-)distribution provided e.g. by the distributed workpool skeleton. A runtime trace visualisation for distributed workpool skeleton with initial block distribution is depicted in Figure 4.20b. The messages in between the working processes reveal work redistribution to balance the load when processes run out of tasks. The results of the distributed workpool skeleton are not in the right order, hence they have to be sorted by the initial distribution order. We use the result transformation function of the distributed workpool to pre-sort the results prior to releasing them. Now we can sort the workers results by successively merging them in the caller process. The trace picture reveals that this result sorting has only minimal impact on the runtime (it starts not before messages arrive on P1:1). However, we will use the Mandelbrot program with this result merging as an example to compare a version of the workpool skeleton with build in parallel reduction vs. the composition of the `parRedRD` (cf. Listing 4.4) and the distributed workpool skeleton (cf. Listing 4.6) implementing the result merging phase in parallel. In the first case, we can simply compose `parRedRDAt o distribWPAt` with appropriate parameters. In the second case we need the specialised monolithic version `distribWPRedAt` (cf. Listing 4.8).

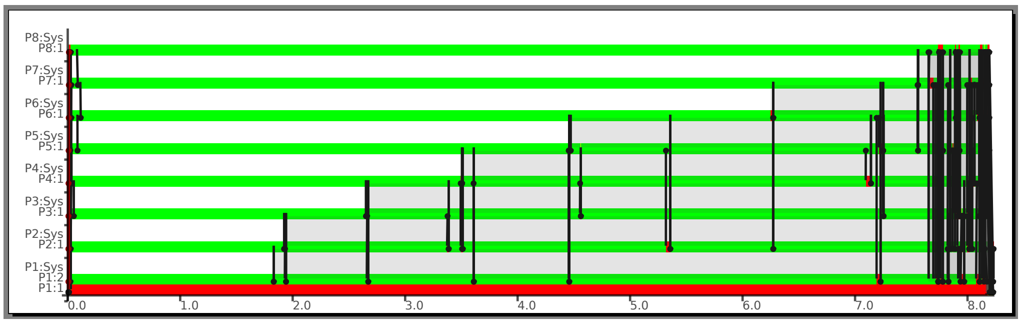
We compare the behaviour of both versions by trace visualisations of Mandelbrot computations with grid resolution 5000×5000 on 8 machines[38] (see Figure 4.21). Apart from the additional processes for the compositional version (see Figure 4.21a), the runtime behaviour of both versions is close to identical. In the composed version (zoomed view, Figure 4.21c) we see that passing the data from the workpool to the `parRedRD` skeleton, right after the dense communication phase (representing termination detection in the distributed workpool) and before the second processes of the even machines start working, happens in an instant. A close observation reveals small blue arrows representing the local messages from P1:2 to P1:3 or PX:1 to PX:2 for the other machines.

A systematic comparison of runtimes and speedups measured on a 64-Core machine[39] with input sizes 5000×5000 and 500×500 is presented in Figure 4.22. Figure 4.22a and Figure 4.22b show runtimes and speedups for the bigger input size, Figure 4.22d and Figure 4.22e show runtimes and speedups for the smaller input size. Runtimes are mean values of 5 program runs. Speedups increase for the bigger input size until 64 PEs, but are far from optimal. Speedups decrease after 32 PEs for the smaller input size due

38 on hex (see Chapter 2.2.1)
39 on hex (see Chapter 2.2.1)



(a) `farm`



(b) `distribWP`

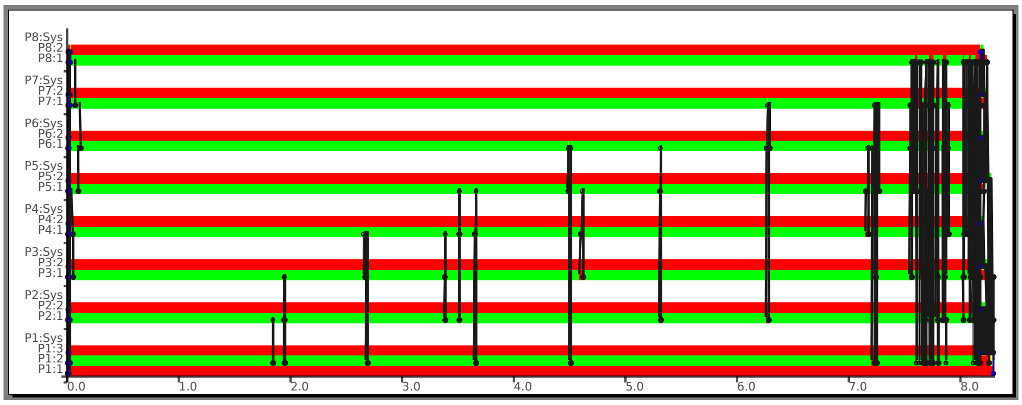
Figure 4.20: Mandelbrot traces on 8 Machines with grid resolution 5000 x 5000 and block distribution

to increasing parallel overhead. Runtimes of compositional and monolithic versions are always close to identical for both input sizes (ranging from −2.6% to +3.0%).

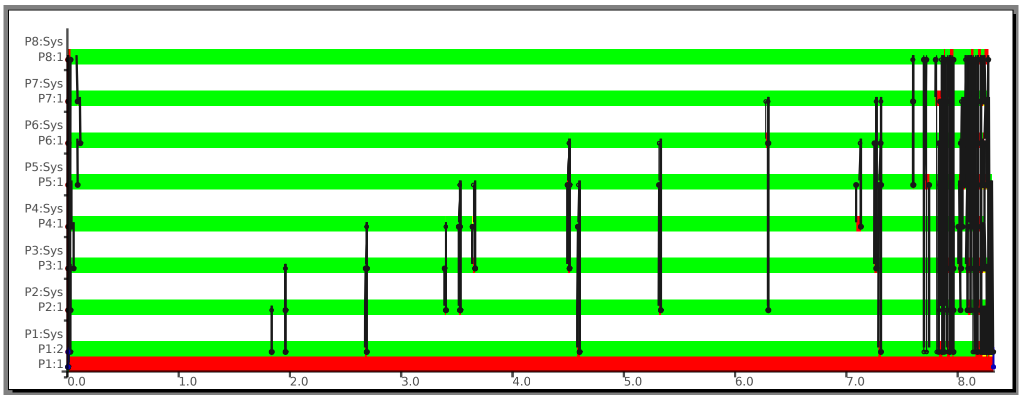
4.4.2 NAS EP with Google map-reduce

The NAS **E**mbarisingly **P**arallel benchmark is part of the NAS parallel benchmarks compilation [NAS94]. The benchmark will “generate a set of Gaussian random deviates [...] and tabulate the number of pairs in successive square annuli”[40]. The generation of the Gaussian random deviates is done by the mappers. The keys are 10 different square annuli and we use one reducer per key. The trace visualisations depicted in Figure 4.23 show the runtime behaviour on a 64 core machine[41] for problem size 10.000.000 and chunking size 100.000, we used the compositional Google map-reduce skeleton to implement the program for Figure 4.23a and the monolithic Google map-reduce skeleton for Figure 4.23b. Map and reduce phases overlap, as we can see clearly in the communication pattern. The

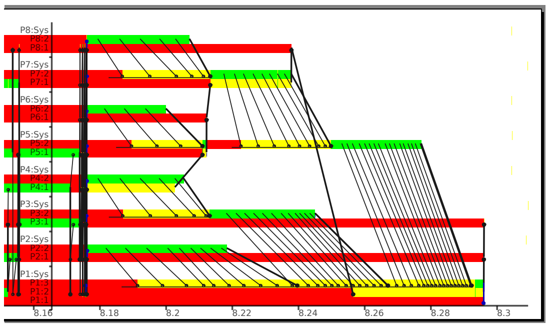
40 [NAS94], page 14
41 on hex (see Chapter 2.2.1)



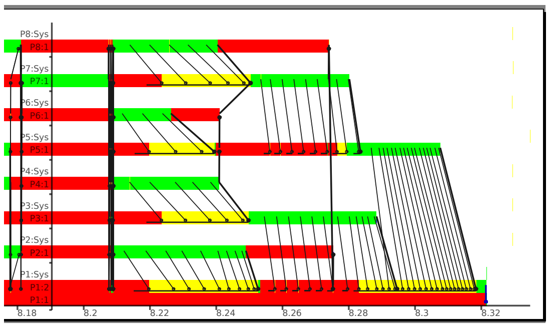
(a) `parRed ∘ distribWP`



(b) `distribWPRed`

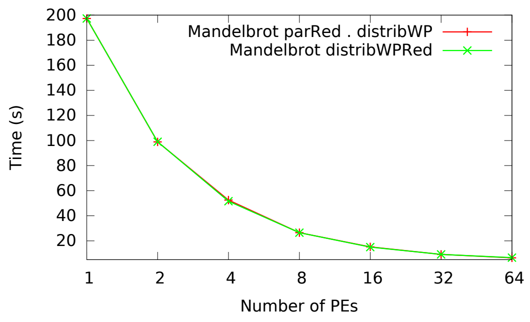


(c) `parRed ∘ distribWP` zoom

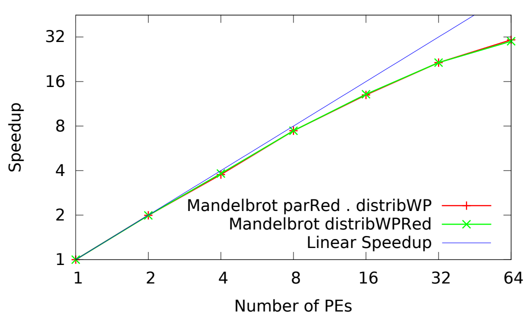


(d) `distribWPRed` zoom

Figure 4.21: Mandelbrot with parallel merging of results on 8 Machines with grid resolution 5000 x 5000 and block distribution



(a) Runtimes for grid size 5000 × 5000

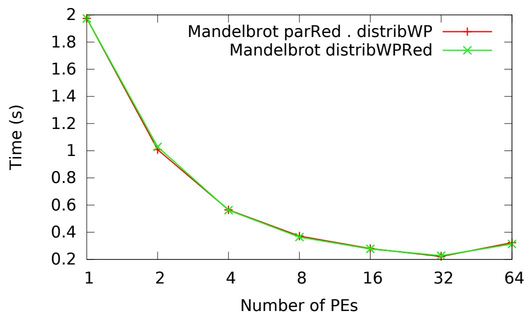


(b) Speedups for grid size 5000 × 5000

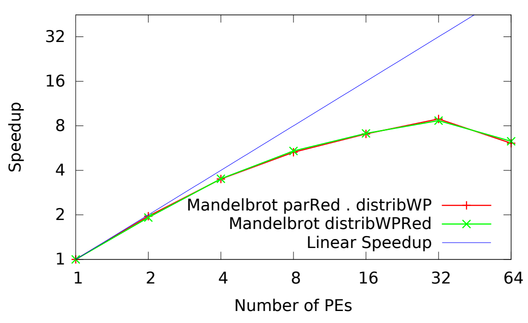
sequential implementation of Mandelbrot: 197.29 sec.

PEs	`distribWPRed`	`parRed . distribWP`	overhead
2	99.01 s	98.90 s	-0.12%
4	51.63 s	52.50 s	1.67%
8	26.54 s	26.50 s	-0.15%
16	15.03 s	15.18 s	1.02%
32	9.19 s	9.18 s	-0.08%
64	6.59 s	6.44 s	-2.21%

(c) Runtime table for grid size 5000 × 5000



(d) Runtimes for grid size 500 × 500



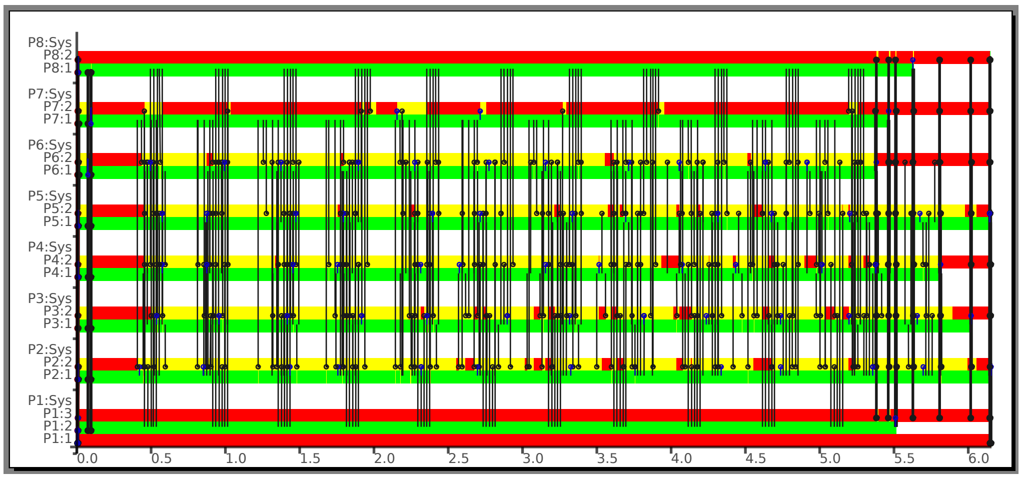
(e) Speedups for grid size 500 × 500

sequential implementation of Mandelbrot: 1.974 sec.

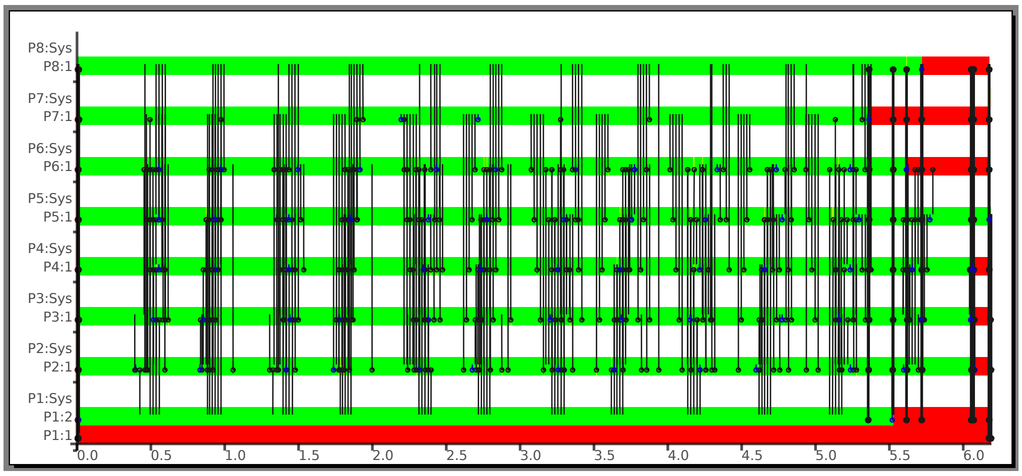
PEs	`distribWPRed`	`parRed . distribWP`	overhead
2	1.03 s	1.01 s	-1.99%
4	0.56 s	0.56 s	0.39%
8	0.36 s	0.37 s	1.93%
16	0.28 s	0.28 s	0.85%
32	0.23 s	0.22 s	-2.56%
64	0.31 s	0.32 s	2.98%

(f) Runtime table for grid size 500 × 500

Figure 4.22: Runtimes and speedups of Mandelbrot program with parallel merging of results and block distribution



(a) compositional



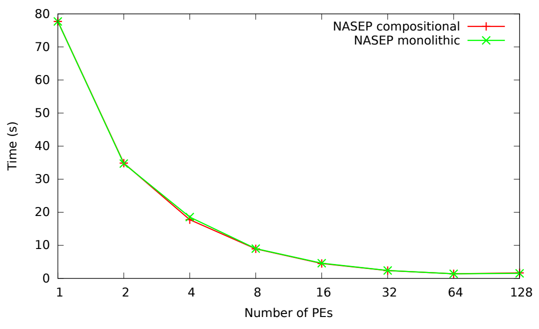
(b) monolithic

Figure 4.23: Traces of the NAS EP benchmark with Google map-reduce on 8 Machines with problem size 10.000.000 and chunk size 100.000 on hex.

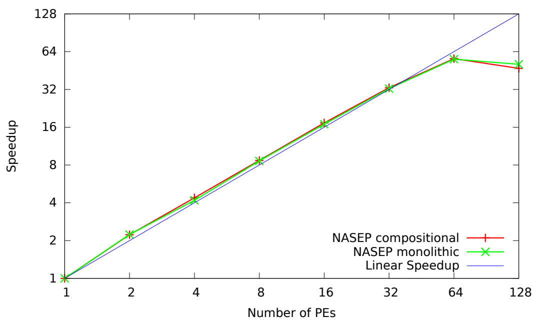
trace of Figure 4.23a reveals that most computation is done by the mappers, while the reducers are running rarely. Apart from the additional reducer processes, the program behaviour of both versions is similar.

Figure 4.24 presents mean runtimes and speedups based on 5 program runs of the NAS EP benchmark for problem size 10.000.000 executed on the Beowulf cluster (see Chapter 2.2.1). Measures scale well up to 64 PEs but runtimes increase for 128 PEs. Compositional and monolithic versions perform equally well (from 4 to 64 PEs the compositional version is up to 4.5% faster), with minimal advantages for the monolithic version for 128 PEs (the difference is 7.21%, with small runtimes around 1.6s).

In a second setup, we measured runtimes for problem size 200.000.000 (see Figure 4.25a).



(a) Runtimes



(b) Speedups

runtime of sequential implementation 77.71 sec.

PEs	monolithic	compositional	overhead
2	34.73 s	34.87 s	0.40%
4	18.55 s	17.74 s	-4.32%
8	9.01 s	8.92 s	-0.97%
16	4.60 s	4.49 s	-2.30%
32	2.39 s	2.35 s	-1.72%
64	1.39 s	1.38 s	-0.93%
128	1.53 s	1.65 s	7.21%

(c) Runtime table

Figure 4.24: Runtimes and speedups of the NAS-EP benchmark with Google map-reduce for problem size 10.000.000 and chunking size 10.000 on Beowulf

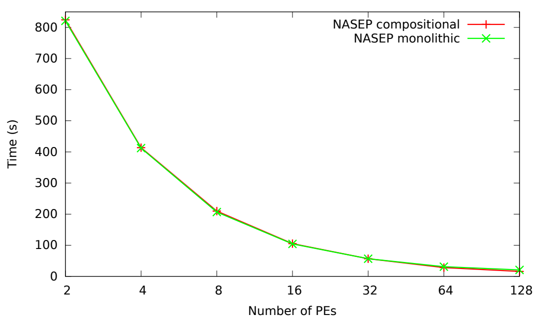
A single machine on the Beowulf cluster ran out of memory, thus Figure 4.25b presents relative speedups based on the doubled runtime for 2 PEs. The benchmark scales well up to 128 PEs. Both versions perform similar, surprisingly with better runtimes for the compositional version with 64 PEs (-9.29%) and 128 PEs (-23.14%). Differences for 2 to 32 PEs are $< 1.6\%$.

4.4.3 FFT with Distributed Homomorphism

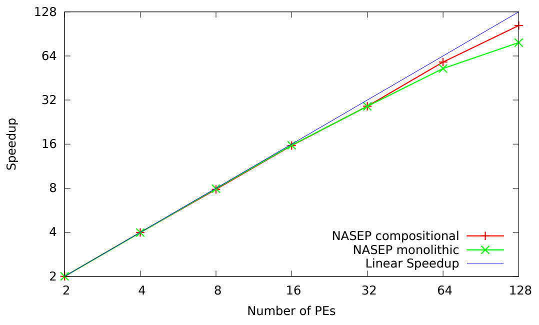
An application which can be implemented using a distributed homomorphism skeleton is the fast Fourier transform (FFT). FFT is a fast way to calculate the discrete Fourier transform using the divide and conquer principle. Function

```
fft :: RealFloat a ⇒ [Complex a] → [Complex a]
```

takes a list of complex numbers and calculates a list of the same length, where the ith



(a) Runtimes



(b) Relative speedups

sequential implementation ran out of memory.

PEs	monolithic	compositional	overhead
2	820.92 s	824.49 s	0.43%
4	412.31 s	413.71 s	0.34%
8	206.72 s	209.95 s	1.54%
16	104.43 s	105.40 s	0.92%
32	56.63 s	56.85 s	0.37%
64	31.25 s	28.35 s	-9.29%
128	20.79 s	15.98 s	-23.14%

(c) Runtime table

Figure 4.25: Runtimes and speedups of the NAS EP benchmark with Google map-reduce for problem size 200.000.000 and chunking size 10.000 on a Beowulf cluster

element of the resulting list is defined from the input list by:

$$\begin{aligned}(\text{fft } x)_i &= \sum_{k=0}^{n-1} x_k \omega_n^{ki} \\ &= \sum_{k=0}^{n/2-1} x_{2k}\omega_{n/2}^{ki} + \omega_n^i \sum_{k=0}^{n/2-1} x_{2k+1}\omega_{n/2}^{ki} \\ &= (\text{fft } u)_{i \bmod (n/2)} + \omega_n^i (\text{fft } v)_{i \bmod (n/2)} \quad \text{where } (u,v) = \text{unshuffle } 2\ x\end{aligned}$$

(see [Gor98]). Here $\omega_n = e^{2\pi\sqrt{-1}/n}$ is the nth root of unity. A simple realisation using divide and conquer skeletons can be implemented straightforwardly from this definition (see [BDLL09]). We focus in the following on an optimised parallel implementation using a distributed homomorphism skeleton.

4.4.3.1 Implementation

We use a version of FFT called `fft3` from [Gor98], which is modified to fit the requirements of the distributed homomorphism definition. Together with the distributed homomorphism computation scheme for $d = 1$, we get the following algorithm:

The following paragraph is a slightly edited version of what we have written in [LBDL09]. The input vector is divided into rows of a matrix with side lengths $l = 2^k$. Thus, the input vector has length $n = l^2 = 4^k$. The algorithm consists of three phases:

1. pre-processing: permutation of input in bit reverse order; tagging input elements with their position and their segments length; split into segments of length 2^k
2. central processing: transpose ∘ local `fft3` ∘ a global transpose ∘ local `fft3`
3. post-processing: remove tags ∘ concat [42]

The key difference between the ordinary sequential FFT and `fft3` is that the latter operates on triples which contain additional information: a position tag and the length of the lists to be joined in the current simulated combine step (the width of the combine step), which is initially 1 and will be doubled at each step. It works with *global* twiddle factors (roots of unity) to simulate a contiguous, single-dimensional FFT algorithm. The `divide` step is a trivial split of lists. The combine step needs to be modified using the additional information in the triples, namely the length of lists in the current *global* divide-and-conquer step and the position of the current element in the global list. Because of the permuted input, it is possible to perform FFT locally on the available subsets of global lists in a global manner. For more details, see [Gor98].
We can use any 2-dimensional distributed homomorphism skeleton to implement the central processing phase of the algorithm described above. These distributed homomorphism skeletons can also be used for the distributed-memory FFT algorithms proposed in [Pea62, GHSJ94].

In Listing 4.11, we present the implementation of a parallel FFT-function based on the

Listing 4.11: Parallel FFT using the monolithic parallel map-and-transpose skeleton

```haskell
fft3_2D :: Int → [Complex Double]
fft3_2D base = out ∘ dh2DMono noPe h ∘ inF $ [1..n] where
  h :: [(Complex Double, Int, Int)] → [(Complex Double, Int, Int)]
  h = fft3 base          --the fft3 implementation
  n = base^2

  inF xs = chunk base $ zip3 (bitReverse xs) [0,1..] [1,1..]
  bitReverse xs = inv n $ map fromIntegral xs

  out = map fst3 ∘ concat
  fst3 (a,_,_) = a

inv :: Int → [a] → [a]          -- permutes [a] in bit-reverse order
```

42 The final transpose step is often omitted in practice, then the transposed output has to be annotated.

`dh2Mono` skeleton of Listing 4.3. In our application the function parameter `h` to the `dh2Mono` skeleton are sequential `fft3` invocations. The skeleton is supplied with $base^2$ input elements `inp` (for $base = 2^i, i \in \mathbb{N}$) in bit-reverse order, which are tagged with each inputs position in the original list and the initial combine step width 1. The output transformation `map fst ∘ concat ∘ transpose` puts the result in the right order and removes the tags.

A direct compositional implementation is presented in Listing 4.12. We use the auxiliary function `unshuffle noPe` to distribute the input rows round robin among the processes prior to the parallel phase and we `shuffle` the results back into the original order after the parallel phase. Thus in each process of the first `parMap` phase, we need to map the `fft3` function to the local subset of rows. In order to distribute the results of this application according to the round robin distribution of tasks, we use `unshuffle noPe ∘ transpose` and thus group the submatrices required by the other processes. In the second `parMap` phase, we use `map (fft3 base ∘ shuffle) ∘ transpose` to reassemble the received submatrices and apply fft3 column-wise.

The same sequence of skeleton calls is encoded in the compositional version of the 2 dimensional distributed homomorphism skeleton `dh2DComp` of Listing 4.10, thus we can equally well use the skeleton `dh2DComp` (see Listing 4.13).

4.4.3.2 Experimental Results

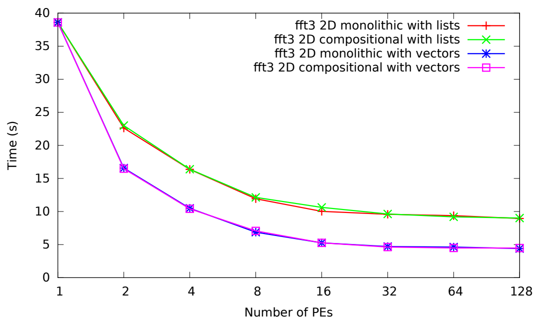
We tested the parallel FFT implementations on a Beowulf cluster (see Chapter 2.2.1) for input sizes 1024^2 and 2048^2. We compare the parallel runtimes to a purely sequential FFT

Listing 4.12: A compositional version of the 2-dimensional distributed homomorphism implementation of FFT using `noPe` processes.

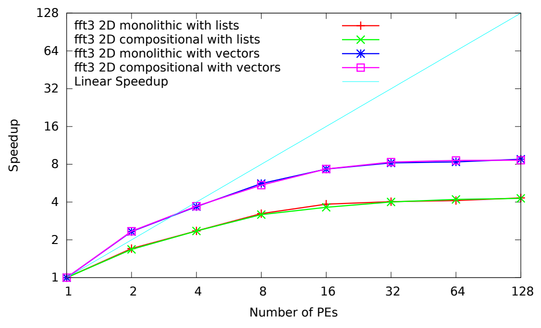
```
fft3_2DComp :: Int → [Complex Double]
fft3_2DComp  base = out
                  ∘ shuffle
                  ∘ parMap (map (h ∘ shuffle) ∘ transpose ∘ fetchAll)
                  ∘ transpose
                  ∘ parMap (releaseAll ∘ unshuffle np ∘ transpose ∘ map h)
                  ∘ unshuffle np
                  ∘ inF $ [1..n] where
  h = fft3 base ...
```

Listing 4.13: A compositional version of the 2-dimensional distributed homomorphism implementation of FFT based on **`parMapTransposeShuffleComp`**.

```
fft3_2DCompSkel :: Int → [Complex Double]
fft3_2DCompSkel base = out ∘ dh2DComp noPe h ∘ inF $ [1..n] where
  h = fft3 base ...
```



(a) Runtimes, input size 1024^2

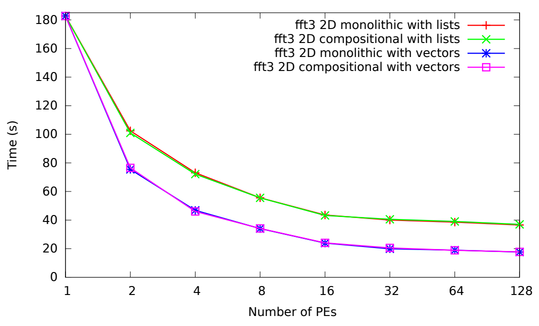


(b) Speedups, input size 1024^2

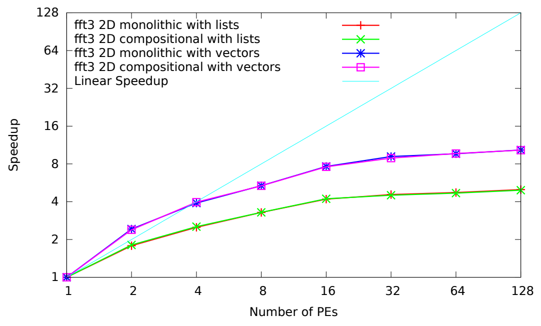
time for sequential fft with 38.59 sec.

PEs	mono. list	comp. list	overhead	mono. vec.	comp. vec.	overhead
2	22.63 s	23.00 s	1.61%	16.57 s	16.47 s	-0.55%
4	16.38 s	16.40 s	0.16%	10.48 s	10.41 s	-0.76%
8	11.93 s	12.14 s	1.68%	6.87 s	7.07 s	2.88%
16	10.03 s	10.62 s	5.59%	5.27 s	5.26 s	-0.11%
32	9.58 s	9.63 s	0.47%	4.70 s	4.63 s	-1.56%
64	9.38 s	9.19 s	-1.99%	4.62 s	4.49 s	-2.81%
128	8.94 s	9.01 s	0.78%	4.40 s	4.48 s	1.77%

(c) Runtime table, input size 1024^2



(d) Runtimes, input size 2048^2



(e) Speedups, input size 2048^2

time for sequential fft with 182.72 sec.

PEs	mono. list	comp. list	overhead	mono. vec.	comp. vec.	overhead
2	102,407 s	100,952 s	-1.42%	75,457 s	76,5002 s	1.36%
4	73,0979 s	72,1585 s	-1.29%	46,8388 s	46,0393 s	-1.71%
8	55,6188 s	55,5654 s	-0.10%	34,0457 s	34,156 s	0.32%
16	43,5637 s	43,2518 s	-0.72%	23,9065 s	24,0775 s	0.71%
32	40,0367 s	40,5543 s	1.28%	19,9424 s	20,5395 s	2.91%
64	38,5933 s	39,0292 s	1.12%	18,9483 s	18,8805 s	-0.36%
128	36,5792 s	36,9997 s	1.14%	17,6825 s	17,7416 s	0.33%

(f) Runtime table, input size 2048^2

Figure 4.26: Runtimes and speedups of FFT with 2 dimensional distributed homomorphism skeleton

implementation. For better performance, we additionally tested slightly modified versions, where we converted the rows of the matrix to vectors before sending them and converted them back to lists after receiving them. All versions include further chunking of rows (always 5 rows per chunk) to reduce the message overhead. The results are presented in Figure 4.26. Most importantly, runtimes for the monolithic versions and the compositional versions are always very close. They differ from -2.9% to $+5.6\%$, but are mostly within a range of $+/-2\%$. The versions using vector based communication are clearly faster than the basic list versions. The runtime improvement is a relatively constant amount of time[43]. Even though speedups for the vector versions are clearly better, both versions do not scale well.

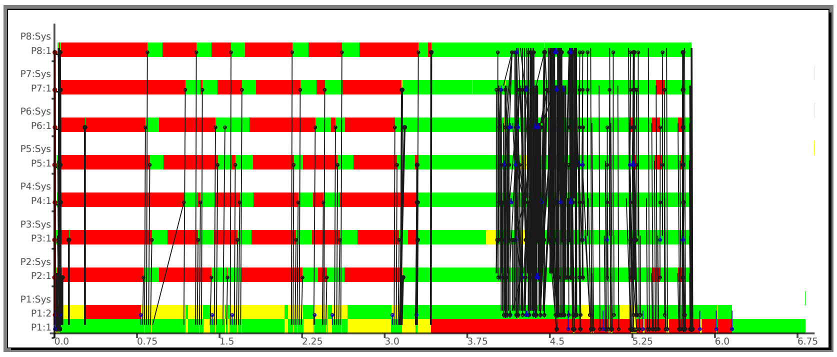
An examination of runtime traces confirms the close resemblance of runtime behaviour between compositional and monolithic versions. Figure 4.27a and Figure 4.27b depict a trace of a program run with 8 PEs and input size 1024^2 for the monolithic and the compositional vector version. Apart from the additional processes, the behaviour is the same. Figure 4.27c and Figure 4.27d show the program behaviour for the monolithic version with 16 and 32 PEs, which reveal the reason for the bad scalability of the program. The sequential input and output transformations take a relatively constant amount of time. Despite the fact that the parallel processing phase decreases constantly, speedups are limited due to the substantial sequential overhead.

We will discuss a FFT implementation based on the general distributed homomorphism skeleton in the iteration Chapter 5.

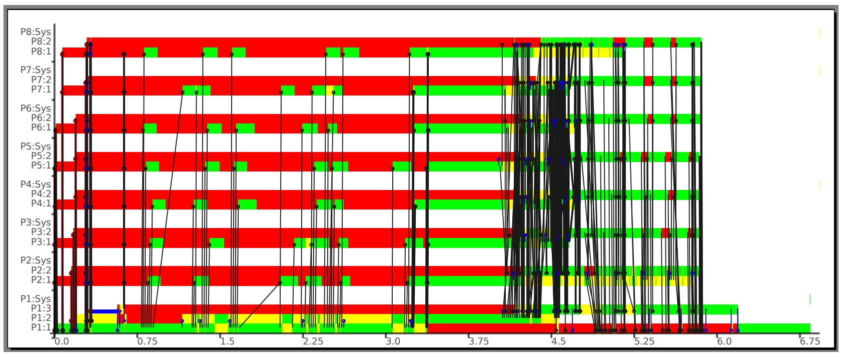
4.4.4 Parallel Sorting by Regular Sampling (PSRS)

The PSRS section is a revised version of [DHLB16]:
In the area of distributed parallel sorting, PSRS: Parallel Sorting by Regular Sampling [LLS+93] 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`[44] when collecting the workers' results. This property ensures that the PSRS algorithm can be efficiently composed with other skeletons for distributed data processing.

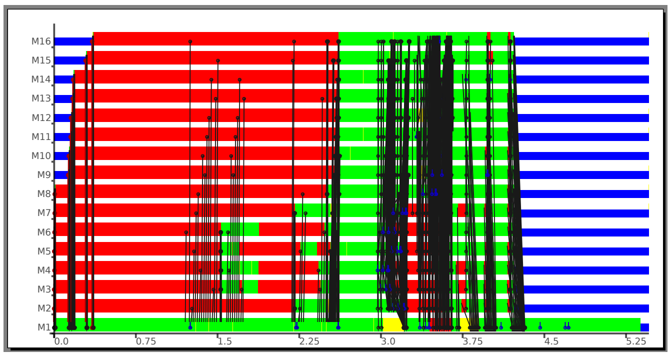
43 5-6 sec. for input size 1024^2, 20-25 sec. for input size 2048^2.
44 Function `sortMerge` merges pre-sorted lists.



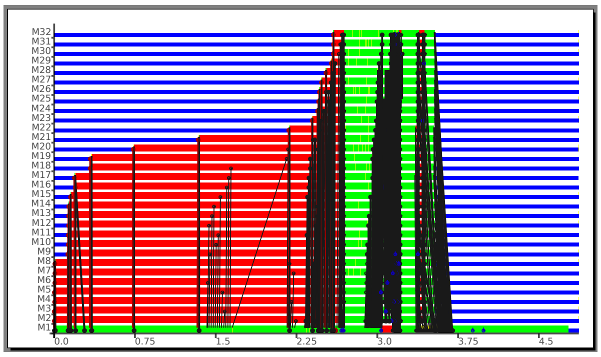
(a) monolithic using `allToAll`, 8PEs, processes per machine view



(b) compositional, 8PEs, processes per machine view



(c) monolithic, 16PEs, machine view



(d) monolithic, 32PEs, machine view

Figure 4.27: Traces of FFT with 2 dimensional distributed homomorphism skeleton with vector based communication for input size 1024^2

The parallel algorithm on p PEs operates by segmenting pre-sorted sub-lists into p segments. Assuming that input is provided in p segments of equal size, PSRS consists of 4 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 $(1 \leq j \leq p)$, keeping one partition;
4. In parallel: Each process merges $p-1$ partitions received from siblings with its own.

4.4.4.1 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 4.28 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[45]. These handles are returned to the parent process, which transposes the matrix of $p \times p$ handles and returns them to the child processes. Hence, every process of Phase 4 is responsible for one of the partitions: It `fetch`es the respective partitions from all other processes, merges these (sorted) partitions, and `release`s its segment of the globally-sorted distributed data.

The process network is depicted in Figure 4.29. Figure 4.30 shows a runtime trace on 8 PEs with input size $n = 1000000$ 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). Phases 3 and 4, after the parent has processed the samples, are marked by communication with other

45 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.

```
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
```

Figure 4.28: PSRS in Eden

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.

4.4.4.2 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 [BDL09], our parallel FFT implementation [LBDL09] or the n-body simulation presented in Chapter 5.2.2 of this paper. Therefore, this pattern is provided as a general topology skeleton `allToAllRD` [DHL10], 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 Fig 4.31, definition of `inp` and two-component worker function `p` in arrow notation).

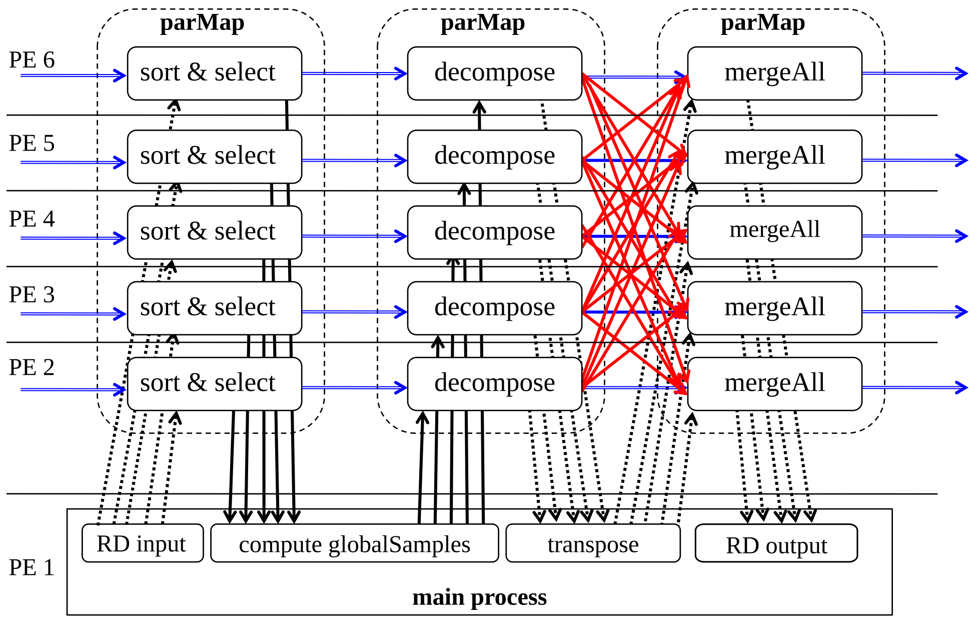
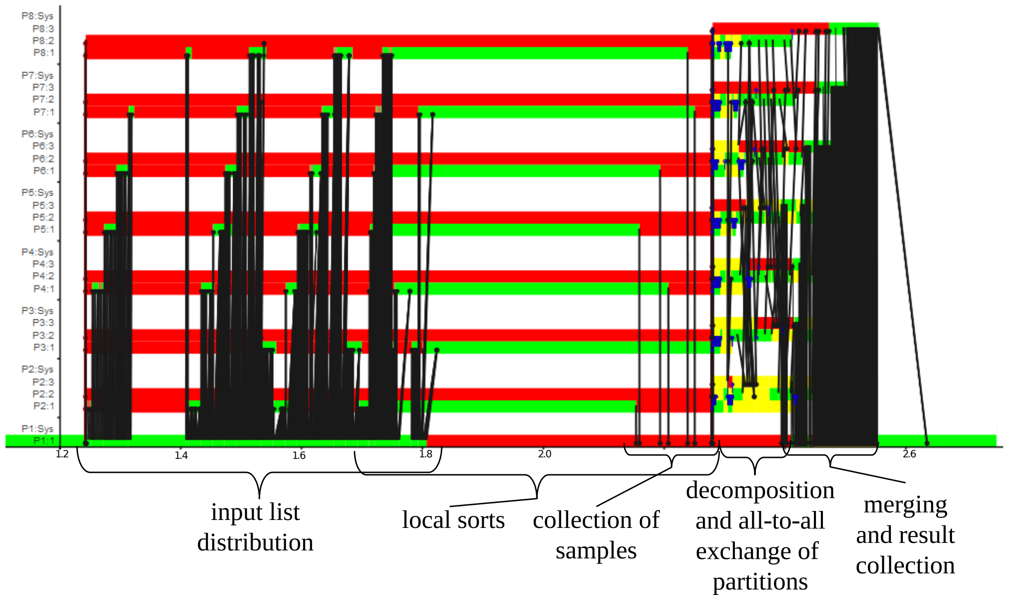


Figure 4.29: PSRS Process Network



PSRS: Input Size 1000000, Chunk Size 1000
Parallel Runtime: 2.76s, 8 Machines, 22 Processes, 177 Threads, 2311 Messages

Figure 4.30: Activity Profile of PSRS: Final Phase and Communication

```
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)
```

Figure 4.31: The `allToAllRD` skeleton

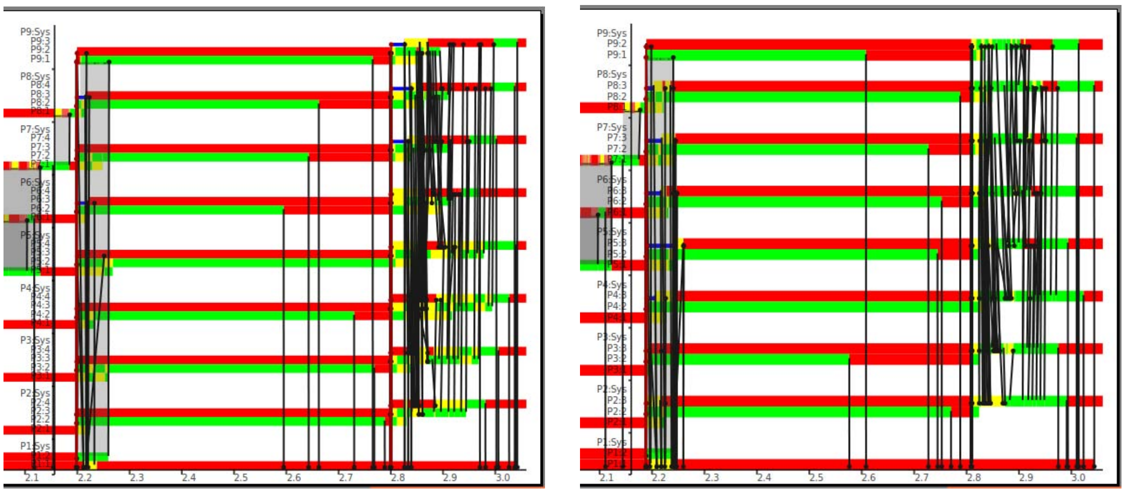
Also note a necessary invariant which cannot be expressed in the (standard) Haskell type system: As the transposed input is supplied to n child processes (second input component), each process should compute n intermediate values from the argument to function `t1` (first input component). `t1` is supplied 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).

4.4.4.3 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 4.33 shows the runtime trace of the two PSRS versions (compositional implementation on the left), with input size 1000000 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. This is confirmed by the runtimes and speedups of PSRS on an increasing number of processors with input sizes 10,000,000 and 60,000,000 (see Figure 4.34). Apart from the 2 PE run for input size 60,000,000 with a difference of −14.26%, runtimes are within a range of −7.5 to

```
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
```

Figure 4.32: PSRS using all-to-all in Eden



PSRS: Input Size 1000000
9 Machines, 33/25 Processes, 282/298 Threads, 2060 Messages

Figure 4.33: PSRS Compositional (left) and using All-To-All (right) with 9 PEs (8 Workers) Input Size 1000000, Zoom into Distributed Phase

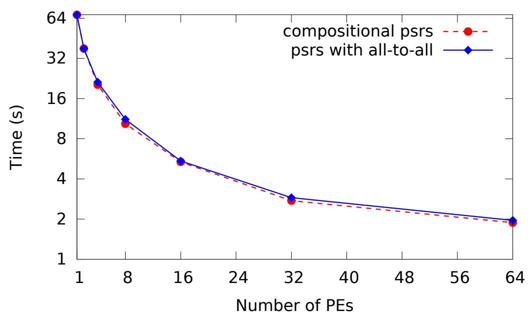
6%. The close correspondence between both versions is independent from the input size. The algorithm scales well when input and output stay distributed.

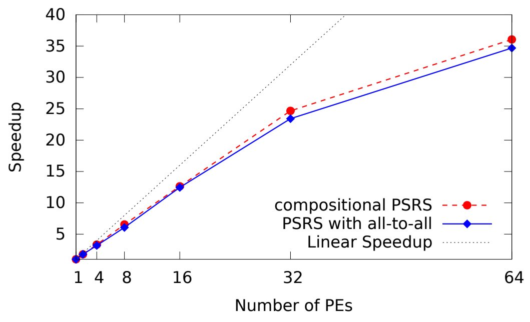
Comparing the results of this chapter, we get the following table:

	Mandelbrot	NAS EP	FFT (2D-DH)	PSRS
min	-2.56 %	-23.14 %	-2.81 %	-14.26 %
max	2.98 %	7.21 %	2.91 %	5.85 %
mean	0.14 %	-2.25 %	0.17 %	-2.55 %
abs mean	1.33 %	3.85 %	1.30 %	3.96 %

Apart of the minimal (min) and the maximal deviation (max) of runtimes between monolithic and compositional versions, we present mean values of the percental deviation (*mean*) and mean values for the absolute percental deviation (*abs mean*). If the *abs mean* value differs substantially from the absolute value of the *mean* value, then there are differences of runtimes in both directions which partially neutralise themselves on an average. One version constantly outperforms the other if the absolute value of the *mean* value and the *abs mean* value are identical.

The *abs mean* values are always below 4%. The mean values are between -2.55 % and 0.17%, thus monolithic and compositional versions perform really close. Surprisingly we get slightly better results for the compositional version, which may as well be caused by the concrete selection of case studies. We compared program versions constructed by a

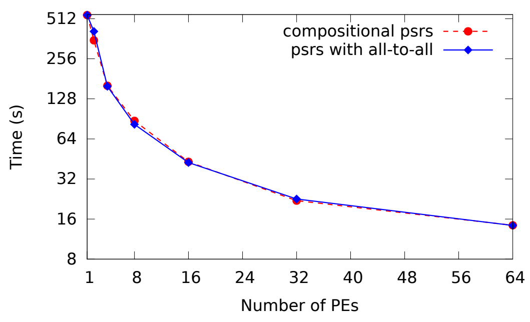


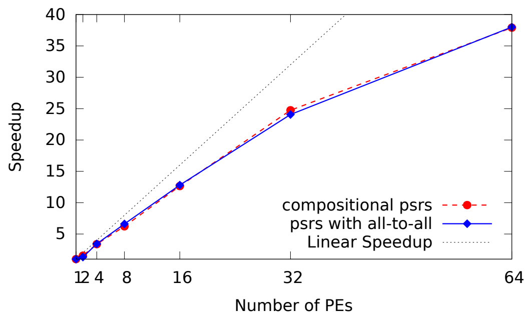


sequential time with `sort` from `Data.List` (mergesort): 67.87 sec.

PEs	psrs with all-to-all	compositional psrs	overhead
2	37.92 s	37.96 s	0.10%
4	21.24 s	20.32 s	-4.32%
8	11.18 s	10.36 s	-7.36%
16	5.45 s	5.37 s	-1.46%
32	2.90 s	2.75 s	-5.06%
64	1.96 s	1.88 s	-3.80%

Input size 10,000,000





sequential time with `sort` from `Data.List` (mergesort): 544.67 sec.

PEs	psrs with all-to-all	compositional psrs	overhead
2	410.90	352.30	-14.26%
4	158.92	160.67	1.09%
8	82.35	87.47	5.85%
16	42.54	43.05	1.19%
32	22.64	22.01	-2.80%
64	14.33	14.37	0.28%

Input size 60,000,000

Figure 4.34: PSRS Runtimes and Speedups, only Distributed Phase

single skeleton composition step to monolithic ones or in the case of PSRS, we compared two composition steps to a single composition step. Our results indicate that skeleton composition using remote data performs competitive to monolithic skeletons for a single composition step.

CHAPTER 5

Skeleton Iteration

Chapter 5 - Chapter 5.2 are revised versions of [DHBL13].

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

Example. Consider a simple *genetic algorithm* which computes the development of a population of individuals under some mutation until a termination criterion is met. The flowchart in Figure 5.1 shows the iterated steps of the algorithm.

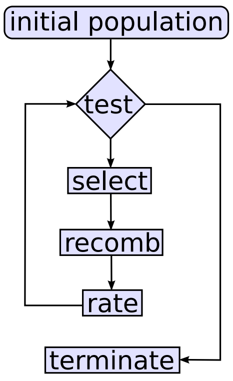


Figure 5.1: Flowchart

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

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

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

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

As the parallel behaviour is encapsulated inside a skeleton's implementation, it is generally very hard to optimise the repeated use of a skeleton without modifying the skeleton itself. On the other hand, a solution that involves rewriting parallel skeletons for every concrete sequence of applications is not favourable; we seek for a more general method to compose skeletons for iterative computations, which we call *skeleton iteration*[46] subsequently.

Our Approach. We propose a general functional iteration scheme `iter` which is a meta-skeleton (combinator) using an iteration control function and an iteration body skeleton as parameter functions. Specific control and body functionality can be freely combined to express a wide range of iterative algorithmic patterns. We show how to lift ordinary skeletons in a systematic way to persistent skeletons which work on communication streams such that they can be used as body skeletons or for control in our iteration scheme. Involving special types for describing iterative processing improves programming comfort and safety. Special support is provided for iteratively processing distributed data structures.

We have implemented our iteration framework in the parallel Haskell dialect Eden [LOMP05]. The functional approach makes it easy to precisely state interfaces and to identify conceptional requirements from our implementation. Two non-trivial case studies, K-means and N-body, have been used to compare the performance of the framework with the straightforward approach of recursive skeleton instantiations and, at least for K-means,

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

with a monolithic customised `parMap` iteration skeleton [PR01]. In the K-means case study, our framework performs much better than the straightforward recursive approach and is competitive with the specialised monolithic skeleton. In the N-body case study, our approach scales better and clearly produces less overhead than the straightforward recursive approach. However, we observe slightly better overall runtimes for the recursive approach on small numbers of processors.

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

5.1 Iterating Skeletons

The Haskell prelude function `iterate` defines the iteration of a parameter function `f`, producing an infinite list (or: stream) of all intermediate results of the iteration: `[x,(f x),(f (f x)),...]`. The same stream can be defined in a different way, using the `map` function and a feedback of the result stream instead of a recursive function call.

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

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

We are especially interested in the case where the parameter `f` of `map` is a parallel skeleton, i.e. when evaluation of `f` involves the creation of threads and/or processes and communication of data between these parallel entities. Both the `iterate` function and the variant `streamIterate` above would in this case repeatedly construct and destroy the parallel process system evaluating `f` in every iteration step. As an illustrative example, consider the case where the parameter function `f` of `map` is itself a parallel `map` skeleton (`parMap`), i.e. creates a parallel process for each input list element to apply the parameter function to it.

The following specialised version of `streamIterate` implements this:

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

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

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

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

Now the iteration via `map` takes place within the processes created by `parMap` only once, saving the process creation overhead.

It is by virtue of streaming and the use of `map` to express the iteration that we can lift the body skeleton to work on streams and push the iteration inside the processes. Just swapping `map` and `parMap` in the definition (leading to `parMap (map g) xss`) would instead lead to a pseudo-parallelisation over the *stream* instead of over the *lists*. However, this variant has correct type, due to a missing distinction between lists (for parallelism) and streams (for iteration). In the following, we will propose special types and mechanisms to generalise this approach and make a clear distinction between the iteration stream and the list of inputs to the parallel processes. We will also add special control functions for the iteration to improve locality and performance.

5.1.1 Iteration Type and Body Skeletons

Communication-related properties of Eden processes are determined by types, using overloaded communication functions in the type class `Trans` for transmissible data. Instances for `Trans` determine different send modes: while the default mode is to fully evaluate and send data as a single item, product types (tuples) can be decomposed and sent concurrently, and recursive types (such as lists) can be transmitted as streams, element by element. The important aspect here is that the type of a process determines the communication mode for its in- and outputs. In our framework, we use self-referential streams of data to describe iteration data as opposed to parallel input to worker processes.

Special Stream Type for Iteration.

In our example above, streams were modeled as lists, leading to a potential pseudo-parallelisation of the algorithm when parallelisation is applied at the outer level. In order to have a clear distinction of the (sequential) iteration stream and the (parallel) input to the iteration body, we introduce a special iteration type `Iter` (see Figure 5.2), which is isomorphic to lists but different with respect to the communication mode. This enables the programmer and the type checker to identify iteration inputs and outputs in type signatures and thereby increases readability and type safety. Furthermore, the intended streaming behaviour can be defined in a targeted manner by an appropriate `Trans` instance for `Iter`, while other lists can be communicated as single items[47].

Aside from the new data type, Figure 5.2 shows auxiliary functions for common uses of `Iter` data when defining efficiently iterable skeletons. The functor instance of Iter provides `fmap`, lifting a function of type `a → b` to iteration streams, `Iter a → Iter b`. `fmap` is used to realise iteration of the body skeleton. Function `distribWith` splits a single iteration stream into many iteration streams, where the i'th element of each output stream is generated from the i'th element of the input stream. The function parameter `f` produces a list of output elements for each element of the input iteration stream; these lists are then distributed into a list of output streams using `map Iter ∘ transposeRt`. Consider the special case of `f = id`, which implies `a = [b]` and merely interchanges an outer `Iter` and an inner list. One subtle detail here is that `f` must produce lists of identical length for all its arguments (elements of the iteration stream), and the transposition needs to use a custom function `transposeRt` for rectangular matrices which fixes the length of its result list to the length of the first inner list of its input. In other words, the number of output streams, which defines the parallelism degree, is determined by the first incoming stream element. Finally, the function `combineWith` defines the inverse transformation (and does

```
newtype Iter a = Iter {fromIter :: [a]}

instance Functor Iter where
    fmap f = Iter ∘ map f ∘ fromIter

distribWith :: (a→[b]) → Iter a → [Iter b]
distribWith f = map Iter ∘ transposeRt ∘ map f ∘ fromIter

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

Figure 5.2: Iter type and auxiliary functions

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

not impose restrictions on the transposition used).

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

Figure 5.3: Parallel map as an iteration body

With these tools at hand, it is easy to define the efficient iterable version of `parMap` in a more readable and type-safe way (see Figure 5.3). Now transforming inputs of type `Iter [b]` element by element to outputs of type `Iter [c]`, `simpleParMapIter` can be used as an iteration body, transforming a stream of input lists (of equal length, see above) in a set of parallel mapper processes; suitable for an iteration loop. Effectively, this is achieved by only using the type conversion and the `transposeRt` inside `distribWith` (but setting the transformation function to `id`), to generate a list of streams from the stream of lists. The resulting stream of lists is transformed using `parMap (fmap f)`, and the results of type `Iter c` streamed back to the caller (bound by `yss' :: [Iter c]`), after reordering them again using function `combineWith`.

In the same way we can define a iterable version of function `spawnF`:

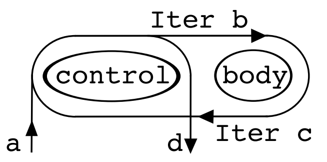
```
simpleSpawnFIter :: forall b c.
                   (Trans b, Trans c)
                   ⇒ [Iter b → Iter c]
                   → Iter [b]
                   → Iter [c]
simpleSpawnFIter fs = combineWith id ∘ spawnF fs ∘ distribWith id
```

Here we do not map the functions to the inputs. We rather allow that the functions work on the whole input of a process.

5.1.2 Iteration Scheme

Iteration control consists of linking together the two "loose ends" of the iteration streams, to produce new input and decide termination. The body skeleton's input stream must be started by initial data, and the result stream must be conditionally fed back to the body skeleton, or terminated by closing the input stream and returning a final result. This can be defined in terms of a generic iteration scheme:

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



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

The body skeleton is allowed to transform input of type `Iter b` to a different type `Iter c`. Thus, output cannot be fed back directly by the control function, but needs to be transformed back from `Iter c` to `Iter b`, in an element-wise fashion. The iteration control function also needs to check a termination condition, and to produce the final output from the iteration body's output upon termination. Two simple examples for iteration control functions are `loopControl`, which performs exactly *n iterations* by forwarding n inputs without any transformation, and `whileControl`, which takes a function parameter `checkNext` to transform the initial input and iteration output of type `a` to a new iteration input of type `b` (`Left` alternative). It stops the iteration with a result of type `d` (`Right` alternative).

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

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

The expressiveness of `whileControl` is limited because the control function `checkNext` only considers the output of a single iteration step to decide termination or to compute the input for the next step. The general control function type in `simpleIter` is much more liberal, in fact it is not even required that the control function generates exactly one iteration body input for each iteration body output. It appears more suitable to use a *state-based* control function. This can be provided by a generic function parameterised with a state transformation function for a single iteration step, thereby combining safety and flexibility of the approach. We stick to stateless versions at the moment for reasons of simplicity. We have implemented a generic stateful version, use it in our measurements and present it in Chapter 5.3.1.

5.1.2.1 Running Example.

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

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

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

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

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

5.1.3 Performance Tweaking

The main potential for optimisation of iteration steps lies in reducing communication overhead. One obvious bottleneck is that data is gathered in the control function and then redistributed to the body skeleton in each step. One approach to optimise communication is to *keep all data distributed* between the iterations. In Eden, this can be done using remote data [DHL10].

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

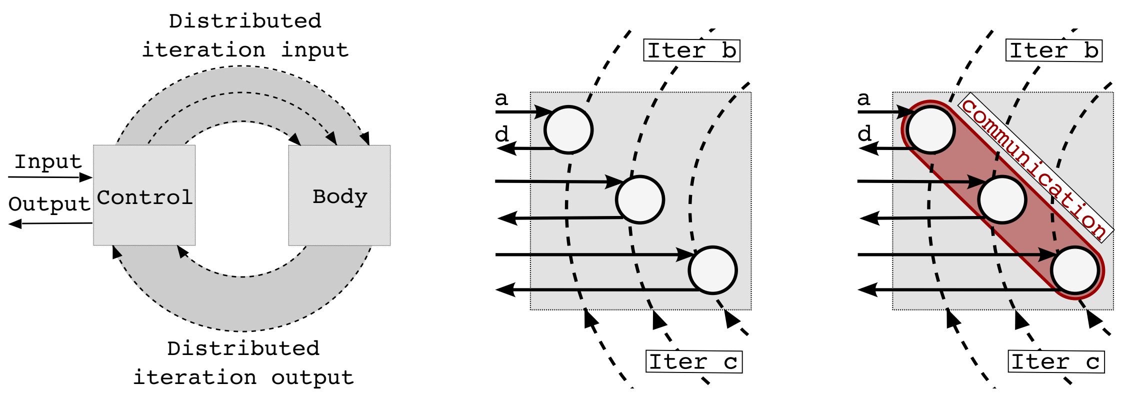
```
simpleParMapIterRD :: (Trans b, Trans c)
                       ⇒ (b → c) → Iter [RD b] → Iter [RD c]
simpleParMapIterRD f = simpleParMapIter (release ∘ f ∘ fetch)
```

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

5.1.4 Parallel Iteration Control Skeletons

In many cases where the iteration body uses a skeleton to work on distributed data, a corresponding *control skeleton* with parallel processes can be used to inspect the distributed data, exchanging only the parts of it that are needed globally (see Figure 5.4a[48]). In addition, corresponding processes of control and body skeleton can be placed on the same processor element to avoid communication[49]. This concept can be used with arbitrary distributed data structures, in our implementation we focused on the special case of iterations over distributed lists (lists of remote data).

Two different types of parallel iteration control can be distinguished: *local* and *global* iteration control, with respect to the data dependencies in each one of the control processes.



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

Figure 5.4: Parallel iteration control

48 Note that the number of arrows in between control and body is not the same as the other way around. This indicates that the parallelism degree does not have to be identical for both directions in general, even though this is typically the case.

49 The parallel Haskell dialect Eden supports explicit placement of computations in a multi-node parallel system. We have omitted placement aspects from our code for simplicity throughout.

5.1.4.1 Local Iteration Control

Local iteration control means that tasks of iteration control can be fulfilled without exchanging data with other control processes – data dependency is *local*, as depicted in Figure 5.4b. Otherwise, a *global* data exchange is necessary, as depicted in Figure 5.4c.

A local iteration control skeleton for lists of remote data is given in Listing 5.1. The implementation is similar to the implementation of `parMapIterRD`, but takes the two input values and the tuple output –both containing remote data handles– into account. The control processes will connect both to their predecessor processes that produce the distributed list beforehand and to the processes of the body skeleton, fetching required data on-demand, or else passing on the `RD` handles. Functionality in each process is described by the process-local control function which transforms the initial input and the output of a process in the iteration body (stream-wise) in the respective control process. This skeleton can implement several common iteration control variants simply by partially applying the control skeleton to a suitable control function. E.g. a variant of `whileControl` where termination can be decided from local data would be:

```
simpleLocalWhileCtrl :: (Trans a, Trans b, Trans d)
                        ⇒ (a → Either b d)
                        → [RD a] → Iter [RD a] → (Iter [RD b],[RD d])
simpleLocalWhileCtrl checkNext = simpleLocalControl (whileControl checkNext)
```

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

Listing 5.1: Process-local iteration control skeleton

```
simpleLocalControl :: forall a b c d. (Trans a, Trans b, Trans c, Trans d)
  ⇒ (a → Iter c → (Iter b, d))    -- ^process local control:
  → [RD a]                        -- ^initial Input
  → Iter [RD c]                   -- ^output of loops
  → (Iter [RD b], [RD d])         -- ^input for loops, final result
simpleLocalControl controlF as css = (combineWith id bss, ds) where

  css'     = distribWith id css
  (bss,ds) = unzip $ parMap f $ zip as $ lazy css'

  f :: (RD a, Iter (RD c)) → (Iter (RD b), RD d)
  f (a, cs) = (fmap release bs, release d) where
    (bs,d) = controlF (fetch a) (fmap fetch cs)
```

5.1.4.2 Global Iteration Control

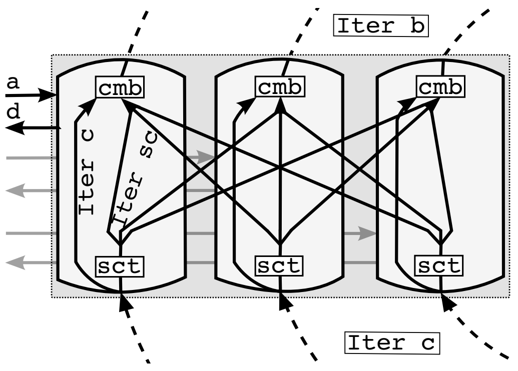


Figure 5.5: all-gather control

If the control function needs information from multiple processes to calculate the next input for the body or to determine termination, the processes of the control skeleton need to exchange these data. As an example of this kind of control skeleton, consider an *all-gather* pattern where all processes gather selected data of all other processes in a distributed manner. We only discuss the signature of the skeleton here, given in Figure 5.6. Aside from the iteration body output (distributed list of type `[RD c]`, iterated), the input for the next iteration and the final result (distributed lists `[RD b]` and `[RD d]`) depend on additional synchronisation data (of type `sc`, iterated). Combine function `cmb` produces the local next input and result, but considers the entire list of synchronisation data (iterated) and the own position in the list of processes (`Int`). Select function `sct` yields the local synchronisation data which will be communicated to all other control processes.

A skeleton `simpleAllGatherWhileControl` can be defined as a specialisation of skeleton `allGatherControl` with simpler interface, where type `a=c`. The select and combine function of this skeleton work on single elements of the iteration stream. The encoding of the termination condition in `cmb` is similar to the simple `whileControl` function presented in Chapter 5.1.2.

5.1.4.3 Running Example.

In order to implement our genetic algorithm with a parallel control skeleton, we use a global control variant, as functions `test` and `select` from the genetic algorithm must

```
simpleAllGatherControl::(Trans a, Trans b, Trans c, Trans d, Trans sc)
  ⇒ (a → Iter c → Iter sc)                            --sct
  → (Int → a → Iter c → Iter [sc] → ((Iter b),d))  --cmb
  → [RD a]→ Iter [RD c] → (Iter [RD b],[RD d]) --controlType

simpleAllGatherWhileControl :: (Trans a, Trans b, Trans d, Trans sc)
  ⇒ (a → sc)                                          --sct
  → (Int → a → [sc] → Either b d)                     --cmb
  → [RD a]→ Iter [RD a] → (Iter [RD b],[RD d]) --controlType
```

Figure 5.6: Global control: the `simpleAllGatherControl` Skeleton

consider the whole population. We use `allGatherWhileControl` with `test_select` and `parMapIterRD` with function `recomb_rate` to define control and body of the iteration.

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

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

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

In the control part, we synchronise the whole *local* population (`sct=id`) with all processes, such that every process can use the whole *global* population in function `cmb`. To implement the latter, we use `test_select` on the whole global population `pop`. If this doesn't yield termination (`Right res`), `test_select` returns the input `Left next` for all the body processes. We use the process position (`self`) to select the part for one body process.

5.1.5 Inlining the Iteration Streams

In the previous two sections, we focused on input/output data of type `[RD a/b]`, which is something like a distributed list type. For the iterated body and the control skeletons, we used the interface `Iter [RD a/b]` because we defined in the `simpleIter` scheme that data of type `a` will be passed during the iteration using type `Iter a`. There are two drawbacks of the implementations based on this signature:

1. The channel connections between the processes of the body and the control skeleton have to be rebuilt in every iteration step.
2. In the skeleton definitions, we have to drag the iteration stream from the outside of the iterated list to its elements.

In the definition of `simpleParMapIter`, we used function `distributeWith` and reversed its effects by `combineWith` when defining the skeleton's result. Similar transformations are necessary for other body skeletons as well as control skeletons. What is actually desired is a remote data connection list that itself carries iterated data, leading to type `[RD (Iter a)]`. If we had this type, a stream of data would be communicated over remote data connections established only once. The following `parMap` variant with modified interface implements these static remote data connections:

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

Notice that we can express the iterable skeleton simply by transforming the function parameter. We observed that the transformation of more complex topology skeletons, such as `allToAllRD` and `allReduceRD` (both developed in the context of remote data [DHL10]), are similarly easy, only involving the respective function parameters (all transformations done by the nodes are function parameters to these skeletons). The `allToAllRD` skeleton can e.g. be adapted for Iteration streams essentially by lifting its parameter functions appropriately. Function `t1` generates the inputs for the all-to-all connection, `t2` combines the outputs from the all-to-all connections, and both are lifted using `distribWith` and `combineWith'`[50]:

```
allToAllIter :: (Trans b, Trans c, Trans i) ⇒
  (Int→b→[i]) → (b→[i]→c)
  → [RD (Iter b)] → [RD (Iter c)]
allToAllIter t1 t2 = allToAllRD t1Iter t2Iter where
    t1Iter p = distribWith $ t1 p
    t2Iter   = combineWith' t2

combineWith' :: (c→[b]→a) → Iter c → [Iter b] → Iter a
combineWith' f (Iter bs) = Iter ∘ zipWith f bs ∘ transpose ∘ map fromIter
```

Lifting skeleton `allReduceRD` to `allReduceIter` (which uses a butterfly scheme for a more efficient reduction than the former `allToAllRD`), is similarly easy.

```
allReduceIter :: (Trans b, Trans c) ⇒
  (b → c) → (c → c → c) → [RD (Iter b)] → [RD (Iter c)]
allReduceIter t r = allReduceRD (fmap t) (liftIter2 r)
 liftIter2 :: (a → b → c) → Iter a → Iter b → Iter c
 liftIter2 f (Iter bs1) (Iter bs2) = Iter $ zipWith f bs1 bs2
```

It uses two function parameters, function `t` transforms the initial input of each process to the reduction type. The reduce function `r` is then applied $log(n)$ times in all the nodes of the butterfly scheme. We lift `t` using `fmap`, and `r` using `liftIter2`. The latter is implemented similarly to `fmap` but uses `zipWith` instead of `map` because `r` takes two parameters.

The iteration streams to and from all body processes have to be processed by a control function or skeleton which exactly matches the particular distributed data shape. This constraint can be fulfilled by adjusting the previous iteration meta-skeleton to a different type signature (`iterD`, with an implementation identical to the earlier `simpleIter`):

```
iterD :: (a → [RD (Iter c)] → ([RD (Iter b)],d))
         → ([RD (Iter b)] → [RD (Iter c)])
         → a → d
```

Further to using `iterD`, we need to define specialised versions of local and global iteration control for this interface, which is again a simplification of the existing implementations.

50 `combineWith'` is a variant of `combineWith` with an additional input.

Here the implementation of `localControl`:

```
localControl :: forall a b c d. (Trans a, Trans b, Trans c, Trans d)
  ⇒ (a → Iter c → (Iter b, d))  -- ^process local control
  → DList a                     -- ^initial Input
  → Iterated (DList c)          -- ^output of loops
  → (Iterated (DList b),DList d)      -- ^input for loops and final result
localControl controlF as css  = (bss, ds) where

  (bss,ds) = unzip $ parMap f $ zip as $ lazy css

  f :: (RD a, RD (Iter c)) → (RD (Iter b), RD d)
  f (a, cs) = (release bs, release d) where
    (bs,d) = controlF (fetch a) (fetch cs)
```

This version works without `distribWith` and `combineWith` on the iteration stream and manages to `fetch` and `release` each `Iter` stream on the processes as a whole.

5.1.6 Unifying the Interface

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

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

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

```
type family Iterated a :: *

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

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

We use the simple type mapping `type instance Iterated a = Iter a` for ordinary types `a` — those types that use the `simpleIter` scheme. It is not possible to allow overlapping

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

Figure 5.7: General iteration skeleton

instances for type families, so we have to define these instances for every base-type separately. The following constraint `CSimpleIter a` for a type `a` can be used in the type context of a skeleton to allow only input with according type family mapping:

```
type CSimpleIter a = Iterated a ~ Iter a
```

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

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

The type family enables us to define a generic but type-safe iteration skeleton `iter` (see Figure 5.7) that works for both `DList`s and for any other reasonable type instance of `Iterated`. The small caveat is that two dummy parameters `b` and `c` are introduced in the control function, in order for the typechecker to check the types `Iterated b` and `Iterated c`. This is needed because the type family mapping might not be injective.

5.2 Evaluation

We measured the performance of our iteration framework on the Beowulf cluster (see Chapter 2.2.1). The cluster provides a total of 256 processor cores. However, as it could not be used exclusively, measurements are limited to a maximum of 128 processor cores. All program versions where tested on 2^i processors with i ranging from 0 to 7. The reported runtimes are mean values of 5 program runs. They are presented in diagrams with logarithmically scaled axes, with runtimes corresponding to a linear speedup indicated by dotted lines. In the following we present measurement results for two non-trivial case studies: k-means and n-body.

5.2.1 K-means

K-means clustering is a heuristic method to partition a given data set of n d-dimensional vectors into k clusters. In an iterative approximation, the method identifies clusters such that the average distance (a metric such as the euclidian or Manhattan distance) between each vector and its nearest cluster centroid is minimal [Mac03]. The algorithm proceeds as follows:

- Randomly choose k vectors from the data set as starting centroids.
- Define clusters by assigning each vector to the nearest centroid.
- Compute the centroids of these clusters.
- Repeat the last two steps until the clusters do not change anymore.

The iteration body takes a list of cluster centroids as input and computes the list of new centroids as output. The iteration continues until two subsequent iteration results are equal or their differences fall below a threshold.

The cluster assignment and part of the centroid computation can be parallelised using `parMap`. Each parallel process receives a subset of the vectors, and the whole list of centroids. Every process then computes a list of weighted sub-centroids which are combined to the list of new centroids by the iteration control.

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

- *recursive parMap* is a naïve implementation which creates new processes and re-distributes not only the centroids but also the (unchanged) list of vectors in each iteration step. Here we do not use remote data. As the parallel processes are newly created for each step, there is no way to share the vector list across iterations.
- *untilControl/simpleParMapIter* uses our iteration scheme with *stateful* versions of `untilControl` and `simpleParMapIter`. Only the centroids are gathered and distributed for each iteration step, while the data vectors are once distributed and then kept in the worker states during the iteration.
- *monolithic iterUntil* uses the monolithic iteration skeleton `iterUntil` presented in [PR01]. Like the composed version above, it uses a stable process system and holds the data set in local states. While being a perfect match for the parallel k-means, other iterative algorithms would require a complete re-design and re-write of the skeleton.

Figure 5.8 shows the mean runtimes plotted against the number of processors. The modular skeleton *untilControl/ simpleParMapIter* performs as well as the specialised *monolithic iterUntil* version. Both scale well, showing an almost linear speedup up to 8 processors. On more than 32 processors, initialising and distributing the vectors increasingly influences runtime, leading to lower speedup.

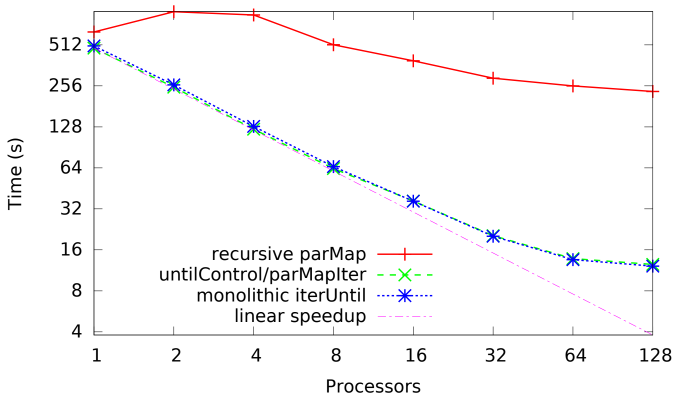


Figure 5.8: Runtimes for k-means with 600000 vectors, 25 clusters, 142 iterations

PEs	iterUntil (1)	untilControl/ parMapIter (2)	recursive parMap (3)	(1) vs (2)	(1) vs (3)	(2) vs (3)
1	503.37 s	485.21 s	637.08 s	-3.61%	20.99%	23.84%
2	260.56 s	250.66 s	897.20 s	-3.80%	70.96%	72.06%
4	128.92 s	122.98 s	849.33 s	-4.61%	84.82%	85.52%
8	65.30 s	63.24 s	512.98 s	-3.16%	87.27%	87.67%
16	36.40 s	36.54 s	390.97 s	0.37%	90.69%	90.65%
32	20.15 s	20.34 s	291.33 s	0.94%	93.08%	93.02%
64	13.56 s	13.79 s	255.73 s	1.67%	94.70%	94.61%
128	12.14 s	12.51 s	232.88 s	2.98%	94.79%	94.63%

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

5.2.2 N-body

The n-body problem is a simulation of the movement of n particles in a 3-dimensional space taking into account their mutual gravitational forces. In a straightforward parallel n-body algorithm, particles are distributed to processes and each process computes the new velocity and position for its own particles. To update its particles' velocities, each process needs position and mass (but not velocity) of all other particles. This information needs to be exchanged in-between the iterations, leading to considerable communication between the parallel processes, in contrast to the parallel k-means algorithm described earlier. These functions are used before and after the exchange:

```
getMassPoint :: Body → MassPoint
updateAll    :: [Body] → [[MassPoint]] → [Body]
```

We used variants of the skeleton `allToAllRD`:

`allGatherRD`, `allGatherIter` and `simpleAllGatherIter`

to parallelise the iteration body. These skeletons simply are defined by setting the transformation function `t1'` of the respective all-to-all variant to `t1' n x = replicate n (t1 x)`, where `t1 :: a → b` is a transformation function of the `allGather` skeleton. Each process holds a subset of the particles and processes exchange particle information in every iteration step in a distributed manner using the all-to-all topology. We implemented the following versions:

(1) *recursive allGatherRD* recursively instantiates the skeleton `allGatherRD`. As the corresponding processes are allocated on the same processor in each iteration, all data transfers occur between processes on the same processors. The Eden runtime system optimises this processor-local communication by passing references to existing data instead of serialising and sending it. I.e. processor-local communications do not incur any overhead apart from evaluating `rnf` when sending local messages.

```
doStepsAllGatherStream :: Int → [Body] → [Body]
doStepsAllGatherStream n = shuffle ∘ fetchAll
                          ∘ doStepsAllGatherStream' n
                          ∘ releaseAll ∘ unshuffle noPe

doStepsAllGatherStream':: Int → [RD [Body]]
                              → [RD [Body]]
doStepsAllGatherStream' 0 bs = bs
doStepsAllGatherStream' s bs = doStepsAllGatherStream' (s-1) new_bs where
  new_bs = allGatherRD (map getMassPoint) updateAll bs
```

In the *recursive allGatherRD* implementation, the particles are passed locally between the processes of the different skeleton instances. Thus, the only remaining overhead consists of the repeated process and all-to-all topology creations.

(2) *localLoopControl/allGatherIter* instantiates our `iterD` scheme with the skeletons `localLoopControl` for the iteration control and `allGatherIter` for the body[51]. It uses persistent remote data connections. The drawback is that the output streams from the body processes have to be fetched in the control processes for iteration counting and released again when forwarding them to the body-processes. This involves an additional use of `rnf`.

```
doStepsAllGatherIterStream :: Int → [Body] → [Body]
doStepsAllGatherIterStream n bs = shuffle ∘ fetchAll $ rdBs' where
  dlBs = DList [1..noPe] (releaseAll ∘ unshuffle noPe $ bs)

  (DList _ rdBs') = iter (localLoopControl n) (allGatherIter t1 t2) dlBs
  t1 = map getMassPoint
  t2 = updateAll
```

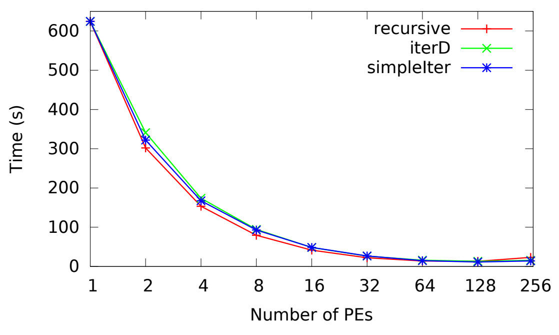
51 Note: here we use a slightly extended version of `DList` including `Places`

(3) *simpleLoopControl/simpleAllGatherIter* instantiates our `simpleIter` scheme with the skeletons `simpleLoopControl` for the iteration control and `simpleAllGatherIter` for the body. The remote data from body-processes output to body-processes input are repeadedly established (not the all-to-all connections, these are persistent).

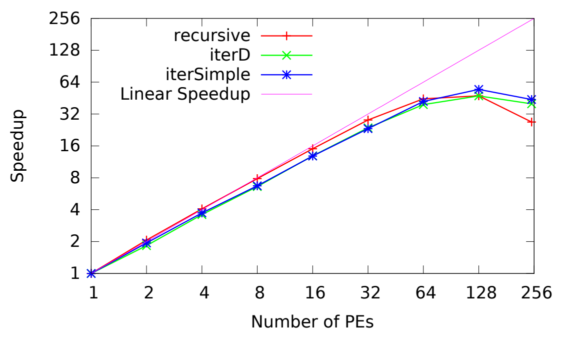
```
doStepsSimpleAllGatherIterStream :: Int → [Body] → [Body]
doStepsSimpleAllGatherIterStream n bs = shuffle ∘ fetchAll $ dBs' where
  dBs = releaseAll ∘ unshuffle noPe $ bs

  dBs' = iter (simpleLoopControl n) (simpleAllGatherIter t1 t2) dBs
  t1 = map getMassPoint
  t2 = updateAll
```

In the first setting, we ran the n-body simulation for 20 iterations with 20000 bodies. This constitutes a relatively high workload and large amounts of data have to be exchanged in every iteration. Runtimes and steedups are presented in Figure 5.9.

Surprisingly, the *recursive allGatherRD* version performs better than the iteration framework versions on up to 64 processors. Only on 128 and 248 processors, the iteration framework versions are faster than the recursive version. An analysis of runtime behaviours revealed that the *recursive allGatherRD* has no disadvantage in the communication steps



(a) runtime



(b) speedup

sequential runtime: 624.35 s

PEs	rec (1)	iterD (2)	iterSimple (3)	(1) vs (2)	(1) vs (3)	(2) vs (3)
2	302.11 s	340.94 s	322.03 s	11.39%	6.19%	-5.55%
4	153.43 s	173.73 s	167.47 s	11.68%	8.38%	-3.60%
8	79.47 s	94.55 s	92.87 s	15.95%	14.43%	-1.77%
16	41.48 s	48.42 s	48.59 s	14.33%	14.62%	0.34%
32	22.20 s	26.32 s	26.81 s	15.68%	17.20%	1.81%
64	14.02 s	15.89 s	14.85 s	11.75%	5.54%	-6.58%
128	13.13 s	13.15 s	11.42 s	0.14%	-13.03%	-13.15%
248	23.19 s	15.63 s	14.23 s	-32.61%	-38.63%	-8.93%

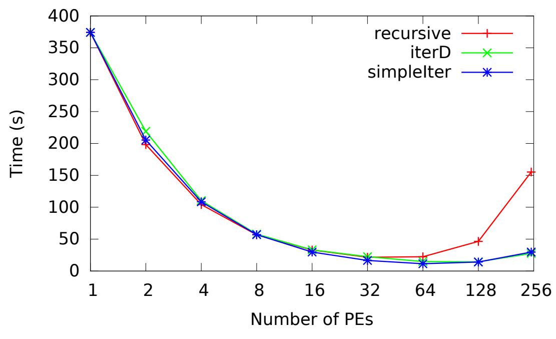
(c) Runtime table

Figure 5.9: N-body with 20000 bodies, 20 iterations and chunk size 30

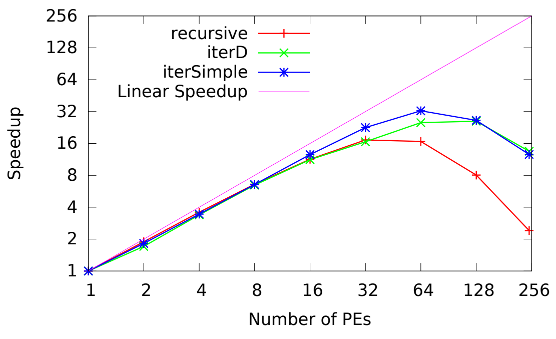
due to the optimised local communications, but the computation phases seem to be shorter, although sharing the same sequential code base with the iteration scheme version. Pending further investigation, we assume that the differences originate from the runtime system, maybe the garbage collection does not work as effectively for the data streams in our iteration scheme version. The `simpleIter` version performs mostly better than the `iterD` version, but both versions are close.

In the second setting, we reduced the workload and amount of data to be communicated for every iteration step. We used 5000 bodies but increased the number of iteration steps to 200. Runtimes and steedups are presented in Figure 5.10. This time, the *recursive allGatherRD* version performs slightly better than the iter framework versions up to 8 processors, but the latter versions are clearly faster than the recursive version from 16 processors on. Also the `simpleIter` version performs slightly better than the `iterD` variant, except for 248 processors.

In the third setting, we used only 1000 bodies but increased the number of iteration steps to 600, in order to measure the parallelism overhead. Runtimes and speedups are presented in Figure 5.11. Here, both iteration framework versions clearly outperform the *recursive allGatherRD* version, which has about half the speed of the other versions for 16 processors,



(a) runtime



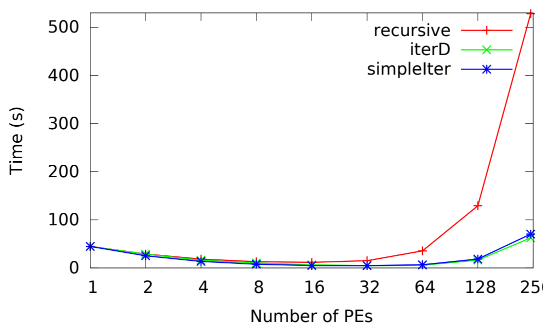
(b) speedup

sequential runtime: 374.17 s

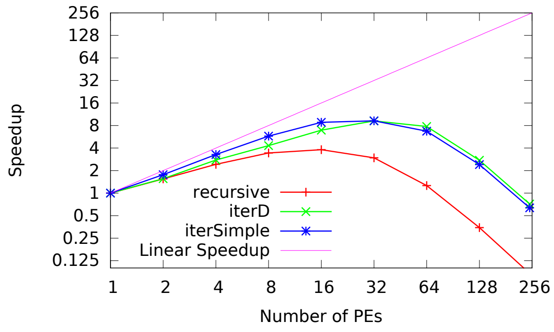
PEs	rec (1)	iterD (2)	iterSimple (3)	(1) vs (2)	(1) vs (3)	(2) vs (3)
2	198.31 s	219.02 s	205.36 s	9.46%	3.43%	-6.24%
4	104.29 s	110.40 s	108.55 s	5.54%	3.92%	-1.68%
8	56.73 s	57.84 s	57.04 s	1.92%	0.54%	-1.39%
16	32.92 s	33.21 s	29.80 s	0.87%	-9.48%	-10.27%
32	21.67 s	22.53 s	16.54 s	3.82%	-23.68%	-26.59%
64	22.37 s	14.87 s	11.49 s	-33.50%	-48.63%	-22.75%
128	46.51 s	14.40 s	14.14 s	-69.04%	-69.60%	-1.82%
248	155.22 s	27.70 s	29.69 s	-82.15%	-80.87%	6.71%

(c) Runtime table

Figure 5.10: N-body with 5000 bodies, 200 iterations and chunk size 30



(a) runtime



(b) speedup

sequential runtime: 44.75 s

PEs	rec (1)	iterD (2)	iterSimple (3)	(1) vs (2)	(1) vs (3)	(2) vs (3)
2	28.91 s	28.69 s	25.40 s	-0.76%	-12.13%	-11.46%
4	18.47 s	16.15 s	13.64 s	-12.54%	-26.13%	-15.54%
8	13.01 s	10.37 s	7.75 s	-20.27%	-40.42%	-25.27%
16	11.76 s	6.45 s	5.06 s	-45.12%	-56.92%	-21.51%
32	15.10 s	4.85 s	4.83 s	-67.88%	-68.00%	-0.37%
64	35.45 s	5.76 s	6.65 s	-83.74%	-81.24%	13.32%
128	129.08 s	16.27 s	18.58 s	-87.39%	-85.61%	12.39%
248	528.44 s	62.60 s	70.35 s	-88.15%	-86.69%	11.03%

(c) Runtime table

Figure 5.11: N-body with 1000 bodies, 600 iterations and chunk size 30

with increasing differences for increasing numbers of processors. The `simpleIter` version performs better than the `iterD` version up to 32 processors and worse afterwards. However, speedups are decreasing for all versions with more than 32 processors, the results in this area are not crucial for practical purposes.

The *recursive allGatherRD* version performs better than the iteration framework versions when the number of bodies per processor is relatively high, while the iteration framework versions perform better than the *recursive allGatherRD* version when the number of bodies per processor falls beneath a certain threshold – e.g. when we increase the number of processes for a fixed input size. There seems to be a trade off between the process and channel creation overhead for the recursive version (here the overhead for repeatedly creating the all-to-all topology increases quadraticly) and some other overhead for the iteration framework version (increasing with the number of bodies per process), which we have not yet identified.

5.3 Introducing State

The control function and the iteration body skeleton must adhere to constraints that cannot be expressed in its Haskell type: They have to process their argument stream incrementally and produce an incremental output stream. Even more severe a restriction, they have to produce *exactly one element* of the output stream when consuming one element of the (second) argument stream. In practice, there exist pairs of iteration control and iteration body that do not have these properties, but are still useful to compute iteration results. However, our intention is definitely lost in these cases. As often, the price of the increased flexibility is a less robust interface.

Introducing an explicitly stateful control function to the iteration interface would give the desired robustness, but the final iteration result could not be incremental. Therefore, we make this optional by providing a safe interface without jeopardizing the general character of `iter`.

5.3.1 Stateful Iteration Control

We distinguish between simple control functions from the `simpleIter` scheme where the iteration stream is not inlined and control functions with inlined iteration streams as presented in Chapter 5.1.5.

5.3.1.1 Simple Iteration Control

A suitable control function for the `simpleIter` scheme can be constructed from a stateful interface, as shown in Listing 5.2. The construction takes two stateful parameter functions

Listing 5.2: Stateful control interface

```
simpleControlS :: (a → State s (Either b d))
                  → (c → State s (Either b d))
                  → s          -- ^initial state
                  → b          -- ^dummy b needed to typecheck Iterated b
                  → c          -- ^dummy c needed to typecheck Iterated c
                  → a          -- ^initial Input
                  → (Iter c)   -- ^input from body
                  → (Iter b,d) -- ^output to body and final result
simpleControlS c1 c2 s _ _ a (Iter cs)
  = (Iter $ lefts bs, head $ rights ds) where
      (bs,ds) = break isRight $ evalState sCas s
      sCas = do init ← c1 a
                rest ← mapM c2 cs
                return (init:rest)
```

Listing 5.3: Loop control with transform function and simple interface

```
simpleLoopControlFS :: Int
                    → (Int → b → b)
                    → b           -- ^dummy b needed to typecheck Iterated b
                    → b           -- ^dummy c needed to typecheck Iterated c
                    → b           -- ^initial Input
                    → (Iter b)    -- ^input from body
                    → (Iter b,b)  -- ^output to body and final result
simpleLoopControlFS n cf _ _ x xs = simpleControlS' loopFC loopFC 0 x xs where
  loopFC x' = do i ← get
               if i<n then do put (i+1)
                              return $ Left $ cf i x'
                 else return $ Right x'
```

that define how to convert single control inputs (which are the initial input of type `a` or the bodie's outputs of type `c`) to either the bodie's input or the final result. Together with an initial state, this yields an internally stateful control function. This version includes the two dummy parameters needed to typecheck the function `iter`. For convenience, we define function `simpleControlS'`, a slight verison of `simpleControlS` without the dummy parameters:

```
simpleControlS' c1 c2 s = simpleControlS c1 c2 s undefined undefined
```

We can use `simpleControlS'` to define specialised control functions, e.g. `simpleLoopControlFS` of Listing 5.3 which triggers exactly `n` iteration steps and transforms each input using function parameter `cf` with the current iteration step.

5.3.1.2 Distributed Iteration Control

We will now focus on local iteration control skeletons as introduced in Chapter 5.1.4.1. An implementation of a stateful skeleton for distributed lists is given in Listing 5.4. When run, the control processes will connect both to their predecessor processes that produced the distributed list beforehand and to the processes of the body skeleton. The functionality of each process is expressed by function `simpleControlS'`, which is used in the respective control process to transform the initial input and the output of an iteration body's process (stream-wise) into the iteration body's input and the final result. Two stateful control functions `c1` and `c2` are parameters for `simpleControlS'` and specify the concrete local functionality, and likewise, a list of initial states is needed to run `simpleControlS'` with `c1` and `c2`. The functionality of the processes is defined by function `f`. They each take one of the initial states `sts`, one of the distributed input elements `as` and the loop-back output of the body `css`. The latter is of course initially not available, thus we supply the list lazily, such that function `zip3` can assemble the processes' input immediately.

Listing 5.4: Stateful process-local iteration control skeleton

```
localControlS ::
  forall a b c d s. (Trans a, Trans b, Trans c, Trans d, Trans s)
  ⇒ (a → State s (Either b d))
  → (c → State s (Either b d))
  → [s]                          -- ^initial states
  → DList b                      -- ^dummy b needed to typecheck Iterated b
  → DList c                      -- ^dummy c needed to typecheck Iterated c
  → DList a                      -- ^initial Input
  → Iterated (DList c)           -- ^input from loops
  → (Iterated (DList b),DList d) -- ^output to loops and final result
localControlS c1 c2 sts _ _ (DList places as) ~(DList _ css)
 = (DList places bss, DList places ds) where

    (bss,ds) = unzip $ parMapAt places (uncurry f) $ zip sts inp
    inp = zip as $ lazy css

    f :: s → (RD a, RD (Iter c)) → (RD (Iter b),RD d)
    f s (a, cs) = (release bs, release d) where
       (bs,d) = simpleControlS' c1 c2 s (fetch a) (fetch cs)
```

This skeleton can implement several common iteration control variants, e.g. a simple counter of fixed iterations (`localLoopControlS`, see Listing 5.5)

Listing 5.5: Control Skeleton **`localLoopControlS`**

```
localLoopControlS n = localControlS loopC loopC (repeat 0) where
  loopC :: a → State Int (Either a a)
  loopC inp = do i ← get
                 if (i < n) then do put (i+1)
                                    return $ Left inp
                   else return $ Right inp
```

Listing 5.6: Stateful map skeleton **`simpleParMapIterSAt`** with simple iteration type

```
simpleParMapIterSAt :: forall b c s. (Trans b, Trans c, Trans s)
                       ⇒ Places
                       → (b → State s c)
                       → [s]
                       → Iter [b]
                       → Iter [c]
simpleParMapIterSAt places f states xss = combineWith id yss where
  xss' = distribWith id xss                         :: [Iter b]
  yss  = parMapAt places (uncurry f') $ zip states xss' :: [Iter c]
  f' :: s → Iter b → Iter c
  f' s (Iter bs) = Iter $ evalState (mapM f bs) s
```

5.3.2 Stateful Body Skeletons

Listing 5.6 presents a stateful iterable `parMap` variant with simple interface. The skeleton uses a stateful[52] map-function and a list of initial states has to be provided, one for every process. In each process, the map function `f` is mapped on the iteration stream with the initial state, transforming the state in each iteration step. Function `evalState` is needed to run the stateful computation and retrieve the stream of outputs. The skeleton uses auxiliary functions `distribWith` and `combineWith` similar to `simpleParMapIter` to distribute and combine the iteration stream to and from the processes.

The `parMapIterS` skeleton works similar to the `simpleParMapIterS` skeleton, but additionally uses `liftRD` to establish the remote data connections while not using functions `distrib` and `combine`.

Listing 5.7: Stateful map skeleton **`parMapIterS`** with distributed iteration type

```
parMapIterS :: (Trans b, Trans c, Trans s)
               ⇒ (b → State s c)
               → [s]
               → Iterated (DList b)
               → Iterated (DList c)
parMapIterS f states (DList places xsRDs)
  = DList places $ parMapAt places (uncurry f') $ zip states xsRDs
  where f' s = liftRD (λ(Iter bs) → Iter $ evalState (mapM f bs) s)
```

52 using State from module Control.Monad.State.Lazy, a strict state would not work

5.4 Case Studies

5.4.1 FFT

We already implemented FFT with distributed homomorphism skeletons. The 2-dimensional distributed homomorphism skeletons of Chapter 4.1.4 and Chapter 4.3.4 are a special case of the general distributed implementation of a distributed homomorphism described by Gorlatch [Gor98].

5.4.1.1 Distributed Homomorphism - the General Case

Remember, a distributed homomorphism can be implemented in parallel as follows:

$$\overset{\sim(d)}{h} = \mathop{\circ}_{i=1}^{d} chdim^{(i,i+1)} \circ \mathop{\circ}_{i=1}^{d+1} map^{d} h \circ chdim^{(i,d+1)}$$

Here h is a distributed homomorphism function, the input list is of size 2^l. Function $\overset{\sim(d)}{h}$ ($d \in 0...l-1$) works on the list distributed into $d+1$ dimensions where d dimensions are distributed among the processes. $map^d h$ maps h to the inner list of the $d+1$-dimensional input. $chdim^{(i,j)}$ flips dimension i and j of the input and $\mathop{\circ}_{i=m}^{n}$ denotes function compositions for i from m to n[53].

A recursive transient general distributed homomorphism skeleton `dhSkelRec` can be constructed using remote data (see Listing 5.8). To keep it simple, we work with a list of lists, where each inner list contains the input for one process. We do not generalise the data type to a d dimensional structure, because this would involve type arithmetic if d shall be variable. We rather assume a virtual d dimensional space, with a fixed base length `base`

Listing 5.8: A simple recursive distributed homomorphism skeleton for the d-dim. case

```
dhNDFlatRec :: Trans a
              ⇒ Int --base
              → Int --n = d+1 dims, including local dim
              → ([a] → [a])
              → [[RD a]]
              → [[RD a]]
dhNDFlatRec base d h = chdims base d ∘ loop d where
  loop 0 xss = xss
  loop i xss = loop (i-1) (parMap h' ∘ chdim base d d i $ xss)
  h' = releaseAll ∘ h ∘ fetchAll
```

53 The order is the typing order, not the evaluation order: $\mathop{\circ}_{i=1}^{2}(+i) = (+1) \circ (+2)$

for all dimensions (the length of each inner list):

The mapping of each sublist to a single process implies that each data exchange between two processes due to `chdim` invocations involves only a single data element. The missing agglomeration leads to massive parallel overhead. Even though this is not a competitive implementation, we can use this version very well for runtime analysis. Each element of an inner list is lifted to remote data, such that each element of each process can be exchanged with an arbitrary process. The functionality is composed of `loop d` and `chdims`, where the latter implements $\overset{d}{\underset{i=1}{\circ}} chdim^{(i,i+1)}$. `loop d` calls `d` times `chdim` and `parMapAt` with the homomorphism `h`, lifted to remote data.

We need to implement function `chdim`, which swaps 2 dimensions of an n dimensional grid (`chdim base n i j`$=chdim^{(i,j)}$) and function `chdims`, which is defined as `chdims base n`$= \overset{n-1}{\underset{i=1}{\circ}} chdim^{(i,i+1)}$. We implement these via auxiliary function `chdimArr` (see Listing 5.9), which works on a flat array and reorders the elements efficiently using prelude function `ixmap`.

The input array is a flattened version of a $n = d + 1$ dimensional grid with base length `base`. Function `partnerF` calculates for each position of the flat array the position after the dimension swap in the n dimensional grid. We chose this low-level approach based on bit arithmetic because it is reliable, efficient and easy to implement. A high level transformational implementation where the nested lists are recursively recombined like for the `reduce` and `allReduce` skeletons of Chapter 4.2.1.3 and Chapter 4.2.1.4 seems very

Listing 5.9: Function `chdim` for arrays

```
chdimArr :: (Ix i, Integral i) ⇒
         i       --base length
         → i    --n-Dims
         → i    --ChDim1
         → i    --ChDim2
         → Array i a
         → Array i a
chdimArr base d cd1 cd2 arr
  | cd1 > 0 && cd1 ≤ d && cd2 > 0 && cd2 ≤ d
    = if cd1 == cd2 then arr else ixmap (bounds arr) partnerF arr
  | otherwise = error "ChDim ≤ 0 || > n-Dims"
  where pot1     = base^ (d-cd1)
        pot2     = base^ (d-cd2)
        partnerF ix | val1 == val2 = ix
                    | otherwise = ix - bigVal1 - bigVal2 + bigVal1' + bigVal2'
          where
            val1     = ix `div` pot1 `mod` base
            bigVal1  = val1 * pot1
            bigVal1' = val1 * pot2
            val2     = ix `div` pot2 `mod` base
            bigVal2  = val2 * pot2
            bigVal2' = val2 * pot1
```

difficult to implement, not least because of the (virtually) arbitrary number of dimensions of the input.

Apart from the bit arithmetics, the implementation of the distributed homomorphism skeleton is simple and straight forward. The drawback of this implementation is that it creates one channel for each element in each `parMap` phase. It is hard to optimise the number of channels by agglomerating n dimensional blocks, because of the variable number of dimensions. Anyway, it is not the purpose of this example to create a highly optimised version of the distributed homomorphism skeleton, rather we want to show how it can be done conceptually.

Further, we call the `parMap` skeleton in the loop function n times, which includes the overhead of process creation for every recursion step.

A version of skeleton `dhNDFlatRec` called `dhNDFlatIter` using persistent iterable skeleton `simpleParMapIter` instead of `parMap` is presented in Listing 5.10. It uses the stateful *simple* control function `simpleLoopControlFS` from Listing 5.3 with function `chdimCF` to transform the input, where:

```
chdimCF :: Int              -- n-Dims
           → Int            -- base length
           → Int            -- act-dim
           → [[a]]      -- n-dim Grid (flatened) in
           → [[a]]      -- n-dim Grid (flatened) out
chdimCF base d i = chdim base d (d-i) d
```

The *simple* versions of the iteration skeletons use a persistent process topology, but do not use persistent channel connections between the processes. This includes more overhead for the channel creation, but gives more flexibility allowing for different communication partners in every iteration step. This is exactly what we need, as the change of dimensions will always happen at different dimensions involving different communication partners.

Listing 5.10: A simple persistent distributed homomorphism skeleton for the d-dim. case

```
dhNDFlatIter :: (Trans a, CSimpleIter ([[RD a]]))
                ⇒ Int --base
                → Int --n = d+1 dims, including local dim
                → ([a] → [a])
                → [[RD a]]
                → [[RD a]]
dhNDFlatIter base d h
  = chdims base d
    ∘ iter (simpleLoopControlFS d $ chdimCF base d) (simpleParMapIter h')
    where h' = releaseAll ∘ h ∘ fetchAll
```

Listing 5.11: A transient (processes and channels); n-dimensional distributed homomorphism FFT implementation

```
fft3_NDRec :: Int   --base
           → Int   --n-Dims
           → [Complex Double]
fft3_NDRec base d = out ∘ dhNDFlatRec base d h $ inF [1..n] where
  h :: [(Complex Double, Int, Int)] → [(Complex Double, Int, Int)]
  h = fft3 base                                --the fft3 implementation
  n = base^d

  inF :: [Int] → [[RD (Complex Double, Int, Int)]]
  inF xs = chunk base $ releaseAll $ zip3 (bitReverse xs) [0,1..] [1,1..]
  bitReverse xs = inv n $ map fromIntegral xs

  out = map fst3 ∘ fetchAll ∘ concat
  fst3 (a,_,_) = a
```

Listing 5.12: A persistent processes; transient channels; n-dimensional distributed homomorphism FFT implementation

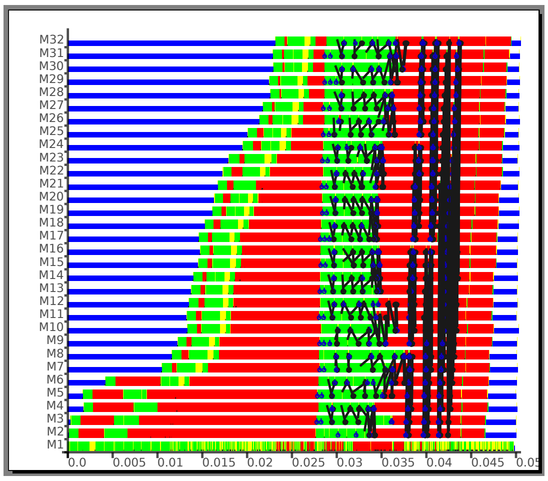
```
fft3_NDIter :: Int   --base
            → Int   --n-Dims
            → [Complex Double]
fft3_NDIter base d = out ∘ dhNDFlatIter base d h $ inF [1..n] where
  h = fft3 base ...
```

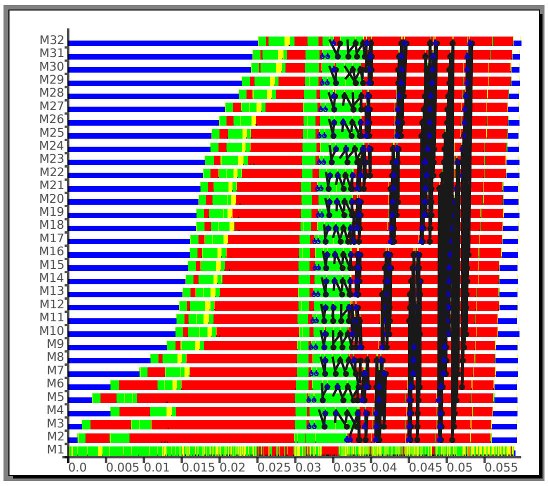
5.4.1.2 FFT Implementation

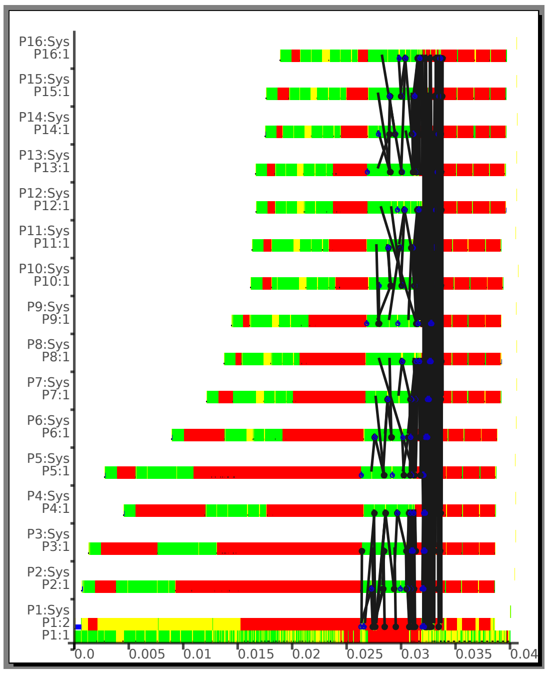
We compare two FFT implementations based on the two distributed homomorphism skeletons from the previous section. A version for the recursive skeleton with transient processes (see Listing 5.11) and a version using the iteration framework with persistent processes (see Listing 5.12). Both versions are identical apart from the different skeletons used. They differ from the two dimensional versions of Chapter 4.4.3 by the additional dimension parameter `d` and the additional use of `releaseAll` to prepare the skeleton input and `fetchAll` to gather the skeleton output.

5.4.1.3 Experimental Results

We conducted our experiments on a Beowulf cluster (see Chapter 2.2.1) and used different distributions for each input size. E.g. we can distribute input size 64 into 2^6 elements ($base = 2$, $d = 5$: 2 elements on each process and $2^5 = 32$ processes in a virtual hypercube) or into 4^3 elements ($base = 4$, $d = 2$: 4 elements on each process and $4^2 = 16$ processes in a virtual hypergrid). In general: Input distribution $base^n$ means that we have $d = n - 1$ dimensions in the hypergrid topology, where the n'th dimension with $base$ elements is process local. The runtime trace visualisations of Figure 5.12 presents parallel FFT


(a) iter framework, input distribution 2^6, 32PEs, machine view


(b) recursive, input distribution 2^6, 32PEs, machine view


(c) iter framework, input distribution 4^3, 16PEs, process per machine view

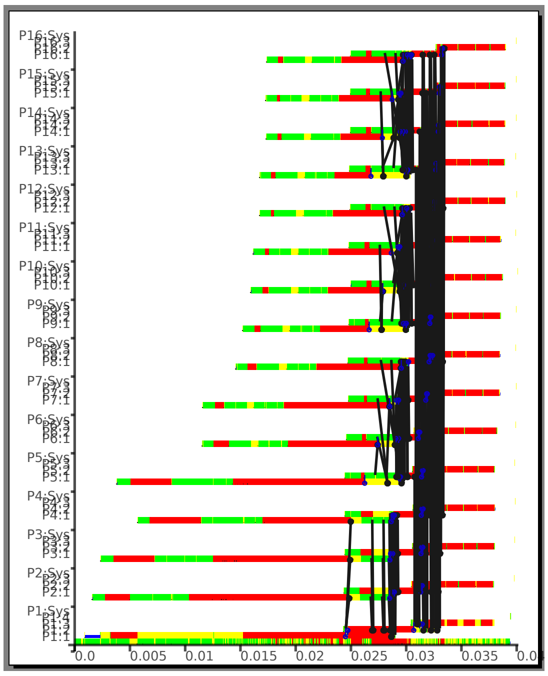
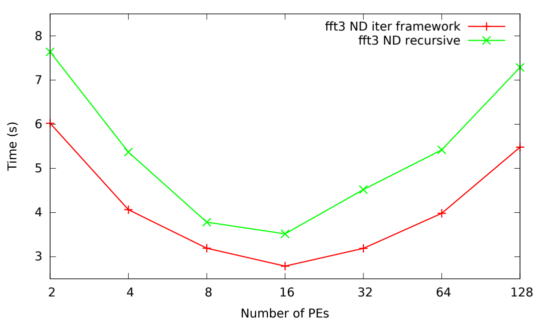
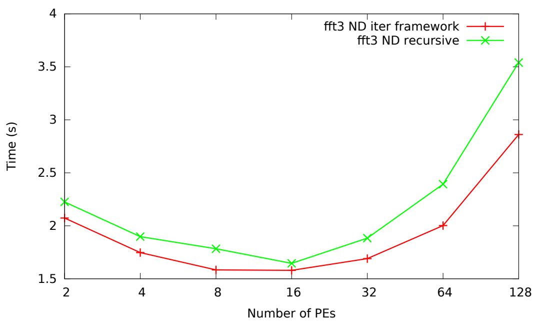

(d) recursive, input distribution 4^3, 16PEs, process per machine view

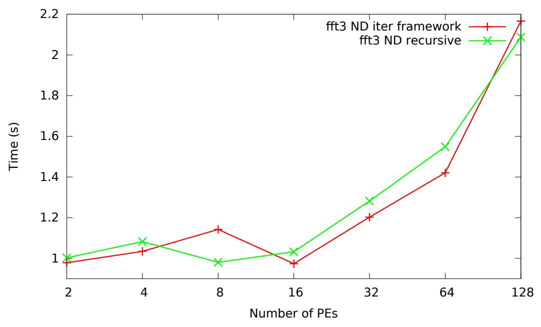
Figure 5.12: Traces of FFT with N dimensional distributed homomorphism skeleton for input size 64



(a) distribution 2^{12}



(b) distribution 4^{6}



(c) distribution 16^{3}

input	2^{12}			4^{6}			16^{3}		
PEs	iter	rec	overhead	iter	rec	overhead	iter	rec	overhead
2	6.02 s	7.64 s	21.17%	2.07 s	2.23 s	6.82%	0.98 s	1.00 s	2.40%
4	4.06 s	5.37 s	24.29%	1.75 s	1.90 s	7.92%	1.03 s	1.08 s	4.45%
8	3.19 s	3.78 s	15.60%	1.58 s	1.78 s	11.18%	1.14 s	0.98 s	-14.15%
16	2.79 s	3.52 s	20.78%	1.58 s	1.65 s	4.02%	0.97 s	1.03 s	5.68%
32	3.19 s	4.52 s	29.44%	1.69 s	1.88 s	10.26%	1.20 s	1.28 s	6.19%
64	3.98 s	5.42 s	26.51%	2.00 s	2.39 s	16.33%	1.42 s	1.55 s	8.26%
128	5.48 s	7.29 s	24.77%	2.86 s	3.54 s	19.18%	2.17 s	2.09 s	-3.64%

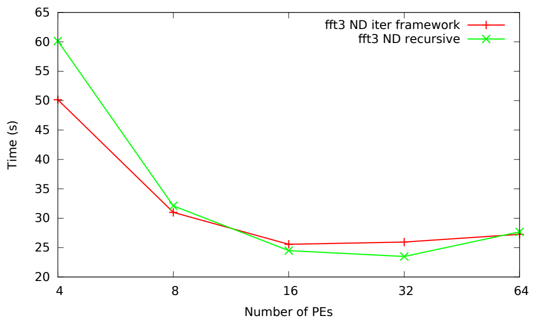
(d) Runtime table

Figure 5.13: Runtimes of FFT with N dimensional distributed homomorphism skeleton, input size 4096

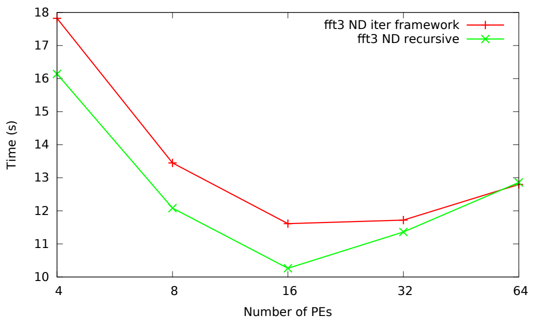
invocations for input size 64. We only show communication between the worker processes and hide input and output communication, which would overlap much of the traces. Figure 5.12a and Figure 5.12b show program invocations for the iteration framework and the recursive version with input distribution 2^6 on 32 PEs in the machine view. Here, the iteration framework version is slightly faster than the recursive version and communication phases are somewhat closer to each other, but the overall behaviour is similar. The message communication pattern of the hypercube is visible, where in the initial step, neighbouring elements are exchanged and subsequently the step width of communication increases. Figure 5.12c and Figure 5.12d show program invocations for data distribution 4^3 on 16 PEs in processes per machines view. Except using a single or three processes per machine, both versions are again similar. Here, the runtime of both versions is similar, but communication of the recursive version seems to be slightly more dense.

Figure 5.13 depicts runtime comparisons of both versions for input size 4096. We used varying input distributions: 2^{12} (Figure 5.13a), 4^6 (Figure 5.13b) and 16^3 (Figure 5.13c). Apart from the 16 PE and 128 PE measurements in the 16^3 setting, the iteration framework always performs clearly better. Runtimes are best with a number of processes around 8 to 16.

The second setup compares runtimes for input sizes 16384 for input distribution 2^{14} in



(a) distribution 2^{14}



(b) distribution 4^7

input	2^{14}			4^7		
PEs	iter	rec	overhead	iter	rec	overhead
4	50.16 s	60.14 s	16.60%	17.82 s	16.14 s	-9.44%
8	30.99 s	32.11 s	3.48%	13.45 s	12.08 s	-10.16%
16	25.56 s	24.48 s	-4.22%	11.61 s	10.26 s	-11.62%
32	25.94 s	23.49 s	-9.43%	11.72 s	11.36 s	-3.06%
64	27.27 s	27.68 s	1.51%	12.80 s	12.86 s	0.48%

(c) Runtime table

Figure 5.14: Runtimes of FFT with N dimensional distributed homomorphism skeleton, input size 16384

Figure 5.14a and 4^7 in Figure 5.14b. Here, the recursive version often performs better than the iteration framework implementation, but runtimes are mostly close.

In the last setting, we used 65536 input elements with input distributions 2^{16} in Figure 5.15a, 4^8 in Figure 5.15b, 16^4 in Figure 5.15a and 256^2 in Figure 5.15e. The results are inverse to the first setting. Here the recursive version performs clearly better in most cases and slightly worse in few cases.

5.4.2 Conjugate Gradient (CG)

This section is a revised versions of [DHLB16], Section 3.2.
Conjugate gradient is an efficient iterative algorithm for solving large linear systems whose matrix is symmetric and positive definite [Saa96]. 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.

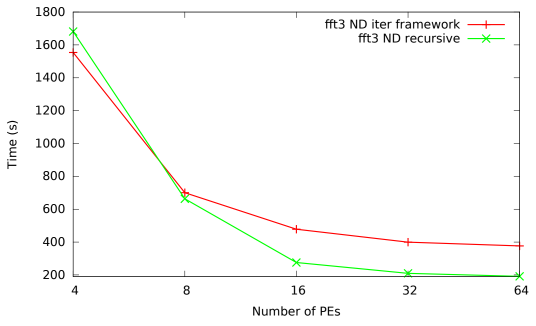
The Haskell code given in Figure 5.16 follows the presentation of [Bre98]. A data type `ISol` is used for representing an iterative solution comprising an iterate `x`, a residual `r`, a search direction `p` and an iteration counter. The `cg` code uses common basic matrix and vector operations (scalar 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.

5.4.2.1 Recursive Process Instantiations

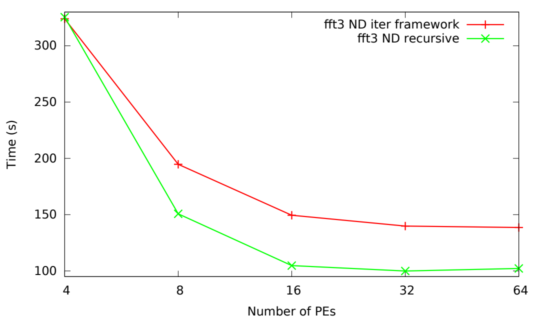
This iterative program is parallelised by decomposing the matrix-vector multiplication into as many tasks as PEs are available. The matrix is split into chunks of row vectors and the vector is broadcast to all worker processes, such that each worker process can compute a chunk of the result vector. The 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



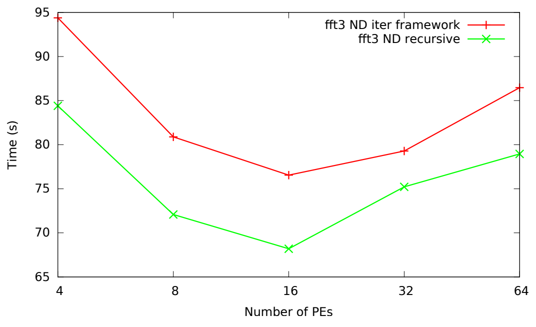
(a) distribution 2^{16}



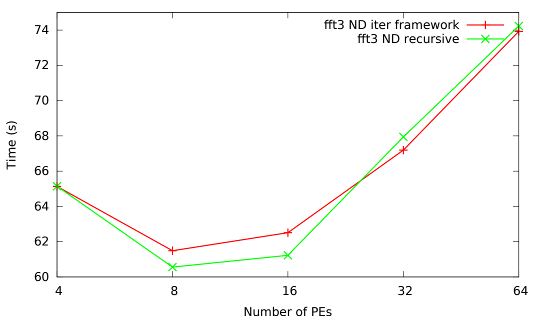
(b) distribution 4^8

input	2^{16}			4^8		
PEs	iter	rec	overhead	iter	rec	overhead
4	1554.11 s	1681.51 s	7.58%	324.01 s	325.48 s	0.45%
8	700.39 s	663.57 s	-5.26%	194.72 s	150.77 s	-22.57%
16	478.01 s	275.35 s	-42.40%	149.50 s	104.76 s	-29.92%
32	399.47 s	209.84 s	-47.47%	139.87 s	99.97 s	-28.52%
64	376.68 s	191.25 s	-49.23%	138.62 s	102.23 s	-26.25%

(c) Runtime table



(d) distribution 16^4



(e) distribution 256^2

input	16^4			256^2		
PEs	iter	rec	overhead	iter	rec	overhead
4	94.38 s	84.41 s	-10.57%	65.13 s	65.14 s	0.01%
8	80.89 s	72.08 s	-10.88%	61.48 s	60.55 s	-1.51%
16	76.55 s	68.20 s	-10.91%	62.51 s	61.23 s	-2.05%
32	79.29 s	75.22 s	-5.12%	67.20 s	67.94 s	1.10%
64	86.47 s	78.95 s	-8.70%	73.92 s	74.23 s	0.41%

(f) Runtime table

Figure 5.15: Runtimes of FFT with N dimensional distributed homomorphism skeleton, input size 65536

```
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
```

Figure 5.16: Haskell Implementation of Conjugate Gradient

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 stream 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 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 5.17.

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 processor elements (PEs) starting with

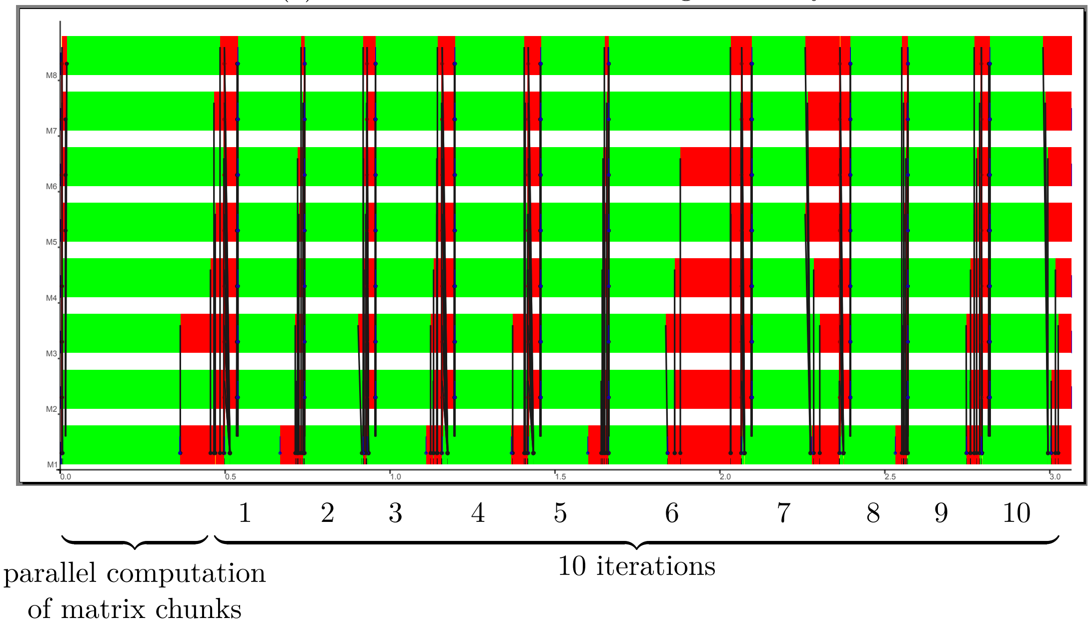
PE 2 (indicated by the first parameter `[2..]`).

Figure 5.18 shows three views of a trace of a program run for a matrix and vectors of size 20000 on 8 PEs. 10 iterations are performed. The machines view in Figure 5.18 (a) and the processes-per-machine view in Figure 5.18 (b) have messages overlayed. In Figure 5.18 (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 5.18 (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 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

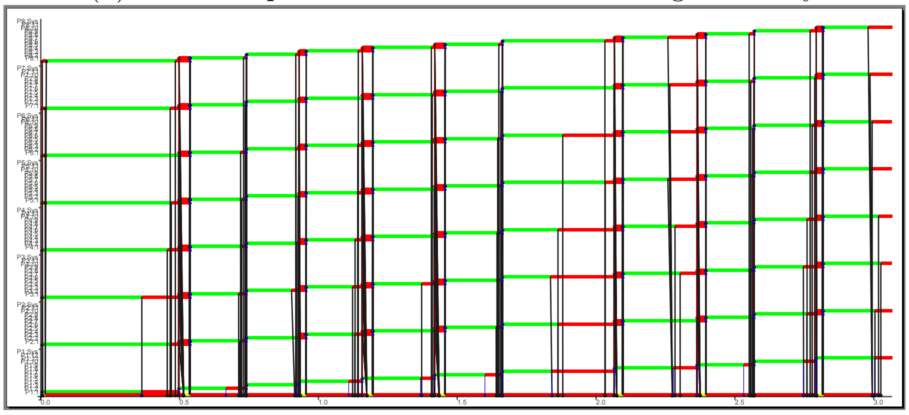
```
cgpar  :: Int → Matrix  → Vector →  ISol
cgpar np a v
  = snd $ until converge nextIter (aSplit, (initSol v))
  where
    converge :: ([RD (LBox Matrix)],  ISol) → Bool
    converge (_, (IterSol _ r _ _)) = dotProd r r < epsilon
      where epsilon = 0.01

    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 =∘ ..
```

Figure 5.17: Parallel Conjugate Gradient With Recursive Process Instantiations

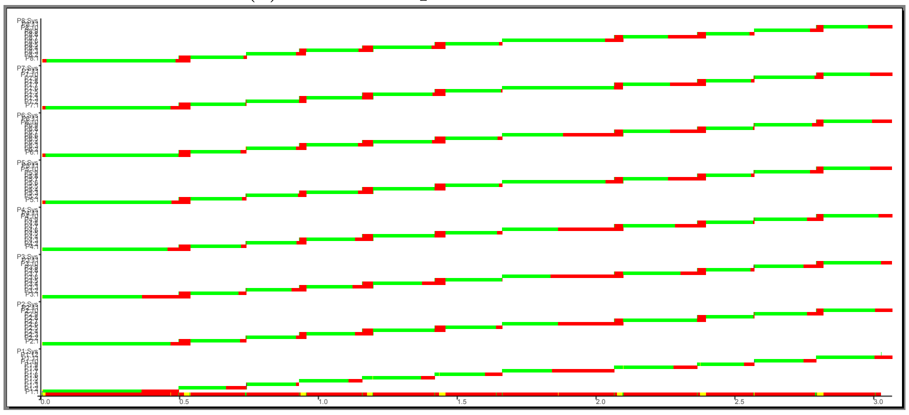
(a) Machines View With Messages Overlay



(b) Processes per Machines View With Messages Overlay



(c) Processes per Machines View



(d) Statistics

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

Figure 5.18: Activity Profile of Parallel CG Program with Recursive Process Instantiations

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 5.18 (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 5.18 (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 PEs heap, passing the matrix chunk reduces to a simple pointer exchange.

The statistics in Figure 5.18 (d) shows that 89 processes have been created, the main processes, 8 auxiliary processes to compute the matrix chunks and finally 10 times 8 processes for the ten parallel iteration steps on 8 PEs. 513 threads are created: the main thread, 8 system threads (1 per PE), 1 input and 2 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 $1 + 8 + 3 \cdot 8 + 6 \cdot 80 = 513$ threads.

5.4.2.2 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.

The `iterUntil` skeleton's interface is shown in Figure 5.19. 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

```
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
```

Figure 5.19: Interface of `iterUntil` iteration skeleton

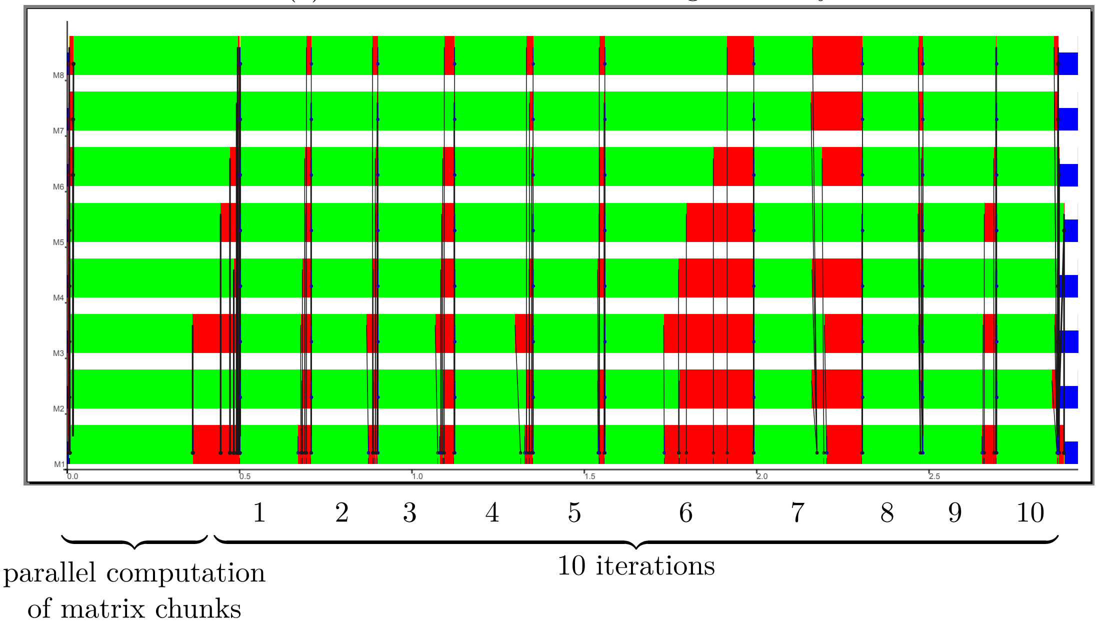
list of worker inputs and a master state.

The conjugate gradient algorithm can easily be parallelised using the `iterUntil` skeleton. Figure 5.20 shows the necessary definitions of the three function parameters. In each iteration, the current vector is `replicate`d 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

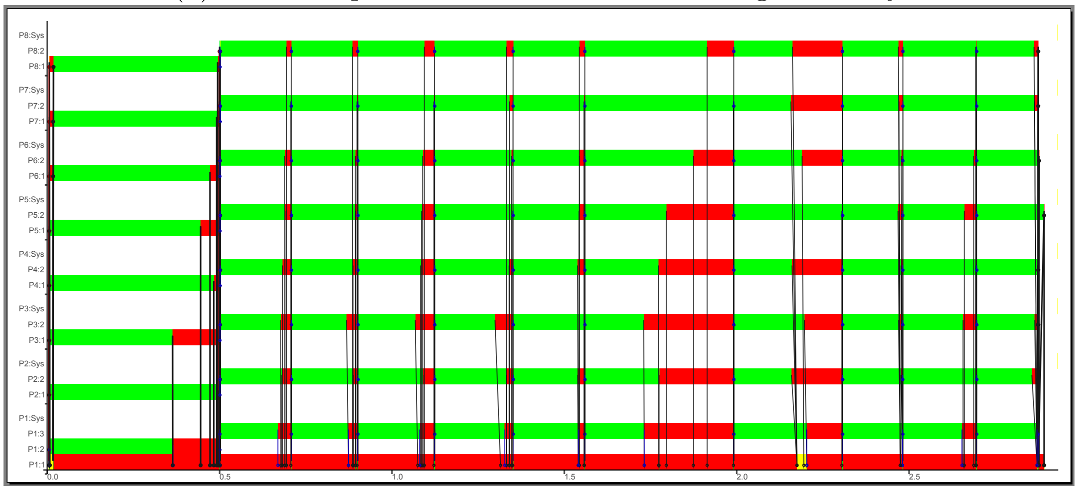
```
cg_par_skel  :: Int → Matrix → Vector →  ISol
cg_par_skel  np   mat  vec
    =  iterUntil transform workerfct combine (mat,vec)
    where
      transform :: (Matrix,Vector) → ([Matrix],[Vector],ISol)
      transform    (matrix,vector)
         =  (splitIntoN np matrix, replicate np vector,
     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

Figure 5.20: Parallel Implementation of Conjugate Gradient Using `iterUntil` skeleton

(a) Machines View With Messages Overlay



(b) Processes per Machines View With Messages Overlay



(c) Statistics

Input Parameters: Matrix Dimension 20000, 10 Iterations, Time for Iterations: 2,393s
Parallel Runtime: 2,932605s, 8 Machines, 17 Processes, 209 Threads, 233 Messages

Figure 5.21: Activity Profile of Parallel `iterUntil` CG Program

the matrix as parameter, as the matrix vector multiplication is computed in parallel by the worker processes.

Figure 5.21 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 5.18 for the compositional (recursive) program version. For this program version, the machines view in Figure 5.21 (a) is very similar to the machines view in Figure 5.18 (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 5.21 (b). The message streams overlay would cover most of the activity bars. As in Figure 5.18 there is an imbalance, by chance in the same iterations 6 and 7 as in Figure 5.18, but due to different PEs causing the delays. The machines views in Figures 5.21 and 5.18 (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-per-machine view, one detects the different approaches. The statistics in Figure 5.21 (c) show that only 17 processes have been created, namely the main process, 8 auxiliary processes and 8 `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.

5.4.2.3 Parallel Implementation using the Iteration Framework

We used also the iteration framework to implement the conjugate gradient algorithm using a stable process system. The implementation (see Listing 5.13) is similar to the `iterUntil` version, but we have to chose additionally the concrete type of the iteration. We use the stateful control function from Listing 5.2 and the iterable version of spawn. The latter allows us to map over the input iteration stream and to fetch an initial static input – the subset of the matrix – which is different for each process. This allows initially to receive an already distributed matrix, which will give this version a small advantage in our runtime experiments of the next Section. The control function is similar to the one from the `iterUntil` version, but uses the (lazy) `State` monad and a flipped encoding of the result in the `Eiter` type.

5.4.2.4 Experimental Evaluation

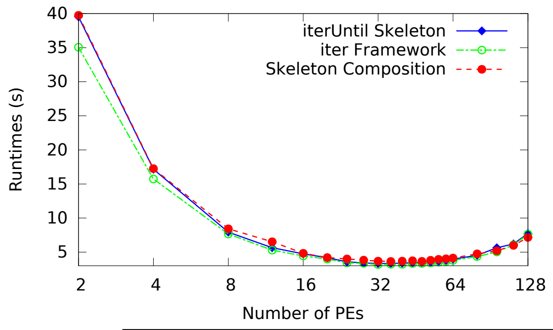
The advantage of the compositional parallel CG program is that it is very close to the original sequential program. Only local changes to the sequential program led to a parallel program which shows the same overall behaviour as the `iterUntil` instantiation when looking at the machine level only.

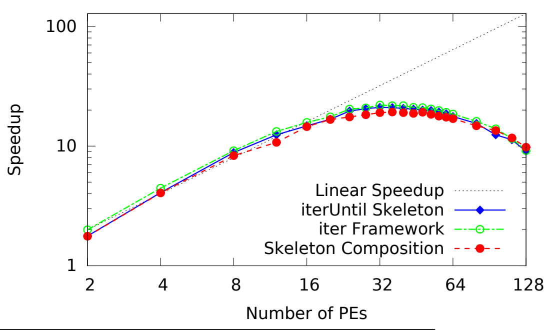
Listing 5.13: Parallel Implementation of Conjugate Gradient Using the iteration framework

```
cg_par_iter :: Int → Matrix → Vector → ISol
cg_par_iter np a v = iter (simpleControlS combine' combine' $ initSol v)
                          (simpleSpawnFIter $ map workerfct' aSplit) $
                          replicate np v where
  converge :: ISol → Bool
  converge (IterSol x r p k) = dotProd r r < epsilon where
      epsilon = 0.1

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

  combine' ::  [Vector] → State ISol (Either [Vector] ISol)
  combine' qs = do itersol@(IterSol x r p k) ← get
                   let itersol'@(IterSol _ _ p' _)
                         = nextIter (joinVectors qs) itersol
                   put itersol'
                   return $ if converge itersol' then Right itersol'
                              else Left $ replicate np p'

  workerfct' :: RD (LBox Matrix) → Iter Vector → Iter Vector
  workerfct' hwl ts = srs where
    (LBox wl) = fetch hwl
    f t = matVec wl t
    srs = fmap f ts

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

We compare program runs with input sizes 20,000 and 50,000 for all three versions on the Beowulf cluster (see Chapter 2.2.1). The results are shown in Figure 5.22. We measured runtimes from 2 to 128 PEs. We show relative speedups based on the doubled runtime for 2 PEs. The program scales well until 32 PEs with speedups up to 18. Speedups are better for the bigger input size. Runtimes of the three different implementations are close. Differences are within a range of 14 %, mostly beneath 10%. Here the iteration framework version performs slightly better than the monolithic `iterUntil` skeleton and the compositional version. Runtimes of the two lattes versions are mostly very close. With the modular approach of the iteration framework, we can match the problem more specifically then using the `iterUntil` skeleton and gain more performance, even though we use the same problem specific functions.

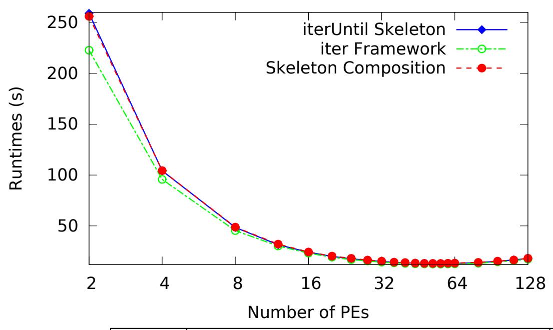
In order to measure only the core phase of the parallel computation, we compute the corresponding matrix chunks in parallel on all PEs and pass remote data handles to the processes of the `iterUntil` skeleton and the iteration framework, respectively. In the `iterUntil` case, we are forced to repeatedly fetch and release the local matrix chunks

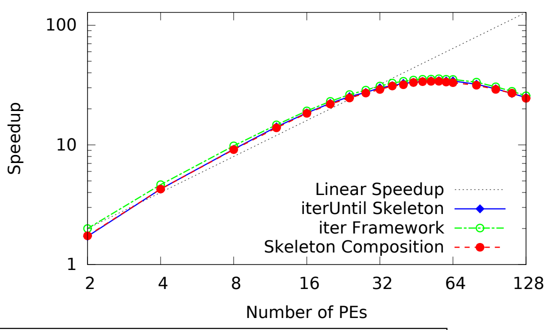




PEs	runtimes			overhead		
	mono	iter	rec	mono *vs* iter	mono *vs* rec	iter *vs* rec
2	39.54 s	35.06 s	39.75 s	-11.32%	0.52%	11.78%
4	17.17 s	15.74 s	17.26 s	-8.34%	0.53%	8.82%
8	7.94 s	7.66 s	8.43 s	-3.48%	5.77%	9.05%
16	4.75 s	4.44 s	4.83 s	-6.52%	1.50%	7.93%
32	3.33 s	3.18 s	3.68 s	-4.38%	9.60%	13.56%
64	3.97 s	3.78 s	4.14 s	-4.70%	4.13%	8.63%
128	7.74 s	7.64 s	7.16 s	-1.31%	-7.42%	-6.19%

Input Parameters: Matrix Dimension 20000, Number of Iterations 30





PEs	runtimes			overhead		
	mono	iter	rec	mono *vs* iter	mono *vs* rec	iter *vs* rec
2	259.12 s	222.84 s	256.14 s	-14.00%	-1.15%	13.00%
4	104.22 s	95.76 s	104.35 s	-8.12%	0.12%	8.23%
8	48.50 s	45.38 s	48.83 s	-6.44%	0.69%	7.08%
16	24.09 s	23.17 s	24.31 s	-3.82%	0.90%	4.69%
32	14.89 s	14.35 s	15.35 s	-3.60%	3.00%	6.50%
64	13.04 s	12.65 s	13.46 s	-2.95%	3.12%	5.97%
128	17.91 s	17.33 s	18.19 s	-3.22%	1.53%	4.70%

Input Parameters: Matrix Dimension 50000, Number of Iterations 30

Figure 5.22: Runtimes (in seconds) and Speedup of Parallel Conjugate Gradient Programs for Input Sizes 20000 and 50000 and 30 Iterations including Iteration Framework Measurements

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 runtime system, 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.

In this chapter we compared compositional versions with many composition steps to persistent process systems constructed by the monolithic `iterUntil` skeleton or the compositional iteration framework:

	K-means: iterUntil vs i-framework	n-body: rec vs simpleIter[54]	FFT(ND-DH)	CG: iterUntil vs Iter	CG: rec vs Iter
min	-4.61%	-80.87 %	-49.23 %	-14.00 %	-6.19 %
max	2.98 %	17.20 %	29.44 %	-1.31 %	13.56 %
mean	-1.15 %	-13.10 %	-1.37 %	-5.87 %	-7.41 %
abs mean	2.64 %	22.39 %	-13.02 %	5.87 %	8.3 %

For the K-means example we compared the `iterUntil` skeleton to the iter framework and a compositional version **without remote data**. The compositional version with the sub-optimal communication structure performed dramatically worse. We summarise here only the other two versions which perform competitive. The *mean* value is only -1.15% and the *abs mean* is only 2.64%.
For the n-body example the results depend a lot on the input size. The recursive version performed better for bigger input sizes and the iteration framework for smaller sizes. Thus in this overview we leave out the results for the small input size 1000 because we do not want to overemphasise small input sizes. Yet these numbers show how much the results differ. The *abs mean* value (22.39 %) is around 9% bigger than the absolute value of the *mean* value (-13.1%). The *mean* result can be influenced a lot by the chosen input sizes. If we would include bigger input sizes, then the sign of the *mean* value would switch from $-$ to $+$.
The FFT ND-distributed homomorphism example displays runtimes which also differ a lot. However, we should not attach much weight to this case study as it was not agglomerated. Runtimes are dominated by parallel overhead, there are no speedups but only slowdowns.
For the conjugate gradient example, the results are more homogeneous. The mean values are close to the abs mean values. The iter framework version performs best, but the mean differences with 5.87 % to the `iterUntil` skeleton version and with 7.41 % to the recursive version are still small.

54 without input size 1000

In our experiments the iteration framework performed competitive to the monolithic `iterUntil` skeleton. We observed varying performance gaps between the iteration framework and recursive skeleton compositions using remote data.

CHAPTER 6

Types and Type Class Support for Efficient Composition

By now, we treated composition techniques using remote data and its optimisations for iterated skeletons. In doing so, we detected some conceptual gaps which we seek to fill in this chapter.

Custom communication: We want to allow stream communication for user defined Trans instances, which is not possible with the original Eden module. An example is the streaming of the `Iter` type in Chapter 5.1, where we already had to modify the Eden module to enable this. Likewise, we want to allow to send not only tuples, but arbitrary product types concurrently. → *Chapter 6.1*

Interfaces: As yet, we used lists of remote data as parallel interfaces to compose skeletons. We want to generalise this to process e.g. trees or matrix shaped input data in parallel. Thus, we need a more general version of function `spawn` to allow eager process instantiation for various process topologies. → *Chapter 6.2.1*

Process Placement: We distinguish between skeleton versions with and without explicit placement. There are actually two versions of most skeletons in our library. We propose a type class based solution which makes this distinction superfluous for most skeletons and still allows simple skeleton interfaces without placement parameters, leading to a more concise skeleton library. → *Chapter 6.2.2*

Distributed Data: As a generalisation of type `DList` introduced in Chapter 5.1, we define type support for arbitrary distributed data types. → *Chapter 6.2.3*

Optimising Communication: Communication cost is crucial for the performance of parallel programs. We discuss boxes [DHLB16] as a way of changing the communication behaviour of arbitrary data. → *Chapter 6.3*

A major feature of Eden is its overloaded communication through type class Trans. This means that the communication behaviour of Eden programs and skeletons is usually implicitly determined by the types of the input data. On the one hand, this bears the risk of sub-optimal performance when used by an unaware user. However, this can be prevented by experienced programmers or with well-designed *end user* skeletons. On the other hand, the overloaded communication has important advantages e.g. when lifting skeletons to iterable skeletons, where the iterable ones can often be derived through mere specification of parameter functions of the original skeleton. We want to strengthen this unique characteristic of Eden – the data driven controlability of skeletons – by the modifications we present hereafter.

6.1 New Trans and Transmissible Class

The design of Eden's `Trans` class enables overloaded communication, but defining further `Trans` instances with concurrent or streaming behaviour is no simple task. In order to define concurrent sending behaviour for a custom type, one needs to encode the concrete concurrent sending behaviour explicitly in the Trans instance. This low-level task should not be left to an ordinary skeleton programmer. To a greater degree, the definition of further `Trans` instances with Streaming behaviour is impossible, because it is hard-coded in the runtime system that the received elements are combined with `(:)` constructors of Haskell's list type.

We therefore undertook a general redesign of the `Trans` class[55] to provide more flexibility in the definition of user defined `Trans` instances. The new `Trans` class provides a *transformation* function from the origin type to a representation which will actually be sent. The new `Transmissible` class defines the actual sending behaviour. Accordingly, the `Transmissible` class defines what before was done in the old `Trans` class from an implementational point of view, where instances of the new `Trans` class are used exactly how and where old `Trans` instances have been used by an Eden programmer.

6.1.1 The Transmissible Class

The Transmissible class has basically three instances and must not be extended by Eden users:

Mono a describes data that will be sent all at once, in a single message.

```
newtype Mono a = Mono {fromMono :: a}
```

(a,b,...): [56] Tuples will be sent concurrently, by one thread per tuple element.

55 This is joint work with my colleague Thomas Horstmeyer.
56 These are truly many instances –one for each arity– which work analogously.

Stream a contains a list of elements, which will be sent one by one in stream mode.

```
newtype Stream a = Stream {fromStream :: [a]}
```

```
class NFData a ⇒ Trans a where
    createComm :: IO (ChanName a, a)
    write :: a → IO ()

    createComm = do (cx,x) ← createC
                    return (Comm (sendVia cx) , x)

    write x = rdeepseq x `pseq` sendData Data x

sendVia :: Trans a ⇒ ChanName' a → a → IO()
sendVia c d = do connectToPort c
                 sendData Connect d
                 write d
```

(a) Old `Trans` class with default definitions

```
class NFData a ⇒ Transmissible a where
    createComm' :: PA (ChanName a, a)

instance NFData a ⇒ Transmissible (Mono a) where
    createComm' = PA $ do (cx,x) ← createC
                          return (Comm (sendVia cx), x)
      where
        sendVia :: NFData a ⇒ ChanName' a → a → PA()
        sendVia cx x = do link cx x
                          PA (rdeepseq x `pseq` sendData Data x) --"write x"

link :: ChanName' a → a → PA()
link c d = PA $ do connectToPort c
                   sendData Connect d
```

(b) New `Transmissible` class with `Mono` instance

Figure 6.1: Comparison of old `Trans` with new `Transmissible` class

In the following, we will compare the definition and instances of the old `Trans` class and the new `Transmissible` class, to show the differences in detail.

Comparing Class Definitions and Mono Instance

The old type class `Trans` (Figure 6.1 (a)) had two functions:

`createComm` returns a channel handle and the value to be received via the associated channel in the `IO` monad.

`write` sends a value via an already connected channel.

The problem with the old type class is that we use overloaded function `write` inside `createComm`, which might belong to another type instance. This is problematic e.g. when we use tuples of streams, where `write` of the stream instance is called by `createComm` of the tuple instance. We discuss this in detail at the end of this Section. Class Transmissible (Figure 6.1 (b)) has only one function: `createComm'` which works like `createComm`, but encodes also the functionality of `write`.

The default instance of old class `Trans` and the `Mono` instance of class `Transmissible` are presented in Figure 6.1. In function `createComm` and `createComm'` respectively, `createC` is used to create a primitive channel/value pair. The primitive channel is then wrapped inside the send function `sendVia`. Function `sendVia` works basically the same way in both cases. The changes are owed to the fact, that in the `Transmissible` class we can not overload `write` in the definition of `sendVia`, so the whole function `sendVia` must be a local definition to `createComm'`. The new `link` function corresponds to the part of the old `sendVia` version which is not overloaded and can thus be defined toplevel for reuse. The `write` functionality is now inlined because we only need it here.

```
instance (Trans a, Trans b) ⇒ Trans (a,b)
    where createComm = do (cx,x) ← createC
                          (cy,y) ← createC
                          return (Comm (write2 (cx,cy)),(x,y))
--write from default instance

write2 :: (Trans a, Trans b)
          ⇒ (ChanName' a, ChanName' b) → (a,b) → IO ()
write2 (c1,c2) (x1,x2) = do fork (sendVia c1 x1)
                            sendVia c2 x2
```

(a) Old `Trans` instance of `(a,b)`

```
instance (Transmissible a, Transmissible b) ⇒ Transmissible (a,b)
    where createComm' = do (cx, x) ← createComm'
                           (cy, y) ← createComm'
                           return (Comm (write2 (cx,cy)), (x, y))

write2 :: (Transmissible a, Transmissible b)
          ⇒ (ChanName a, ChanName b) → (a,b) → PA ()
write2 (Comm send1,Comm send2) (x1,x2) = do forkPA (send1 x1)
                                            send2 x2
```

(b) New `Transmissible` instance of `(a,b)`

Figure 6.2: Comparison of tuple instances (concurrent mode)

Comparing Instances for Concurrent Sending

The tuple instances of old class `Trans` and class `Transmissible` (see Figure 6.2) both create two channel/value pairs, one for each tuple component. In Figure 6.2 (a), primitive channels are created using `createC`. In Figure 6.2 (b) we use overloaded function `createComm'` on the inner tuple types. The new implementation will therefore **send nested tuples**

concurrently on all levels, where the original implementation only sends the first level concurrently.

The concurrency is created in function `write2`, which will be wrapped in the returned channel handle. It defines the send function by forking an extra thread for the send operation on the first channel. In the first case, it uses `sendVia` and thus overloaded function `write` on the primitive channels. In the second case, the overloaded send functions of the tuples inner types are used directly.

```
instance Trans a ⇒ Trans [a]  where
  --createComm from default instance
  write l = do mapM write' l
               sendData Data []
      where
        write' :: NFData a ⇒ a → IO ()
        write' x = rdeepseq x `pseq` sendData Stream x
```

(a) Old `Trans` instance of `[a]`

```
instance NFData a ⇒ Transmissible (Stream a) where
  createComm' = PA $ do (cx,x) ← createC
                        return (Comm (sendVia cx), Stream x)
       where
         sendVia :: NFData a ⇒ ChanName' [a] → Stream a → PA()
         sendVia cx (Stream l) = do link cx l
                                    mapM write l
                                    PA $ sendData Data []
         write :: NFData a ⇒ a → PA ()
         write x = PA $ rdeepseq x `pseq` sendData ParPrim.Stream x
```

(b) New `Transmissible` instance of `Stream a`

Figure 6.3: Comparison of streaming instances

Comparing Instances for Stream Sending

The list instance of the old `Trans` class and the `Stream` instance of class `Transmissible` are semantically identical (see Figure 6.3). The technical differences are, that in the latter case, `createComm'` is completely overloaded, where in the former case only write has to be redefined. The `link` part of `sendVia` of the latter version is also part of the former `sendVia` version, called by `createComm`. The only remaining differences are the different, but isomorphic types. In the second version, we use pattern matching to get rid of the `Stream` constructor in the second parameter of `sendVia` and we wrap the resulting list with the `Stream` constructor in the second result value of `createComm'`.

Note: Something similar would not work inside the old `Trans` class. Assume we want to define an old `Trans` instance for a given type `a` and actually send a transformed type `b`. We would need a transformation inside `createComm` and the inverse transformation for function `write`. Assume further that we want to send a tuple `(a,b)`. The `createComm` instance of the tuple would call `createC` and not `createComm` for the inner type `a` to establish a channel connection. Thus, `write` would transform `a` before sending, but `createC` would not do the inverse transformation after receiving the result. Hence, the recursive use of `createComm` in the tuple instance is necessary to send a type different to the one indicated by the `Transmissible` instance.

6.1.2 The Trans Class

The Trans class describes data which is *transformable* to `Transmissible` data (see Figure 6.4). Instances can be defined for arbitrary types. Each instance declaration contains an *associated type* `ComType a`, which has to be one of the just defined `Transmissible` types. A `transform` function defines the transformation to the representation which will actually be sent. The `retransform` definition gives the inverse function, which will be applied after a value is received.

```
class Transmissible (ComType a) ⇒ Trans a where
  type ComType a :: *
  transform :: a → (ComType a)
  retransform :: (ComType a) → a
```

Figure 6.4: Class Trans

With the functions of class `Trans` and `Transmissible` at hand, we define function `createComm`, giving us a channel handle and the value to be received via the associated channel.

```
createComm :: Trans a ⇒ PA (ChanName a, a)
createComm = do (Comm sendVal, out) ← createComm'
                return (Comm (λinn → sendVal (transform inn)),retransform out)
```

It uses overloaded function `createComm'` to create the channel handle and the placeholder for the received value. The returned (channel, value) pair is then modified by using `transform` and `retransform` before and after sending.

`createComm` itself is not overloaded. We use `createComm'`, `transform` or `retransform` only to define `createComm`, which is used to define core Eden functions like `process`, `instantiate` or `new`.

Dual to the PA monadic version of `createComm`, we define a `PA-monadic` send operation. Following Erlang [Arm07] syntax, we use the `!` operator for sending a value over a channel:

```
(!) :: Trans a ⇒ ChanName a  -- ^ @ChanName@ to connect with
                → a           -- ^ Data that will be send
                → PA ()
```

```
(!) (Comm c) a =  forkPA $ c a where
```

Trans instances

To define custom `Trans` instances is now very simple. One does not need to tackle the challenge to define a custom sending behaviour with a new `Transmissible` instance, but has only to define a transformation to a type with predefined sending behaviour. Typical `Mono` transformations would be like:

```
instance Trans Int where
  type ComType Int = Mono Int
  transform = Mono
  retransform = fromMono
```

Or more general as CPP pattern:

```
#define instanceTransSimple(myType)    \
instance Trans myType where{           \
  type ComType myType = Mono myType;   \
  transform = Mono;                    \
  retransform = fromMono}
```

We chose to stream the new type `Stream`.

```
instance (Trans a) ⇒ Trans (Stream a) where
  type ComType (Stream a) = Stream (ComType a)
  transform = fmap transform
  retransform = fmap retransform
```

but stop streaming lists, because we want to have streaming behaviour only if it is explicitly indicated by the user.

```
instance Trans a ⇒ Trans [a] where
  type ComType [a]  = Mono [ComType a]
  transform = Mono ∘ map transform
  retransform = map retransform ∘ fromMono
```

In both cases we use `transform` recursively on the inner elements. This is because we can use `transform` not only to change sending behaviour but also e.g. to automatically compress or encrypt data before sending it.

Streaming nested streams

In Eden, lists are only streamed top-level or inside of tuples. Inner lists of lists or lists inside other data types are sent in a single message. When confronted with iteration skeletons, we had to find a solution to stream inner lists, as for iterations over lists, the streaming behaviour should be preserved in the iteration steps.

We want to define the `Trans` instance of `Iter` similar to the instance for `Stream`, with the difference that streams inside of `Iter` shall also be streamed. The first problem is to define different instances for `Trans (Iter a)` and `Trans (Iter (Stream a))`, such that they do not overlap.

Our solution is based on the definition of the `Show` class, where similar, it is desired to have a different output method for `[a]` and `[Char]`[57]. The trick in the `Show` class definition for type `a` is, that there are functions `showsPrec :: Int → a → String → String` and `showList :: [a] → String → String`. One can define a `Show` instance by defining only function `showPrec`. The `Show` instance for `[a]` is defined as `showsPrec _ = showList`, which is overloaded by it's inner type `a`. The default instance for `showList` gives the comma separated list representation, where the `Char` instance for `showList` returns the double quotes style.

For the `Trans` class, we avoid the overlap by defining functions

```
transformStream, retransformStream :: Stream a → Stream a
```

in class `Transmissible` for type `a` with default implementation `id`, which we had hidden so far for the sake of simplicity:

```
class NFData a ⇒ Transmissible a where
    createComm' :: PA (ChanName a, a)
    transformStream :: Stream a → Stream a
    retransformStream :: Stream a → Stream a

    transformStream = id
    retransformStream = id
```

We use these functions to define the `Trans` instance for `Iter`:

```
instance (Trans a) ⇒ Trans (Iter a) where
  type ComType (Iter a) = Stream (ComType a)
  transform = transformStream ∘ Stream ∘ fromIter ∘ fmap transform
  retransform = fmap retransform ∘ Iter ∘ fromStream ∘  retransformStream
```

In the `transform` case, we use the `transform` function recursively on the elements of the `Iter` stream. Then we transform `Iter` into a `Stream`. The nested streaming behaviour should apply for those types `Iter a`, where `ComType a ~ Stream a`. This is realised through applying overloaded function `transformStream`, which in the case of nested streams flattens the inner stream. Accordingly in the `retransform` step, we first use function `retransformStream` to reconstruct the originally nested `Stream` from the flattened version. Then we convert the `Stream` to `Iter` and use `retransform` recursively on the

57 `show[1..3]` $\Rightarrow^*$ `[1,2,3]`, but `show['a'..'c']` $\Rightarrow^*$ `"abc"`,

elements.

In the default case, `transformStream` and `retransformStream` do not do anything (`id`). We overload this behaviour only in the `Transmissible` instance of `Stream`:

```
instance NFData a ⇒ Transmissible (Stream a) where
  -- createComm' as before
  transformStream = liftStream $ concatMap f where
    f (Stream xs) = map (λx→Stream[x]) xs ++ [Stream []])

  retransformStream = Stream ∘ unfoldr break' ∘ fromStream
      where break' [] = Nothing
            break' xs = let (xs1,xs2) = (break (null ∘ fromStream)) xs
                        in Just (Stream $ concatMap fromStream xs1,tail xs2)

liftStream :: ([a]→[b]) → (Stream a → Stream b)
```

For the instance for `Stream a`, we get types:

`transformStream, retransformStream :: Stream (Stream a) → Stream (Stream a).`

So when using `transformStream` on a nested Stream[58] the flattening takes place as follows: We encode each element of an inner list as a singleton `Stream` in the resulting top-level `Stream` and encode each end of an inner `Stream` as `Stream []`.

Example:

```
      transformStream (Stream[Stream[1,2,3],Stream[4,5,6]])
   ⇒ * Stream [Stream [1],Stream [2],Stream [3],Stream [],
         Stream [4],Stream [5],Stream [6],Stream []]
```

◁

Function `retransformStream` does the inverse transformation. Fortunately there is no type conflict, these transformations have the same type as `id`.

Here an example for the whole function `transform`, applied to a nested iteration of streams, `transform (Iter[Iter[Stream[1,2],Stream[3]],Iter[Stream[4,5]]])`, where in the following `Stream` was replaced by `[]` for simplicity.

Example:

```
     transform $ Iter[Iter[Stream[1,2],Stream[3]],Iter[Stream[4,5]]]
⇒*   transformStream ∘ Stream ∘ fromIter ∘
       fmap transform $ Iter[Iter[[1,2],[3]],Iter[[4,5]]]
⇒*   transformStream ∘ Stream ∘ fromIter $ Iter[[[1],[2],[],[3],[]][[4],[5],[]]]
⇒*   transformStream [[[1],[2],[],[3],[]][[4],[5],[]]]
⇒*   [[[1]],[[2]],[[]],[[3]],[[]],[],[[4]],[[5]],[[]],[]]
```

58 after the recursive `transform` step, which may have transformed inner types to `Stream` beforehand, thus may have flattened nested inner streams already.

◁

Of course, functions `transformStream` and `retransformStream` can be used equally in other `Trans` instances where nested streaming is required.

Example: **Automatic chunking**

Typically we use chunking of input and output lists to control the granularity of input and output streams. Usually we hide these details from the presented code because frequent calls to `chunk` and `unchunk` divert the attention from the relevant details. In this example we show how to move the applications of `chunk` and `unchunk` for a special chunk list type to the chunk list instance of the new `Trans` class.
First, we define a *chunked list* type which carries a flat list `cList` and the desired chunk size `cSize`:

```
data CList a = CL {cSize :: Int, cList :: [a]}
```

Then, we add common typeclasses to make the data type usable, here e.g. a `Functor` instance:

```
instance Functor CList where
  fmap f cl = cl{cList = map f (cList cl)}
```

With the subsequent modifications to the type class `Trans`, we can add an adequate `Trans` instance for `CList`s which defines the `chunk` and `unchunk` behaviour. We do this by defining `transform` and `retransform` functions, which are implicitly applied before and after sending and an associated `ComType`. This determines the type of the transformation result, which is actually the type of the data that will be transmitted:

```
instance NFData a ⇒ Trans (CList a) where
  type ComType (CList a)        = (Mono Int, Stream [a])

  transform (CL cSize as)       = (Mono cSize, chunk cSize as)
  retransform (Mono cSize, ass) = CL cSize (unchunk ass)
```

Here type `Mono a` means that a value of type `a` will be send in a single chunk and type `Stream a` determines a stream of type `a`. Function `transform` returns a tuple with the chunk size and a stream of chunks (sub-lists), such that `retransform` has all the necessary information to reassemble the original chunked list `CList` on the receiver side.

Here a simple example taken from a Mandelbrot program, where the rows of the grid are distributed in chunks among the processes[59]:

59 Note: In this example we use an explicit type `Stream` for streams. Thus we need two versions for the auxiliary functions: one working with streams and one working with lists. Auxiliary functions using the list interface are imported here "qualified" as L

```
skel :: [[Complex Double]] → [String]
skel = L.shuffle ∘ map unchunk
       ∘ parMap (chunk cs ∘ fmap wf ∘ unchunk)
       ∘ map (chunk cs) ∘ L.unshuffle np
```

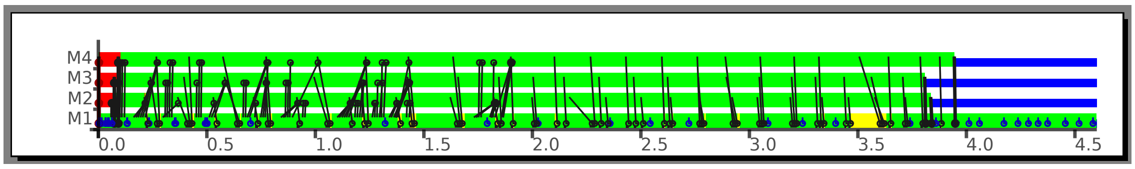
Functions `chunk` and/or `unchunk` are applied at the caller after distributing the input into sub-lists, before and after applying the function `wf` in the worker processes and again in the caller before recombining the results.

Here the same program segment with the new type class:

```
skel :: [[Complex Double]] → [String]
skel = L.shuffle ∘ map cList
       ∘ parMap (fmap wf)
       ∘ map (CL cs) ∘ L.unshuffle np
```

We have to construct/destruct the chunked lists before and after calling the `parMap` skeleton. This is similar to the calls to `chunk` and `unchunk` in the previous version. A simple call to `fmap wf` suffices on the worker process.

Observing the following trace visualisation for a program run with grid resolution 1000 × 1000 and chunk size 10 we conclude by the number of messages that the input and output is actually chunked:



Sometimes we can not implement the skeleton merely by using overloaded functions with the chunk list type. Then it is often the easiest way to unwrap and wrap the data type in the same way we would have applied `chunk` and `unchunk`. This may put the usefulness of the whole approach into question. There is another advantage when using the chunk list data type: The type signatures are closer to the original problem, as the types are not artificially nested. We added an example for this to the Appendix (Chapter A.4).
Moreover, when defining custom data types for other reasons than defining specialised `Trans` classes, there is no drawback in including custom communication functionality in the `Trans` instance. ◁

6.2 Parallel Input

We usually use lists for parallel in- and output to our skeletons. For most parallel programs a simple list interface is sufficient. Therefore we never took an effort to generalise this approach. Our tool chain is limited to a list interface, as is the case with auxiliary functions `spawn`, `fetchAll`, `releaseAll` and so on.

A key role comes to function `spawn`, which is used in all our skeletons. Generalising it allows to use other data types with different intrinsic topological structure like trees or matrices. Moreover we can use different data types with list like linear structure like arrays, vectors or queues.

6.2.1 Generalising `spawn`

Eden's process instantiation function

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

instantiates a list of processes, one for each function in the first argument list and the corresponding input in the second argument list. The list based interface makes it necessary to transform any other input data structures to lists before process `spawn`ing. We want to overcome this limited interface and develop a more general version of `spawn`, which can easily be used in skeletons for arbitrary data and in particular for distributed data.

First, we want to analyse the implementation of `spawnF`. A naive definition of `spawnF` would be:

```
spawnF0 = zipWith ($#)
```

This definition does not spawn the processes eagerly because Haskell's lazy evaluation demands the *whnf* of a result before instantiating the next process with the next input. We use instead function `instantiateF :: (a → b) → a → PA b` to define an eager version of `spawnF`:

```
spawnF = runPA $ zipWithM instantiateF
```

The sequence of `instantiateF` calls does not block on intermediate results.

Function `spawnFAt` works similar, but takes process placement into account. It is thus based on `instantiateFAt :: Int → (a → b) → a → PA b`.

```
spawnFAt :: (Trans a, Trans b) ⇒ Places → [a → b] → [a] → [b]
spawnFAt []  fs is = spawnFAt [0] ps is
spawnFAt pos fs is
    = runPA $ sequence [instantiateFAt st p i |
                        (st,p,i) ← zip3 (cycle pos) ps is]
type Places = [Int]
```

Here, the arguments for `instantiateFAt` are paired using `zip3` inside the list abstraction. The list of `PA` actions is then executed by `sequence`.

We observe that the `spawnF` versions depend fundamentally on different versions of the function `zip`. Thus we could express a more general version of `spawnF` through a more general version of `zip`.

In the Haskell wiki article *"Foldable and Traversable"* [Aut07], there is a chapter *"generalising zipWith"*, which describes exactly what we need for `Traversable` and `Foldable` `zip` parameters:

```
zipWithTF :: (Traversable t,Foldable f) ⇒ (a → b → c) → t a → f b → t c
zipWithTFM :: (Traversable t,Foldable f,Monad m) ⇒
              (a → b → m c) → t a → f b → m (t c)
zipTF :: (Traversable t, Foldable f) ⇒ t a → f b → t (a,b)
```

The `zip` versions take a `Traversable` and a `Foldable` input. The `Traversable` input is a structural blueprint for the output. The `Foldable` input is grouped element wise with the `Traversable` input by the order given by the `toList` function of the `Foldable` class. When the two input containers are identical, and the input is structured the same way, then the elements at the same position are associated by the `zip..TF` functions. This liberal interface is convenient, since we can group every `Traversable` input structure e.g. with a simple list, which is quite easy to define.

We define an `unzipTF` function using `fmap fst` and `fmap snd`:

```
unzipTF :: Functor f ⇒ f (a,b) → (f a, f b)
unzipTF xs = (fmap fst xs, fmap snd xs)
```

This solution is sub-optimal, as the structure will be traversed twice. However, we do not know a better implementation.

We extended the definitions for `zip..TF`s with three and four parameters. With these tools at hand we can define a version of `spawn` for arbitrary `Traversable` input and `Foldable` function containers:

```
spawnF :: (Traversable t, Foldable f, Trans a, Trans b)
  ⇒ f (a → b) → t a → t b
spawnF fs xs = runPA (zipWithTFM (flip instantiateF) xs fs)
```

The functions of the `Foldable` function container are applied remotely to the inputs of the `Traversable` input container in the order which `toList` on the function container and `traverse` on the input container define. The input data defines exactly the structure of the output data. E.g. instantiating a list of functions with a tree of inputs will generate a tree of outputs.

Example: **Parallel vector input:**

We already mentioned that the use of unboxed vectors allows for optimised communication and improved sequential computation performance. For this reasons they are used more and more commonly. Thus we should allow not only for vectors as processes' input and output, but as well in parallel interfaces. Of course we can not create unboxed vectors of unboxed vectors. But we can support boxed vectors in the parallel interface.
We use the FFT example implemented with the n-dimensional distributed homomorphism skeleton. Here, we used arrays and function `ixmap` internally to efficiently calculate the `chdim` function for (flattened) n-dimensional input. We can as well use boxed vectors and function `backpermute`[60] for this task. We use the vectors as parallel interface as well for the processes' input and do the computations on vector level. Therefore we define the necessary instances and a `VMatrix` type for a shorter notation[61]:

```
instance NFData a ⇒ Trans (B.Vector a) where
  type ComType (B.Vector a) = Mono (B.Vector a)
  transform = Mono
  retransform = fromMono

type instance Iterated (B.Vector a) = Iter (B.Vector a)

type VMatrix a = B.Vector (B.Vector a)
```

The `Trans` instance defines to send vectors in a single chunk. The `Iterated` type family instance allows for using the simple-iter scheme with vectors. The predefined `Traversable` instance (not shown) allows for applying `spawnF` to a `Vector`. With this at hand we define a vector based FFT implementation[62]:

```
fft3_NDIterV :: Int   --base
             → Int   --n-Dims
             → B.Vector (Complex Double)
fft3_NDIterV base d = out ∘ dhNDFlatIterV base d h
                        ∘ inF $ B.generate n (+1) where ...
```

We `generate` the input vector with elements from 1 to n, apply the input transformation function `inF`, then the distributed homomorphism skeleton `dhNDFlatIterV` for vectors and finally the output transformation. The `dhNDFlatIterV` skeleton is implemented with the iteration framework from a generalised version of Edens skeleton library which works with arbitrary `Spawnable` input:

60 `backpermute` is similar to `ixmap`, but instead of an index function it takes a vector with the permutation indexes
61 We use a qualified import of boxed vectors as `B`
62 The whole code including auxiliary functions can be found in the appendix (Chapter A.5).

```
dhNDFlatIterV :: (Trans a, CSimpleIter (VMatrix (RD a)))
                 ⇒ Int --base
                 → Int --n = d+1 dims, including local dim
                 → (B.Vector a → B.Vector a)
                 → VMatrix (RD a)
                 → VMatrix (RD a)
dhNDFlatIterV base d h
  = chdimsV base d
    ∘ iter (simpleLoopControlFS d $ chdimVCF base d) (simpleParMapIter h')
    where h' = releaseAll ∘ h ∘ fetchAll
```

Finally we adjust the `chDim` function for vectors. Here the internal version working on a flat `Vector`[63]:

```
chdimVFlat :: Int   --base length
           → Int   --n-Dims
           → Int   --ChDim1
           → Int   --ChDim2
           → B.Vector a -- n-dim Grid (flatened) in
           → B.Vector a -- n-dim Grid (flatened) out
chdimVFlat base d cd1 cd2 v
  | cd1 > 0 && cd1 ≤ d && cd2 > 0 && cd2 ≤ d
     = if cd1 == cd2 then v
       else B.unsafeBackpermute v (B.generate (B.length v) partnerF)
  | otherwise = error "ChDim␣≤␣0␣||␣>␣n-Dims" ...
```

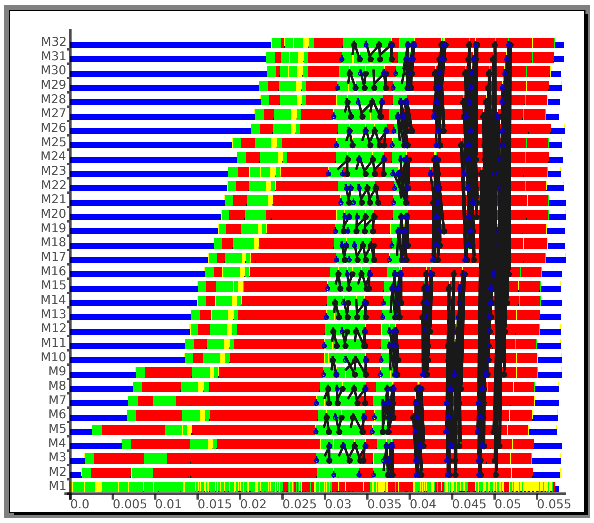
If the switched dimensions are identical we do not modify the vector `v`. Otherwise we permute `v` by the order given by function `partnerF` (see Appendix A.5).

Figure 6.5 depicts traces of program runs for input size 64 with input distribution 2^6 and 4^3. This is the same setup as for the traces in Figure 5.12. The traces show similar runtime behaviour and runtimes compared with the previous version. When increasing the problem size, the vector based version runs clearly slower then the previous version. The problem is that the master is overloaded. We suspect that the `backpermute` function runs notably slower than the `ixmap` function for the array type. However, our goal was not to outperform the previous version, which was already optimised by the use of arrays. We rather showed an interesting use case of how to use vectors consistently to substitute lists in the parallel interface as well as in the processes input and output. ◁

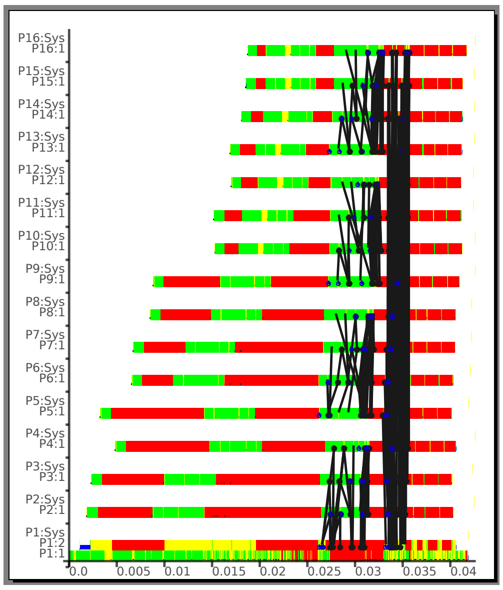
Building Topologies

One might ask the question of the usefulness of this generalisation. The possibility of `spawn`ing data containers like trees, matrices or rings e.g. does not include the topological connections. Channels still have to be created by exchanging remote data or dynamic

63 See Appendix A.5 for the change-dim control function `chdimVCF` and the `chdimsV` function.



(a) input distribution 2^6, 32PEs, machine view



(b) input distribution 4^3, 16PEs, process per machine view

Figure 6.5: Traces of FFT with N dimensional distributed homomorphism skeleton –vector based iteration framework version– for input size 64

channel handles. But as the exchange of handles will follow the result structure of a skeleton, the handles can be moved along the structure's connections. This makes it easy to create topology channel connections among the processes.

A naïve solution to our approach of implementing `spawn` with type class `Traversable` would be to realise a different version for each single container type. But this would lead to a lot of implementational overhead. The more general solution makes it also possible to generalise skeletons based on `spawn` for arbitrary `Traversable` containers.

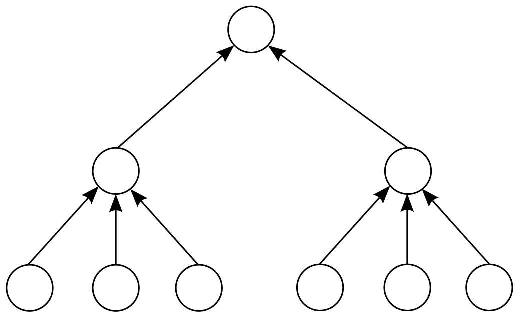
Example: **A tree reduction topology skeleton** with connections to each node's father process, built by exchanging remote data via the tree structure is listed in Figure 6.6a.

The `treeBottomUp` skeleton creates a process tree by spawning a `Tree` structure from module `Data.Tree`. It returns the result tree and the result of the root node. Process function `f` takes a process' part of the input and the inputs from its child nodes and creates one output to its parent and its part of the result tree. In function `f'`, the child/parent related components of function `f` are lifted to remote data. Function `upRot` defines the exchange of remote data from child's to parent. The first result component is the output of the root process. The up rotated remote data is re-fed lazily to the processes.

This skeleton could e.g. be used to implement a distributed function to calculate the sub-tree sums of a process tree (see Figure 6.6c) completely flexibly in the branching

```
treeBottomUp :: forall a b c. (Trans a, Trans b, Trans c)
                ⇒ (a → [b] → (b,c)) → Tree a → (b, Tree c)
treeBottomUp f ts = (fetch resRoot, resTree) where
  (toParents, resTree) = unzipTF ∘ spawnF (repeat f') $ inps
  inps :: Tree (a, [RD b])
  inps = zipTF ts (lazy ∘ toList $ fromChilds)
  (resRoot, fromChilds) = upTree toParents
  f' :: (a, [RD b]) → (RD b, c)
  f' (x, fromChilds) = (release toParent, res) where
    (toParent, res) = f x (fetchAll fromChilds)

upTree :: Tree a → (a, Tree [a])
upTree (Node x []) = (x, Node [] [])
upTree (Node x sTs) = (x, Node xs sTs') where
  (xs, sTs') = unzip $ map upTree sTs
```

(a) The `treeBottomUp` skeleton



(b) `treeBottomUp` skeleton example with 9 processes

```
subtreesums :: (Trans a, Num a) ⇒ Tree a → (a,Tree a)
subtreesums = treeBottomUp f where
  f x xs = let x' = foldl' (+) x xs in (x',x')
```

(c) `subtreesums` example

Figure 6.6: Definition of the `treeBottomUp` skeleton

degree of each node, which is determined by the concrete input tree. The calculated tree (second result component) contains in each node the local sub-tree sum. The first result component contains the sum of all nodes in the tree, which is equal to the value at the root process in the second result component. ◁

Generalising `fetchAll` with a `Traversable` container is straight forward:

```
--import qualified Data.Traversable as T
fetchAll :: (Traversable t, Trans a)
            ⇒ t (RD a) -- ^ The Remote Data handles
            → t a      -- ^ The original data
fetchAll rdas = runPA $ T.mapM fetchPA rdas
```

We only need to use function `mapM` from `Data.Traversable` instead of `Control.Monad`.

6.2.2 Explicit Process Placement

In Eden, we distinguish `spawnF` with automatic process placement and `spawnFAt` with explicit placement. Likewise this distinction is made for almost all Eden skeletons.

> *Note:* Function `spawnFAt` works on `Places` ∼ `[Int]`. As we want to generalise the list interface, we will work with `type Place = Int` inside arbitrary data containers.

We seek to change this with the help of the new type class `Spawnable` (see Figure 6.7). A `Spawnable` container is already `Traversable` and offers further the function `getPlaces`, which specifies where the processes `spawn`ed with the data should be instantiated. The default implementation returns 0 for every element of the structure, which means automatic placement. We can use the default implementation to get automatic placement for predefined `Traversable` types like lists or trees.

To get explicit placement information for `Spawnable` data, we need special instances which either generate the `places` from the structure or contain an additional field with the placement information. The new type `Targetted` (see Figure 6.8) offers such an additional field for arbitrary data containers of kind $* \rightarrow *$, where the placement information `placesT` is stored in the same container type as the data field `dataT`.

A `Targetted t` container is `Foldable`, `Functor` and `Traversable` if the contained container of type `t` is `Foldable`, `Functor` and `Traversable`. The instance declarations for `Targetted t` are all based on the instance functions for `t` applied to the contained data of type `t`. The `Spawnable` instance definition is also straight forward. Function `getPlaces`

```
class (Traversable t) ⇒ Spawnable t where
  getPlaces :: t a → t Place
  getPlaces = fmap (const 0)

instance Spawnable []
instance Spawnable Tree
```

Figure 6.7: Typeclass `Spawnable`

```
data Targetted (t :: * → *) (a :: *) = T {placesT :: t Place,
                                          dataT   :: t a}
-- qualified Functor as F
instance (F.Foldable t) ⇒ F.Foldable (Targetted t) where
  foldr f a ta = F.foldr f a (dataT ta)

instance (Functor t) ⇒ Functor (Targetted t) where
  fmap f ta = ta{dataT = fmap f (dataT ta)}

instance (Traversable t) ⇒ Traversable (Targetted t) where
  traverse f ta = pure toData <*> traverse f (dataT ta) where
    toData a = ta{dataT = a}

instance Traversable t
         ⇒ Spawnable (Targetted t) where
           getPlaces ta = ta{dataT = placesT ta}
```

Figure 6.8: Type `Targetted` and its instances

forwards the `placesT` field to the `dataT` field.

Function `spawnF` with explicit process placement

With type class `Spawnable` at hand, we can refine function `spawnF` to handle data with and without explicit placement information (see Figure 6.9). It is a straight forward modification of `spawnF`'s previous version, where the `Traversable` context for the data container is exchanged by the `Spawnable` context. We use further function `instantiateAt` instead of `instantiate` and places extracted from the input data using function `getPlaces`.

Aside from overloaded communication, we now have *overloaded process placement*. Using the new version of `spawnF`, we will not need to distinguish skeletons with and without explicit placement. The process placement is now driven by the input data, which of course can be modified within a skeleton or restricted by a skeleton's signature.

We profit from this also in the main task of skeleton compositionality, as the placement information is forwarded between skeleton instances without the need to indicate the places repeatedly.

```
spawnF :: (Spawnable t, Foldable f, Trans a, Trans b)
  ⇒ f (a → b) → t a → t b
spawnF fs xs = runPA (zipWith3TFM instantiateFAt' xs (getPlaces xs) fs)  where
  instantiateFAt' x pos f = instantiateFAt pos f x
```

Figure 6.9: Function `spawnF` for `Spawnable` data

Example: We illustrate this with a simple composition of a `parMap` skeleton.

```
parMap :: (Spawnable t, Trans a, Trans b)
          ⇒ (a → b) -- ^worker function
          → t a      -- ^tasks
          → t b      -- ^results
parMap f tasks = spawnF (repeat f) tasks

compPMap :: (Spawnable t, Trans a, Trans b, Trans c)
            ⇒ (a → b) -- ^function1
            → (b → c) -- ^function2
            → t a      -- ^tasks
            → t c      -- ^results
compPMap f g inp = dataT ∘ parMap g ∘ parMap f $ T ([4..] `withStruct` inp) inp
```

Once the places (say starting from 4) are added to the `Targetted` structure, each skeleton in the sequence will take the places from the result of the predecessor skeleton.

> *Note:* Auxiliary function `withStruct` in the example is needed because the container type of the input is unknown and we want to supply the places in form of an arbitrary `Foldable` container like a simple list:
>
> ```
> withStruct :: (Traversable co, Foldable co')
>             ⇒ co' a  -- functor with elements
>             → co b   -- blueprint for new container
>             → co a   -- container with elements of input list
> withStruct = flip (zipWithTF (flip const))
> ```
>
> Function `flip const` returns the second argument, `zipWithTF (flip const)` returns the first input structure with the elements of the second input structure. With the outermost `flip`, `withStruct` returns the elements of the first structure inside the second structure.

We do not change the places in the `parMap` skeleton, such that the initially supplied places are forwarded to the second skeleton instance, leading to a co-allocation of the processes of both instances. This is especially important for the performance when using remote data in `f`, `g` and `inp`. ◁

6.2.3 Distributed Data Types

In Chapter 5.1.6, type

```
data DList a = DList [RD a]
```

has been introduced. Internally, we always used a representation containing places:

```
data DList a = DList places [RD a]
```

With this container for distributed lists, where all list elements are remote data handles, we were able to define a special instance for type family `Iterated` to optimise the case of iterations over distributed lists. We want to define a distributed data type not only for lists, but for arbitrary distributed data. Our first approach is to make the list-container type of `DList` a parameter to the distributed data type, similar to the definition of the `Targetted` type:

```
data Distrib0 (c :: * → *) (a :: *) = D0 {placesD0 :: c Place,
                                           dataD0   :: c (RD a)}
```

Problems arise when we try to define instances for `Foldable`, `Functor` and `Traversable`. We need these to provide function `toList` and the generalised `zip`'s from the beginning of this Section. E.g. in the instance definition for `Foldable (Distrib0 c)`, the type for `foldr` is `(a → b → b) → b → Distrib0 c a → b`, where `Distrib0 c a` contains `c (RD a)` and `c` is a `Foldable` container. `foldr` uses a parameter function `f :: a → b → b` but the container contains values of type `RD a`, hence we would be forced to `fetch` all the elements of the container before applying them to function `f`. This would also lead to a definition of `toList :: Distrib0 c a → [a]`, where all the remote data handles would be implicitly fetched. Basically, we want `toList` to change only the container type, we do not want to change the location of the data implicitly. We seek a definition of `toList` where remote handles are not fetched.

This kind of problem occurs similarly at the instance definitions of `Distrib0 c` for `Functor`, where we need to use `liftRD` for the `map`-function `f` – and `Traversable`, where we need to lift the `traverse` function of type `a → f b` to type `RD a → f (RD b)`. The lifting in the `traverse` function would e.g. affect the `spawn` function: we would fetch the distributed data before `spawn`ing the processes and `release` the data after receiving the processes results. This is exactly what we want to avoid by using remote data in the first place.

Thus, we define a similar type `Distrib'`:

```
--Distributed Data intermediate type
data Distrib' (c :: * → *) (a :: *) = D {placesD :: c Place,
                                         dataD   :: c a}
```

without `RD` in the `dataD` field. This is isomorphic to the definition of `Targetted` and we define the instances for `Foldable`, `Functor` and `Traversable` exactly the same way as the `Targetted` instances. Now we can define the proper distributed data as a mere type definition:

```
--Distributed Data
type Distrib (c :: * → *) (a :: *) = Distrib' c (RD a)
```

With this definition, the `Foldable`, `Functor` and `Traversable` instances work on the underlying `Distrib' c` container and do not touch the remote data handles. Everything works as desired.

One might ask why we did not base the `Distrib` type definition directly on `Targetted`. This way, we would get overlapping instances for

```
type instance Iterated (Distrib c a) = Distrib c (Iter a)
```

and

```
type instance Iterated (Targetted c a) = Iter (Targetted c a),
```

because `Distrib c a` $\sim$ `Targetted c (RD a)` and `Targetted c a` overlap. With the data definition for `Distrib'`, these definitions are fine.

For convenience, we define some auxiliary functions to initialise distributed data

- with automatic placement:

```
newDistrib :: (Traversable co, Trans a)
              ⇒ co a → Distrib co a
newDistrib ds = D ([0] `withStruct` ds) (fmap release ds)
```

- or with explicit placement:

```
newDistribAt :: (Foldable f, Traversable co, Trans a)
                ⇒ f Place → co a → Distrib co a
newDistribAt ps ds = D (ps `withStruct` ds) (fmap release ds)
```

- where the container contains already remote data, with automatic placement:

```
newDistribRD :: (Traversable co, Trans a)
              ⇒ co (RD a) → Distrib co a
newDistribRD ds = D ([0] `withStruct` ds) ds
```

- or with explicit placement:

```
newDistribRDAt :: (Foldable f, Traversable co, Trans a)
                  ⇒ f Place → co (RD a) → Distrib co a
newDistribRDAt ps ds = D (ps `withStruct` ds) ds
```

6.2.4 Location Aware Remote Data

Another useful information we missed so far is the location of remote data. With the original definition, we do not know where the data behind a remote data handle is actually located. We add this information to a new remote data definition:

```
data RD a = RD {place :: Place,
                rd :: ChanName (ChanName a)}
```

The original remote data definition was a mere `type` definition, its representation is now contained in the `rd` field of the new `data` definition for `RD`. It additionally contains the `place` where it was created by function `release` or its `PA`-monadic version:

```
release :: Trans a ⇒ a -- ^ The original data
           → RD a      -- ^ The Remote Data handle
release = runPA ∘ releasePA

releasePA :: Trans a
            ⇒ a            -- ^ The original data
            → PA (RD a)    -- ^ The Remote Data handle
releasePA val = do
  (cc , sendValC) ← createComm
  sendValC ! val
  self ← PA $ ParPrim.selfPe
  return (RD self cc)
```

The definition does not change much. Like before, we create the nested channel handle `cc` (the remote data handle) and the handle for the send function `sendValC`, which we will receive after `fetch` is called with the remote data handle. The send operator `(!)` will fork a process to send the remote value `val` subsequently with the send function `sendValC`. The remote data result is then composed of the local machine id `selfPe` and the channel handle `cc`.

The `fetch` implementation does not change technically as the `place` field is not needed here. We can use the new field to extract the place information after remote data handles have been received on another machine. This way, it is easy to explicitly co-locate the processes of subsequent skeleton composition steps exploiting the `place` information of the remote data handles.

Example: Here a small example, a revised version of the `parMap` composition from Chapter 6.2.2.

```
compPMap' :: (Trans a, Trans b, Trans c)
           ⇒ (a → RD b)  -- ^function1
           → (RD b → c)  -- ^function2
           → Tree a       -- ^tasks
           → Tree c       -- ^results
compPMap' f g = dataD ∘ parMap g ∘ (λx → D (fmap place x) x) ∘ parMap f
```

The parameter functions `f` and `g` work on remote data and we use `Tree`s, a data structure without placement information. The process instantiation of `parMap f` for `Tree`s works with automatic placement, which is often just fine. Now we want to make sure that the second `parMap` stage uses the same processes as the first stage for performance reasons. We thus lift the result of the first skeleton to a `Distrib Tree b`. We use the location information of the remote data inside `Tree b` to define the places for the `Distrib Tree b`. ◁

Similar to the example above, but for the common use case of `Targetted` or `Distrib`uted in- and output, we define the new type class `Placeable` with function `placesFromRD` (see Figure 6.10). Function `placesFromRD` takes a `Placeable` structure of type `t (RD a)` and

```
class (Spawnable t) ⇒ Placeable t where
  placesFromRD :: t (RD a) → t (RD a)

instance Traversable t
         ⇒ Placeable (Targetted t) where
           placesFromRD t = t{placesT = fmap place (dataT t)}

instance Traversable t
         ⇒ Placeable (Distrib' t) where
           placesFromRD d = d{placesD = fmap place (dataD d)}
```

Figure 6.10: Type class `Placeable` and its instances

returns a modified structure of the same type. The placement information should be replaced by the places of the remote data inside the structure, like in the instance definitions for `Targetted` and `Distrib'`.

Example: In our example `compPMap''`, we use `Placeable` containers `t` :

```
compPMap'' :: (Placeable t, Trans a, Trans b, Trans c)
           ⇒ (a → RD b)      -- ^function1
           → (RD b → c)      -- ^function2
           → t a             -- ^tasks
           → t c             -- ^results
compPMap'' f g = parMap g ∘ placesFromRD ∘ parMap f
```

As before, we want to apply the two `parMap` stages. Despite the `Placeable` context, we can not know that the input data carries explicit placement information. The places field might as well contain zeros. Thus, we can use function `placesFromRD` between the stages to get the places from the remote data. ◁

6.2.5 Comparison: Old and New Skeleton Versions

How does the new generalised interface effect our skeleton library? In this section, we present a sample of Eden skeletons before and after the generalisation and examine, if the surplus flexibility has significant impact on the simplicity of code.

Map Skeletons

The `ranch` differs only in the generalised type from its basic version:

```
ranch :: (Spawnable t, Trans b, Trans c) ⇒
         (a → t b)      -- ^input distribution function
         → (t c → d)    -- ^result combination function
         → (b →  c)     -- ^worker function
         → a            -- ^input
         → d            -- ^output
ranch transform reduce f xs = reduce $ parMap f (transform xs)
```

The same holds true for the `farm`. Here, the implementation problem lies in the definition of the parameter functions `distrib` and `combine` which have to transform the input container to an agglomerated one and vice versa, providing the agglomerated values as a stream. The following is a straight forward generalisation of the basic `farm`s type and may be too restrictive for some input containers. If the agglomeration does not fit the interface, then the `ranch` with a more liberal interface can be used likewise:

```
farm :: (Spawnable t, Trans a, Trans b)
        ⇒ (t a → t (Stream a)) -- ^input distribution function
        → (t (Stream b) → t b) -- ^result combination function
        → (a →  b)             -- ^mapped function
        → t a                  -- ^input
        → t b                  -- ^output
farm distr combine f tasks = ranch distr combine (fmap f) tasks
```

Iteration Framework

The iteration framework can be generalised from `DList a` to distributed containers `Distrib co a` for `Traversable` containers `a` and inner types `a`. We have to exchange `zip` and `unzip` with `zipTF` and `unzipTF` and function `lazy` with `lazy ∘ toList` like in the case of `localControl`:

```
localControl ::
  forall a b c d co.
  (Trans a, Trans b, Trans c, Trans d, Traversable co)
  ⇒ (a → Iter c → (Iter b, d))            -- ^process termination control
  → Distrib co b                          -- ^dummy type
  → Distrib co c                          -- ^dummy type
  → Distrib co a                          -- ^initial Input
  → Iterated (Distrib co c)               -- ^output of loops
  → (Iterated (Distrib co b), Distrib co d) -- ^input for loops, final result
localControl controlF _  _ as css
  = (bss, ds) where
    (bss,ds) = unzipTF $ parMap f $ zipTF as $ lazy $ toList css
    f :: (RD a, RD (Iter c)) → (RD (Iter b), RD d)
    f (a, cs) = (release bs, release d) where
      (bs,d) = controlF (fetch a) (fetch cs)
```

Function parMapDistrib generalises parMapRD, with the same implementation:

```
parMapDistrib :: (Trans a, Trans b, Traversable co)
                 ⇒ (a → b) → Distrib co a → Distrib co b
parMapDistrib f as = parMap (liftRD f) as
```

We use it to generalise parMapIter – which apart from its type – is again equal to the original version:

```
parMapIter :: (Trans b, Trans c, Traversable co)
              ⇒ (b → c)
              → Iterated (Distrib co b)
              → Iterated (Distrib co c)
parMapIter f = parMapDistrib $ fmap f
```

More challenging is the definition of a generalised version allToAllDistrib for allToAllRD. The problem is to span the allToAll connections based on an arbitrary container. We chose to reduce the problem using a list interface for the remote data handles of the inner allToAll connections. This works especially well because we can zip the initial Traversable input with lazy list input of remote data handles.

```
allToAllDistrib :: forall b c i co.
           (Trans b, Trans c, Trans i, Traversable co)
           ⇒ (Int→b→[i])          -- ^transform before transpose
           → (b→[i]→c)          -- ^transform after transpose
           → Distrib co b         -- ^remote input for each process
           → Distrib co c         -- ^remote output for each process
allToAllDistrib t1 t2 xs = res where
  n = length $ toList xs          --same amount of procs as #xs
  (res,iss) = n `pseq` unzipTF $ parMap f inp
  inp       = zipTF xs (lazy $ transposeRt $ toList iss)

  f :: (RD b,[RD i]) → (RD c,[RD i])
  f (x,theirIs) = (resF theirIs, myIsF x') where
    x' = fetch x
    myIsF = releaseAll ∘ (t1 n)
    resF  = release ∘ (t2 x') ∘ fetchAll
```

The iterable version of allToAll is identical to the original allToAllIter implementation, apart from using different allToAllDistrib skeletons.

```
allToAllIter ::
  forall b c i co. (Trans b, Trans c, Trans i, Traversable co)
  ⇒ (Int→b→[i])              -- ^transform before transpose
  → (b→[i]→c)                -- ^transform after transpose
  → Iterated (Distrib co b)   -- ^remote input for each process
  → Iterated (Distrib co c)   -- ^remote output for each process
allToAllIter t1 t2 = allToAllDistrib t1Iter t2Iter where
    t1Iter p         = map Iter ∘ transposeRt ∘ map (t1 p) ∘ fromIter
    t2Iter (Iter bs) = Iter ∘ zipWith t2 bs ∘ transpose ∘ map fromIter
```

In the following, we will study the benefits of the generalised spawn implementation on the gentleman algorithm [Gen78].

6.2.6 Case Study: Torus Skeleton and Gentleman Algorithm

The `torus` skeleton unfolds a grid topology with round about connections at the borders. Each process receives its initial input of type `c` from the master and torus input from its left and lower neighbour (types `c` and `d`). The input/output is a nested list of type `c`/`d`. Working with the nested list means that we have to nest list functions appropriately. The processes input `t_inss` e.g. is combined from the initial input and the remote data handles of each processes neighbours', which are supplied lazily. We use `zipWith3 lazyzip3` on the input lists to generate an input triple for each process. Analogously, we use `unzip3 (map unzip3...` to separate the output triples of the processes into lists. The lazy input for the processes is composed of the rotated remote data handles of the output, which generate the torus connections.

Finally, the `ptorus` process function lifts the latter input and output components of function `f` to remote data. Spawning the nested list of process functions with function `map spawnF` would not be eager enough, so we define `spawnFss`, a nested `sequence`[64] of `instantiateF` calls, to ensure eager instantiation.

```
torus :: (Trans a, Trans b, Trans c, Trans d) ⇒
         (c → a → b → (d,a,b)) -- ^ node function
         → [[c]] → [[d]]        -- ^ input-output mapping
torus f inss = outss
  where
    t_outss = spawnFss (repeat (repeat (ptorus f))) t_inss
    (outss,outssA,outssB) = unzip3 (map unzip3 t_outss)
    inssA   = map rightRotate outssA
    inssB   = rightRotate outssB
    t_inss  = zipWith3 lazyzip3 inss (lazy inssA) (lazy inssB)
    lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)

-- | Spawn a matrix of processes
spawnFss :: (Trans a, Trans b) ⇒ [[a → b]] → [[a]] → [[b]]
spawnFss pss xss
  = runPA $ zipWithM (zipWithM instantiateF) pss xss

ptorus :: (Trans a, Trans b, Trans c, Trans d)
          ⇒ (c → a → b → (d,a,b))
          → (c,RD a,RD b)
          → (d,RD a,RD b)
ptorus f (fromParent, inA, inB) =
               let (toParent, outA, outB) = f fromParent inA' inB'
                   (inA',inB')            = fetch2 inA inB
               in  (toParent,   release outA,  release outB)
```

The new version of the `torus` skeleton works with an explicit `Matrix` type and an appropriate `Traversable` instance. We use `Traversable` versions for `zip3`, `unzip3` and `spawn` directly and thus have no need for sophisticated, type specific solutions:

64 `zipWithM f xs ys = sequence (zipWith f xs ys)`

```
newtype Matrix a = Matrix {fromMatrix :: [[a]]}
                     deriving (Eq, Show)
-- with Foldable, Functor and Traversable instances
liftMatrix f = Matrix ∘ f ∘ fromMatrix

torus :: (Trans a, Trans b, Trans c, Trans d) ⇒
         (c → a → b → (d,a,b)) -- ^ node function
         → Matrix c → Matrix d  -- ^ input-output mapping
torus f inss = outss
  where
    t_outss = parMap (ptorus f) t_inss
    (outss,outssA,outssB) = unzip3TF t_outss
    inssA   = liftMatrix (map rightRotate) outssA
    inssB   = liftMatrix rightRotate outssB
    t_inss  = zip3TF inss (lazy2L inssA) (lazy2L inssB)
    lazy2L = lazy ∘ toList
```

Other differences are:

- In order to make inputs `inssA` and `inssB` lazy[65], we need to use function `toList` in combination with function `lazy :: [a] → [a]`. This pattern is typical for topology skeletons. We `zip` a `Traversable` initial input with the lazy input. In more general settings, the lazy input is of an arbitrary `Foldable` type. Even though, after using `toList`, we can define the resulting list constructors lazily. The use of `toList` here is perfectly fine, as all `zip..TF` versions only use the first input as template structure for the output.
- The use of `liftMatrix` allows to apply ordinary list functions to the `newtype Matrix`.

We use the torus skeleton to implement a parallel matrix multiplication according to Gentleman [Gen78]:

```
gentlemanMul :: forall a. (Num a,Trans a)
                ⇒ Matrix a → Matrix a → Matrix a
gentlemanMul (Matrix ass) (Matrix bss)
  = torus f (zipTF (Matrix ass') (Matrix bss')) where
  -- function on torus elements
  f :: (a,a) → Stream a → Stream a  → (a, Stream a, Stream a)
  f (initR, initD) (Stream fromR) (Stream fromD)
   = (res, Stream streamL, Stream streamU) where
    res = foldl1' (+) $ zipWith (*) streamL streamU
    streamL = initR : take (n-1) fromR
    streamU = initD : take (n-1) fromD
    n = length bss
  -- pre-rotation
  ass' = zipWith leftRotIXs [0..] ass
  bss' = transpose $ zipWith leftRotIXs [0..] (transpose bss)
  leftRotIXs i xs = xs2 ++ xs1 where
    (xs1,xs2) = splitAt i xs
```

65 to define all the (inner) list constructors beforehand

The algorithms parameters are two matrices `ass` and `bss` which have to be pre-rotated in a sequential preparation step. We rotate row i of matrix `ass` i elements to the left and column j of matrix `bss` j elements up. We zip the matrices to a matrix of pairs (using `zipTF`) and put each pair on a process of the torus. We use function `f` on the torus to implement the processes functionality of the matrix multiplication. Function `f` receives a `Stream fromR` from its right neighbour and a `Stream fromD` from the neighbour beneath. It uses $n-1$ elements from each stream. We calculate the sum of the products of our initial elements `initD` and `initR` concatenated with the incoming streams (forming `streamL` and `streamU`) to determine the result of the matrix multiplication at this processes' position. The function forwards `streamL` and `streamU` to the left and upper neighbour processes.

In this version, the streams to the neighbours are forwarded directly without waiting for the intermediate results of the local sum. The advantage of calculating the local sums before the streams are forwarded to the neighbouring processes is that the process need only to keep two matrix elements in the local memory. If all streams are forwarded directly, then every process has to keep a whole row and column of the matrix (worst case). We added such a demand control to the algorithm. The code of this `gentlemanMulDemand` version can be found in Chapter A.3.

The code of an implementation using the old torus skeleton is similar and differs only at the upper part, because in the previous version we fell back to using a list representation of the matrix in the lower part of the function body:

```
gentlemanMul :: forall a. (Num a,Trans a)
                ⇒ [[a]] → [[a]] → [[a]]
gentlemanMul ass bss = torus f (zipWith zip ass' bss') where
  -- function on torus elements
  f :: (a,a) → [a] → [a]  → (a, [a], [a])
  f (initR, initD) fromR fromD
   = (res, streamL, streamU) where ...
```

On the one hand this implementation is more simple, as we can work directly on lists. On the other hand it is less expressive as the types are less meaningful. E.g. the type `[[a]]` may as well refer to a list of streams and the stream input of function `f` may also refer to parallel input for a sub-skeleton. Further, zipping the matrices for the torus input using function `zipTF` is also more convenient in the new version, in contrast to using `zipWith zip` in the old version.

To optimise the performance of the implementation we have to agglomerate the input. We use thus a block distribution of the input matrices. An elegant way to do this is to use the fact that the `gentlemanMul` and `gentlemanMulDemand` implementation do work for any instance of typeclass `Num`. Thus both implementations can be used with a matrix of matrices of type `a` (`Matrix (Matrix a)`), if `Matrix a` is an instance of `Num`:

```
mPlus :: Num a ⇒ Matrix a → Matrix a → Matrix a
mPlus ass bss = zipWithTF (+) ass bss

mMul :: Num a ⇒ Matrix a → Matrix a → Matrix a
mMul (Matrix ass) (Matrix bss) = Matrix css where
  css = map (flip vecMatProd (transpose bss)) ass
  vecMatProd vec mat = map (dotProd vec) mat

dotProd :: Num a ⇒ [a] → [a] → a
dotProd vec1 vec2 = foldl1' (+) $ zipWith (*) vec1 vec2

instance Num a ⇒ Num (Matrix a) where
  (+) = mPlus
  (*) = mMul
  ...
```

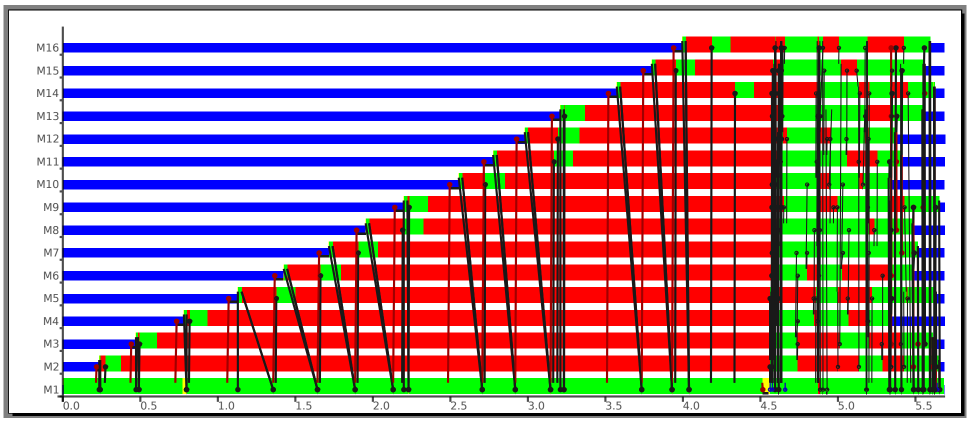
Thus we can define a wrapper function `gentlemanFarm` to slice the sub-matrices and call `gentlemanMulDemand`:

```
gentlemanFarm :: forall a. (Num a,Trans a,NFData a)
                ⇒ Int → Matrix a → Matrix a → Matrix a
gentlemanFarm n m1 m2
  = concatMatrix n n
    ∘ gentlemanMulDemand (splitMatrix n n m1) $ (splitMatrix n n m2)

splitMatrix :: Int → Int → Matrix a → Matrix (Matrix a)
concatMatrix :: Int → Int → Matrix (Matrix a) → Matrix a
```

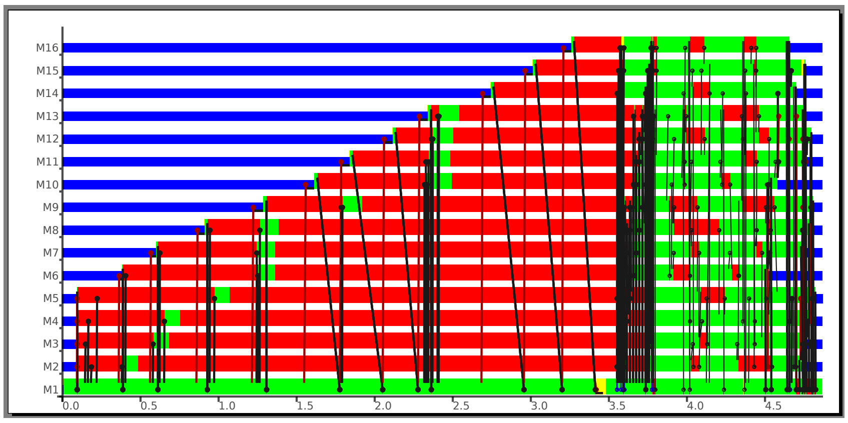
This is as well possible with the old Eden implementation, but we would have to work with a `[[Matrix a]]`[66] type. We prefer the `Matrix (Matrix a)` version, which is more elegant in our opinion. One advantage of the old version has been that we did not have to define the `newtype Matrix` with all kinds of typeclass instances. In the end we would have to do this work anyhow.

In Figure 6.11 we show trace visualisations of both `gentlemanFarm` versions. The process instantiation differs in both versions. The version with the modified Eden modules (*new version*) instantiates at least the first 4 processes instantly. The version with the classic Eden modules (*old version*) instantiates the processes one after the other. The computation phases are quite similar, even though there seems to be a different communication structure. Both – the different instantiation behaviour as well as the different communication structure – may imply that the processes are instantiated in a different order. We could not analyse this in more detail.

66 As we can not define a `Num` instance for `[[a]]`.



(a) old version



(b) new version

Figure 6.11: `gentlemanFarm` with 400 inputs on 16 PEs

6.2.7 Case Study: Mandelbrot with different Eden Versions

We compared different versions of the Eden- modules and skeletons using a simple Mandelbrot program[67]. We use a `farm` with round robin distribution of inputs to apply the worker function `wf` on the available machines:

```
farm (unshuffle (np-1)) shuffle wf
```

We used basic *Eden* in the first program version, the modified implementation of the Eden modules and the farm skeleton of this Chapter in version *Eden mod* and a similar version, which was slightly changed by stopping the recursive calls to `transform` and `untransform` at a certain level (version *Eden mod no rec*). We stop transforming recursively when further transformations may no longer change the communication mode. Thus we do not

67 ... with a different implementation then in Chapter 4.4.1. Here we use inputs and outputs with streams of lists instead of vectors (type `[[a]]` or `Stream[a]`) which do not scale well.

transform outputs of type list or `Stream` or `Maybe`, but we still transform outputs of tuple type or the `Iter` type recursively.
We performed our measurement on the Beowulf cluster (see Chapter 2.2.1). Mean runtimes of 5 program runs for each of the different versions and input size 1000 are listed in the following table:

PEs	*Eden* (1)	*Eden mod* (2)	*Eden mod no rec* (3)	(1) vs (2)	(1) vs (3)	(2) vs (3)
2	10.82	13.03	10.59	16.95%	-2.12%	-18.70%
3	6.99	8.03	6.99	12.90%	-0.10%	-12.99%
4	6.99	7.93	7.07	11.84%	1.07%	-10.88%
5	7.24	8.09	7.15	10.44%	-1.29%	-11.60%
6	7.27	8.08	7.28	9.97%	0.16%	-9.82%
7	7.21	8.13	7.20	11.35%	-0.17%	-11.51%
8	7.17	8.16	7.16	12.15%	-0.10%	-12.24%
9	7.21	8.04	7.33	10.34%	1.72%	-8.77%
∅				11.99%	-0.10%	-12.07%

Versions *Eden* and *Eden mod* differ around 12%. This is because of the overhead for recursively applying the transformations on the inner lists, as we can see easily when comparing versions *Eden* and *Eden mod no rec*. Both perform close to identical (the mean difference is -0.1%). Thus, for performance reasons we should stop transforming recursively when the communication mode is not any more effected. This should at least be implemented for standard container types like list, `Maybe`, `Either`, `Tree`, etc.. Our motivation for transforming recursively has been to enable for example automatic compression or encryption of data which is nested in other container types. However, recursive transformations may still be defined for special purpose container types, while stopping them for standard container types.

6.3 Boxes

In Eden, data is evaluated to normal form before sending it and the sending mode is determined from the type of the data. From the beginning, Eden programs used “lift types” to disable streaming behaviour of list input. The lifted type has default `Trans` instance, such that the carried list will be sent in a single chunk. As a generalisation of this, we introduced boxes [DHLB16] in Eden.

There are *strict boxes:*

```
newtype SBox a = SBox {fromSBox :: a}

instance NFData a ⇒ NFData (SBox a)
         where rnf (SBox a) = rnf a

instance NFData a ⇒ Trans (SBox a) where
  type ComType (SBox a) = Mono (SBox a)
  transform = Mono
  retransform = fromMono
```

This version contains an `NFData` instance inherited from the boxes inner type and an instance for the new `Trans` class where transformation maps to `Mono`, with no recursion on the inner type. Thus, in addition to the monolithic send mode of the original `SBox` definition, our interpretation for the new `Trans` class prevents recursive transformations on the inner type. This is characteristic to all box definitions for the new `Trans` class.

Where strict boxes are basically equivalent to previous lift types, we offer also *lazy boxes:*

```
newtype LBox a = LBox {fromLBox :: a}

instance NFData (LBox a)
         where rnf = r0

instance Trans (LBox a) where
  type ComType (LBox a) = Mono (LBox a)
  transform = Mono
  retransform = fromMono
```

which are similar to strict boxes, but define the normal form reduction of the `NFData` instance as `r0` (no reduction). This means, that all values inside of lazy boxes are not further evaluated prior to sending them. With lazy boxes, we do not need special offline versions of our skeletons. We can send unevaluated data via arbitrary connections, e.g. we can also provide unevaluated output and send unevaluated data over remote data connections. This is especially useful to avoid the evaluation cost for applying `rnf` to the inner type, when data is known to be evaluated already.

Thus, boxes are container types which carry inner types in order to change their

- *sending mode* – boxes are sent as monolithic chunks,
- *evaluation degree* – boxes have associated strategies to evaluate their content,
- *transformation behaviour* – inner types of boxes are not transformed.

We define a type class for boxes to give a common API for treating boxes:

```
class Box b where
  toBox    :: a → b a               -- ^box from value.
  fromBox  :: b a → a               -- ^value from box.
  stratBox :: (NFData a)
             ⇒ b a → Strategy a    -- ^return the strategy of a box.
  boxMap   :: (a → a) → b a → b a -- ^like fmap, but with a single map type.
  copyBox :: b a → a → b a          -- ^puts new content in a given box.
  copyBox b a = boxMap (const a) b
```

Function `toBox` puts data in a new box and `fromBox` extracts the data from a box. `stratBox` returns the strategy of a box. Function `boxMap` is like `fmap` of the `Functor` class, but with a single map type, thus the associated strategy can be passed to the resulting box. Finally, `copyBox` puts new content in a given box, preserving the associated strategy.

Instances for lazy and strict boxes are straight forward:

```
instance Box SBox where
  toBox = SBox
  fromBox = fromSBox
  stratBox _ = rnf
  boxMap = fmap

instance Box LBox where
  toBox = LBox
  fromBox = fromLBox
  stratBox _ = r0
  boxMap = fmap
```

Functions `toBox`/`fromBox` simply apply/remove the `Box` constructor, `boxMap` is `fmap` with the restricted type and `stratBox` returns `rnf` or `r0` respectively.

With *dynamic boxes*, we add a more sophisticated representative to the set of boxes[68]:

```
data DBox a = DBox {fromDBox :: a,
                    stratDBox :: Strategy a,
                    initStratDBox :: Strategy a}
              | RnfPending {fromDBox :: a}
              | RnfApplied {fromDBox :: a}

toDBox :: Strategy a → a → DBox a
toDBox s a = DBox a s s

evalDBox :: NFData a ⇒ DBox a → DBox a
evalDBox b@(RnfApplied _) = b
evalDBox (RnfPending a)   = rnf a `seq` RnfApplied a
evalDBox (DBox a s1 s0)   = s1 a  `seq` DBox a r0 s0

instance NFData a ⇒ NFData (DBox a) where
  rnf (DBox a s1 _)  = s1 a
  rnf (RnfApplied _) = ()
  rnf (RnfPending a) = rnf a
```

68 This is joint work with my colleague Thomas Horstmeyer

```
instance NFData a ⇒ Trans (DBox a) where
  type ComType (DBox a) = Mono (DBox a)
  transform = Mono
  retransform (Mono (DBox a _ s0)) = DBox a r0 s0
  retransform (Mono b)             = RnfApplied (fromDBox b)

instance Box DBox where
  toBox = RnfPending
  fromBox = fromDBox
  boxMap f (DBox a _ s) = toDBox s (f a)
  boxMap f b = toBox (f $ fromBox b)
  stratBox (DBox _ _ s) = s
  stratBox _            = rnf
```

A dynamic Box (DBox) carries two Strategies. We store the associated strategy of the box `initStratDBox` and also a current strategy for the box `stratDBox`. The purpose of the current strategy is runtime optimisation, it is used for evaluation when `rnf` is called. Initially, we require both strategies to be identical, as it is when we create a `DBox` from a value and a strategy with function `toDBox`.

Function `toDBox` generates a box with the provided strategy. Constructors `RnfPending` and `RnfApplied` allow to generate `DBox`es with `rnf` strategy, even if the `NFData` context is not available. Here:

```
RnfPending a ∼ DBox a rnf rnf
RnfApplied a ∼ DBox a r0 rnf
```

This allows e.g. to define an instance for `DBox` of class `Box`, where `toBox` and `boxMap` do not contain the `NFData` context. This is essential because instances for `Functor` and `Applicative` use `toBox` and `boxMap`, and both classes do not contain the `NFData` context.

Like every value, a `DBox` will be evaluated before it is sent. Subsequently – on the receiver side – function `retransform` will set the current strategy `stratDBox` to `r0` (no evaluation). Thus, further uses of `rnf` will not lead to additional evaluation cost if the box is not modified. Modifying the box using `boxMap` will reset the current strategy `stratDBox` to the initial strategy.

Function `rnf :: NFData a ⇒ a → ()` itself will not modify the current strategy `stratDBox`, as it returns unit and not the box. Thus we provide function

```
evalDBox :: NFData a ⇒ DBox a → DBox a,
```

which returns the box, where the *next* strategy `stratDBox` is set to `r0` after evaluating the *current* strategy `stratDBox`.

CHAPTER 7

Related Work

Remote Data

Remote data resembles closely the remote references concept used to implement future based RMI in the context of a Java based skeleton framework for the grid. Alt and Gorlatch [Alt07, AG03, AG04] used future based RMI to implement *single server* skeleton composition, but implemented *multi server* skeleton composition using distributed data types. We get the main benefit of future based RMI[69] for free, as the channel creation used to implement remote data returns the channel handle immediately. Similar to the optimisation step for localised RMI [Alt07] (see Chapter 1.2), runtime support has been implemented for optimised communication on the same machine[70]. Communication is avoided if the released data of the sending process is referenced by the fetched data on the receiving process which both share the same heap (machine). This optimisation applies to all Eden channels, independent of their construction (remote data channels, dynamic channels or implicit channels). However, this is most relevant for skeleton composition using remote data with co-location of processes.
Aldinucci, Gorlatch et al. use future based RMI to optimise communication and coordination in the Java based skeleton library Lithium [ADD04] for the grid. In Lithium, all skeletons consume streams of inputs and produce streams of results, programs are compiled to a data flow graph. The optimisations do not require changes to Lithium programs, they affect rather how the data flow graph is executed by the scheduler. Optimisations comprise of *task lookahead* to avoid idle times on the servers and *Server-to-server lazy bindings* to avoid communication between the scheduler and the servers: The system trys to determine computation chains where each task depends on the result of the previous one and places those tasks on the same server. Thus, future based RMI is used to optimises

69 With future based RMI, the remote reference can be instantly returned to the caller.
70 This was implemented by Jost Berthold, thanks.

composition implicitly, but is not used explicitly in skeleton programming.

Remote data is related to futures. Harper describes futures as follows:

> "A future is a computation that is performed before its value is needed.
> ... a future represents a value that is to be determined later.
> ... a future is always evaluated, regardless of whether its value is required."[71]

In contrast, remote data represents a value on a remote machine. The evaluation degree of the value is unknown. The normal form evaluation of a value referenced by remote data is demanded when `fetch` is called on the receiver side, which is when it is needed. When a remote data handle is not used, the referenced value will not be evaluated[72].

`IVar`s (imutable variables) where introduced to model communication between threads in the Par-Monad [MNPJ11] for deterministic multicore programming in Haskell. They resemble closely I-structure components and especially `ICell`s, which where introduced in the context of PH [Nik01]. Like `MVar`s (mutable variables) of Concurrent Haskell [PJGF96], `IVar`s can be used to put data to and get data from, which allows for communication and synchronisation among different threads. Unlike `MVar`s, it is only allowed once to put a value into an `IVar`. From an abstract point of view, Eden's dynamic channel concept provides channels with a loose end at the sender side, the remote data concept provides channels with a loose end at the receiver side and an `IVar` is a channel with loose ends at sender and receiver side. Thus with `IVar`s, a third party (for example the caller) can create handles which are passed to a sender and a receiver to exchange data directly. `IVars` are well suited for a monadic style of programming in the PAR-Monad, but do not match the functional style of programming in Eden.

Skeleton Nesting and Skeleton Composition

The categories of skeleton nesting and skeleton composition are usually not distinguished in literature, as skeleton composition is often achieved by nesting skeletons in a pipeline skeleton or a composition skeleton.

P3L [BDO$^+$95] (based on C + MPI), Kuchen and Cole's skeleton library [KC02, Kuc02] and its successor, the Münster skeleton library *Muesli* [Kuc07] (based on C++ and MPI) use a two tier model, where the data parallel skeleton may be nested in task parallel skeletons, but not the other way around. The task parallel layer is compounded of a

71 [Har15, page 355]
72 With our current implementation, a thread used to evaluate the remote handle will be waiting for a receiver to call `fetch` and the referenced value will not be garbage collected until the program terminates. Thus, once created remote data must be fetched to avoid space leaks and unnecessary threads.

persistent skeleton nesting interconnected by streams of tasks. These skeleton systems also have in common, that at the root level programs are implemented by a pipeline skeleton, where the initial stage produces the input stream and the final stage consumes the output stream. For P3L, performant skeleton nesting is achieved by compiler optimisations, for example by optimising the sequence of collective MPI-operations used to implement the skeleton chain. The latter two skeletal systems use distributed data types for performant skeleton composition and model data (re-)distribution explicitly.

Darlington et al. [DGTY95b, DGTY95a] also use distributed data types for parallel composition in SCL, but borrow the (re-)distribution model from High Performance Fortran's [Lov93] data distribution directives. He "uses an imperative base as well but describes the composition of (predefined) skeletons itself functionally[.] ...

Although programming with distributed data structures is comfortable and efficient, the number of predefined data structures is limited and their use is thus not as flexible as working with remote data. Remote data can be nested in arbitrary algebraic data structures and manipulated by standard functions on those structures."[73]

Also the Java based skeleton libraries Lithium (mentioned already above) and its successor Muscle [GVL10] as well as Calcium [CL07] and its successor Scandium [GVL10] allow arbitrary skeleton composition. The latter two are inspired by the prior two and Scandium is a re-implementation of Calcium for multicore architectures. While Lithium uses data flow graphs as execution model, Calcium's execution model produces and processes a taskpool-tree, where processing of a task may lead to subtasks which are added to the taskpool-tree. A task is finished if it has no more subtasks and no pending skeleton code to execute. For Calcium, no indications are made whether communication overhead is optimised somehow, which is not relevant for Scandium because of the shared memory environment.

Skeletons are rarely used in the context of other parallel Haskells. The Par-Monad [MNPJ11] uses algorithmic skeletons as parallelism abstraction concept. Because of the shared memory setting, data flow optimisations for skeleton composition are not relevant. Cloud Haskell [EBPJ11] is another relevant parallel Haskell. It is aimed at a distributed setting and uses monadic style of programming like the Par-Monad, but unlike the Par-Monad it is nondeterministic. Cloud Haskell adepts the actor [HBS73] based parallel programming model of Erlang [Arm07]. Basic communication in Cloud Haskell uses untyped messages like in Erlang. The `send` function can address an arbitrary process, the receive function `expect`s messages from arbitrary senders. The messages can be matched by type or a predicate. Cloud Haskell allows also to send messages via typed channels. These work similar to dynamic channels in Eden, but communication channels in Cloud

73 [DHL10], page 87

Haskell are not overloaded. They are thus more like primitive channels in Eden. A channel is created at the receiver side and the typed receive port can be passed to the sender via existing channels. Thus, it should be possible to implement a Cloud Haskell version of remote data in the same way as we implemented remote data in Eden, allowing for similar skeleton composition mechanisms. But Cloud Haskell "concentrates on portability of network layers and basic control support, rather than providing high-level skeleton libraries."[74] Even though, a blog series on communication patterns in Cloud Haskell dedicates a part [dV12] to the implementation of the Google map-reduce skeleton in Cloud Haskell. Thus, skeleton programming is basically possible.

Skeleton Iteration

For our approach of skeleton iteration the distinction of skeleton composition in *transient* or *persistent* composition [BC05] is most relevant. In the iteration framework, we used persistent skeletons interconnected by streams of data. Skeleton frameworks using the two-tier model of P3L (P3L, Kuchen and Cole's skeleton library and Muesli) all use persistent compositions of task parallel skeleton interconnected by streams of data. Thus they already have the optimisation we used for iteration incorporated by design. They treat skeleton iteration by a simple control construct.

"The original skeleton work by Murray Cole [Col89] contains a chapter on an iterative completion, parallelised on a grid of processes, but does not generalise iteration as we do. Slightly more general is the iteration skeleton proposed in earlier Eden skeleton work [PR01], realising an iteration of a stateful parallel `map`. This work lays the grounds for our investigation, but does not generalise iteration bodies and types, nor does it consider parallel control skeletons.

Many skeleton libraries, especially those based on imperative programming languages, provide the constructs `while` for conditional iteration or `for` for fixed iteration and support skeleton nesting, see e.g. the Scandium library [LP10], which uses Java as computation language. However, no indications are made about whether iterated body skeletons will be optimised with respect to process creation overhead. A slightly larger corpus of related work can be found in the cloud computing community but usually restricted to map-reduce [DG08, BDL09] computations, like e.g. [ZGGW12][ELZ$^+$10]. HaLoop [BHBE12] is another Map-Reduce extension, which mainly capitalises on caching mechanisms for unmodified data and reduction results across several iterations of one map-reduce computation over the same dataset. A small API extension is provided to specify how existing map-reduce (Hadoop) computations should be iterated.

74 citeJFP16DHLB

None of these publications addresses parallel iteration as a general concept or distills out algorithmic patterns as we do. This generalising conceptual angle is present in very recent work in the data-flow framework Stratosphere [ETKM12]. The authors propose the concept of "incremental" iteration and "microsteps" to exploit sparseness of data dependencies and optimise read-only data accesses, but thereby break up the iterative nature of the computation.”[75]

Cost Models

Cost models are an important part of skeleton systems (e.g. in P3L and Eden). They are used to determine the cost of using the different skeleton implementations for a concrete algorithm and input on a specific parallel machine. Cost models can be used to provide mere information to the programmer, which can be included in the decision process for a specific implementation or the compiler may choose the skeleton implementation automatically, based on the cost model.

For the Eden system, cost models have been introduced by Rubio [Rub01] and Loogen et al. [LOP^{+}03] as mere information for the programmer. They describe the cost of a critical path, a sequence of actions including sequential times before and after parallelism is created and the maximum time of the single processes. Eden’s cost model takes problem dependent parameters like input size or the CPU time to compute a function, RTS dependent parameters like the time to create a process and architecture dependent parameters like the latency of a message or the start-up cost and the per-word cost for sending/receiving a message into account.
Hammond et al. [HBL03] present an approach of “automatic skeletons” in Eden, where a pre-processing step using Template Haskell [SJ02] at compile time supplied with problem-specific and environmental parameters, selects automatically a skeleton implementation based on the Edens cost model and generates optimised code.
Thus, it would be possible to use cost models for skeleton composition in Eden. This would be especially helpful for the following decisions:

- use remote data (if data is send via other processes and we have streams of data or bigger data chunks) or local data (if we have small data chunks or data is send directly to the final receiver)
- use a gather-distribute (if we have small data chunks) or an all-gather/all-to-All pattern (if we have bigger data chunks or streams)
- use the `iter` framework (in flavours `simpleIter` or `iterD`) or the recursive composition pattern using remote data.

75 [DHBL13], page 34

On the other hand, we observed some strange program behaviour when comparing skeleton composition and skeleton iteration with persistent process networks compared to transient skeleton composition. Here the persistent process networks have less overhead for process and channel creation and behave furthermore similar, as we could observe using runtime trace visualisations. Nevertheless the transient approach to skeleton composition achieved better performance in some cases. We could not figure out what causes this behaviour, but we guess it may be caused by garbage collection or other effects of the GHC runtime system[76], thus these effects would probably neither be detected by a cost model approach. We chose not to treat cost models in this thesis and rather to concentrate on the technical aspects of skeleton composition and programming methodology. This area is left for future work.

76 runtime trace analysis suggests that the surplus time is spent in sequential activity on the processes and not in the communication phases.

CHAPTER 8

Conclusions and Future Work

Initially we state the guiding question of this thesis from the introduction:

> What are alternative conceptual building blocks, which enable *performant* skeleton compositionality, are *easy to use* and provide *high flexibility* in terms of connectivity, extensibility and transformability?

Remote data is the key feature of the composition approach presented in this thesis. It allows the connection of two processes directly using a data handle. The sender does not need to know anything about the receiver to initiate the connection and return a data handle. The receiver needs just the data handle and no further information about the sender itself to establish a direct connection with the sender and receive the data. This offers *performant* composition of two processes with communication overhead substantially reduced.

It is the loose combination of remote data with arbitrary sequential data containers which gives our approach its *flexibility*. A skeleton that returns a container of remote data – typically created by its processes – offers a parallel interface. We can compose it with another skeleton with compatible signature (*connectivity*), possibly permute the positions of the remote data handles between the skeleton composition to define a redistribution of data or transfer the data in a different container if the skeleton signatures are not compatible (*transformability*). Skeletons can be composed directly using the prelude operator `( ∘ )` for function composition. Furthermore, programs do not need to be changed structurally to optimise skeleton composition by using remote data (*easy to use*).

Apart from skeleton composition, remote data is well suited to define topology skeletons. It "uses an existing communication topology to build direct connections between different processes. [...] With remote data the explicit channel handling using `new` and `parfill` can in most cases be abandoned. This improves the elegance and usability of Eden even more."[77] However, this raises the question if we should remove `new` and `parfill` from the Eden API and replace it by remote data? There are cases where `new` and `parfill` suit better than remote data. For example in the definition of the distributed workpool skeleton. Here the workers send requests within the ring when they are running out of work. Requests are replied by workers which have surplus tasks. The requests are implemented as bare channel names created by `new` and are used by `parfill` to send new tasks. Another example is a broadcast skeleton that can be defined conceptionally by reversing the communication direction of the `parRed` skeleton (see Chapter 4.2.1.3), thus we can implement it in principle by exchanging remote data with dynamic channels. However, `new` and `parfill` fit badly in the functional style of programming with Eden. It would be better to use `createComm` and the `(!)` operator (see Chapter 6.1.2) in the PA monad for the few cases where they are really needed.

The iteration framework is another approach to skeleton composition for the special case of skeleton iteration. It uses skeletons with static process networks where processes are connected by streams of data (persistent skeleton composition) to reduce the overhead of the naïve approach to skeleton iteration: recursive skeleton invocations using remote data. The concept of the iteration framework is orthogonal to the latter approach of transient skeleton composition. Apart from reducing the overhead of process and channel creation, it allows us to use state for iteration control and iteration body which render many data transfers unnecessary (*performant*) — particularly communications from the processes of the iteration body skeleton to the iteration control and back again. Even though remote data is no essential part of the framework itself, the combination of remote data and the iteration framework allows the creation of persistent channels not only between the processes of the iteration body and a single control process, but also between arbitrary processes of a parallel iteration control and a body skeleton. Remote data can be used to create transient connections between iteration control and body skeletons using the `simpleIter` scheme. This allows to *flexibly* change communication partners in the different iteration steps *(transformability).* Through the type family definition for `Iterated` skeleton input, we can define the `simpleIter` scheme allowing transient channel connections and the `iterD` scheme for persistent interprocess connections by a single function `iter` *(connectivity).*
"We allow for arbitrary parallel body skeletons and supply some parameterised control functions including step counting and termination conditions on local and global data [*easy to use*]. We have shown how body skeletons can be transformed in such a way that the body processes will be re-used for all iterations [*extensibility*], how to handle streams of input and output data, and how to optimise communication between distributed processes in a parallel execution."[78]

77 [DHL10]
78 [DHBL13]

The Eden extensions of Chapter 6 are designed to increase Edens support for skeleton compositionality further on, but are also of general use. This is most true for the redesign of Edens `Trans` class for transmissible data, which was split up into a new `Trans` class for transformable data, transforming arbitrary input into instances of the new `Transmissible` class. The `Transmissible` class defines instances for different sending modes: `Mono` for single messages, tuples for concurrent messages and `Stream` for stream messages. This instances must not be extended. *Extensibility* is rather achieved by defining simple transformations in the `Trans` class. This redesign was necessary to enable recursive streaming of the `Iter` type, but it has various other use cases, for example streaming instances for various common data types like vectors or trees and automatic chunking, compression or encryption of messages. Boxes allow to change the predefined communication behaviour, evaluation degree prior to sending and transformation depth for a given type. They can be used to optimise the communication behaviour of skeletons, skeleton compositions and applications (*performance*).
Other improvements aim more directly at skeleton composition like the generalisation for the function `spawn` which allows for polymorphic parallel interfaces of skeletons and thus facilitates the use of varying parallel interfaces for skeleton composition (*extensibility and connectivity* of parallel interfaces). Further the `Spawnable` class integrates placement information in the data containers which makes co-placement of processes among succeeding skeleton instances easier (*performance*). We implemented the extension of remote data to location aware remote data with the same objective of co-placement. With the type `Distrib` we introduce finally a type for distributed data in Eden – without jeopardising the advantages of remote data stored in arbitrary data containers compared to regular distributed data types: *transformability* and *extensibility.*

Comparing the results of the skeleton composition chapter, we get the following table:

	Mandelbrot	NAS EP	FFT (2D-DH)	PSRS
min	-2.56 %	-23.14 %	-2.81 %	-14.26 %
max	2.98 %	7.21 %	2.91 %	5.85 %
mean	0.14 %	-2.25 %	0.17 %	-2.55 %
abs mean	1.33 %	3.85 %	1.30 %	3.96 %

Apart of the minimal (min) and the maximal deviation (max) of runtimes between monolithic and compositional versions, we present mean values of the percental deviation (*mean*) and mean values for the absolute percental deviation (*abs mean*). If the *abs mean* value differs substantially from the absolute value of the *mean* value, then there are differences of runtimes in both directions which partially neutralise themselves on an average. One version constantly outperforms the other if the absolute value of the *mean* value and the *abs mean* value are identical.

The *abs mean* values are always below 4%. The mean values are between -2.55 % and 0.17%, thus monolithic and compositional versions perform really close. Surprisingly we get slightly better results for the compositional version, which may as well be caused by the concrete selection of case studies. We compared program versions constructed by a single skeleton composition step to monolithic ones or in the case of PSRS, we compared two composition steps to a single composition step. Our results indicate that skeleton composition using remote data performs competitive to monolithic skeletons for a single composition step. In the iteration chapter we compare compositional versions with many composition steps to persistent process systems constructed by the monolithic `iterUntil` skeleton or the compositional iteration framework:

	K-means: iterUntil vs i-framework	n-body: rec vs simpleIter[79]	FFT(ND-DH)	CG: iterUntil vs Iter	CG: rec vs Iter
min	-4.61%	-80.87 %	-49.23 %	-14.00 %	-6.19 %
max	2.98 %	17.20 %	29.44 %	-1.31 %	13.56 %
mean	-1.15 %	-13.10 %	-1.37 %	-5.87 %	-7.41 %
abs mean	2.64 %	22.39 %	-13.02 %	5.87 %	8.3 %

For the K-means example we compared the `iterUntil` skeleton to the iter framework and a compositional version **without remote data**. The compositional version with the sub-optimal communication structure performed dramatically worse. We summarise here only the other two versions which perform competitive. The *mean* value is only -1.15% and the *abs mean* is only 2.64%.
For the n-body example the results depend a lot on the input size. The recursive version performed better for bigger input sizes and the iteration framework for smaller sizes. Thus in this overview we leave out the results for the small input size 1000 because we do not want to overemphasise small input sizes. Yet these numbers show how much the results differ. The *abs mean* value (22.39 %) is around 9% bigger than the absolute value of the *mean* value (-13.1%). The *mean* result can be influenced a lot by the chosen input sizes. If we would include bigger input sizes, then the sign of the *mean* value would switch from $-$ to $+$.
The FFT ND-distributed homomorphism example displays runtimes which also differ a lot. However, we should not attach much weight to this case study as it was not agglomerated. Runtimes are dominated by parallel overhead, there are no speedups but only slowdowns. For the conjugate gradient example, the results are more homogeneous. The mean values are close to the abs mean values. The iter framework version performs best, but the mean differences with 5.87 % to the `iterUntil` skeleton version and with 7.41 % to the recursive version are still small.

79 without input size 1000

In our experiments the iteration framework performed competitive to the monolithic `iterUntil` skeleton. We observed varying performance gaps between the iteration framework and recursive skeleton compositions using remote data.

Another focus of this thesis 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 (usually monolithic skeletons) 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 de-serialisation of data to be communicated."[80]
The iteration framework integrates both, stable process systems and skeleton composition. Even though the skeleton framework is designed to be *flexible*, *extensible* and *easy to use*, it still restrains the liberty of the programmer compared to the simple recursive approach using remote data. Regarding performance: On the one hand, the iteration framework performed often better than the recursive versions. On the other hand, the recursive version clearly outperformed the iteration framework version of our n-body implementation for bigger input sizes. Thus, the programmer should choose carefully in each case between the poles of flexibility and performance.

"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 runtime system, 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."[81]

Future Work

We left the adaptation of Eden's cost model to our composition framework for future work. A further possibility would be to add the use of `fetch` and `release` automatically by a pre-processing step based on an adapted cost model. However, this might be quite

80 [DHLB16]
81 [DHLB16]

complicated as we can use `fetch` and `release` on many places: for the whole in- and output, for all elements of a list or stream, for tuple components or inside arbitrary nested data structures.

An important limitation of our framework is that remote data has to be used exactly once, a property which we inherited from dynamic channels. We could overcome this limitation by implementing a distributed garbage collection. There are common mechanisms to do this, e.g. distributed reference counting like in GUM [THM+96]. This includes a certain runtime overhead and would be implemented in the runtime system, which is conflicting with our goal to create a full library version of Eden.

Another important point regarding the further development of Eden's implementation is the integration of mechanisms supporting fault-tolerance. This is crucial for the productive use of Eden programs.

In addition to Eden, our skeleton composition framework based on remote data could be transferred to other languages, e.g. Cloud Haskell or Scala actors. Cloud Haskell introduces typed, receiver initiated channels to the actor model, similar to dynamic channels in Eden. Thus, it would be possible to implement remote data in Cloud Haskell in the same way we implemented it in Eden. We could further extend Scala actors by typed channels like in Cloud Haskell and transfer then remote data to Scala. The main difference between the Eden implementation of remote data and the transferred implementations would be that both Scala actors and Cloud Haskell do not use overloaded communication modes. Of course we could additionally add this feature to both libraries.

Appendices

APPENDIX A

Extended Code Reference

A.1 The `mwNestedAt` Skeleton

```
mwNestedAt :: forall t r. (Trans t, Trans r) ⇒
              Places →         -- tickets
              [Int] → [Int] → -- branching/prefetches per level
              ([t] → [r]) →    -- worker function
              [t] → [r]        -- tasks, results
mwNestedAt tickets@(_:_) (n:ns) (pf:pfs) wf ts
  = mwAt' childTickets n pf subMWs ts where
  (childTickets, restTickets) = splitAt n tickets
  subMWs = [mwNestedAt tickets' ns pfs wf| tickets' ← unshuffle n restTickets]
mwNestedAt _ _ _ wf ts = wf ts


mwAt' :: (Trans t, Trans r)
         ⇒ Places
         → Int → Int      -- #workers, prefetch
         → [[t] → [r]]    -- worker functions
         → [t] → [r]      -- what to do
mwAt' places np prefetch wf tasks = ress
  where
   (reqs, ress) = unzip ∘ merge ∘ tagF $ outs
   tagF         = zipWith zip [[i,i..] | i ← [0..np-1]]
   outs         = spawnFAt places wf inputs
   inputs       = distribute np (initReqs ++ reqs) tasks
   initReqs     = concat $ replicate prefetch [0..np-1]
```

A.2 The `allReduceRD` Skeleton Generalised for Arbitrary Input Sizes and Non-Commutative Functions

```
allReduceGenRD :: forall a. Trans a
               ⇒ (a → a → a)  -- ^reduce function
               → [RD a] → [RD a]
allReduceGenRD redF toReduce = reduced where
  steps = (ceiling ∘ logBase 2 ∘ fromIntegral ∘ length) toReduce
  (intermediates,reduced) = steps `pseq` unzip $ parMap (uncurry p) inp
  inp          = zip toReduce $ lazy $ buflyF $ transposeRt intermediates
  buflyF       = transposeRt ∘ shiftFlipF steps ∘ fillF steps

  p :: RD a → [Maybe (Both (RD a))] → ([RD a], RD a)
  p rdA rdAs = (rdAs'', res) where
    res      = release $ reduced !! steps
    rdAs''   = (releaseAll ∘ take steps ∘ lazy) reduced
    reduced  = scanl redF' a toReduce
    toReduce = fetchAll' rdAs'
    rdAs'    = zipWith (flip maybe Left) (map Right rdAs'') rdAs
    a        = fetch rdA

  --List encoding:
  -- Right: No Partner present, use value b without reduction
  -- Left: RD value comes from partner, then inner encoding:
  --       Right: Partner is positioned at the right hand side
  --       Left: Partner is positioned at the left hand side
  -- needed such that redF does not need to be commutativie
  redF' :: a → Either (Both a) a → a
  redF' _ (Right a) = a
  redF' a (Left (Right a')) = redF a a'
  redF' a (Left (Left a'))  = redF a' a

type Both a = Either a a

--custom fetchAll inside nested Eithers
fetchAll' :: Trans a ⇒ [Either (Both (RD a)) (RD a)] → [Either (Both a) a]
fetchAll' = runPA ∘ mapM fetchPA' where
  fetchPA' (Left (Left rda))  = do a ← fetchPA rda
                                   return $ Left $ Left a
  fetchPA' (Left (Right rda)) = do a ← fetchPA rda
                                   return $ Left $ Right a
  fetchPA' (Right rda)        = do a ← fetchPA rda
                                   return $ Right a

--Fill rows to the power of ldn with Nothing, map Just to the rest
fillF :: Int → [[a]] → [[Maybe a]]
fillF ldn ass = map fillRow ass where
  n = 2 ^ ldn
  fillRow as = take n $ (map Just as) ++ (repeat Nothing)

shiftFlipF :: Int → [[Maybe a]] → [[Maybe (Both a)]]
shiftFlipF ldn rdBss = zipWith shiftFlipRow [1..ldn] rdBss  where
  shiftFlipRow ldi rdBs = (shuffle ∘ flipAtHalfF ∘ unshuffle i) rdBs where
    i = 2 ^ ldi
    flipAtHalfF xs = let (xs1, xs2) = splitAt (i`div`2) xs
                     in map (map (fmap Right)) xs2 ++ map (map (fmap Left)) xs1
```

A.3 The gentlemanMulDemand Skeleton

```
gentlemanMulDemand :: forall a. (Num a,Trans a,NFData a)
                      ⇒ Matrix a → Matrix a → Matrix a
gentlemanMulDemand (Matrix ass) (Matrix bss)
  = torus f (zipTF (Matrix ass') (Matrix bss')) where
  -- function on torus elements
  f :: (a,a) → Stream a → Stream a  → (a, Stream a, Stream a)
  f (initR, initD) (Stream fromR) (Stream fromD)
   = (res, Stream streamL, Stream streamU) where
  -- stream demand code
    (res, streamL, streamU) = streamDemand streamL' streamU'
    streamL' = initR : take (n-1) fromR
    streamU' = initD : take (n-1) fromD

    streamDemand [] [] = error "empty␣streams␣should␣never␣happen"
    streamDemand (st1:sts1) (st2:sts2)
      = let prod = st1 * st2
            (res,sts1',sts2') = streamDemand' prod sts1 sts2
        in rnf prod `pseq` (res,st1:sts1',st2:sts2')
    streamDemand' sum [] [] = (sum,[],[])
    streamDemand' sum' (st1:sts1) (st2:sts2)
      = let prodSum = st1 * st2 + sum'
            (res,sts1',sts2') = streamDemand' prodSum sts1 sts2
        in rnf prodSum `pseq` (res,st1:sts1',st2:sts2')
    n = length bss

  -- pre-rotation
  ass' = zipWith leftRotIXs [0..] ass
  bss' = transpose $ zipWith leftRotIXs [0..] (transpose bss)
  leftRotIXs i xs = xs2 ++ xs1 where
    (xs1,xs2) = splitAt i xs
```

A.4 Automatic vs. Explicit Chunking: Two N-Body Versions

NBody with `simpleAllGatherIter` and *explicit chunking*

```
doStepsSimpleAllGatherIter :: Int → Int → [Body] → [Body]
doStepsSimpleAllGatherIter cSize n bs = bs' where
  dlBs  = releaseAll $ map (chunk cSize) $ unshuffle noPe bs
  dlBs' = iter (simpleLoopControl n) (simpleAllGatherIter t1 t2) dlBs
  bs'   = shuffle $ map unchunk $ fetchAll dlBs'

  t1 :: Stream [Body] → Stream [MassPoint]
  t1 = chunk cSize ∘ map getMassPoint ∘ unchunk
  t2 :: Stream [Body] → [Stream [MassPoint]] → Stream [Body]
  t2 bs = chunk cSize ∘ updateAll (unchunk bs) ∘ map unchunk
```

NBody with `simpleAllGatherIter` and *automatic chunking*

```
doStepsSimpleAllGatherIterC :: Int → Int → [Body] → [Body]
doStepsSimpleAllGatherIterC cSize n bs = bs' where
  dlBs  = releaseAll ∘ map (CL cSize) $ unshuffle noPe bs
  dlBs' = iter (simpleLoopControl n) (simpleAllGatherIter t1 t2) dlBs
  bs'   = shuffle ∘ map cList $ fetchAll dlBs'

  t1 :: CList Body → CList MassPoint
  t1 = fmap getMassPoint
  t2 :: CList Body → [CList MassPoint] → CList Body
  t2 (CL c bs) = CL c ∘ updateAll bs ∘ map cList
```

A.5 FFT with Distributed Homomorphism and Vectors

The vector based fft implementation including local definitions:

```
fft3_NDIterV :: Int   --base
             → Int   --n-Dims
             → B.Vector (Complex Double)
fft3_NDIterV base d = out ∘ dhNDFlatIterV base d h
                         ∘ inF $ B.generate n (+1) where ...
-- end fft3_NDIterV
  h = B.fromList ∘ fft3 base ∘ B.toList
  n = base^d

  inF :: B.Vector Int → VMatrix (RD (Complex Double, Int, Int))
  inF xs = chunkV base ∘ releaseAll $
           B.zipWith3 (,,) (bitReverse xs)
                           (B.generate n id)
                           (B.generate n $ const 1)
  bitReverse xs = invV n $ B.map fromIntegral xs

  out :: VMatrix (RD (Complex Double, Int, Int)) → B.Vector (Complex Double)
  out = B.map fst3 ∘ fetchAll ∘ B.concatMap id
  fst3 (a,_,_) = a
```

The `invV` implementation to permute a `Vector` in bit-reverse order using `backpermute` (relying on the omitted function `binReverse`):

```
invV :: Int → B.Vector a → B.Vector a -- permutes Vector in bit-reverse order
invV n xs = B.unsafeBackpermute xs (B.generate n $ flip binReverse n')
  where n' = realBinSize n
```

The vector based change-dim function including local definitions:

```
chdimVFlat :: Int   --base length
           → Int   --n-Dims
           → Int   --ChDim1
           → Int   --ChDim2
           → B.Vector a -- n-dim Grid (flatened) in
           → B.Vector a -- n-dim Grid (flatened) out
chdimVFlat base d cd1 cd2 v
  | cd1 > 0 && cd1 ≤ d && cd2 > 0 && cd2 ≤ d
    = if cd1 == cd2 then v
      else B.unsafeBackpermute v (B.generate (B.length v) partnerF)
  | otherwise = error "ChDim␣≤␣0␣||␣>␣n-Dims" ...
-- end chdimVFlat
  where pot1     = base^ (d-cd1)
        pot2     = base^ (d-cd2)
        partnerF ix | val1 == val2 = ix
                    | otherwise = ix - bigVal1 - bigVal2 + bigVal1' + bigVal2'
          where
            val1     = ix `div` pot1 `mod` base
            bigVal1  = val1 * pot1
            bigVal1' = val1 * pot2
            val2     = ix `div` pot2 `mod` base
            bigVal2  = val2 * pot2
            bigVal2' = val2 ∗ pot1
```

The change-dim control function `chdimVCF`:

```
chdimVCF :: Int              -- n-Dims
           → Int             -- base length
           → Int             -- act-dim
           → VMatrix a       -- n-dim Grid (flatened) in
           → VMatrix a       -- n-dim Grid (flatened) out
chdimVCF base d i = chunkV base
                  ∘ chdimVFlat base d (d-i) d
                  ∘ B.concatMap id
```

The `chdimsV` function to calculate $\overset{d}{\underset{i=1}{\circ}} chdim^{(i,i+1)}$:

```
chdimsV :: Int          --base length
         → Int          --n-Dims
         → VMatrix a -- n-dim Grid (flatened) in
         → VMatrix a -- n-dim Grid (flatened) out
chdimsV base d = chunkV base
                ∘ chdimsV' 1
                ∘ B.concatMap id where
  chdimsV' i v | i == d    = v
               | otherwise = chdimsV' (i+1) $ chdimVFlat base d (d-i) (d-i+1) v
```

List of Figures

Listings

Publications

- Mischa Dieterle, Thomas Horstmeyer, Rita Loogen, and Jost Berthold.
  Skeleton composition vs. stable process systems in Eden.
  Journal of functional Programming, Vol. 26, to appear. Cambridge University Press, 2016.
- Mischa Dieterle, Thomas Horstmeyer, Jost Berthold and Rita Loogen
  Iterating Skeletons - Structured Parallelism by Composition.
  In Ralf Hinze and Andy Gill (Eds.), IFL 2012, *24th Symposium on Implementation and Application of Functional Languages*, Revised Selected Papers, LNCS 8241, pages 18-36. Springer, 2013.
- Mischa Dieterle, Thomas Horstmeyer and Rita Loogen
  Skeleton Composition using Remote Data.
  In Carro, Manuel; Peña, Ricardo, editors, *Practical Aspects of Declarative Languages 12th International Symposium, PADL 2010*, LNCS 5937, pages 73-87, Madrid, Spain, January 2010. Springer, 2010. (**awarded most practical paper** of PADL'10)
- Mischa Dieterle, Jost Berthold and Rita Loogen
  A Skeleton for Distributed Work Pools in Eden.
  In Matthias Blume, Naoki Kobayashi and German Vidal (Eds.), *Functional and Logic Programming, 10th International Symposium*, FLOPS 2010, LNCS 6009, pages 337-353, Sendai, Japan, April 2010. Springer, 2010.
- Jost Berthold, Mischa Dieterle, and Rita Loogen
  Implementing Parallel Google Map-Reduce in Eden.
  In H. Sips, D. Epema, and H.X. Lin, editors, *Euro-Par 2009*, LNCS 5704, pages 990-1002, Delft, Netherlands. Springer, 2009.
- Oleg Lobachev, Jost Berthold, Mischa Dieterle, and Rita Loogen
  Parallel FFT using Eden Skeletons.
  In *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, Novosibirsk, Russia, LNCS 5698. Springer, 2009.
- Jost Berthold, Mischa Dieterle, Oleg Lobachev, Rita Loogen.
  Distributed Memory Programming on Many-Cores - A Case Study Using Eden Divide-&-Conquer Skeletons.
  In Gröspitsch, K.-E. ; Henkersdorf, A. ; Uhrig, S. ; Ungerer, T. ; Hähner, J, editors, *ARCS*

'09 - 22th International Conference on Architecture of Computing Systems 2009 - Workshop proceedings, Delft, The Netherlands, March 2009.

- Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe
  Hierarchical Master-Worker Skeletons.
  In Paul Hudak and David S. Warren, editors, *Practical Aspects of Declarative Languages*, LNCS 4902, pages 248 - 264. Springer, 2008.

Bibliography

[ADD04] Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47. Universität Münster, Germany, July 2004.

[AG03] Martin Alt and Sergei Gorlatch. Future-Based RMI: Optimizing compositions of remote method calls on the Grid. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 682–693. Springer-Verlag, August 2003.

[AG04] Martin Alt and Sergei Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2004.

[Alt07] Martin Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance.* PhD thesis, Universität Münster, July 2007.

[Arm07] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III. ACM, 2007.

[Aut07] Wiki Authors. Foldable and traversable. Haskell Wiki (https://wiki.haskell.org/Foldable_and_Traversable), 2007.

[BC02] Anne Benoit and Murray Cole. eSkel – The Edinburgh Skeleton Library, University of Edinburgh 2002. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.

[BC05] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In *Computational Science–ICCS 2005*, pages 764–771. Springer, 2005.

[BDL09] Jost Berthold, Mischa Dieterle, and Rita Loogen. Implementing parallel google map-reduce in Eden. In H. Sips, D. Epema, and H.X. Lin, editors, *Euro-Par 2009*, volume 5704 of *Lecture Notes in Computer Science*, pages 990–1002. Springer-Verlag, 2009.

[BDLL09] Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen. Distributed memory programming on many-cores - a case study using Eden divide-&-conquer skeletons. In K.-E. Grosspitsch, A. Henkersdorf, S. Uhrig, T. Ungerer, and J. Haehner, editors, *ARCS '09 - 22th International Conference on Architecture of Computing Systems*, Delft, The Netherlands, March 2009. VDE VERLAG.

[BDLP08] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical master-worker skeletons. In Paul Hudak and David S. Warren, editors, *Practical Aspects of Declarative Languages: 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 2008. Proceedings*, pages 248–264, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BDO$^+$95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

[Ber08] Jost Berthold. *Explicit and implicit parallel functional programming : concepts and implementation.* PhD thesis, Philipps-Universität Marburg, July 2008.

[BHBE12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The HaLoop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169–190, 2012.

[BL07a] Jost Berthold and Rita Loogen. Parallel Coordination Made Explicit in a Functional Setting. In Zoltán Horváth and Viktória Zsók, editors, *18th Intl. Symposium on the Implementation of Functional Languages (IFL 2006)*, LNCS 4449, Budapest, Hungary, 2007. Springer.

[BL07b] Jost Berthold and Rita Loogen. Visualizing Parallel Functional Program Runs – Case Studies with the Eden Trace Viewer. In *Proceedings of the International Conference ParCo 2007, September 2007, Central Institute for Applied Mathematics, Jülich, Germany*, 2007.

[Bre98] Silvia Breitinger. *Design and Implementation of the Parallel Functional Language Eden.* PhD thesis, Philipps-University of Marburg, Germany, 1998. Available at `http://archiv.ub.uni-marburg.de/diss/z1999/0142/`.

[CL07] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *Euro-Par 2007 Parallel Processing*, pages 72–81. Springer, 2007.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[DBL10] Mischa Dieterle, Jost Berthold, and Rita Loogen. A skeleton for distributed work pools in Eden. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS*, volume 6009 of *Lecture Notes in Computer Science*, pages 337–353. Springer, 2010.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGTY95a] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Functional skeletons for parallel coordination. In *EURO-PAR'95 Parallel Processing*, pages 55–66. Springer, 1995.

[DGTY95b] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 19–28, New York, NY, USA, 1995. ACM.

[DHBL13] Mischa Dieterle, Thomas Horstmeyer, Jost Berthold, and Rita Loogen. Iterating skeletons. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 18–36. Springer Berlin Heidelberg, 2013.

[DHL10] Mischa Dieterle, Thomas Horstmeyer, and Rita Loogen. Skeleton composition using remote data. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, PADL'10*, volume 5937 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2010. (**awarded most practical paper** of PADL'10).

[DHLB16] Mischa Dieterle, Thomas Horstmeyer, Rita Loogen, and Jost Berthold. Skeleton composition vs. stable process systems in Eden. *Journal of functional Programming*, 26, 2016. to appear.

[Die07] Mischa Dieterle. Parallel functional implementation of master worker skeletons. Diploma Thesis, Philipps-Universität Marburg, October 2007. (in german).

[DPP97] Marco Danelutto, Fabrizio Pasqualetti, and Susanna Pelagatti. Skeletons for data parallelism in p31. In *Euro-Par'97 Parallel Processing*, pages 619–628. Springer, 1997.

[dV12] Edsko de Vries. Communication patterns in Cloud Haskell (part 3): Map reduce. Blog Series (Web): http://www.well-typed.com/blog/73/, October 2012.

[EBPJ11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, New York, USA, 2011. ACM.

[ELZ$^+$10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of* *the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[ETKM12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proc. VLDB Endowment*, 5(11):1268–1279, jul 2012.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gen78] W Morven Gentleman. Some complexity results for matrix computations on parallel processors. *Journal of the ACM (JACM)*, 25(1):112–115, 1978.

[GHC15] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`, 1991–2015.

[GHSJ94] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, 4(4):477–488, 1994.

[Gor98] Sergei Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *J. of Supercomputing*, pages 85–97, 1998.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[Har15] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2015. second edition preview, available at `https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf`.

[HBL03] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic skeletons in template haskell. *Parallel processing letters*, 13(03):413–424, 2003.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73. Morgan Kaufmann, 1973.

[KC02] Herbert Kuchen and Murray Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.

[Kuc02] Herbert Kuchen. A skeleton library. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 620–629. Springer Berlin Heidelberg, 2002.

[Kuc07] Herbert Kuchen. The Münster Skeleton Library Muesli. Universität Münster, URL: `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/index.php`, 2007.

[LBDL09] Oleg Lobachev, Jost Berthold, Mischa Dieterle, and Rita Loogen. Parallel fft using Eden skeletons. In *10th Intl. Conference on Parallel Computing Technologies (PaCT)'09*, volume 5698 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[LLS$^+$93] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, Oct 1993.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[LOP$^+$03] Rita Loogen, Yolanda Ortega, Ricardo Peña, Steffen Priebe, and Fernando Rubio. Parallelism abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*, pages 95–128. Springer, 2003.

[Lov93] David B Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[LP10] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *PDP*. IEEE Computer Society, 2010.

[Mac03] David MacKay. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 2003. Chapter 20. An Example Inference Task: Clustering PDF, pages 284–292.

[Man80] Benoit B Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z$ (1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[MNPJ11] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *Haskell '11: 4th ACM Symposium on Haskell.* ACM, 2011.

[NAS94] NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division, 1994.

[Nik01] Rishiyur S Nikhil. *Implicit parallel programming in pH.* Oxford University Press, 2001.

[Pea62] Marshall C. Pease. An adaptation of the fast Fourier transform for parallel processing. *JACM*, 15(2):252–264, April 1962.

[Pel03] Susanna Pelagatti. Task and data parallelism in P3L. In *Patterns and skeletons for parallel and distributed computing*, pages 155–186. Springer, 2003.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL 96)*, volume 96, pages 295–308, 1996.

[PR01] R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *PPDP'01 — Intl. Conf. on Principles and Practice of Declarative Programming*, pages 187–198, Firenze, Italy, September 5–7, 2001.

[Pri06] Steffen Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *European Conference on Parallel Computing (Euro-Par) 2006*, LNCS 4128, Dresden, Germany, 2006.

[Qui94] Michael J. Quinn. *Parallel Computing (2Nd Ed.): Theory and Practice.* McGraw-Hill, Inc., New York, NY, USA, 1994.

[Rub01] Fernando Rubio. *Programación funcional paralele eficiente en Eden.* PhD thesis, Universidad Complutense de Madrid, October 2001.

[Saa96] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[THM$^+$96] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96.* ACM, 1996.

[ZGGW12] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. *JOGC*, 10:47–68, 2012.