

Philipps



Universität  
Marburg

# Explicit and Implicit Parallel Functional Programming: Concepts and Implementation

Dissertation  
zur  
Erlangung des Doktorgrades  
der Naturwissenschaften  
(*Dr. rer.nat.*)

dem

FACHBEREICH MATHEMATIK UND INFORMATIK  
DER PHILIPPS-UNIVERSITÄT MARBURG

vorgelegt von

Jost Berthold

aus Kassel

Marburg/Lahn, 2008

Vom Fachbereich Mathematik und Informatik  
der Philipps-Universität Marburg als Dissertation  
am **6.6.2008** angenommen.

Erstgutachter: Prof. Dr. Rita Loogen  
Zweitgutachter: Prof. Dr. Greg Michaelson  
Tag der mündlichen Prüfung am **16.6.2008**

## Zusammenfassung (deutsch)

Die vorliegende Arbeit beschreibt Konzepte zur parallelen Programmierung mit funktionalen Sprachen und deren Implementierung, insbesondere parallele Haskell-Dialekte und die Sprache Eden. Wir gehen der Frage nach, welche grundlegenden Koordinationskonstrukte und welcher Grad an expliziter Ausführungskontrolle nötig und nützlich für eine funktionale Implementierung paralleler Koordination sind.

In der heutigen Zeit von globaler Vernetzung und Mehrkernprozessoren wird die parallele Programmierung immer wichtiger. Dennoch sind nach wie vor Programmiermodelle verbreitet, die kaum von den Hardwareeigenschaften abstrahieren und sich daher zwangsläufig in technischen Details verlieren. Funktionale Sprachen erlauben aufgrund ihrer Abstraktion und mathematischen Natur, die gegenüber sequenziellen Programmen erheblich höhere Komplexität paralleler Programme zu erfassen, sie ermöglichen ein abstrakteres Nachdenken über parallele Programme. Dabei taucht unvermeidlich die oben formulierte Leitfrage auf, zu welchem Grad explizite Kontrolle der Ausführung nötig und nützlich ist, um effiziente parallele Programme zu schreiben, wenn diese wiederum abstraktere Kontrollkonstrukte implementieren und insbesondere in der sog. skelettbasierten Programmierung, welche gängige Muster der Parallelverarbeitung abstrakt als Funktionen höherer Ordnung beschreibt.

Wir beschreiben unsere Implementierung für die Sprache Eden, welche hierarchisch in Schichten organisiert ist. Die unterste, direkt implementierte Schicht stellt nur sehr einfache Primitive für Parallelverarbeitung bereit, komplexere Kontrollkonstrukte werden in der funktionalen Sprache Haskell implementiert.

Neben der Implementierung von Eden stellen die implementierten Primitive für sich genommen bereits Kontrollkonstrukte zur Parallelverarbeitung dar. Aus der Implementierung abgeleitet wird die funktionale Sprache EDI ('ED'en-T'implementierungssprache) vorgeschlagen, die auf niedrigem Abstraktionsniveau Haskell um *orthogonale, grundlegend notwendige* Konstrukte zur Koordination paralleler Berechnungen erweitert: Auswertungskontrolle, Nebenläufigkeit, Prozesszeugung, Kommunikation und Information über verfügbare Ressourcen.

Die grundlegende Systemunterstützung von EDI und das Implementierungskonzept lassen sich mit nur geringen Modifikationen auf andere Implementierungen (und Berechnungssprachen) übertragen. Aufgrund seiner Flexibilität und der funktionalen Basis bietet unser Ansatz großes Potenzial für modellgestützte Ansätze zur automatischen Verwaltung paralleler Berechnungen und für die Verifikation von Systemeigenschaften. Wir beschreiben das allgemeine Design eines generischen Systems zur hierarchischen Implementierung paralleler Haskell-Erweiterungen, eine Prototyp-Implementierung für adaptive Scheduling-Konzepte und eine Machbarkeitsstudie für virtuellen globalen Speicher.

Anwendungsgebiet für die Evaluation der Sprache EDI ist die Implementierung abstrakterer Konstrukte, insbesondere paralleler Skelette, auf die wir uns

---

in einem weiteren Teil der Dissertation konzentrieren. Skelette beschreiben parallelisierbare Algorithmen und parallele Verarbeitungsmuster abstrakt als Funktionen höherer Ordnung und verbergen ihre parallele Implementierung. Somit bieten sie ein (gegenüber der Programmierung mit Eden oder EDI) höheres Abstraktionsniveau, entziehen aber dem Programmierer die explizite Parallelitätskontrolle. Für einen Vergleich der Ausdrucksstärke und Pragmatik werden exemplarisch Implementierungen für verschiedene parallele Skelette in der Sprache Eden und ihrer Implementierungssprache EDI beschrieben. Wir untersuchen neben einschlägigen Vertretern dieser “algorithmischen Skelette” eine andere Art Skelette, welche anstelle der Algorithmik die Interaktion von Prozessen in regulärer Anordnung beschreiben und für die wir den Begriff *Topologieskelette* geprägt haben.

Durch die zusätzlichen nicht-funktionalen Konstrukte in Eden haben Eden und EDI im Ergebnis die gleiche Ausdrucksstärke, wobei aber die abstrakteren Eden-Konstrukte essenzielle Seiteneffekte und Nebenläufigkeit verbergen. Durch die in EDI (im Gegensatz zu Eden) explizite Kommunikation lassen sich Skelett-Implementierungen besser lesen und an besondere Bedürfnisse anpassen. Die explizitere Sprache EDI findet ihr originäres Anwendungsgebiet in der Skelett-Implementierung, -anpassung und -optimierung. Demgegenüber ist ein klarer Pluspunkt von Eden die automatische (durch eine Typklasse und geeignete Überladung definierte) Stream- und Tupelkommunikation. Letztere kann in EDI unverändert übernommen werden, sollte hier aber, gemäß der EDI-Philosophie, explizit und gezielt vom Programmierer eingesetzt werden.

Ergebnisse im Überblick:

- die Definition der Sprache EDI, Haskell mit expliziten Prozesskontroll- und Kommunikationskonstrukten. EDI erweitert Haskell um *orthogonale, grundlegend notwendige* Konstrukte zur Koordination paralleler Berechnungen: Auswertungskontrolle, Nebenläufigkeit, Prozesserzeugung, Kommunikation und Information über verfügbare Ressourcen. Einsatzgebiet der Sprache ist die Implementierung komplexerer Koordinationskonstrukte.
- eine strukturierte Implementierung von Eden, deren Konzepte in einem allgemeineren Kontext tragfähig sind.
- das Design und eine Prototypimplementierung für ein generisches System zur Implementierung abstrakterer Kontrollkonstrukte, insbesondere für eine automatisierte dynamische Steuerung paralleler Berechnungen.
- neu entwickelte bzw. alternative Skelett-Implementierungen für algorithmische und Topologie-Skelette, für einen Vergleich von EDI und Eden.
- der exemplarische Nachweis, dass funktionale Sprachen ein adäquates Abstraktionsniveau bieten, um Strukturen der Parallelverarbeitung zu erfassen.

## Abstract

This thesis investigates the relation between the two conflicting goals of explicitness and abstraction, for the implementation of parallel functional languages and skeletons. Necessary and useful coordination features for implementing parallel coordination in a functional implementation language will be identified, leading to the proposal of a Haskell extension for explicit low-level coordination, and to a concept of structuring implementations for parallel functional languages in layers of increasing abstraction.

The first main part concentrates on implementation techniques and requirements. We are describing the layered implementation of the parallel functional language Eden, pointing out advantages of its layer structure and deriving the coordination features of the proposed explicit low-level language, named EDI. Subsequently, the presented implementation concept is generalised to the design and a prototype implementation of a generic parallel runtime system for management of parallel functional computations, major parts of which are encoded in Haskell.

In a second main part, we concentrate on implementations of parallel skeletons, thereby investigating expressiveness and pragmatics of the proposed low-level language EDI in comparison to the language Eden. Exemplarily, a range of implementations is presented for data parallel skeletons implementing map, map-reduce, and the Google-MapReduce programming model. Furthermore, we present and discuss a new skeleton category: *topology skeletons*, which describe interaction and communication patterns in regularly structured process networks.

In a broader context, the implementation concepts and skeleton implementations which we present underline that functional languages provide a suitable abstraction level to reason about parallel programming and to circumvent its complexity.

## Thank You!

First of all, I would like to thank my supervisor, Prof. Dr. Rita Loogen for her support and inspirations over the last years, which gave me the chance to develop the ideas I am going to present.

I appreciate the opportunity to use the Beowulf clusters of the Heriot-Watt University for my series of tests. Personal acknowledgements go to people who I enjoyed working with and who provided important impulses for my research: Hans-Wolfgang Loidl (formerly Heriot-Watt, now LMU Munich), Phil Trinder and Greg Michaelson (Heriot-Watt), Kevin Hammond (St.Andrews), Abyd Al Zain (Heriot-Watt), and my new colleague Mischa Dieterle. I also would like to thank many nice people who are working in related areas and which I have met at conferences and workshops. It has been a pleasure to meet you, and some of you have become more than just professional colleagues. Furthermore, I may thank other good friends and former friends for the personal support and new perspectives they offered. You know who you are!

Last, but not least, hugs to my parents for their support, especially to my mother, who repeatedly struggled with my computer science slang and my english skills – and who would surely spot a number of flaws on this page.

# Contents

<b>I</b>	<b>Introduction and background</b>	<b>1</b>
1	Introduction	3
2	<b>(Why) parallel functional programming</b>	<b>9</b>
2.1	On parallel programming . . . . .	9
2.1.1	Basics: Hardware and communication. . . . .	9
2.1.2	On parallel programming models . . . . .	10
2.2	Advantages of functional approaches . . . . .	12
2.3	Parallel Haskell classified . . . . .	12
2.4	The language Eden . . . . .	15
<b>II</b>	<b>High-level implementation concepts</b>	<b>19</b>
3	<b>A layered Eden implementation</b>	<b>21</b>
3.1	Implementation of Eden . . . . .	22
3.1.1	Layer structure of the implementation . . . . .	22
3.1.2	Parallel runtime environment . . . . .	23
3.1.3	Primitive operations . . . . .	26
3.1.4	Eden module: Language features . . . . .	27
3.1.5	Simulations for Eden and its implementation . . . . .	31
3.2	Edi: The Eden implementation language . . . . .	34
3.2.1	Degree of control: A compromise . . . . .	34
3.2.2	The EDI language . . . . .	36
3.2.3	Relation to other Haskell extensions . . . . .	38
3.3	Generalising the implementation concept . . . . .	38
4	<b>A generic runtime environment for parallel Haskell</b>	<b>41</b>
4.1	Introduction . . . . .	41
4.2	Design aims of the generic RTE ARTCoP . . . . .	44
4.2.1	Simplest kernel . . . . .	44
4.2.2	Genericity . . . . .	44
4.2.3	Multi-level system architecture . . . . .	44

4.2.4	High-level scheduler control . . . . .	46
4.3	Configurable Haskell scheduler framework . . . . .	46
4.4	Explicit communication . . . . .	49
4.5	System monitoring . . . . .	50
4.6	Example: Adaptive scheduling in GpH . . . . .	51
4.6.1	Hierarchical task management . . . . .	51
4.6.2	Adaptive load distribution mechanisms . . . . .	54
4.7	Feasibility study: Virtual shared memory management in Haskell	56
4.7.1	Virtual shared memory in GpH . . . . .	56
4.7.2	Global address management in Haskell . . . . .	58
4.7.3	Haskell heap access from inside Haskell . . . . .	61
4.7.4	Summary . . . . .	69
<b>5</b>	<b>Visualising Eden program runs: EdenTV</b>	<b>71</b>
5.1	Motivation . . . . .	71
5.2	How EdenTV works . . . . .	72
5.3	Related work . . . . .	76
5.4	Simple examples . . . . .	77
5.4.1	Irregularity and cost of load balancing . . . . .	77
5.4.2	Lazy evaluation vs. parallelism . . . . .	78
5.4.3	Process placement (implementation) bug . . . . .	81
<b>III</b>	<b>Parallel programming with skeletons</b>	<b>83</b>
<b>6</b>	<b>Skeleton programming and implementation</b>	<b>85</b>
6.1	Context: High-level parallel programming . . . . .	85
6.2	Parallel programming with skeletons . . . . .	86
6.2.1	The skeleton idea . . . . .	86
6.2.2	A skeleton typology . . . . .	88
<b>7</b>	<b>Problem-oriented skeletons: Map and reduce</b>	<b>91</b>
7.1	Data parallel mapping . . . . .	91
7.2	Parallel map-and-reduce . . . . .	97
7.3	The “Google MapReduce” skeleton . . . . .	99
7.3.1	MapReduce functionality . . . . .	100
7.3.2	Parallelisation potential . . . . .	102
7.3.3	Example applications . . . . .	106
<b>8</b>	<b>Structure-oriented skeletons</b>	<b>111</b>
8.1	Process pipelines . . . . .	113
8.1.1	Uniform type, implementation variants . . . . .	113
8.1.2	Heterogeneous pipeline stages, and including I/O . . . . .	117



8.2	Process ring skeletons . . . . .	120
8.3	Nesting skeletons . . . . .	128
8.3.1	A toroid topology created as a nested ring . . . . .	128
8.3.2	Two versions of a parallel pipeline . . . . .	134
<b>IV</b>	<b>Conclusion</b>	<b>139</b>
<b>9</b>	<b>Conclusions and future work</b>	<b>141</b>
9.1	Summary of contributions . . . . .	141
9.2	Discussion and future work . . . . .	142
	<b>Appendix</b>	<b>147</b>
<b>A</b>	<b>Bibliography and list of figures</b>	<b>147</b>
	Bibliography . . . . .	147
	List of Figures . . . . .	159
<b>B</b>	<b>Code collection</b>	<b>161</b>
B.1	Implementation . . . . .	161
B.1.1	Eden module: Eden.hs . . . . .	161
B.1.2	Primitives wrapper: ParPrim.hs . . . . .	168
B.1.3	Primitives simulation using Concurrent Haskell . . . . .	170
B.2	Skeletons . . . . .	174
B.2.1	Google MapReduce Skeleton, optimised EDI version . . . . .	174
B.2.2	PipeIO.hs, implementation of multi-type IO-pipelines . . . . .	177
B.2.3	EdiRing.hs: EDI ring skeletons . . . . .	180
B.2.4	PipeRings.hs: definition of a ring using a pipeline skeleton . . . . .	184
<b>C</b>	<b>Formalien (deutsch)</b>	<b>186</b>
	Erklärung des Verfassers (deutsch) . . . . .	186



# Part I

## Introduction and background



# Chapter 1

## Introduction

### **The renaissance of parallel programming**

In today's computer and software development, parallel and concurrent programming is becoming more and more relevant, essentially driven by two evolutions. Local area and wide/global area networks have been developed and consolidated in the 90s, and today are a standard infrastructure for science, industry, and even private users. The recent "Grid" trend [FKT01] has consolidated and standardised these big and unreliable networks to make them usable, and has identified possible applications and shortcomings. As a second evolution, roughly during the last 3 years, single core CPU development is getting closer and closer to a hard limit, for physical and thermal reasons. Only by more efficient chip design and novel hardware techniques can CPU speed be raised further. On the other hand, with advances in chip design and chip production, more and more functionality can be concentrated on smaller and smaller chips. As a consequence, multi-core processors are on the rise, having already become the standard even for consumer computers. And while, today, 8-core CPUs commercially constitute the upper end, efforts concentrate on considerably increasing the amount of processing elements integrated into a CPU, to hundreds and thousands [Chine, All07].

However, writing efficient and correct parallel programs is far more complex than sequential programming. The mainstream in software engineering has long ignored this, neglecting alternative paradigms and conceptual work. To our knowledge, methods and tools for explicitly modeling parallel execution are far from standard in the industry yet, and the emerging new multicores differ substantially from older parallel machines. There is a pressing need to investigate and establish new programming paradigms in the mainstream, suitable for programming parallel machines, well-scaling and efficiently using today's hardware and network technology. Parallel programming needs more conceptual understanding.

## Declarative languages for parallelism

Declarative programming languages have long been a research topic, and have fundamental advantages over imperative languages. Precious ideas emerging from declarative language research have found their way into imperative languages, e.g. type safety by a strong type system, easy code reuse by generic container types (parametric polymorphism), and others.

Declarative programs are often better to maintain and accessible to formal reasoning, and are thus a promising scientific setting to clarify the essence of parallel coordination and to circumvent its complexity. Sound mathematical foundations and their abstract, problem-oriented nature make them amenable to formal reasoning, and to distilling out coordination structure, programming model and algorithm structure of a parallel program. To prove this claim, and to make a contribution to advancing high-level parallel programming, the research presented in this thesis investigates *skeletons* [Col89, RG03], ready-made efficient parallel implementations for common patterns of parallel algorithms. From the functional programming perspective, skeletons are nothing but higher-order functions with a hidden parallel implementation, and constitute a well-established idea in declarative research communities.

## Implementation of parallel coordination

While many approaches to skeleton programming are based on a fixed, established set of efficient skeletons (see e.g. [PK05, Ben07]), some parallel functional languages reveal to the programmer their potential to define new skeletons, or to easily create them by composition [MSBK01, RG03]. However, those languages necessarily provide suitable *lower-level* coordination features and *more explicit* control than the resulting skeleton will offer. The task of *implementing* a skeleton in a high-level language typically deals with a compromise between high abstraction and specific operational control of execution. And in functional programming, the border between library and language is fluid [BKPS03]: Skeleton implementation work can be rightly considered as *developing high-level languages* for parallel coordination (in fact, even independent of the underlying *computation language*). Considering more generally the implementation aspects for parallel functional languages, similar questions arise. Step by step, the implementation needs to bridge the “large gap” to hardware or middleware of the parallel machine.

The work at hand investigates the relation between the two conflicting goals of explicitness and abstraction, for the implementation of parallel functional languages and skeletons, under the following question:

*What coordination features and what degree of explicitness is necessary and useful for implementing parallel coordination in a functional implementation language?*

---

To answer this question, we propose a *Haskell extension for low-level coordination*, suitable to implement different kinds of more abstract and complex coordination features in Haskell modules. This low-level implementation language provides: (a) evaluation control (b) concurrency and remote task creation, (c) support for communication between tasks, (d) location- and resource-awareness. These four points constitute *orthogonal and general requirements* of parallel coordination.

Using the proposed low-level language, implementations of more complex and implicit languages for parallelism can be structured in strictly separated layers. The low-level language serves as a basic layer, and kernel parts underneath can be kept small in favour of library implementations, thereby retaining the fundamental advantages of functional languages. The name of the proposed language is EDI (EDen Implementation language), because our approach issued from implementation concepts for the explicit parallel functional language Eden [LOMP05], which provides explicit process control, implicit concurrency, automatic parent-child communication with stream or single-data mode, and non-functional extensions for explicit communication and reactive systems.

The approach is useful and of general interest: Using the primitives allows one to rapidly prototype, and to reason about requirements for, implementations of other more complex and implicit coordination languages and, in this sense of languages, for parallel skeleton implementation.

Skeleton implementation constitutes another testbed to investigate expressiveness and pragmatics of our low-level implementation language. We will point out its advantages and restrictions in comparison with the, equally very explicit, language Eden, by discussing implementation variants of algorithmic skeletons and *topology skeletons*, a term we have coined for skeletons which capture process interaction in regular structures.

The major contributions of this research are:

- We identify the *orthogonal and general requirements* of parallel coordination by defining a functional low-level language EDI.
- We point out the general applicability of the concepts identified, by explaining implementation concepts for Eden and for a more general system,
- and by comparing skeleton implementations in EDI and Eden.
- Additionally, we propose the notion of *topology skeletons*.
- In the broader context, our work underlines that functional languages provide a suitable abstraction level to reason about parallel programming.

## Plan of the thesis

The thesis is structured as follows: In the first part, we briefly explain the context and background of our work, high-level parallel programming, and especially parallel Haskell dialects and their implementation.

Part II treats implementation concepts for the parallel functional language Eden and an approach to a more generalised implementation concept. The implementation of Eden coordination constructs is explained in detail, and the language EDI (EDen Implementation Language) is proposed, which provides more explicit constructs for communication between parallel computations. Subsequently, we study the underlying implementation concept and its potential as a more general platform for high-level parallel language implementation, presenting design and partial implementation of a prospected generalised prototype system for high-level parallelism. Furthermore, in Chapter 5, we explain concepts and usage of *EdenTV*, a graphical tool to trace and visualise runtime behaviour of Eden programs (which is closely related to the implementation work for Eden).

The other main part (III) focuses on *skeleton programming and implementation*, as a vehicle for comparison between Eden and its implementation language EDI, considered as a language of its own. In Chapter 7, we discuss a range of different implementations for parallel `map` computations (a common transformation applied to all data of a container type), possibly followed by a reduction operation on the results (“map-and-reduce”, for which we also discuss the variant known as Google-MapReduce [DG04]). Chapter 8 discusses a different concept of skeletons: *Topology Skeletons*, which capture interaction and communication in a regularly structured process network. We investigate implementation variants for process pipelines, process rings, and process toroids. Recursive and non-recursive implementations are compared, and we again compare the expressiveness of Eden and EDI for skeleton implementation. Furthermore, we discuss questions related to skeleton *nesting* for two examples.

Conclusions in the last part summarise our results and point out interesting future work. The developed Haskell modules we discuss are reproduced in the Appendix, unless the source code of the thesis is itself compilable.

## References

Part of the material we are going to present in this thesis has been presented and published at workshops and conferences during the last years. The results related to Eden implementation work have been presented and published in several workshops and conference proceedings: [BLPW02] and [BKL<sup>+</sup>03] explain a previous Eden implementation, which already incorporated considerable parts in Haskell. [BL07a] presents the new implementation and a preliminary version of the implementation language EDI, as well as a brief comparison of EDI and Eden for skeleton programming. Our publications related to the idea of gener-



---

alised implementation are [Ber04], which sketches an early vision, and [BLAZ08], which presents idea and design of ARTCoP, and then discusses the scheduling framework and the example. Material related to EdenTV has been published in our conference contribution [BL07b], which contains similar EdenTV implementation concepts and case studies. Several publications contain results related to topology skeletons and their implementation: [BL05a, BL08] describe and quantify the benefit of explicit communication channels, [BL05b] discusses recursive implementations for regular topologies, and [BDLP08] (to which we occasionally refer) discusses concepts for nesting master-worker skeletons.

- [BDLP08] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical Master-Worker Skeletons. In P. Hudak and D. Warren, editors, *PADL'08*, Springer LNCS 4902, San Francisco, USA, January 2008.
- [BLAZ08] Jost Berthold, Hans-Wolfgang Loidl, and Abyd Al-Zain. Scheduling light-weight parallelism in ARTCoP. In P. Hudak and D. Warren, editors, *PADL'08*, Springer LNCS 4902, San Francisco, USA, January 2008.
- [BL07b] Jost Berthold and Rita Loogen. Visualizing parallel functional program executions: Case studies with the Eden Trace Viewer. In C. Bischof et al., editors, *Proceedings of ParCo 2007*, volume 38 of *NIC*, Jülich, Germany, September 2007.
- [BL07a] Jost Berthold and Rita Loogen. Parallel coordination made explicit in a functional setting. In Z. Horváth and V. Zsóka, editors, *18th IFL 2006*, Springer LNCS 4449, Budapest, Hungary, 2007. (**awarded best paper** of IFL'06).
- [BL05b] Jost Berthold and Rita Loogen. Skeletons for recursively unfolding process topologies. In G. R. Joubert et al., editors, *Proceedings of ParCo 2005*, volume 33 of *NIC*, Malaga, Spain, August 2005.
- [BL08] Jost Berthold and Rita Loogen. The impact of dynamic channels on functional topology skeletons. *Parallel Processing Letters (World Scientific Publishing Company)*, 18(1):101–115, 2008.
- [BL05a] Jost Berthold and Rita Loogen. The impact of dynamic channels on functional topology skeletons. In A. Tiskin and F. Loulergue, editors, *HLPP 2005, Coventry, UK*, 2005.
- [Ber04] Jost Berthold. Towards a generalised runtime environment for parallel Haskell. In M. Bubak et al., editors, *Computational Science — ICCS'04*, number 3 in LNCS 3038, Krakow, Poland, 2004. Springer.
- [Ber03] Jost Berthold. Dynamic Chunking in Eden. In Phil Trinder, Greg Michaelson, and Ricardo Peña, editors, *15th IFL 2003*, Springer LNCS 3145, Edinburgh, UK, 2003.
- [BKL<sup>+</sup>03] Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp. High-level Process Control in Eden. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *EuroPar 2003 – Parallel Processing*, Springer LNCS 2790, Klagenfurt, Austria, 2003.
- [HBL03a] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic skeletons in Template Haskell. *Parallel Processing Letters (World Scientific Publishing Company)*, 13(3):413–424, 2003.
- [HBL03b] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic Skeletons in Template Haskell. In F. Loulergue, editor, *HLPP 2003, Paris, France*, 2003.



# Chapter 2

## (Why) parallel functional programming

Or should we add “...matters” in analogy to Hughes [Hug89]? Some computer science publications have adopted the *Why...matters* pattern, and thereby underlined the relevance of that famous seminal paper “Why functional programming matters”. And so does the overview edited by Hammond and Michaelson [HM99], discussing various theoretical and practical aspects of parallel functional programming in general. Somewhat more specifically parallel *Haskells* are presented in the overview by Trinder et al. [TLP02].

The main parts of this thesis treat parallel Haskell implementation issues and skeleton programming techniques. In this chapter, we will briefly give some background of parallel and parallel functional programming.

### 2.1 On parallel programming

Parallel programming is hard, much harder than writing correct sequential programs. But what makes parallel programming hard? To justify what otherwise is nothing but an often-cited commonplace, we have to look at how parallel programs are written.

#### 2.1.1 Basics: Hardware and communication.

To classify parallel machines and parallel processing, Michael Flynn was the first to propose a classification [Fly66] in the late 60s, which is still widely cited. Flynn adopts a stream-based data-processing view, and he divides parallel hardware into the following instruction parallelism and data accesses. The categories are: SISD (Single Instruction, Single Data) – the classical definition of a uniprocessor; SIMD (Single Instruction, Multiple Data) – vector/array processor; MISD (Multiple Instruction, Single Data) – of no practical interest; and MIMD (Multiple

Instructions, Multiple Data) – which covers the range of multiprocessor systems. This very coarse system (which even contains the redundant MISD class) is the first of only a few systems that have been widely accepted.

Today, we hardly find special-purpose vector processors any more; dominating hardware architectures are parallel machines where the processing elements (PEs) simply are the general-purpose processors found in single computers – and increasingly several PEs on one chip. MIMD has become the only model of interest in practical use (all 500 supercomputers of the world are MIMD architectures [Top]). Subsequent taxonomies introduced the loosely coupled and tightly coupled categories for MIMD architectures, taking into account the interconnection network and thereby capturing a highly relevant machine characteristic. Communication between the PEs of a parallel computer can be realised by shared memory, or by exchanging messages over an interconnection network. In addition, modern (multicore) processors commonly organise the shared memory in a memory hierarchy with several levels of cache memory, where the cores are equipped with on-chip 1st-level cache, but subsets share a common lower-level cache. Memory consistency is managed in hardware for these cache architectures, but lead to non-uniform memory latencies in an extended memory hierarchy, which Flynn’s simple model does not take into account (as well as its successors).

Consequently, basic issues of parallel programming are either synchronisation between concurrent memory accesses to shared data, avoiding inconsistent read/write operations, or else, the fact that message passing is inherently non-deterministic and easily introduces errors due to race-conditions. Parallel programming models build on the basic infrastructure to more or less hide synchronisation and communication issues and to provide a more abstract programming model.

### 2.1.2 On parallel programming models

Useful classifications of parallel programming models refer to the *degree of explicitness* provided by the respective programming language, or paradigm, for instance Skillicorn and Talia [ST98], who subdivide along several aspects: communication, division, mapping, and synchronisation of parallel subcomputations, and whether the parallelisation is statically determined (at compile time) or dynamic.

However, the predominating programming model in practice today is still closely connected to the underlying hardware characteristics: message-passing using MPI [MPI97], or shared-memory parallelism using OpenMP [DM98] or threading libraries [IEE92, Int]. So, parallel programming is mostly done by means of libraries which *facilitate* and *standardise*, but do not *abstract from* the basic infrastructure explained.

And basically *this* is why “parallel programming is hard”. When every detail of parallelism is left to the programmer, the program complexity becomes excessive, and a large fraction of code deals with purely technical issues.

During decades of research in high-level parallel programming, a range of more abstract models have been developed, but did not find broad acceptance. Although industry becomes more and more interested today, we hardly find examples where modern high-level techniques are in practical use. Far from it, commercial efforts sometimes concentrate on porting the hardware-oriented low-level view and model to different architectures. For instance, Intel’s Cluster-OpenMP [Hoe06] enables to apply a shared memory model to distributed memory architectures, and claims that unmodified OpenMP-programs can be used. The different hardware architecture will, of course, lead to a dramatic performance loss for programs not optimised for data locality, but optimisation is completely left to the programmer.

Of particular interest for the remainder of the thesis are the following approaches, which are therefore mentioned and briefly described in their characteristics here.

The Cluster-OpenMP mentioned is an example of **virtual shared memory** (VSM). VSM can be implemented on top of message-passing, to abstract from the real interconnection network. VSM reduces communication and synchronisation issues to simpler synchronisation problems in a shared-memory environment, and thereby facilitates parallel programming for different platforms. Based on well-established techniques (global addresses and weighting), data sharing, consistency of distributed data and memory management can be automatically managed by a library implementation (we will explain technical details later on). However, low-level programming in a VSM model may lead to poor program performance, because the transparent access to remote memory cells hides data dependencies and latencies.

A related approach is the **tuple-space** paradigm, most commonly known as its realisation in the **Linda** coordination language [CG92, CG90]. This model differs from plain shared-memory in that the PE’s local memory is cleanly separated from the shared data in the tuple space, and the latter is only accessed via special (library) operations. However, both approaches cannot be considered as a substantially new paradigm; programming essentially follows the shared-memory paradigm and either the programmer or a complex runtime system needs to adequately synchronise concurrent memory accesses.

**Data parallelism** is a fundamentally different and more abstract model, based on special container data types (lists, arrays etc.) and operations with a (hidden) parallel implementation. Operations on these data containers provide inherent parallelism which can be exploited by that transparent implementation. Parallelism can even be introduced automatically (an approach which is, however, said to be “comprehensively dead” [HM99]), or when the programmer explicitly chooses to use a parallel operation for data in a container. Mapping parallel operations onto PEs, data transfer, and synchronisation are completely hidden in the implementation.

**Algorithmic skeletons** [Col89] (already mentioned before) constitute another high abstraction for parallel programming. An algorithmic skeleton captures a common algorithmic structure with inherent parallelism, or more generally, a common pattern of parallel programming, as a higher-order function. Real parallel algorithms can be programmed merely by supplying parameter functions which execute the concrete computation inside the fixed algorithmic pattern. The skeleton approach and its functional nature are of primary interest for the work we present, and we will discuss skeletons in detail in our main part III.

## 2.2 Advantages of functional approaches

As Hammond and Michaelson summarise [HM99], the advantages of functional programming underlined by Hughes carry over to the parallel world. Programs are easy to read and understand, program construction and code reuse are simplified, and thereby program transformations, optimisations and formal reasoning about program behaviour is easier.

Some additional advantages specific to parallelism can be added: Specific benefits of *parallel* functional programming, versus imperative parallelism, are that the absence of side-effects makes data-dependencies and inherent parallelism obvious: Programs can be easily parallelised and analysed. Furthermore, results of (purely) functional parallel programs are determined, just as sequential functions will always produce one and the same output for the same input. The particular evaluation order does not matter (Church-Rosser theorem), and purely functional computations can exploit parallelism inherent in the reduction semantics [HM99]. Moreover, a parallel program is usually closely connected to a sequential one, which is useful for development and testing.

Last but not least, functional languages with support for higher-order functions are the languages of choice to express algorithmic skeletons. Put another way, functional languages enable to abstractly describe common parallelisation patterns without getting lost in technical details or particularities of the concrete algorithm. In all, irrespective of the concrete programming model, the high abstraction provided by functional languages makes them suitable languages to conceptually describe parallelism, in an executable specification – which, however, will not instantly deliver maximum performance.

## 2.3 Parallel Haskell classified

Similar to Skillicorn’s classification of programming models, parallel functional languages are often classified along their *explicitness*, ranging from completely implicit to completely explicit coordination. As one might expect, the predominant category is a mid-level of “controlled parallelism” [HM99], where program-

mers specify parallelism, while details are left to the language implementation. However, the understanding of explicitness varies, and especially the interesting mid-level remains vague and open to interpretation. We will follow the categorisation suggested by Loogen in her relevant chapter [Loo99], and illustrate the classification by referring to relevant Haskell dialects.

### Implicit parallelism

In functional languages, it is possible, and has been carried out, to find and exploit the parallelism which is inherent in the reduction semantics. The Haskell example is *parallel Haskell* (pH) [NAH<sup>+</sup>95, AAA<sup>+</sup>95], in which Haskell’s lazy evaluation is changed to eager evaluation for performance. However, completely implicit approaches turned out to be less useful than giving programmers (limited) execution control.

### Indicating parallelism

Using annotations or combinators, inherent parallelism in a functional program may be indicated by the programmer, to inform the compiler or runtime system about whether an independent computation *should* be done in parallel. The programmer annotates a program with (semantically transparent) compiler directives and thereby decides, or suggests<sup>1</sup>, a parallelisation. Examples are data parallel languages, which use special bulk types and operations with parallel implementation (such as the data parallel Haskell NEPAL [CKLP01, CLJ<sup>+</sup>07]), as well as the `par`, `seq` combinators of Glasgow-parallel Haskell (GpH) [THM<sup>+</sup>96]. GpH is described in more detail here, because we will refer to its implementation later.

**Glasgow parallel Haskell (GpH)** [THM<sup>+</sup>96] is a well-known parallel dialect of Haskell investigated since the 90’s. The overall paradigm of GpH is semi-implicit data and task parallelism, following annotations in the source program. In every definition, subexpressions can be marked as “suitable for parallel evaluation” by a `par`-expression in the overall result. The coordination construct `par` takes 2 arguments and returns the second one after recording the first one as a “spark”, to be evaluated in parallel. An idle processor can fetch a spark and evaluate it. The built-in `seq` is the sequential analogon, which forces evaluation of the first argument before returning the second one.

```
par,seq :: a -> b -> b
```

These coordination atoms can be combined in higher-order functions to control the evaluation degree and its parallelism without mixing coordination and computation in the code. This technique of *evaluation strategies* described in [THLP98]

---

<sup>1</sup>Whether or not the execution is parallel can either be decided depending on the workload, or be mandatory (as the annotation in Concurrent Clean [PvE93]).

offers sufficient evaluation control to define constructs similar to skeleton-based programming. However, as opposed to usual skeletons, parallelism always remains semi-implicit in GpH, since the runtime environment (RTE) can either ignore any spark or eventually activate it.

The implementation of GpH, GUM [THM<sup>+</sup>96], essentially relies on the administration of a distributed shared heap and on the described two-stage task creation mechanism, where potential parallel subtasks first become local sparks before they may get activated. The only access point to the system is the spark creation primitive; parallel computations and their administrative requirements are completely left to the RTE and mainly concern spark retrieval and synchronisation of a distributed heap. Once a spark gets activated, the data which is evaluated in parallel could subsequently reside on a different processor and therefore has to receive a global address, so it can be sent back on request.

The main advantage of the implicit GpH concept is that it dynamically adapts the parallel computation to the state and load of nodes in the parallel system. The GpH implementation would even allow to introduce certain heuristics to reconfigure the parallel machine at runtime. However, parallel evaluation on this dynamic basis is hardly predictable and is accessible only by simulation and tracing tools like GranSim [Loi98].

### Controlled parallelism

A higher degree of execution control is achieved when the programmer explicitly specifies parallel scheduling. Programs with controlled parallelism are real parallel programs that expose their parallel behaviour. Examples in Haskell are Hudak's para-functional programming approach and successors [MH04]), or the evaluation strategies approach (as a high-level GpH [THLP98]), which enables to force evaluation of subexpressions to a certain degree (in parallel or sequentially).

Skeleton-based parallelisation can be ranged in this category since, commonly, the programmer has to explicitly choose the algorithmic pattern implemented by a certain skeleton, and to follow it. However, Trinder et al. [TLP02] categorise the HDC language [HL00] (implementing a subset of Haskell) as "implicit". In HDC, common higher-order functions for lists have an implicit parallel implementation which is completely transparent to the programmer.

### Explicit parallelism

Other, even more explicit, languages give the programmer *complete* control over parallel execution. These languages are not only able to speed up transformational systems (which map input to output), but can also be used to implement concurrent, interactive and distributed systems, i.e. augment the language expressiveness. Explicit parallel languages often use a concept of *processes* and *channels* between them to define process networks.



In the language Caliban [Tay97], (static) process networks can be specified declaratively. The compiler generates code for all processes and their interconnection, which is thus statically determined. The language Eden (presented and explained in detail in the next section) takes a similar approach: Processes are specified by their input-output mapping and connected via channels, which may transfer data as *streams*. In contrast to Caliban, Eden processes are instantiated dynamically, and the process network can thus evolve during runtime.

Both Caliban and Eden are implicit about the communication details and synchronisation. Going even further, we find functional languages with explicit message-passing and concurrency. Examples (not based on Haskell) are Concurrent ML [Rep99], Facile [GMP89], and Concurrent Clean [SP99]. In the Haskell world, we find Concurrent Haskell [JGF96] and Port-based distributed Haskell [TLP02]. However, these languages are not primarily targeted towards parallelism (speeding up a single computation), but intended for distributed and interactive systems. Glasgow-distributed Haskell [PTL01] is another example, an unconventional one, since it uses the virtual shared memory model of GpH.

## 2.4 The language Eden

The parallel Haskell dialect Eden [LOMP05] has been developed in the 90s by research groups in Marburg and Madrid [BLO95, BLOP96, BLOMP97]. Several implementations based on message passing have been constructed since 1998 [Bre98, Klu, BKL<sup>+</sup>03], and a broad range of publications investigate semantics and implementation aspects, as well as its application to skeleton programming (the overview [LOMP05] summarises).

### Basic coordination constructs

Eden extends Haskell [PH99] by syntactic constructs for explicitly defining processes, providing direct control over process granularity, data distribution and communication topology. Its two main coordination constructs are process abstraction and instantiation.

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> (a -> b)
```

embeds functions of type  $a \rightarrow b$  into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` ensures that both types `a` and `b` belong to the type class `Trans` of transmissible values. A *process abstraction* `process (\x -> e)` defines the behavior of a process with parameter `x` as input and expression `e` as output.

The evaluation of an expression `(process (\x -> e1)) # e2` leads to dynamic creation of a new (remote) *child process* which evaluates the expression `e1[x->e2]`. The instantiating or *parent process* evaluates and sends `e2` to the child process,

while the latter sends the result back to the parent, both using implicitly created communication channels. The (denotational) meaning of the above expression is identical to that of the ordinary function application  $(\lambda x \rightarrow e1) e2$ .

### Communication semantics: Streams and tuples

In general, Eden processes do not share data among each other and are encapsulated units of computation. All data is communicated eagerly via (internal) channels, avoiding global memory management and data request messages, but possibly duplicating data.

Data which is communicated between Eden processes is generally evaluated to normal form by the sender. In principle, arbitrary data could be sent, but this property requires that its type belongs to type class `NFData` (providing a normal form evaluation strategy [THLP98]). The mentioned `Trans` class is thus a subclass of `NFData`, and additionally provides communication operations<sup>2</sup>: Depending on the data type, Eden specifies special communication for data communicated as a process input or output.

- If a list is communicated, its elements will be successively evaluated to normal form and immediately sent to its destination one by one. The list is communicated *element-wise*, as a stream.

This property can be used to profit from lazy evaluation, namely by using infinite structures and by reusing the output recursively as, e.g., in the *workpool* skeleton [KLPR00].

- If process input or output is a tuple, its components will be evaluated to normal form and sent *concurrently*. Thus, several inputs and outputs of a process do not interfere with each other, and do not block process creation.

### Eager evaluation

Both input and output of a process can be a tuple, in which case one concurrent thread for each output component will be created, so that different values can be produced independently. Whenever one of their outputs is needed in the overall evaluation, the whole process will be instantiated and will evaluate and send all its outputs eagerly. This deviation from lazy evaluation aims at increasing the degree of parallelism and at speeding up the distribution of the computation. Local garbage collection detects unnecessary results and stops the evaluating remote threads. Another obvious effect is increased responsiveness of remote processes and the interleaving of parameter supply and parallel computation.

---

<sup>2</sup>In fact, the presence of an additional class has technical reasons. To require a normal-form evaluation strategy would be sufficient for Eden processes, following the language definition.

## Additional non-deterministic features

The basic constructs of Eden internally install channels between parent and child processes, and handle their communication automatically. To increase expressiveness and optimise communication in arbitrary process networks, two additional Eden language constructs allow one to dynamically create and use *dynamic reply channels*. Direct connections between arbitrary processes can be established. The difference between *static* and *dynamic* channels is that the former are installed during process creation while the latter are created by a running process.

A type constructor `ChanName` is used to represent a *dynamic channel*, which can be created and passed to another process to receive data from it. Dynamic channels are installed using the following two operators:

```
new      :: Trans a => (ChanName a -> a -> b) -> b
parfill :: Trans a => ChanName a -> a -> b -> b
```

As can be seen from their type, operations on dynamic channels in Eden are type-safe. Furthermore, which is not expressible in terms of types, channels are restricted to 1:1 communication. If a channel is used by more than one sender simultaneously, behaviour will be undefined, usually a runtime error.

Evaluating an expression `new (\ ch_name ch_vals -> e)` has the effect that a new channel name `ch_name` is declared as a reference to the new input channel, via which the values `ch_vals` will eventually be received in the future. The scope of both is the body expression `e`, which is the result of the whole expression. The channel name has to be sent to another process to establish direct communication. A process can reply through a channel name `ch_name` by evaluating an expression `parfill ch_name e1 e2`. Before `e2` is evaluated, a new concurrent thread for evaluation of `e1` is generated, whose normal-form result is transmitted via the dynamic channel. The result of the overall expression is `e2`; the new thread is generated as a side effect. Its execution continues independently from the evaluation of `e2`. This is essential, because `e1` could yield a (possibly infinite) stream which would be communicated element by element. Or, `e1` could even (directly or indirectly) depend on the evaluation of `e2`.

As another non-functional feature, Eden defines a non-deterministic operation to `merge` a list of lists into a single output list, in the order in which elements are available, similar to the Concurrent Haskell construct `nmergeIO`. Many-to-one communication, essential for expressing reactive systems, can be realised using this operation. A controversial fact is that the Eden definition provides `merge` not as a monadic operation, but as a purely functional one – thereby spoiling referential transparency. However, a good reason for this (also explained in [LOMP05]) is that mutual recursive value-passing between processes, as in the master-worker process topology, would otherwise need to use monadic fixpoint operators. To guarantee determined behaviour of functions which use `merge` internally is a skeleton programming task; its intention is internal use.

In summary, Eden is an explicit parallel Haskell which changes the evaluation order for the parallelism constructs and adds useful non-functional features. As a general-purpose language, it allows to express more implicit parallelism constructs, like skeletons, internally. Eden programs are fully explicit regarding parallelism, but the built-in communication modes provide additional implicit concurrency, which adds programming comfort and is a reasonable compromise between programmer control and automatic management. The next chapter will describe the concepts for the Eden implementation and point out possible alternative coordination constructs for even more explicit control.

## Part II

# High-level implementation concepts



# Chapter 3

## A layered Eden implementation

Any high-level approach to parallel programming contains an inherent trade-off for its implementation: providing operational control of the execution while abstracting over error-prone details. The explicit parallel runtime support needed for an implementation must coordinate the parallel evaluation operationally, i.e. express *operational* properties of the execution entities. It will thus – in the end – rely on an imperative-style description. Parallelism support in its basic form must be considered as imperative (and thus encapsulated in monads in the purely functional language Haskell). Yet programmers wish for a higher level of abstraction in their parallel programs; they do not want to deal with side-effects or communication and prefer to use skeletons [RG03] (higher-order functions for common parallel patterns), because they are not interested in gory details of implementation. Some parallel languages and libraries offer a fixed set of predefined skeletons and special, highly optimised implementations. On the other hand, with a more explicit general-purpose parallel language (like Eden), a programmer can express *new* skeletons specific to the application.

It follows that whether to hide or show the imperative basics of a coordination language for parallel functional computation is purely a question of language design. Eden tries to achieve a compromise between extremes in these matters: it exposes the execution unit of parallel processes to the programmer, but sticks to a functional model for their use. Eden processes differ from functions only by additional strictness and remote evaluation. However, the advanced Eden language features `merge` for nondeterministic stream merging, and `new` and `parfill` for explicit channel communication, allow for reactive systems and an arbitrary programmer-controlled communication structure, which is (necessarily) opposed to referential transparency. Furthermore, we can say from our practical experience that a modified instantiation operation with explicit placement of the newly created process is an indispensable feature, for both skeleton programming and application programming in Eden.

In this chapter, we describe the Eden implementation, based on the Glasgow-Haskell Compiler (GHC, currently at version 6.8.2), which we have developed,

maintained and refined during several years. The implementation has been structured in essentially two layers: a kernel runtime system which provides directly implemented primitive operations for basic parallelism support, and two Haskell modules which wrap these primitives and combine them to the more complex operations. The first section describes the essential functionality needed for Eden specifically. But the Eden implementation primitives may as well be considered as a language of their own, the *EDen Implementation language*, EDI for short, presented in section 3.2. In contrast to Eden, EDI uses explicit communication and the IO monad to encapsulate side-effects. Like Eden, EDI is implemented by a small Haskell module which builds some safety around the primitive operations, restricting their types and usage.

Our work in [BL07a] describes the Eden implementation, but also compares expressiveness and performance of Eden and EDI (for an earlier stage). While the differences in performance can be neglected, the programming styles are substantially different. EDI allows more accurate control of parallelism, useful for system programming, whereas the higher abstraction of Eden is favourable for application programming, but often obscures what exactly is happening during parallel execution.

As we outline in the last section of this chapter, the structured Eden implementation using a low-level implementation language can be a valuable approach for other parallel Haskell as well. The next chapter goes into more detail about, and presents selected aspects of, this generalising approach.

## 3.1 Implementation of Eden

### 3.1.1 Layer structure of the implementation

The implementation of Eden extends the runtime environment (RTE) of the Glasgow-Haskell-Compiler (GHC) [GHC] by a small set of primitive operations for process creation and communication between processes. These primitives merely provide very simple basic actions for process creation, data transmission between the machines' heaps, and system information. More complex operations are encoded in functional modules: a *Parallel Primitives* (ParPrim) module, which adds a thin wrapper around the primitives proper and some Haskell types for runtime system access, and the *Eden module*, defining all the language construct in terms of the wrapped primitives.

This module relies on the side-effecting primitive operations to encode Eden's process creation and communication semantics. The code on module level abstracts from many administrative issues, profiting from Haskell's support in genericity and code reuse. Moreover, it will protect the basic primitives from being misused. This leads to an organisation of the Eden system in layers (see Fig. 3.1): program level – skeleton library – Eden module – primitive operations – parallel



runtime environment. The strict structuring greatly improves the maintainability of the highly complex system and also enables to think more generally about the needed runtime system support for parallel coordination in general.

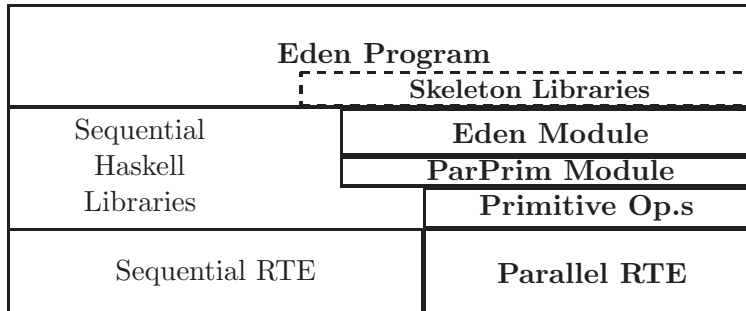


Figure 3.1: Layered Eden implementation

### 3.1.2 Parallel runtime environment

The basic layer implementing the primitive operations is based on the GHC runtime environment, and manages communication channels and thread termination. The GHC runtime environment (RTE) has been extended such that it can execute in parallel on clusters. Furthermore, small changes have been made to the compilation process, so that the compiled program is accompanied by a run script to make it execute in parallel with suitable parameters. We briefly summarise and systemise the extensions made to the RTE.

**Communication infrastructure** inside the runtime system is concentrated inside one single “Message Passing System” interface (file *MPSsystem.h*). The module provides only very basic functionality assumed to be available in virtually any middleware solution, or easily self-implemented, which enables different implementations on different hardware platforms. Fig. 3.2 shows the functions to provide. Apparently, the parallel runtime system has to start up in several instances on a whole group of connected machines (PEs). The primitive operations, and also the entire runtime system code, address the  $n$  participating PEs simply by numbers from 1 to  $n$ . Mapping these logical addresses to the real, middleware-dependent addressing scheme is one task to implement. Two implementations of the MPSsystem interface have been carried out, for MPI [MPI97] or PVM [PVM] as a middleware.

**Startup and shutdown infrastructure** manages that, upon program start, the runtime system instances on all PEs get synchronised before the main evaluation can start, and that the distributed system does a controlled shutdown both upon success and failure.

```
/******  
 * Startup and Shutdown routines (used inside ParInit.c only) */  
  
/* - start up the PE, possibly also spawn remote PEs */  
rtsBool MP_start(char** argv);  
  
/* - synchronise participating PEs  
 * (called by every node, returns when all synchronised */  
rtsBool MP_sync(void);  
  
/* - disconnect current PE from MP-System */  
rtsBool MP_quit(int isError);  
  
/******  
 * Communication between PEs */  
  
/* - a send operation for p2p communication */  
void MP_send(int node, OpCode tag, long *data, int length);  
  
/* - a blocking receive operation. Data stored in *destination */  
int MP_recv(int maxlength, long *destination, // IN  
            OpCode *code, nat *sender);      // OUT  
  
/* - a non-blocking probe operation */  
rtsBool MP_probe(void);
```

---

Figure 3.2: RTE message-passing module (interface *MPSystem.h*)

The protocol for the startup procedure is deliberately simple and depends on the underlying middleware system. For middleware with the ability to spawn programs on remote nodes (such as PVM [PVM]), a “main” PE starts up first, and spawns RTE instances on all other participating PEs. PEs are synchronised by the main PE broadcasting the array of all PE addresses, which the other PEs acknowledge in a reply message (`PP_READY`). Only when the main PE has received all acknowledgements, it starts the main computation.

When the middleware manages the startup of programs on multiple PEs by itself (this is the case for MPI implementations, where the MPI report [MPI97] imposes that MPI processes are synchronised by the `mpirun` utility upon startup), no additional synchronisation for the runtime system needs to be implemented.

In order to implement the controlled system shutdown, basic message passing methods had to be implemented, and the scheduling loop of GHC has to regularly check for arriving messages before executing the next runnable thread.

Shutdown is realised by a system message `PP_FINISH`. Either this message is broadcasted by the main PE (with address 1), or from a remote PE to the main PE, when the remote PE fails. In the failure case, the parallel computation cannot be recovered, since needed data might have been lost. Remote

PEs receiving `PP_FINISH` simply stop execution, while the main PE, in the failure case, broadcasts the message to all other remote PEs, thereby initialising a global shutdown.

**Basic (Runtime) Computation Units**, managed by the runtime system, are addressed by globally unique addresses as follows.

A running parallel Eden program splits up, in the first instance, into a set of PEs 1 to  $n$  (also called *machines* in the following). Machine 0 is invalid. Furthermore, already the sequential GHC runtime system internally supports *thread concurrency* addressed using (locally) unique thread identifiers (IDs). Multiple threads at a time can thus run inside one machine. Threads are uniquely identified by their machine number and ID.

A useful mid-level abstraction of a thread group in a machine is introduced by the Eden language definition: a *process*. Each thread in Eden belongs to a process, a conceptual unit of language and runtime system. A process consists of an initial thread, and can add threads by forking a subcomputation (concurrent haskell). All threads in one process share a common heap, whereas processes are not assumed to share any data; they need to communicate explicitly. Grouping threads inside one machine to processes like this is useful in general, and also relates to the extensions made to garbage collection with respect to heap data transfer.

**Support for data transfer between PEs** is a more than obvious requirement of any parallel system implementation. In the context of extending GHC, specifically, any data is represented as a graph in the heap. Data transfer between PEs thus means to serialise the subgraph reachable from one designated start node (or: heap closure), and to reconstruct it on the receiver side. In our implementation, heap data structures are transferred as *copies*, which potentially duplicates work, but avoids implementing a virtual global address space in the runtime system (we will come back to this in Section 4.7). An important property of the data serialisation routine is that on the one hand, it does not evaluate any data (but sends it as-is, in its current evaluation state). On the other hand, serialisation is instantly aborted when a placeholder for data under evaluation is found in the subgraph. Thus, in terms of concurrent heap access, data serialisation behaves like evaluation, even though it does not evaluate anything.

Data is always sent via *channels* previously created on the receiver side, where the placeholder nodes which synchronise concurrent threads in the sequential system may now stand for *remote* data as well. The RTE keeps a list of open channels and manages the replacement of placeholder by data which has been received through the channel.

Several data message types are implemented: the normal `Data` and the `Stream` mode. Data sent in `Data` mode just completely replaces the placeholder when it is

received. When data is sent in `Stream` mode, the receiver inserts it into the heap as the first element of a list and leaves the channel open for further list elements (until the closing `nil`, `[]` is eventually sent in `Data` mode). Another communication mode `Connect` serves to establish a producer-consumer link between two PEs early, before results of a potentially expensive evaluation are transmitted. Finally, because computations, as data, are first-class citizens in Haskell, and therefore nothing but a heap graph structure, the creation of a remote computation could be implemented as yet another data communication mode `Instantiate`, where the transmitted data is actually the unevaluated computation to be executed.

Please note that our extensions for data transfer change the meaning of placeholder nodes in the heap, which has consequences for the GHC garbage collection mechanisms. In the sequential system, a thread may only find a placeholder in the heap if there is another thread that evaluates the data behind it. Garbage collection in the sequential system evacuates data needed by *runnable* threads in the first instance. If none of the runnable threads will ever update a certain placeholder any more, threads blocked on this placeholder are effectively garbage and will be removed. This is not the case any more in our system, where placeholders may also stand for remote data. But the implementation of the Eden language constructs (described later) ensures that a remote data source sender exists. Thus, the modified garbage collection keeps threads alive whenever they are registered as *members of a process* (i.e. not created for internal reasons).

**Changes to the compilation process** have been made only for the linking phase and for convenience reasons. The compilation of an Eden program in the extended GHC remains largely the same as compiling a sequential program with GHC. Differences are that libraries for the message passing system have to be linked to the application, and that the compiled and linked program needs custom mechanisms to be started in parallel. The latter issue is solved by generating a separate startup script, depending on the middleware in use. It must be mentioned that the start script is a minimalistic solution, and might cause problems for unfamiliar users or in custom-configured clusters. However, the Eden System in its current state was not developed as a commercial off-the-shelf solution but is a research software system.

### 3.1.3 Primitive operations

The current implementation of Eden is based on six primitive operations. These primitives, rather than the runtime system support described before, represent the basic requirements for any Eden implementation: system information (amount of participating PEs and local PE), explicit data transfer, remote process and local thread creation, upon which Eden coordination constructs can be built.

Primitive operations are directly implemented in the runtime system, and consequently do not use Haskell data types as parameters. The lowest Haskell

---

<code>data Mode = Stream   Data</code>	data modes: Stream or Single data
<code>      Connect   Instantiate Int</code>	special modes: Connection, Instantiation
<code>data ChanName' = Chan Int# Int# Int#</code>	a single channel: IDs from RTE
<code>createC :: IO ( ChanName' a, a )</code>	channel name creation
<code>connectToPort :: ChanName' a -&gt; IO ()</code>	channel installation
<code>sendData :: Mode -&gt; a -&gt; IO ()</code>	send data on implicitly given channel
<code>fork :: IO () -&gt; IO ()</code>	new thread in same process (Conc.Haskell)
<code>noPe :: IO Int</code>	number of processor elements
<code>selfPe :: IO Int</code>	ID of own processor element

---

Figure 3.3: Primitive operations to implement Eden (module *ParPrim.hs*)

module of the implementation merely consists of embedding the primitives in the IO monad to encapsulate the side-effects, and adds Haskell data types for communication mode and channels. Fig. 3.3 shows the interface, the full code is reproduced in Appendix B.1.2.

The first two primitives provide system information: the total number of processor elements (`noPe`) or the number of the processor element running a thread (`selfPe`).

For communication between processes, `createC` creates a new channel on the receiver side. It returns a channel name, containing three RTE-internal IDs: (PE, processID, portID) and (a handle for) the channel contents. Primitives `connectToPort` and `sendData` are executed on the sender side to connect a thread to a channel and to asynchronously send data. The send modes specify how the receiver sends data: either as an element of a stream (mode `Stream`), or in a single message (mode `Data`), or (optionally) just opening the connection (mode `Connect`). The purpose of the `Connect` mode is to provide information about future communication between processes to the runtime system. If every communication starts by a `Connect` message, the runtime system on the receiver side can terminate threads on the sender side evaluating unnecessary data.

Please note that thread management reduces to only one primitive `fork`, which creates a new thread in the same process (and which is simply a variant of a Concurrent Haskell [JGF96] construct). We have explained that starting a new remote thread (and process) can be implemented as sending data with the send mode `Instantiate`. The `Int` argument allows to explicitly place the new process on a certain machine. If it is zero, the RTE automatically places new processes in round-robin manner.

### 3.1.4 Eden module: Language features

#### Overloaded communication

The primitives for communication are used inside the Eden Module to implement Eden's specific data transmission semantics. The module defines type class `Trans`

---

```

newtype ChanName a = Comm (a -> IO())

class NFData a => Trans a where
  -- overloading for channel creation:
  createComm :: IO (ChanName a, a)
  createComm = do (c,v) <- createC
                 return (Comm (sendVia c), v)
  -- overloading for streams:
  write      :: a -> IO()
  write x = rnf x 'seq' sendData Data x

sendVia ch d = do connectToPort ch
                 write d

```

---

Figure 3.4: Type class `Trans` of transmissible data

---

```

-- list instance (stream communication)
instance Trans a => Trans [a]
  where write l@[_] = sendData Data l
        write (x:xs) = do (rnf x 'seq' sendData Stream x)
                          write xs

-- tuple instances (concurrency by component)
instance (Trans a, Trans b) => Trans (a,b)
  where createComm = do (c1,v1) <-createC
                       (c2,v2) <-createC
                       return (Comm (send2Via c1 c2), (v1,v2))

send2Via :: ChanName' a -> ChanName' b -> (a,b) -> IO ()
send2Via c1 c2 (v1,v2) = do fork (sendVia c1 v1)
                             sendVia c2 v2

```

---

Figure 3.5: Eden module: Overloading for communication

of transmissible data, which contains overloaded functions, namely `createComm` to create a high-level channel (type `ChanName`), and `write` to send data via channels.

As shown in Fig.3.4, the high-level channel `ChanName` is a *data communicator*, a function which performs the required send operation. It is composed by supplying the created primitive channel as a first argument to the auxiliary function `sendVia`. The latter, evaluated on sender side, first connects to the channel and then calls the `write` function to evaluate its second argument to normal form<sup>1</sup> and send it to the receiver in `Data` mode.

The two functions in `Trans` are overloaded as follows: `write` is overloaded for streams, which are communicated elementwise, and `createComm` is overloaded

---

<sup>1</sup>The `NFData` class provides an evaluation strategy [THLP98] `rnf` to force normal-form evaluation of any member type.

for tuples, which are evaluated concurrently by one thread for each component. Fig. 3.5 shows the instance declarations for lists and pairs. `write` communicates lists elementwise in `Stream` mode, and `createComm` for pairs creates two primitive channels, using the auxiliary function `sendVia` for forking threads.

## Process abstraction and instantiation

The Eden constructs `process` and `( # )` render installation of communication channels between parent and child process, as well as communication, completely implicit, whereas the module internally uses *explicit* communication channels provided by `Trans` and the primitive operations.

---

```

data Process a b = Proc (ChanName b -> ChanName' (ChanName a) -> 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 -- input communicator
            connectToPort inCC             -- sent back...
            sendData Data sendInput       -- ...to parent
            sendResult (f input)         -- sending result

( # ) :: (Trans a, Trans b) =>
    Process a b -> a -> b
p # x = unsafePerformIO (instantiateAt 0 p x)

instantiateAt :: (Trans a, Trans b) =>
    Int -> Process a b -> a -> IO b
instantiateAt pe (Proc f_remote) procInput
    = do (sendResult, r) <- createComm -- result communicator
        (inCC, Comm sendInput) <- createC -- input comm. (reply)
        sendData (Instantiate pe) -- spawn process
            (f_remote sendResult inCC)
        fork (sendInput procInput) -- send input concurrently
        return r -- return placeholder

-- variant of ( # ) which immediately delivers a whnf
data Lift a = Lift a
deLift (Lift x) = x

createProcess :: (Trans a, Trans b) =>
    Process a b -> a -> Lift b
createProcess p i
    = unsafePerformIO (instantiateAt 0 p i >>= \x ->
        return (Lift x))

```

---

Figure 3.6: Eden module: Process abstraction and instantiation

Fig. 3.6 shows the definition of process abstractions and instantiations in the Eden module. Process abstractions embed a function `f_remote` that is executed by a newly created remote process. When the process is instantiated, this function is applied to a communicator `sendResult` to return the results of the process to the parent process, and to a primitive channel `inCC` to send a communicator function (of type `ChanName a`) for its input channels to the parent process. The remote process first creates input channels, i.e. the corresponding communicator functions and the handle to access the input received. It connects to the channel `inCC` and sends the input communicator with mode `Data` on it. Afterwards, the process will evaluate the expression `(f input)` and send the result to the parent process, using the communicator function `sendResult`.

The instantiation operator `(#)` relies on the function `instantiateAt`, which defines the parent side actions for the instantiation of a new child process. The embedded function `f_remote` is applied to a previously created result communicator and a primitive channel for receiving the input, and the resulting IO action is sent to the designated machine *unevaluated*. A new thread is forked to send the input to the new process. As its name suggests, `instantiateAt` may place the new process on the PE specified by the parameter `pe`; or else uses the automatic round-robin placement if the parameter is 0.

Additionally, the Eden module provides a variant `createProcess` of the instantiation, which differs in the type of the result value, *lifted* to immediately deliver a value in weak head normal form (whnf). In the past, this was necessary to create a series of processes without waiting for process results.

The basic Eden coordination constructs implemented up to now have a purely functional interface, as opposed to the IO-monadic primitive operations. The nature of the primitive operations is either to trigger side-effects (sending messages, spawning new processes, creating and connecting to channels), or to return values which depend on the execution environment (number of machines and executing machine). To maintain Haskell's functional nature, we consequently provide the primitives as IO monadic actions, the common way to encapsulate effects of this kind in Haskell. However, Eden's process instantiation has purely functional type, requiring that the implementation hides their inherent side-effecting nature. Process instantiation is thus internally a sequence of IO actions based on the primitives but, finally, the functional type of the instantiation operator `(#)` will be obtained by `unsafePerformIO`, the back door out of the IO monad.

### Advanced, non-functional Eden features

With the additional features of Eden, `merge` and *dynamic reply channels*, Eden loses its purely functional face and exposes nondeterminism and side-effects to programmers explicitly (without them having a monadic type, however). The implementation of these advanced constructs is straightforward, using the IO-monadic primitives and `unsafePerformIO` again.



---

```

merge :: [[a]] -> [a]
merge = unsafePerformIO . nmergeIO

new :: Trans a => (ChanName a -> a -> b) -> b
new chanValCont
  = unsafePerformIO (do (chan , val) <- createComm
                        return (chanValCont chan val))

parfill :: Trans a => ChanName a -> a -> b -> b
parfill (Comm sendVal) val cont
  = unsafePerformIO (do fork (sendVal val)
                        return cont)

```

---

Figure 3.7: Implementation of Eden’s advanced, non-functional features

The nondeterministic `merge` of a list of streams into a single stream is already provided by Concurrent Haskell as an IO-action `nmergeIO :: [[a]]  $\mapsto$  IO [a]`. Thus we can simply define `Eden.merge = unsafePerformIO . nmergeIO`. The implementation of `nmergeIO` offers some potential for optimisations in cases where short lists are merged [DBL08].

The functions for dynamic reply channels in Eden are channel creation with `new`, and concurrent sending with `parfill`. They offer arbitrary explicit communication to the programmer. Conforming to Eden’s implicit communication semantics, data has to be evaluated to normal form before sending, and lists are transmitted as streams. It follows that, aside from the particular continuation-passing style shown in their types (explicit continuation in `new` and result “continuation” in `parfill`), the implementation of dynamic reply channels is simply a variant of using the `Trans` member functions `createComm` and `write` (inside the communicator `sendVal`).

Haskell, and especially Concurrent Haskell which is heavily used in the Eden module, encapsulate nondeterminism and side-effects in the IO monad, while the two Eden constructs in question here use continuation-passing style – which leads to a programming paradigm mixture. Given the roughly similar functionality of `new,parfill` and its implementation using data communicators, these two constructs could, and should, be replaced by IO-monadic versions easily. Which brings us to the point where we are ready to define a *whole language* as an alternative to Eden, which is closer to, and integrates much nicer with, the Concurrent Haskell base.

### 3.1.5 Simulations for Eden and its implementation

#### Simulating Eden constructs?

For debugging and developing Eden programs, a sequential environment which *simulates* the Eden coordination constructs is a useful feature. This is not a hard

problem for the purely functional parts, `process` and `( # )`. What has to be simulated is the additional eager evaluation introduced by the implicit inter-process communication, which enforces evaluation of process inputs and outputs. This additional evaluation does not happen in a particular order, but concurrently in separate threads. It is therefore too naïve an attempt to simulate this normal-form evaluation using only `rnf` and `seq`. Process input and output may depend on each other, and include complex interdependencies between elements of communicated streams. Figure 3.8 sketches such a simulation, as well as examples where it does not model the real Eden process behaviour.

Consequently, approximating the behaviour of Eden processes requires starting concurrent threads that evaluate process instantiation input and output to normal form independently, in order to trigger potential exceptions. Any simulation has to use an analog overloading as in the `Trans` class, to model concurrent tuple component evaluation and stream evaluation.

What still remains to be simulated is explicit communication, available through `new` and `parfill` in Eden. At the bottom line, this amounts to simulating the explicit communication features of the *primitives*. And once we reach this point, it is apparently a lot easier to simulate the implementation primitives instead of the Eden constructs.

---

```

data Process a b = Proc (a -> b)
process f = Proc f
( # ) :: (NFData a, NFData b) => Process a b -> a -> b
(Proc f) # x = let fx = f x
               -- in rnf x 'seq' rnf fx 'seq' fx -- INCORRECT (1)
               -- in rnf fx 'seq' rnf x 'seq' fx -- INCORRECT (2)
               in fx                               -- INCORRECT (3)

```

Counter-examples for the sequential `rnf`-based simulations (1) and (2) are processes where either a list output, or a tuple component, determines parts of the input. The parallel Eden-based semantics of the following two functions is the identity, while the additional `rnf` evaluation as sketched above will lead to deadlock.

```

seqDeadlock n = let loop = process (map (+1)) # (0:loop)
                 in loop!!(n-1)
seqDeadlock2 n = let (a,b,c) = swapP # (n,a,b)
                   swapP = process \(in1,in2,in3) -> (in2,in3,in1)
                   in c

```

Vice-versa, variant (3) without the sequential `rnf` evaluation is incorrect for the following function. It will return `x` when evaluated sequentially, while in the real parallel execution, it yields a runtime error in the input-sending thread:

```

parError x = process (const x) # undefined

```

---

Figure 3.8: *Inadequate* simulation for Eden coordination constructs

## Simulating, or specifying, the primitives

A simulation *ParPrimConcHs.hs* for the primitives is included in Appendix B.1.3. It simulates the behaviour of the implemented primitives by means of concurrent Haskell, and thereby also constitutes a specification of the primitives semantics.

**selfPe** In order to simulate **selfPe**, every new thread is registered in a global thread table, indicating its PE and process. The first thread which registers is placed on the main PE (number 1).

**noPe** The number of PEs (**noPe**) is a simulation parameter which can be arbitrarily set to a value (initially set to four).

Data communication is simulated by **MVars**, which will hold the “transmitted” data. A global simulation table holds all these **MVars** and also registers the sending thread to check the 1 : 1 channel restriction.

**createC** Channel creation with **createC** is thus simulated by creating a new **MVar** and registering it in the global table. As in the runtime system, channels are addressed by globally unique addresses, built from the machine ID, the process ID, and a third ID. The return value is a channel name and an action which reads the value from the **MVar** (which will block the reading thread until data has been written).

**connectToPort** Threads need to **connectToPort** before sending data, and can as well use **sendData** with mode **Connect** to be registered on the receiver side.

**sendData** Sending data in **Data** or **Stream** mode is simulated by filling the respective **MVar** from the table with data. Either the channel is closed and deleted from the table, or a new **MVar** replaces the old one, for data sent in **Stream** mode. The ID of the sender thread is registered, and it will be checked for subsequent communications, triggering a runtime error if a previously registered ID and the currently sending thread’s ID do not match.

**fork** Forking a local thread is a concurrent Haskell functionality. For the simulation, the forking thread additionally has to add the new thread to the simulation thread table (in the same process), and the new thread will delete itself from it upon termination.

Remote process creation using **sendData** with mode **Instantiate** is simulated by forking a new initial thread and registering it in a new process. Automatic local round-robin placement is simulated by registering an appropriate PE number for the new process. A global **MVar** holds a list of current places for all PEs.

## 3.2 Edi: The Eden implementation language

Eden provides a purely declarative interface, but aims to give the programmer *explicit* control of parallelism in the program. Eden programs can be read in two ways, from a computational and from a coordinational perspective:

- Instantiation of a previously defined process abstraction denotationally differs from function application by the additional strictness due to Eden's eager communication policy, but yields the same result as application of a strict function.
- Process abstraction and instantiation will hide any process communication, but expose the degree of parallelism of an algorithm directly by the number of instantiations.

However, the additional strictness introduced by eager communication is a crucial point for tuning parallel programs. On the one hand, it is required to start subcomputations at an early stage and in parallel. On the other hand, adding too much artificial strictness to a program can easily lead to deadlock situations. A complex Eden program normally uses a suitable skeleton library, optimised for the common case and circumventing common pitfalls of parallelism. Eden can also describe new specialised *skeletons*, and programming these is a different matter. Efficiently programming skeletons in Eden requires intimate knowledge of Eden specifics and a clear concept of the evaluation order in a demand-driven evaluation. Concentrating on the coordination view of Eden, programming skeletons can profit from a more *explicit* approach, which we name EDI, the EDEN Implementation language. Another part of this thesis will discuss skeleton programming in more detail, but we can state here already that more explicit coordination features can help optimise skeletons for particular cases and save time in spotting errors due to Eden's complex implicit communication semantics.

### 3.2.1 Degree of control: A compromise

In the first instance, the primitives used for the Eden implementation (see Fig. 3.3) can be taken as a – very low-level, but fully-fledged – alternative Eden-type language, which will render communication and side-effects explicit and will force one to use the IO monad for parallel execution. However, directly using the primitives for programming has the fundamental drawback that these had been designed with the Eden implementation requirements in mind, and, more substantially, with a deliberately simple, non-restrictive interface, to obtain a lean interface between Haskell and the RTE. Subsequently, we are pointing out reasons for an additional layer above the primitives.

## Evaluation and communication decoupled

In contrast to Eden’s communication semantics, EDI communication is completely independent of the underlying computation. If a communicated value is not needed by the sender for a local computation, it will be left unevaluated by sending. This, of course, is not intended for parallel processes supposed to compute subresults. Programs in EDI therefore have to use evaluation strategies [THLP98] to explicitly initiate the computation of a value to be sent. Although EDI does *not* encode coordination by strategies, using the class `NFData` and its normal form evaluation strategy `rnf` is a necessary part of EDI programming.

The implemented `sendData` primitive does not imply any prior evaluation. As EDI is purely monadic and deliberately simple, the programmer has to specify *every* single action. This means that programming with the primitives considerably inflates the code, and may introduce an excess of unwanted details.

### Too liberal type of `sendData`

Another possible source of errors is the all-purpose character of `sendData`, which uses the same primitive for data transmission, communication management, and process instantiation, distinguished only by the different send modes. Sending data by the wrong mode may lead to, e.g., a bogus process without any effect, as shown here:

```
badIdea_no1 :: Int -> a -> IO ()
badIdea_no1 pe data = sendData (Instantiate pe) data
```

If the data sent is, say, a number, its remote evaluation will have no effect at all, although its type is perfectly correct, due to the liberal typing of the primitive. In the example above, an auxiliary function for instantiation should enforce that the data sent is an action of type `IO()`.

```
spawnProcessAt :: Int -> IO () -> IO ()
spawnProcessAt pe action = sendData (Instantiate pe) action
```

### Two-step communication

For data communication, threads are supposed to *connect* to a channel prior to sending values or stream elements. There is a good reason to support separate `Connect` messages: Remote evaluation can be stopped earlier when the sender thread is known before it evaluates and sends data. However, both steps need to occur together, and might cause obscure runtime errors if the wrong connections are created. Moreover, another source of errors is that the wrong communication mode may be used, and unexpected data structures in the heap are created by receiving messages, leading to severe runtime errors. The simple channels of EDI are strongly typed, but even though, the two-step communication and the liberal choice of send modes allow one to create erroneous communication sequences,

which cannot be discovered at compile time. The following (perfectly well-typed) function expects a *wrong* channel type and then does not connect prior to sending in one case, or alternatively uses the wrong send mode.

```
badIdea_no2 :: ChanName' Double -> [Double] -> IO () -- types do not match
badIdea_no2 c (n:ns) = do sendData Stream n -- not yet connected
                        badIdea_no2 c ns
badIdea_no2 c [] = do connectToPort c
                      sendData Stream [] -- wrong send mode
```

When evaluating this function, a run-time error will occur because the receiver's heap becomes corrupted.

As above, a type-enforcing auxiliary function can detect the error, in this case combining connection and send operation. The only disadvantage here is that a separate function for sending lists is needed, since the send mode becomes hard-coded. To allow controlling the evaluation degree of transmitted data, it is sensible to also include an evaluation strategy (applied prior to transmission) in such a combined function.

```
sendWith :: (Strategy a) -> ChanName' a -> a -> IO ()
sendWith strat c d = connectToPort c >> (strat d 'seq' sendData Data d)

sendStreamWith :: (Strategy a) -> ChanName' [a] -> [a] -> IO ()
sendStreamWith strat c xs = connectToPort c >> send xs
  where send l@[] = sendData Data l
        send (x:xs) = (strat x 'seq' sendData Stream x) >> send xs
```

In order to streamline the interface between Haskell and the runtime system, the primitive `sendData` has been given the liberal type `Mode -> a -> IO ()`, which is why erroneous usage of the primitive will not be detected at compile time. Hence, the solution to these problems consists in typed auxiliary functions which will restrict the argument types in such a way that the primitives will be used as intended.

### 3.2.2 The EDI language

Obviously, it is necessary to superimpose a layer of type-checking auxiliary functions over the primitive operations to improve error detection during type checking in EDI. On the other hand, this superstructure is purely for safety reasons (and perhaps also decorative). The auxiliary functions presented up to now hardly go beyond the primitive operations in (reasonable) functionality, abstraction, or syntactic sugar. What may be added in this respect is a specialised send operation that hard-wires the normal form evaluation by the `rnf` strategy – which is the desired behaviour in most cases, and a convenience function that creates a whole *list* of channels instead of just one.

---

```

-- send operations with rnf evaluation (no connect message)
sendNF :: NFData a => ChanName' a -> a -> IO ()
sendNF = sendWith rnf

sendNFStream :: NFData a => ChanName' [a] -> [a] -> IO ()
sendNFStream = sendStreamWith rnf

-- creation of n channels in one call, "safe" evaluation
createCs :: NFData a => Int -> IO ([ChanName' a],[a])
createCs n | n >= 0 = do list <- sequence (replicate n createC)
              let (cs, vs) = unzip list
                  rnf cs 'seq' -- channels fully evaluated
                  return (cs,vs)
              | otherwise = error "createCs: n < 0"

```

The latter function also eliminates a potential error: programmers who write this function on their own might be unaware that the sequence of IO actions is executed, but the channel list `cs` is produced by a lazy function application `unzip`. Thus, without the evaluation `rnf cs`, other functions accessing the channel list `cs` for export might access unevaluated placeholders upon packing data. Put together in a module, we obtain the definition of new coordination features for parallel Haskells, which use the IO monad: the EDI language shown in Fig. 3.9.

---

*Edi.hs*: EDen Implementation language

```

module Edi
-- interface:
(fork,          -- :: IO () -> IO (), from conc.hs, without ThreadID
 spawnProcessAt, -- :: Int -> IO () -> IO ()
 ChanName',     -- EdI channel type
 createC,       -- :: IO (ChanName' a,a) , prim.Op.
 createCs,      -- :: Int -> IO ([ChanName' a],[a])
 sendWith,      -- :: (Strategy a) -> ChanName' a -> a -> IO ()
 sendNF,        -- :: NFData a => ChanName' a -> a -> IO ()
 sendStreamWith, -- :: (Strategy a) -> ChanName' [a] -> [a] -> IO ()
 sendNFStream,  -- :: NFData a => ChanName' [a] -> [a] -> IO ()
 noPe, selfPe,  -- :: IO Int
 module Strategies)
where

```

---

Figure 3.9: EDI, IO-monadic coordination features for parallel Haskell

We will assess this experimental language in a subsequent part of this thesis for implementing parallel skeletons, higher-order functions with a parallel implementation. Especially implementing optimised skeletons requires a high degree of explicitness, and is thus an interesting testbed, subject of the next thesis part. Another interesting area related to EDI's expressiveness is, what other high-level language constructs (besides Eden) can be implemented? In this thesis part

about language implementation, we will subsequently discuss how to apply and extend the presented implementation concepts.

### 3.2.3 Relation to other Haskell extensions

EDI, considered as a language, provides extensions to existing concepts of Concurrent Haskell [JGF96], as implemented in GHC. Thread concurrency is *extended* by process parallelism, communication in EDI is handled using channel communication instead of the shared synchronised heap cells (`mVars`) of Concurrent Haskell. Both approaches can be sensibly combined.

Latest efforts in Haskell implementations aim to extend Concurrent Haskell's thread concurrency to OS level for multiprocessor support in the threaded GHC runtime system [HMJ05, LMJT07]. Combining this multicore support with the distributed-memory parallelism provided by EDI is one of our future goals.

In the field of parallel functional languages, many language concepts follow more implicit approaches than Eden and, necessarily, its implementation language. Although intended as a low-level implementation language, EDI can be used as a language for distributed programming with explicit asynchronous communication.

Glasgow Distributed Haskell (GdH) [PTL01] is the closest relative to EDI in this respect and provides comparable language features, especially location-awareness and dynamically spawning remote IO actions. However, GdH has been designed with the explicit aim to extend the virtual shared memory model of Glasgow Parallel Haskell (GpH) [THM<sup>+</sup>96] by features of explicit concurrency (Concurrent Haskell [JGF96]). Our implementation primarily aimed at a simple implementation concept for Eden and thus does not include the shared-memory-related concepts of GdH.

Port-based distributed Haskell (PdH) [TLP02] is an extension of Haskell for distributed programming. PdH offers a dynamic, server-oriented port-based communication for first-order values between different Haskell programs. In contrast to our implementation, its primary aim is to obtain open distributed systems, interconnecting different applications – integrating a network library and a stock Haskell compiler.

## 3.3 Generalising the implementation concept

We have previously described in detail the implementation principle of Eden, extending the RTE by only few primitive operations, and a Haskell module for more complex features. This concept is not entirely new: Previous Eden implementations partially used the same ideas [BKL<sup>+</sup>03], without lifting them to a concept or paradigm, nor developing the approach towards other languages. We have developed it to maturity, even leading to a new language proposal. During



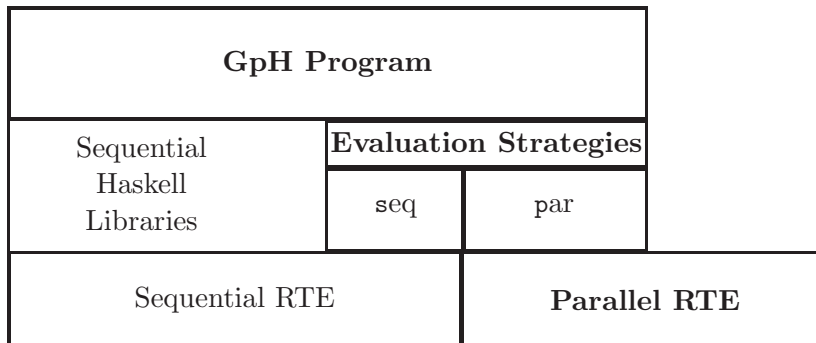


Figure 3.10: Layer view applied to GpH implementation

the Eden implementation work, ideas for a more general approach became more and more self-suggesting, given that the basic requirements for Eden are about as simple as they possibly can be for any other coordination language concepts.

In particular, the previous Eden implementations shared some essential RTE features with its more implicit relative GpH [THM<sup>+</sup>96]. The GpH implementation is essentially different in that the parallel RTE automatically manages load balancing between PEs, thereby creating the need for a global virtual shared memory; however, roughly the same RTE support for data transmission and process creation is needed for GpH.

To systemise the common parts of parallel Haskell implementations, we follow the approach of Eden’s layered implementation, i.e. thick layers of functionality exploited strictly level-to-level to avoid dependencies across abstraction levels. Apart from maintenance of only one system, the concept of layered implementation is promising for the implementation of other coordination languages based on Haskell, since it facilitates maintenance and experimental development. With one flexible basic layer, different language concepts can be easily implemented by a top-layer module (making use of the underlying RTE support) where the appropriate coordination constructs are defined. Its functionality must be exposed in an API which offers general support for parallelism coordination without introducing characteristics of one particular language concept in the runtime system.

As shown in Fig. 3.10, the *Evaluation Strategy* module for GpH [THLP98] is just an example of these high-level parallelism libraries. GpH, and similar annotation-based coordination languages, are not as explicit as Eden, and are not using distributed memory and communication channels between parallel processes. The generalised runtime system assumed for GpH will thus have to support virtual shared memory and implicit, load-dependent task creation.

Like Eden, one could implement other parallel coordination extensions, e.g. automatic parallel operations for some data-parallel language, by adding an appropriate module for all parallel operations to their sequential implementation. As Fig. 3.11 exemplifies, the approach is also interesting for automatic, compiler-

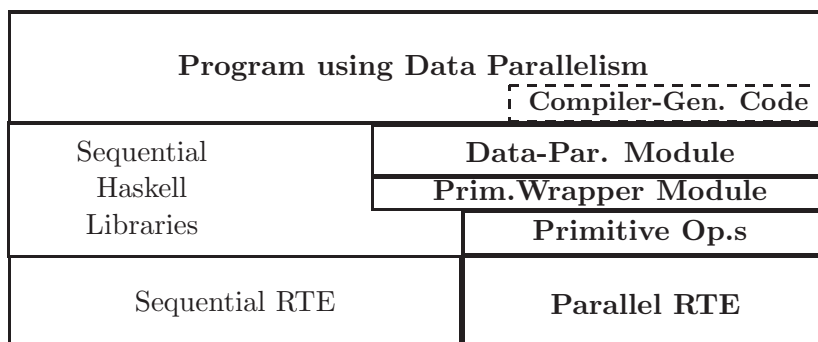


Figure 3.11: Layered implementation of a data-parallel language

directed parallelisations: compiling certain data structure operations to use specific parallel implementations, which are essentially defined in Haskell itself. The basic RTE support for this Haskell-implementation will be more or less similar to the one needed for Eden: Primitives used by the top-layer modules, and which can thereby be considered as a common *Parallelism API*, the essential, most general and most practical features of which we were to discover in more systematic case studies [Ber04].

Some apparently inherent restrictions of the approach come to mind. First of all, the Haskell-based implementation is based on parallel graph reduction with synchronisation nodes representing remote data. The question is, where actually do the implemented primitives use this fact at all? And, in order to support implicit parallelism, a generalised RTE needs to support virtual shared memory and implicit, load-dependent task creation. Another question is, to which extent can these features be made accessible from the language (i.e. Haskell) level and, thus, be part of the API?

The ideas outlined in this section are more systematically developed in the next chapter, where we will present the design and prototype implementation of a generic and modular RTE for parallel Haskell execution — if not high-level parallel languages in general.

# Chapter 4

## A generic runtime environment for parallel Haskells

### 4.1 Introduction

A runtime environment to exploit the computational power of today’s parallel architectures – ranging from multi-core machines to large-scale computational Grids – must reflect the underlying target architecture and either take into account its specific properties in automatic management functionality, or expose the architecture to the programmer by specific coordination constructs. However, targeting one specific architecture and proposing a (yet more specific) precisely-taylorred coordination language is of course anything but future-proof, and might be outdated in short time, given the rapid advances in today’s hardware development.

In this chapter, we present the design of a new parallel runtime environment for executing parallel Haskell code on complex, hierarchical architectures. By this design study, we aim to identify and implement the minimal and most general runtime support required for parallel Haskells. As we have already pointed out, the layer concept applied for the Eden implementation has proven useful and inspiring, and can be useful for the maintenance of other language implementations. Even more interesting is that the modular concept of the Eden implementation allows to use the implemented primitives for defining different coordination constructs *in Haskell*, fitted to particular future architectures.

Aiming to support various architectures, our design must allow *deep memory and process hierarchies*. The system should be able to use different control mechanisms at different levels in the hierarchy, either automated in the implementation, or exposed to language level. For memory management, this provides a choice of using explicit data distribution or virtual shared memory. For process management, this means that units of computation are very light-weight entities, and we explicitly control the scheduling of these units.

Our modular design defines a minimal *micro-kernel*, which is a slightly extended version of the Eden RTE described before. As in the Eden implementa-

tion, more complex operations are implemented in a high-level system language (Concurrent Haskell) outside this kernel. We arrive at a system with a clear modular design, separating basic components by their functionality and employing a hierarchy with increasing levels of abstraction. The micro-kernel is accessed via a narrow interface, and most of the coordination of the system is realised in a functional language. Immediate benefits of this design are the ease of prototyping and of replacing key components of the RTE — issues of particular importance in the rapidly evolving development of parallel systems.

Our design describes a generic and adaptive system for parallel computation, combining features of existing parallel RTEs for GpH [THM<sup>+</sup>96] and Eden [Ber04, BL07a]. We present a *prototype implementation* of key concepts in such a system in the form of an executable specification, amenable to formal reasoning.

As an example, we demonstrate the flexibility of the system by refining the GpH scheduling mechanisms towards a Grid environment, adding sophisticated work distribution policies – which previously had to be implemented in C inside the GpH RTE [ATLM06]. Supporting such *computational Grids* [FKT01], which incorporate thousands of machines on a global scale, requires taking into account the different memory access times and process hierarchies when distributing the parallel work. Additionally, the system needs to be *adaptive* in the sense that it dynamically adapts its behaviour to a dynamically changing environment.

The system presented in this chapter is partially implemented, but does not consider the memory management component, for which we will discuss the design space in Section 4.7. We plan to continue working on the prospected system in the near future, to further explore its potential and to implement more features, under the acronym ARTCOP (Architecture-Transparent Control of Parallelism). Subsequently, as well as in our related workshop publication [BLAZ08], we mainly concentrate on scheduling policies. The scheduling mechanisms we present are both *executable* and simple enough to serve as a *specification*, with the potential to easily provide formal proofs of runtime system properties. The code presented in this chapter is executable Haskell, and has been tested on GHC Version 6.6, extended with the Eden implementation primitives and only minor other extensions.

## Related work

Historically, using high-level languages for system programming has long been a research area, and the functional paradigm was even believed to lead to radically different future computer architectures in the late 70s [HHJW07] (while architectural work quickly came to the conclusion that sophisticated implementations on stock (von-Neumann) hardware are the better path). Work in the 80s on high-level languages for system-level programming mainly focused on how to implement O/S concepts in a functional [Hen82, Sto84, Per88] or logic [Sha84] style. Most of these systems introduce specific primitives to deal with non-

determinism, whereas later approaches either insisted on maintaining deterministic behaviour [HDD95] or used special data structures to control interactions between concurrent threads (such as MVars in Concurrent Haskell [JGF96]). Early implementations of functional operating systems are NEBULA [Kar81] and KAOS [Tur87]. More recent functional systems are Famke [vP03] and Hello [BF].

An early system that uses a micro-kernel (or substrate) approach in the RTE is the Scheme-based Sting [JP92] system. Sting defines a coordination layer on top of Scheme, which is used as a computation language. Genericity is demonstrated by directly controlling concurrency and processor abstractions, via Scheme-level policy managers, responsible for scheduling, migration etc. This general framework supports a wide range of features, such as (first-order) light-weight threads, thread pre-emption, and asynchronous garbage collection. Common paradigms for synchronisation (e.g. master-slave parallelism, barrier communication etc.) are implemented at system level and demonstrate the possibility to easily define application-optimised synchronisation patterns. However, since Sting uses Scheme as a system level language, it lacks the clear separation of pure and impure constructs at system level as offered by Haskell. We also consider Haskell's static type safety for system level code as an advantage.

Most closely related to our high-level implementation approach is [HJLT05]. It defines a Haskell interface to low-level operations and uses a hardware monad to express stateful computations. It focuses on safety of system routines, using its own assertion language and Haskell's strong type system. This interface has been used to code entire O/S kernels (House, Osker) directly in Haskell, reporting satisfactory performance. In contrast to this proof-of-concept approach, we want to improve maintainability by realising the more complex RTE routines in Haskell, but still keeping a micro-kernel implemented in a low-level language.

The Manticore [FFR<sup>+</sup>07] system, a recent project of the University of Chicago, targets parallelism at multiple levels and enables the programmer to combine task and data parallelism. Manticore's computation language is a subset of ML, a strict functional language. Compiler and runtime system provide support for parallel arrays and tuples and a number of scheduling primitives. Similar in spirit to our approach, only a small kernel is implemented in low-level C; other features are implemented in external modules, in an intermediate ML-like language of the compiler. A prototype implementation is announced, and aims to be a testbed for future Manticore implementations and language design. As opposed to ARTCOP's genericity in coordination support, Manticore explicitly restricts itself to shared-memory multi-core architectures and does not support networked computing, nor location-awareness and monitoring features.

The Famke system [vP03] is implemented in Clean and explores the suitability of Clean language features such as dynamic types and uniqueness typing for O/S implementation. Using these features, type-safe mobile processes and concurrency are implemented. The latter uses a first class continuation approach and implements scheduling at system level.

Most recently, Peng Li et al [LMJT07] have presented a micro-kernel (substrate) based design for the concurrent RTE of GHC, including support for software transactional memory (STM). This complements our work, which focuses on control of parallelism, and we intend to combine the design of our interface with that currently produced for GHC.

## 4.2 Design aims of the generic RTE ARTCoP

### 4.2.1 Simplest kernel

We aim to provide support for parallel programming from the conceptual, language designer perspective. A major goal in designing a generic runtime environment is to explore *how many* of the coordination constructs can be specified at higher levels of abstraction, and to identify the *minimal and most general runtime support* for parallel coordination. As in the Eden implementation, major parts are implemented in a high-level language, which keeps the kernel runtime small. Following a functional paradigm has the advantage that specifications can, more or less, be executed directly and that it facilitates theoretical reasoning such as correctness proofs.

### 4.2.2 Genericity

Our study concentrates on identifying and structuring the general requirements of parallel coordination, with the only assumption that concurrent threads are executing a functionally specified computation, explicitly or implicitly coordinated by functional-style coordination abstractions.

The genericity we aim at is two-fold: By providing only very simple actions as primitive operations, our system, by design, is not tied to particular languages. We avoid language-specific functionality whenever possible, thus ARTCoP supports a whole spectrum of coordination languages. Secondly, the coordination system can be used in combination with different computation engines and is not restricted to a particular virtual machine. Furthermore, this coordination makes minimal assumptions on the communication between processing elements (PEs). We thus concentrate *key aspects of parallelism* in one place, without introducing specific coordination constructs or being tied to a certain parallelism model.

### 4.2.3 Multi-level system architecture

High-level parallel programming manifests a critical trade-off: providing operational control of the execution while abstracting over error-prone details. In our system, we separate these different concerns into different levels of a multi-level

system architecture. As shown in Figure 4.1, ARTCoP follows the concept of a *micro-kernel*, proven useful in the domain of operating system design.

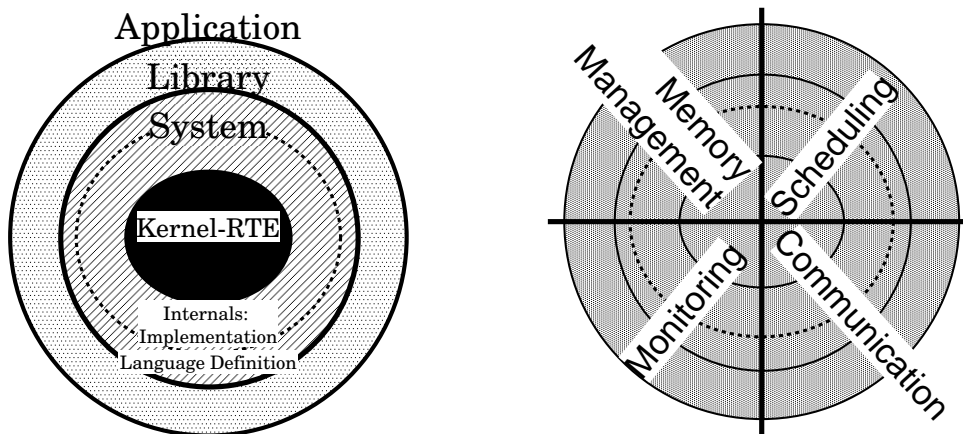


Figure 4.1: Layer view of ARTCoP Figure 4.2: Component view of ARTCoP

At *Kernel level*, the most generic support for parallelism is implemented. The system offers explicit asynchronous data transfer between nodes, means to start and stop computations, as well as ways to retrieve machine information at runtime. Operations at this level are very simple and general. *System Modules* build on the kernel to restrict and combine the basic actions to higher-level constructs, i.e. the constructs of a proper parallel functional language. The runtime support is necessarily narrowed to a special model at this level. The implemented parallel coordination language is nothing else but the interface of the system level modules. At *Library level* and *Application level*, concrete algorithms, or higher-order functions for common parallel algorithmic patterns (called skeletons [RG03]) can be encoded using the implemented language.

Focusing more on functionality and modularity, the kernel can be divided vertically into four interacting components, as shown in Figure 4.2: Parallel subtasks are created and sent to other processing elements (PEs) for parallel execution by the *scheduling* component, which controls the local executing units. Explicit *communication* between several scheduler instances on different PEs is needed to coordinate and monitor the parallel execution. The *memory management* component is responsible for (de-)allocating dynamic data and distributing it over the available machines, interacting in this task with the communication component. Explicit message passing is possible, but not mandatory for data communication, and it is possible to implement a shared address space instead. In order to decide which PE is idle and suitable for a parallel job, static and dynamic system information is provided by a *monitoring component*.

### 4.2.4 High-level scheduler control

The key issue in efficiently using a wide-area network infrastructure for parallel computations is to control the parallel subtasks that contribute to the overall program, and to *schedule* the most suitable task for execution, depending on the current machine load and connectivity (whereas efficiently combining them is an algorithmic issue). Likewise, modern multicore CPUs will often expose uneven memory access times and synchronisation overhead. Parallel processes have to be placed with minimal data dependencies, optimised for least synchronisation, and dynamically consider system load and connectivity. ARTCOP aims to be a common framework for different coordination concepts. Adaptive scheduling support will thus be specified in the *high-level language* and not in the runtime system.

## 4.3 Configurable Haskell scheduler framework

We propose a parallel RTE which allows system programmers and language designers to define appropriate scheduling control at the system level in Haskell. In our parallel system, the scheduler is a monadic Haskell function using an internal scheduler state, and *monitors* all computations on one machine. Subtasks are activated and controlled by a separate manager thread, which can take into account properties of the subtask as well as static and dynamic machine properties. The scheduler thread runs concurrently to the controlled computations and relies on a low-level round-robin scheduler inside the RTE. To specify it, we use the state monad and features of Concurrent Haskell, combining stateful and I/O-actions by a monad transformer [KW93].

The internal scheduler state type depends on the concrete job type and leads to another type class, which provides a start state and a termination check. A third type class `ScheduleMsg` relates jobs and state to messages between the active units and provides a message processing function. Table 4.1 summarises the overloaded functions in the scheduler classes. The scheduler `schedule` which is provided as a default (shown in Fig. 4.3) only starts the main computation (i.e. the jobs), then repeatedly passes control to one of the job threads (calling `kyield`, a redefinition of `yield` from Concurrent Haskell), and checks for termination, and returns the final scheduler state upon termination (using function `get` from the state monad).

Parallel tasks in a coordination language implemented by ARTCOP will appear as a new type of job. The scheduling behaviour is consequently defined at system level in Haskell. Haskell's type system allows one to specify the respective scheduler for a certain kind of parallelism by overloading instances of type class `ScheduleJob`. Thus, language designers will not deal with runtime system code, but simply define the scheduling for such jobs at system level, extending this



Table 4.1: Overview of class functions (implemented at system level)

<code>type StateIO s a = StateT s IO a</code>	type alias combining State and IO monad
<pre>class ScheduleState st where startSt      :: st killAllThreads :: StateIO st () checkTermination :: StateIO st Bool checkHaveWork  :: StateIO st Bool</pre>	<p>the initial state of the scheduler shutdown function check state, return whether to stop check state, return whether any local work available</p>
<pre>class ScheduleJob job st   job -&gt; st where runJobs  :: [job] -&gt; IO st schedule :: [job] -&gt; StateIO st st forkJob  :: job -&gt; StateIO st ()</pre>	<p>run jobs with default start state schedule jobs, return final state fork one job, modify state accordingly</p>
<pre>class ScheduleMsg st msg   st -&gt; msg where processMsgs :: [msg] -&gt; StateIO st Bool</pre>	<p>process a set of message for the scheduler, modify state accordingly. Return <code>True</code> immediately if a global stop is requested.</p>

```
runJobs jobs = evalStateT (schedule jobs) startSt
schedule (job:jobs) = do forkJob job
                        schedule jobs
schedule [] = do liftIO kYield           -- pass control
                 term <- checkTermination -- check state
                 if term then get        -- return final state
                 else schedule ([]::[job]) -- repeat
```

Figure 4.3: Default scheduler

Haskell scheduling loop.

As a simple extension for parallel execution, every machine could control a pre-determined subset of the jobs, running one instance of the scheduler. With the appropriate RTE support (described next), this behaviour can be expressed easily, even without the need to modify the default instances, by the function shown in Fig. 4.4. The code shown here uses two hard-wired kernel operations, indicated by a leading `k`: `kRFork` sends an IO action to a given processor, where it is executed asynchronously, providing basic remote execution control. The other operation – `kNoPe`, a system observer primitive – returns the number of processors in the system. In the example, the `runparallel` action retrieves the number of available PEs (`kNoPe`), portions the available jobs accordingly, and (asynchronously) spawns a scheduler instance which runs the respective subset of the jobs on each other PE (`kRFork`).

Further primitive operations are needed to get more dynamic system informa-

Table 4.2: Overview of primitive operations (provided by the kernel)

Functionality at Kernel Level (primitive operations)	
<b>Execution Control:</b>	
<code>kRFork :: PE -&gt; IO() -&gt; IO()</code> <code>kFork  :: IO() -&gt; IO ThreadId</code> <code>kYield :: IO()</code>	start a remote computation start a local thread (Conc. Haskell) pass control to other threads (Conc. Haskell)
<b>Explicit Communication:</b>	
<code>kOpenPort :: IO( ChanName' [a],[a])</code> <code>kSend  :: ChanName' [a] -&gt; a -&gt; IO()</code>	open a stream inport at receiver side, return port handle and placeholder basic communication primitive, send an element of type a to a receiver (a port handle)
<b>System Monitoring:</b>	
<code>kThisPe,kNoPe :: IO Int</code> <code>kThreadInfo :: ThreadId -&gt; IO ThreadState</code> <code>kPEInfo  :: Int -&gt; IO InfoVector</code>	get own node's ID / no. of nodes get current thread state (Runnable, Blocked, Terminated) info about a node in the system (cpu speed, latency, load, location etc)

```

runParallel jobs = do pes <- kNoPe
  let (mainjobs:others) = splitJobs pes jobs
      doJobs pe = do runJobs (others!!(pe-2))
                    return ()
      mapM_ (\pe -> kRFork pe (doJobs pe)) [2..pes]
      runJobs mainjobs

```

Figure 4.4: A simple parallel scheduler

tion and to allow communication between different Haskell execution units. The basic kernel support we assume, which is only a minor extension to the primitives for the Eden implementation, can be grouped into scheduler control, communication, and system information. All primitive operations provided by the kernel (indicated by the leading `k`), and their types, are shown in Table 4.2.

As in the Eden implementation, the operations shown here are not directly implemented, but provided by a minor wrapper module to use Haskell data structures and monads. Internally, the operations will call real primitives directly implemented in the kernel. Figure 4.5 shows that most of the functionality is already provided by the Eden implementation primitives, only the monitoring functionality `kThreadInfo` and `kPEInfo` rely on newly implemented primitives. Whereas `kThreadInfo` simply observes an available runtime system thread state, `kPEInfo` requires a more elaborate implementation by a monitoring component (outlined below in 4.5).

---

```

kRFork :: Int -> IO () -> IO ()
kRFork = spawnProcessAt          -- == sendData (Instantiate pe) action

kFork  = Control.Concurrent.forkIO -- from concurrent Haskell
kYield = Control.Concurrent.yield  -- from concurrent Haskell

-- channel creation, restricted to streams by the given type
kOpenPort :: IO (ChanName' [a], [a])
kOpenPort = createC
kSend :: ChanName' [a] -> a -> IO ()
kSend c d = do connectToPort c
               sendData Stream d -- stream communication only

kThisPe, kNoPe :: IO Int
kThisPe = thisPe
kNoPe   = noPe

```

---

Figure 4.5: Kernel operations, as far as implemented by EDI primitives

## 4.4 Explicit communication

If additional jobs are created dynamically, they may be transmitted to a suitable PE, and received and activated by its scheduling loop. The scheduler instances may also exchange *requests* for additional work and receive jobs as their answers. This model requires communication between the scheduler instances. The kernel provides the infrastructure for explicit message passing between any two running threads, which we have already presented in the context of the Eden implementation. The ports are created from Haskell by `kOpenPort` and managed by the kernel internally. A `kOpenPort` returns a placeholder for the stream, and a Haskell port representation to be used by senders for `kSend`. The difference to EDI's `createC` is that communication support in the Generic RTE is intended mainly for scheduler instance communication, and thus restricted to *open streams*. Figure 4.5 shows that the type of created channels is restricted to lists.

As in EDI, sending data by `kSend` does not imply any evaluation; data has to be explicitly evaluated to the desired degree prior to sending.

Startup synchronisation and stream communication between all scheduler instances are easy to build on this infrastructure, using a simple startup protocol for exchanging the scheduler instance's channels. The scheduler also needs to communicate with locally running threads (e.g. to generate new jobs), which can be handled by enabling all threads to write to the scheduler stream. It is up to language designers to define suitable message types for parallel coordination between the schedulers, accompanied by an instance declaration which provides the message processing function in the class `ScheduleMsg`.

Figure 4.6 sketches a scheduler for such a language, assuming the existence of a globally managed job pool. If an instance runs out of work, it will send a request.

```
instance ScheduleJob MyJob MySchedulerState where
  schedule (j:js) = do forkJob j
                      mapM_ addToPool js
                      schedule ([] :: [MyJob])
  schedule empty = do stop <- do { ms <- receiveMsgs ; processMsgs ms }
                      term <- checkTermination
                      if (term || stop)
                        then do { killAllThreads; get }
                        else do work <- checkHaveWork
                               if (not work)
                                 then sendRequest
                                 else liftIO kYield
                      schedule empty
```

---

Figure 4.6: Scheduler for a parallel job-pool

It will eventually receive an answer, and the next call to `processMsgs` will activate the contained job. This example enables reasoning about appropriate workload distribution and the consequences and side conditions, while the scheduling loop itself remains small and concise. All essential functionality is moved from the scheduling loop into separate functions, e.g. we leave completely unspecified how a scheduler instance decides that it needs work (in `checkHaveWork`), and how jobs are generated and managed in the job pool, and the message type. All these aspects are defined in the helper functions, allowing a clear, structured view on the scheduling implemented. Modifying the job scheduling policy concentrates on these (`checkHaveWork`, `addToPool`, and `sendRequest`) instead of a big monolithic loop.

## 4.5 System monitoring

Programmable scheduling support at system level requires knowledge about static and dynamic system properties at runtime. Our system is geared towards adaptive techniques developed for GridGUM, GpH on computational Grids [ATLM06], and the kernel will be extended to provide the necessary information. For location awareness, we have `kNoPe` for the total number of PEs in the parallel system, and `kThisPe` for the own PE. Another primitive, `peInfo :: PE -> IO InfoVector` is supposed to return a vector of data about the current system state of one PE. This information will be continuously collected by the kernel and held in local tables *PEStatic* and *PEDynamic*.

**Load information at system level:** A list of load information represented in a Haskell data structure `PEInfo` is a self-suggesting component of the scheduler state in many cases. The concrete selection, postprocessing and representation of the information provided by the kernel depends on how the scheduler at system

level wants to use the information. An example of a Haskell type `PEInfo` is shown in Fig. 4.7. It includes selected components of the scheduler state: the number of threads controlled by the local scheduler, and how many sparks (potential parallel computations in GUM) it holds. Other information comes directly from the kernel, as it cannot be obtained at system level alone: communication latency (continuously measured when exchanging messages), overall CPU load, and static machine characteristics.

---

```

data PEInfo = PE { runQ_length :: Int,    -- system level information
                  noOfSparks  :: Int,
                  clusterId   :: Int,
                  clusterPower:: Double,
                  cpuSpeed    :: Int,    -- kernel level information
                  cpuLoad     :: Double,
                  latency     :: Double,
                  pe_ip       :: Int32,
                  timestamp   :: ClockTime }

startup :: StateIO s ()
startup = do infos <- buildInfos -- startup, returns initial [PEInfo]
           let ratios = zipWith (\lat str -> fromIntegral str / lat)
                               (map latency infos) (map cpuSpeed infos)
               myVote = fromJust (findIndex (== maximum ratios) ratios)
           votes <- allGather myVote
           setMainPE (1 + hasMostVotes votes)

```

---

Figure 4.7: System level code related to load information

As exemplified in the figure, the scheduler can do arbitrary computations on `PEInfo` structures. For instance, to start the computation on a “strong” machine with good connectivity, all PEs might *elect* the main PE by a strength/latency ratio. Each PE votes for a relatively strong neighbour, where neighbourhood is a function of latency, varying for different electing PEs. A collective (synchronising) message-passing operation `allGather` is easily expressed using explicit communication. Referential transparency guarantees that all PEs will then compute the same value without further synchronisation.

## 4.6 Example: Adaptive scheduling in GpH

### 4.6.1 Hierarchical task management

We now express the scheduler of the GUM RTE [THM<sup>+</sup>96], which implements the GpH parallel extension of Haskell, in terms of the generic framework presented in the previous section. Instead of the single global job pool we sketched in the previous example, GpH maintains local job pools in each PE, which contain

“sparks”. We want to model not only GUM, but the GridGUM extension to GpH, which, in short, has the following characteristics [ATLM06]:

- As in plain GpH, *hierarchical task management* distinguishes between potential parallelism (sparks) and realised parallelism (threads); the former can be handled cheaply and is the main representation for distributing load; the latter, representing computation, is more heavy-weight and fixed to a processor;
- Its *adaptive load distribution* uses information on latency and load of remote machines when deciding how to distribute work.

We will see that, in this high-level formulation of the scheduler, the code modifications necessary to realise these two features are fairly simple. Hereafter, we first describe how to model the hierarchical task management in GUM. These changes only affect the scheduling component. In tuning load distribution, we then interact with the monitoring and communication components.

### GUM scheduler state

First we specify the machine state in the GUM RTE (shown in Fig. 4.8). As discussed earlier, it is a triple consisting of:

- a *thread pool* of all threads; these are active threads controlled by the scheduler, each with its own stack, registers etc;
- a *spark pool* of all potential parallel tasks; these are modelled as pointers into the heap;
- *monitoring information* about load on other PEs; this information is kept, as a partial picture, in tables on each processor;

We model the two pools and load infos as simple lists (more efficient container types could be used instead). The `GumJob` type is left unspecified for now.

The messages between running PEs must be specified as well: Messages for the work-stealing functionality of GUM are FISH (by which an idle PE fishes for work at other PEs), and the respective reply SCHEDULE, which contains a job to execute. Besides, we will find the global stop request and other internal messages, left out for now.

Now making `GumState` an instance of `ScheduleState`, we specify how to handle and run jobs, and especially how the scheduler should FISH for work when idle. Equally, we will define the message processing for GUM messages as an instance of `ScheduleMsg`.

---

```

type Threadpool = [ThreadId]
type Sparkpool = [GumJob]
data PEInfo = PE ... -- see before
data GumState = GSt { gThreads :: [ThreadId]
                    , gSparks  :: Sparkpool
                    , gLoads   :: [PEInfo] }
                    deriving Show
data GumMsg = FISH [PEInfo] Int -- steal work, share PEInfo on the way
            | SCHEDULE [PEInfo] GumJob -- give away work (+ share PEInfo)
            | GSTOP | ... -- and other (system) messages

```

---

Figure 4.8: Types for modelling the GUM system

```

instance ScheduleJob GumJob GumState where
  runJobs jobs = evalStateT (initLoad >> (schedule jobs)) startSt
  forkJob (GJ job) = error "left out for now"

  schedule (j:js) = do forkJob j
                       schedule js
  schedule empty = do
    (runThrs, blThrs) <- updateThreadPool -- update and
    term <- checkTermination -- (1) check local state
    if term then do bcast GSTOP -- finished: return state
                 get
                 else do localWork <- if runThrs > 0 -- (2) local work available?
                          then return True -- yes: runnable thread
                          else activateSpark -- no: look for spark
    stop <- if localWork
            then do reqs <- readMs
                   processMsgs reqs
            else do sendFish -- (3) get remote work
                   waitWorkAsync
    if stop then do killAllThreads -- finished: return state
                  get
                  else do liftIO kYield -- (4) run some threads
                          schedule empty

-- essential helper functions:
activateSpark :: StateIO GumState Bool -- tries to find local work
sendFish :: StateIO GumState () -- sends request for remote work
waitWorkAsync :: StateIO GumState Bool -- blocks on receiving messages

updateThreadPool :: StateIO GumState (Int,Int)
updateThreadPool = do
  gst <- get
  tStates <- liftIO (mapM kThreadInfo (gThreads gst))
  let list = filter (not . isFinished . snd) (zip threads tStates)
      blocked = length (filter (isBlocked . snd) list)
      runnable = length (filter (isRunnable . snd) list)
  put (gst {gThreads = map fst list })
  return (runnable, blocked)

```

---

Figure 4.9: GUM scheduler

## Plain GUM scheduler

The code for the GUM scheduler is summarised in Figure 4.9. The arguments to `schedule` are jobs to be executed. These jobs are forked using a kernel routine, and added to the thread pool (`forkJob`). The case of an empty argument list describes how the scheduler controls the machine’s workload. First the scheduler checks for termination (1). Then the scheduler checks the thread pool for runnable tasks, otherwise it will try to activate a local spark (2). If local work has been found, it will only read and process messages. The handlers for these messages are called from `processMsgs`, which belongs to the communication module. If no local work has been found, a special FISH message will be sent to search for remote work (3). Finally, it yields execution to the micro-kernel, which will execute the next thread (4) unless a stop message has been received, in which case the system will be shut down. The thread pool is modeled as a list of jobs, and `updateThreadPool` retrieves the numbers of runnable and blocked jobs.

### 4.6.2 Adaptive load distribution mechanisms

The above mechanism will work well on closely connected systems but, as measurements show, it does not scale well on Grid architectures. To address shortcomings of the above mechanism on wide-area networks, we modify the thread management component for better load balancing, following concepts of the adaptive scheduling mechanism for computational Grids [ATLM06]. The key concept in these changes is *adaptive load distribution*: the behaviour of the system should adjust to both the static configuration of the system (taking into account CPU speed etc.) and to dynamic aspects of the execution, such as the load of the individual processors. One of the main advantages of our high-level language approach to system-level programming is the ease with which such changes can be made. Looking for remote work (`sendFish` and its counterpart in `processMsgs`) and picking the next spark (`activateSpark`) are the main functions we want to manipulate in tuning scheduling and load balancing for wide-area networks. Note that by using index-free iterators (such as `filter`), we avoid risks of buffer-overflow. Furthermore, the clear separation of stateful and purely functional code makes it easier to apply equational reasoning.

Adaptive load distribution deals with: *startup*, *work locating*, and *work request handling*, and the key new policies for adaptive load distribution are that work is only sought from relatively heavily loaded PEs, and preferably from local cluster resources. Additionally, when a request for work is received from *another cluster*, the receiver may add more than one job if the sending PE is in a “stronger” cluster. The necessary static and dynamic information is either provided by the kernel or added and computed at system level, and propagated by attaching load information to every message between PEs (as explained in Section 4.5).



**Placement of the main computation.** During startup synchronisation, a suitable PE for the main computation is selected, as already exemplified in Section 4.5. GRIDGUM 2 starts the computation in the 'biggest' cluster, i.e. the cluster with the largest sum of CPU speeds over all PEs in the cluster, a policy which is equally easy to implement.

**Work location mechanism.** The Haskell code in Figure 4.10 shows how the target PE for a FISH message is chosen adaptively by `choosePE`. A ratio between CPU speed and load (defined as `mkR`) is computed for all PEs in the system. Ratios are checked against the local ratio `myRatio`, preferring nearby PEs (with low latency, sorted first), to finally target a nearby PE which recently exposed higher load than the sender. This policy avoids single hot spots in the system, and decreases the amount of communication through high-latency connections, which improves overall performance.

**Work request handling mechanism.** To minimise high-latency communication between different clusters, the work request handling mechanism tries to send multiple sparks in a SCHEDULE message if the work request has originated from a cluster with higher relative power (see Figure 4.11). The relative power of a cluster is the sum of the speed-load ratios over all cluster elements. If the originating cluster is weaker or equally strong, the FISH message is served as usual.

---

```

sendFish :: StateIO GumState ()
sendFish = do infos <- currentPEs -- refresh PE information
              me <- liftIO kThisPe
              pe <- choosePe me
              liftIO (kSend pe ( FISH infos me ))

-- good neighbours for work stealing: low latency, highly loaded
choosePe :: Int -> StateIO GumState (ChanName' [GumMsg])
choosePe me = do
  lds <- gets gLoads
  let mkR pe = (fromIntegral (cpuSpeed pe)) / (cpuLoad pe)
      rList = [ ((i,mkR pe), latency pe) -- compute 'ratio'
              | (i,pe) <- zip [1..] lds ] -- keep latency and PE
      cands = filter ((< myRatio) . snd) -- check for high load
              (map fst -- low latencies first
               (sortBy (\a b -> compare (snd a) (snd b)) rList))
      myRatio = (snd . fst) (rList!!(me-1))
  if null cands then return (port 1) -- default: main PE
  else return (port ((fst . head) cands))

```

---

Figure 4.10: GRIDGUM 2 work location algorithm

```

instance ScheduleMsg GumState GumMsg where
  processMsgs ((FISH infos origin):rest) = do processFish infos origin
                                              processMsgs rest
  processMsgs ((SCHEDULE ...) :rest) = ...

processFish :: [PEInfo] -> Int -> StateIO GumState ()
processFish infos orig = do
  updatePEInfo infos          -- update local dynamic information (1)
  me <- liftIO kThisPe
  if (orig == me) then return () -- my own fish: scheduler will retry
  else do
    new_infos <- currentPEs    -- compare own and sender cluster (2)
    let info    = new_infos!!(orig-1)
        myInfo  = new_infos!!(me-1)
        amount  = if (clusterPower info > clusterPower myInfo)
                    then noOfSparks myInfo `div` 2 -- stronger: many
                    else 1                        -- weak or the same: one
    sparks <- getSparks amount True -- get a set of sparks (3)
    case sparks of
    [] -> do target <- choosePe me -- no sparks: forward FISH
            liftIO (kSend target (FISH new_infos orig))
    some -> liftIO (sequence_ -- send sequence of SCHEDULE messages
                  (map ((kSend (port orig)).(SCHEDULE new_infos)) some))

```

---

Figure 4.11: GRIDGUM 2 work request handling algorithm

In Figure 4.11, after updating the dynamic information (1), the sender cluster is compared to the receiver cluster (2), and a bigger amount of sparks is retrieved and sent if appropriate (3). In this case, the RTE will temporarily switch from passive to active load distribution.

## 4.7 Feasibility study: Virtual shared memory management in Haskell

### 4.7.1 Virtual shared memory in GpH

In the GpH scheduler shown previously, we have seen how work is distributed adaptively, as sparks which are sent to remote machines across the network. The GpH programming model and its GUM implementation assumes that exported and activated sparks are evaluated remotely. Later, in the main computation, the result of such a spark evaluation will be needed on the exporting machine. The underlying implementation is then responsible for synchronisation of the two accesses to the same data, and therefore has to emulate a global shared address space on a distributed architecture, Virtual Shared Memory (VSM).

Up to now, we have left out the memory management component from our

description. We want to model this part rather generally, but make the basic assumption that the implemented computing engine uses garbage collection and a graph heap for evaluation. First of all, it is clear that the elementary graph heap management – garbage collection – cannot possibly be handled from the Haskell level, since the execution of Haskell memory management code needs an underlying heap and management. Thus garbage collection itself should remain completely transparent to the Haskell system code. But aside from this elementary local heap management, the question arises whether it is possible – and favourable – to lift some parallelism-specific parts, especially global address space simulation, to Haskell level.

### Global addresses and weighted reference counting

The basic technique used for the distributed shared memory implementation in GUM is *weighted reference counting* [JL92, Bev89]: globally unique addresses accompanied by a *weight*, indicating whether remote references to the data in question exist on other machines in the system. In a local heap, the data is either available, or represented by a placeholder. The semantics of such placeholders is that represented data is fetched from the possessing remote machine automatically as soon as a thread tries to evaluate the placeholder. When data is exported to another machine for the first time, it receives a fresh global address, for which half of the initial weight remains on the exporting machine. Whenever further copies of globalised data are made, the weight is split in half. Local garbage collection will either evacuate placeholders, or else detect that the represented data is not needed any more for local evaluation, in which case the weight is *returned* to the possessing machine. Furthermore, garbage collection must keep alive any data which is registered with a global address: Remote references to local data might exist, even if local evaluation does not refer to it any more. By checking the local weight against its total sum, garbage collection at the possessing machine detects whether all remote references have returned their weight. In this case, the data can be garbage-collected.

The described mechanism can be, and *has been* in the past, entirely implemented in the runtime system kernel; and the decision where to implement global addressing will turn out to be an all-or-nothing question, since it involves complex interactions between evaluation (blocking on placeholders), global address implementation, communication semantics (to model spark export at system level) and, last but not least, garbage collection (of placeholders and globalised data) – elementary kernel parts. However, there is some potential to profit from lifting certain parallelism-related parts of memory management to Haskell level.

### Potential benefit of high-level VSM

To begin with, all management functions for global addresses are straightforward and easily expressed in a high-level language. Coding them in C is a lot more

error-prone and cumbersome. Additionally, the management code in Haskell would be much easier to read.

Secondly, since garbage collection remains transparent, it is completely decoupled from global address management. From the fact that weighted reference counting is a garbage collection algorithm, one might think in the first instance that both have to be implemented together, and are inherently kernel tasks, but the opposite is the case: In our architecture, a Haskell implementation of the global address mechanism will use a mapping of addresses to data as part of the scheduler state. As long as the scheduler is running, there will automatically be a reference to any data to which a global address has been assigned, thus *automatically* keeping alive all remote data. Whenever a remote reference to globalised data is garbage-collected, the other PE returns the respective weight, realisable by *finalizers in Haskell* [JME99]. Finalizers are specified as Haskell code attached to data of a special type, which is executed as soon as the data is garbage-collected. The handler for such a weight-returning message will update the table of global addresses accordingly, and eventually remove data from the table when all weight has been returned. The garbage-collection parts of a system-level implementation of VSM management will thus come more or less for free; only the weight-returning functionality has to be realised (by attaching finalizers to the placeholders).

The GHC implementation of Haskell provides some means to influence memory management: The module `System.Mem.Weak` allows one to create *weak pointers* (not taken into account during garbage collection), and to give explicit *finalisers* for Haskell data structures. Using `Foreign.StablePtr` in the foreign function interface (FFI), one can create fixed addresses for heap objects, and cast them to a simple pointer type. Usage of both features is documented in [JME99]; the code we will present is inspired by that practical paper.

## 4.7.2 Global address management in Haskell

### Global address data structures

Weighted reference counting requires mappings from global addresses to the heap and vice-versa on each participating PE. These tables could become part of a scheduler state, or else be maintained as global mutable data, a *constant applicative form* (CAF) defined at the top-level of the program (as sketched by Peyton-Jones in [Pey00]). The latter “CAF trick” alternative is more convenient, since every running thread can access the table independently (otherwise, a thread-scheduler communication would be required for each data request).

We first give types for global addresses and provide a supply for (locally) unique IDs, from which a global address is constructed. This `idSupply` is implemented by a CAF at the top level of the program. The stateful ID counter is wrapped in a mutable variable (MVar). Each local thread can now equally call functions using

the `idSupply` as an implicit parameter (as shown in `mkFreshId`), and the counter is automatically protected against concurrent accesses by the surrounding `MVar`.

```

type Weight = Int
data GA = GA PE Int Weight
    deriving Show

{-# NOINLINE idSupply #-} -- CAF trick:
idSupply :: MVar Int      -- accessible to all local threads
idSupply = unsafePerformIO (newMVar 1)

mkFreshId :: IO Int
mkFreshId = do next <- takeMVar idSupply
               putMVar idSupply (next+1)
               return next

newGA :: StateIO GumState' GA
newGA = do id <- liftIO mkFreshId
           pe <- liftIO kThisPe
           return (GA pe id initWeight)
initWeight :: Weight
initWeight = ... -- system constant, power of 2

```

**Typing the address table** is the first obstacle on the way. What type of data do we get when looking up the global address of something? What needs to be stored in the table that maps global addresses (GAs) to data and vice-versa? *Any* data item can be assigned a global address. To store all these GAs in one and the same Haskell structure, we have to cast the original types to something homogeneous. We could use *dynamic types*, a Haskell 98 extension available in GHC and Hugs. However, since we will need to perform reverse lookups in the address tables, it is desirable that addresses of globalised data should be ordered, and remain stable over garbage collections. We use `StablePtrs` (which provide exactly this in GHC) in the first instance, and additionally cast to `Ptr()` to obtain a list of homogeneous items.<sup>1</sup>

```

{-# NOINLINE galaVar #-}
galaVar :: MVar [(GA,Ptr())]
galaVar = unsafePerformIO (newMVar [])
instance Eq GA where -- do not look at weight when comparing GAs
    (==) (GA pe1 id1 _) (GA pe2 id2 _) = pe1==pe2 && id1 == id2

insertGA :: GA -> a -> IO ()
insertGA ga x = do takeMVar galaVar
                   p <- liftIO (newStablePtr x)           -- p :: StablePtr a
                   p' <- liftIO (castStablePtrToPtr p)    -- p' :: Ptr ()
                   putMVar galaVar ((ga,p'):gala)

```

<sup>1</sup>We are interested in general feasibility and do not consider any performance issues here, therefore the simple list implementation. Mapping between global addresses and data can be realised by any other container structure, optimised for the most common lookup.

```

lookupGA :: GA -> IO (Maybe a)
lookupGA ga = do gala <- readMVar galaVar
  case lookup ga gala of
    Nothing -> return Nothing
    Just p' -> do p <- castPtrToStablePtr p'
      x <- deRefStablePtr p
      return (Just x)

lookupData :: a -> IO (Maybe GA)
lookupData x = do p <- (newStablePtr x >>= castStablePtrToPtr)
  t <- readMVar galaVar -- obtain and invert table
  let t' = map (\(x,y) -> (y,x)) t
  return (lookup p t')

```

We have a similar problem when we model the spark pool: *Subexpressions of any type* may be sparked during execution, and we need to store sparks in one common Haskell data structure. Additionally, sparks are either evaluated *locally* or sent away to another PE that assigns a global address and thereby acknowledges the data. We could, of course, simply identify `IO()` and `GumJob`. Evaluation of the contained data could be enforced as a side effect:

```
(sparkclosure 'seq' return ())::IO()
```

The problem with this representation is that we will not get at the evaluated data any more in order to send it back when needed (however, working for a simulation, since it does the same work). It is more practical to use the pointer cast for sparks as well, and to identify the yet unspecified `GumJob` type with `Ptr()`. Locally activating a spark will thus involve a cast back to the original type.

```

type GumJob = Ptr()

putInSparkPool :: Typeable a => a -> StateIO GumState ()
putInSparkPool x = do p <- liftIO (newStablePtr x >>= castStablePtrToPtr)
  modify (\ st@(GSt {gSparks = pool} ) ->
    st {gSparks = (p:pool)})

activateSpark :: StateIO GumState Bool
activateSpark = do st <- get -- get state
  let sparks = gSparks st
  case sparks of
    [] -> return False
    (s:rest) -> do put (st {gSparks = rest} )
      forkJob s
      return True

forkJob :: GumJob -> StateIO GumState ()
forkJob ptr = do stPtr <- castPtrToStablePtr ptr
  dat <- deRefStablePtr stPtr
  tid <- liftIO (kFork (dat 'seq' return ()))
  modify (\ st@(GSt {gThreads = ts}) -> st {gThreads = tid:ts})

```

**Weight-related functionality for GAs** can be realised using *finalisers* and weak pointers. Whenever a GA for remote data is inserted into the table, a

weight-returning finaliser is attached to the placeholder pointed at by the GA. When the placeholder (for data on another PE) is garbage-collected, its current local weight is retrieved from the table and returned to the possessing PE. As mentioned, the GAs assigned to local data will remain alive as long as they are in the table. The table needs to be cleaned up regularly when weight is returned by other PEs.

Another essential functionality is to half the stored weight in the table, and to return a copy of the GA for export to another PE. We will not go into further details on this, since more serious problems need to be solved.

### 4.7.3 Haskell heap access from inside Haskell

Having reached this point, we have to state how structures in the Haskell heap will receive a global address with weight, and move from one PE to another. We have prototyped code and will now sketch a Haskell-based solution and the required kernel functionality for moving heap elements from one heap to another.

First of all, the explicit communication primitive `sendData` normally *copies* the data to be sent, but it can as well *move* the data instead, and we could just use a kernel implementation of VSM, as we have said. But our intention is to investigate feasibility, advantages and drawbacks of a Haskell (system) implementation.

After spark *creation*, the data (unevaluated) is still available, since it might be that the spark never gets fetched. Only when a spark is *exported*, unevaluated parts inside its computation subgraph should be replaced by placeholders with blocking semantics, and global addresses assigned. When a thread evaluates a closure, it may block, observable only by the RTE. When data arrives, the thread must be unblocked, and the blocking node updated. It should be noted that manipulating evaluation and blocking normally is kernel business and needs to be disclosed to system level now.

**Moving globalised data to a remote PE** is not too hard to encode in Haskell. A primitive operation which (unsafely!) `replaces` one heap object by another can be easily added to GHC.

```
replaceBy# :: a -> a -> IO ()
replaceBy# c1 c2
  = ...-- replaces c1 by indirection to c2, activates threads blocked on c1
```

We need to model how data is exported and replaced by blocking placeholders, and how data is retrieved when the placeholder is evaluated. Placeholders can be created from Haskell by `createC`, which simultaneously creates a channel to receive data. Now, when *moving* away heap structures, they must be replaced not only by normal placeholders, but evaluation must trigger an action to actively fetch back the data for the blocked local evaluation (using the channel created).

---

```

moveData :: a -> ChanName' a -> StateIO GumState ()
moveData x receiver@(Chan pe _ _) = do
  maybeGaChan <- if (isUnevaluated x) -- include GA request if not NF data
  then do
    (chan,blocker) <- createC -- data placeholder
    (gaChan, newGa) <- createC -- GA placeholder
    -- evaluation of x should trigger the following code:
    let fetch_x = unsafePerformIO (
          kSend pe (FETCH chan newGa) -- remote scheduler to reply
          y <- (castPtrToStablePtr blocker >>= deRefStablePtr )
          return y
        liftIO (replaceBy# x fetch_x)
        insertGA newGa fetch_x
        return (Just gaChan)
    else return Nothing
  toSend <- serialise x -- pack subgraph into some byte store...
  send pe (RESUME toSend maybeGaChan) -- send away

```

---

Figure 4.12: Sketch: Replacing heap data by a fetching node

The data to be moved is a subgraph in the heap, serialised into a suitable data structure and sent to the receiver. If the data is already evaluated (we will revert to this later), it is not moved but only copied, without any problems. Otherwise, data in the local heap will be replaced by an I/O action which triggers to fetch the real data back when needed, using a fresh channel to receive it.

Figure 4.12 sketches how data  $x$ , which is already globalised and requested by another PE, can be moved and replaced by a fetch node. What makes this very complicated is the fact that the global address has to be updated to point to the new location of  $x$ . The new global address has to be assigned by the receiving PE and cannot be propagated to other PEs, thus the sending PE has to store a mapping between GAs. We need to modify the mapping table so that it will also be able to hold other global addresses, and the reply message to the request has to contain a channel to send the new global address.

We would have to specify as well how data is globalised, i.e. when exporting a spark. The essentials are similar to the data export shown, except that a *new* global address has to be assigned by the receiver (we have given all needed helper functions in code fragments earlier).

The *essential obstacle* in this approach is yet another problem: The subgraph needed to compute the data  $x$  may contain unevaluated inner parts and references to other globalised data. Any unevaluated parts could simply be copied, leading to potential duplicate evaluation. Yet, duplicating the global references requires to adjust the weights of their global addresses. However, the “subgraph reachable from a node” is traversed, up to now, inside the kernel only, transparent to Haskell level. To be able to access the inner nodes directly from Haskell level, we still need to proceed further, to define the heap graph traversal *in Haskell* as well.



## Heap graph traversal routines

Communication of Haskell data structures is a salient part of any parallel Haskell on distributed memory, especially in implementations which emulate shared memory by global addresses and an internal fetch protocol. We concretely aim at the GUM implementation of Glasgow parallel Haskell. Long-standing experience with implementations of GpH and Eden has shown that Haskell data communication routines are by far the most problematic code in a parallel Haskell implementation. Maintaining a working system essentially means to keep routines for packing and unpacking heap structures alive, bug-free, and consistent with internals of the sequential base implementation.

To serialise the subgraph reachable from the node to be transferred, a breadth-first traversal of the subgraph is performed and the data is sent to the receiver for reconstruction. Thus, packing and unpacking is an instance of a more general graph traversal routine. Classical graph traversal (breadth-first and depth-first) is easily programmed in Haskell, presuming that we can retrieve all nodes referenced from one node. We have developed a Haskell prototype for graph traversal, and identified the obstacles and issues of a high-level, language-internal approach to heap access. The main problems here are: sufficient knowledge of GHC internals to specially encode handling for closures with non-standard layout (and basically presuming a standard layout at all), as well as typing.

**Typing** is an issue again: When traversing a graph structure, we are not interested in the Haskell type it might represent. However, our code needs to typecheck somehow. We simply cast all references to a (wrong) unit type, and cast them back when needed.

```

type Untyped = ()

untyped :: a -> Untyped
untyped x = unsafeCoerce# x

retype :: Untyped -> a
retype x = unsafeCoerce# x

```

**Knowledge of GHC internals** is required to retrieve all child nodes referenced from a node to be analysed. The recent ghci debugger [IM07, Ibo06, MIPG07] uses two primitive operations<sup>2</sup> to access the heap representation of

---

<sup>2</sup>Since May 2007, these primitives have been replaced by a single one with a different name `unpackClosure`. The name is no problem, but looking at the code, the primitive apparently omits pointers for thunks and other interesting node types, which is exactly the information we need. It is essential for us that this information should be accurate for all closure types, especially that it reflects the most subtle implementation details.

The code presented here uses the primitives as of January 2007, changed by ourselves to return correct reference pointers for thunks.

data, on which we could build our heap graph traversal code.

```
infoPtr#      :: a -> Addr#          -- address of info table
closurePayload# :: a -> (# Array# , ByteArray# #) -- (pointers,nonpointers)
```

The code of the ghci debugger (inside ghc itself) uses these primitives in Haskell. However, it is an integrated part of GHC and accesses many other internal GHC structures, which we want to avoid. In order to abstract from the particular GHC representation, we used a small wrapper module which provides an IO-monadic interface and returns an opaque Haskell representation (inspired by the ghci-debugger work) of closures. Implementation of the module is, of course, heavily GHC-dependent, but the closure representation `GHCClosure` can remain opaque if the right helper functions are provided: access to referenced heap cells and data inside a heap node, and information about the evaluation state of a heap cell.

```
data GHCClosureType = Constr | Fun | Thunk Int | ThunkSelector
                    | Blackhole | AP | PAP | Indirection Int | Other Int
  deriving (Show, Eq)
data GHCClosure = Closure { tipe          :: GHCClosureType
                          , infoTable    :: Ptr ()
                          , ptrs         :: Array Int Untyped
                          , nonPtrs      :: ByteArray#
                          }
getClosureData :: a -> GHCClosure -- opaque

withChildren :: GHCClosure -> (Untyped -> IO r) -> IO [r]
withData    :: GHCClosure -> (Byte -> IO r) -> [r]

isUnevaluated :: GHCClosureType -> Bool
isIndirection :: GHCClosure -> Bool          -- detect indirections
unwind       :: Untyped -> IO Untyped -- skip possible indirections
```

Any implementation detail should better remain hidden inside a single module which reflects the GHC implementation and requires maintenance by a GHC-expert.

**Graph traversal functions.** Provided the implementation-dependent parts mentioned above work properly, we can traverse a reachable subgraph in depth-first manner using recursion and the implicit call stack, or in breadth-first manner, which additionally needs a queue. Code is shown in Fig. 4.13. The monadic traversal is specified as a higher-order function applying one of two parameter functions to each visited node: Function `firstMet` is applied to nodes not visited before, `metAgain` is applied upon further visits. To keep track of visited nodes, both traversal functions use a hash table which holds the previously obtained result of applying `firstMet` to the node. The code uses a hash table specialised to closures, which are represented as `Untyped` values comparable by raw memory address. In addition, a queue for nodes to be visited is required for breadth-first traversal.

```

type HTClosures c = HashTable Untyped c
newHTC :: IO (HTClosures c)
htLookup :: HTClosures c -> Untyped -> IO (Maybe c)
htInsert :: HTClosures c -> Untyped -> c -> IO ()

traverseB :: (Untyped -> IO (c,d) ) -> -- for first time closure is met
           (d -> Untyped -> IO c) -> -- for closure seen before (result d)
           a -> IO [c]
traverseB firstMet metAgain rootnode = do ht <- newHTC
                                           q  <- newQueue
                                           do x <- unwind (untype rootnode)
                                               enqueue q x
                                               traverseRecB ht q

where traverseRecB ht q -- :: HTClosures d -> Queue Untyped -> IO [c]
      = do e <- empty q
          if e then return []
            else do
              cl <- dequeue q
              haveIt <- htLookup ht cl
              r <- case haveIt of
                  Just stored -> metAgain stored cl
                  Nothing -> do cld <- getClosureData cl
                                (res,storeHt) <- firstMet cl
                                htInsert ht cl storeHt
                                withChildren cld
                                (\cl -> unwind cl >>= enqueue q)
                                return res
              rs <- traverseRecB ht q
              return (r:rs)
-----
traverseD :: (Untyped -> IO (c,d) ) -> -- action for first time closure is met
           (d -> Untyped -> IO c) -> -- action for closure seen before (result d)
           a -> IO [c]
traverseD firstMet metAgain rootnode = do ht <- newHTC
                                           traverseRecD ht (untype rootnode)

where traverseRecD hasht closure -- :: HTClosures d -> Untyped -> IO [c]
      = do x <- unwind closure
          maybeRes <- htLookup hasht x
          case maybeRes of
              Just stored -> do r <- metAgain stored x
                              return [r]
              Nothing -> do cld <- getClosureData x
                            (ret,store) <- firstMet x
                            htInsert hasht x store
                            rss <- withChildren cld
                                (traverseRecD hasht)
                            return (ret:concat rss)

```

Figure 4.13: Heap graph traversal (breadth-first/depth-first)

```
printGraph :: a -> IO ()
printGraph rootnode = do indentVar <- newMVar ["-->"]
                        traversed (printIndent indentVar)
                                (printRepeatIndent indentVar)
                                rootnode
                        return ()

printIndent :: MVar [String] -> Untyped -> IO ((),Int)
printIndent indentVar closure
  = do (myIndent:rest) <- takeMVar indentVar
       id <- mkFreshId
       clD <- getClosureData closure
       -- push indent strings for children on the stack
       ...
       -- output the current node
       ...
       return ((), id)

printRepeatIndent :: MVar [String] -> Int -> Untyped -> IO ()
printRepeatIndent indentVar id closure
  = do (myIndent:rest) <- takeMVar indentVar
       putMVar indentVar rest
       putStrLn (myIndent ++ show id ++ " (met again)")
```

---

Figure 4.14: Worker functions to pretty-print a heap graph structure

**Usage.** Suitable worker functions for the traversal have to be supplied by the caller. For instance, IDs could be assigned to every closure met, and counted how often the same closure has been found.

```
data ClosureMet = New Int Untyped | Again Int
instance Show ClosureMet where
  show (New id cl) = show id ++ showAddr cl
  show (Again id)  = show id ++ "(again)"
firstMet :: Untyped -> IO (ClosureMet,Int)
firstMet cl = do iD <- mkFreshId
               return (New iD cl, iD)
metAgain :: Int -> Untyped -> IO ClosureMet
metAgain iD cl = return (Again iD)
```

**Stateful Graph Traversal.** To use the graph traversal for packing heap structures, the worker functions have to support an internal state. For packing, this storage will be continuously filled during packing and sent away afterwards. A more simple, but analogous problem is to pretty-print the subgraph below a rootnode with an appropriate indentation. This can be encoded easily using depth-first traversal and suitable worker functions, as shown in Fig. 4.14. We should underline that all code fragments shown for heap graph traversal are compilable and have been tested; here is the code and output of a small test program

which prints a heap graph before and after evaluation:

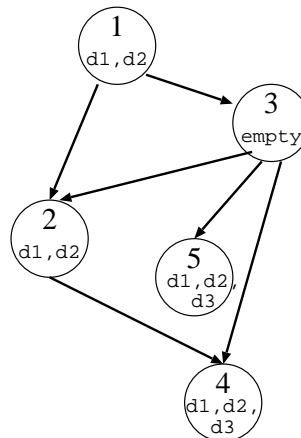
<pre>Haskell Program:  import HeapAccess  d2 = [1,2,3] d3 = 0:d2 d4 = tail d3  main = do hSetBuffering stdout NoBuffering   let testlist = [d2,d3,d4]   printGraph testlist -- (1)   print testlist     -- (*)   printGraph testlist -- (2)  printGraph :: a -&gt; IO () printGraph rootnode = ... -- see above</pre>	<pre>Before *, at (1):  --&gt;1: Thunk 17 +-2: Constr   -3: Constr     -4: Constr     +-5: Constr       -6: Constr       +-7: Constr         -8: Constr         +-9: Constr +-10: Constr   -11: Constr     -12: Constr     +-3 (met again) +-13: Constr   -14: Thunk 22 +-9 (met again)</pre>	<pre>After *, at (2):  --&gt;1: Thunk 17 +-2: Constr   -3: Constr     -4: Constr     +-5: Constr       -6: Constr       +-7: Constr         -8: Constr         +-9: Constr +-10: Constr   -11: Constr     -12: Constr     +-3 (met again) +-13: Constr   -3 (met again) +-9 (met again)</pre>
---	---	---

**Packing and unpacking.** The standard layout of heap closures in GHC starts with a header section, followed by all pointers to other graph nodes, and then by all non-pointers. Assuming this, we have prototyped a packing implementation which is largely equivalent to the packing routine in the Eden RTE.

Newly met closures are packed as their info pointer address (the same on all machines when using a homogeneous network), zeros for the pointer fields, and data directly copied into some storage. The size of the closure in the packet is returned. Internally, the start index of the packed data is stored in the hash table we use in the traversal function. When a closure is met again, the second worker function applies, which will not pack it again, but only pack a back reference (stored earlier when meeting the closure for the first time). The resulting packet layout is as follows (example):

info	0,0	$d_1, d_2$	info	0	$d_1, d_2$	info	0,0,0	info	$d_1, d_2, d_3$	REF	6	info	$d_1, d_2, d_3$	REF	14					
1:graphroot			6:closure 2			10:closure 3			14:cl. 4			ref. to 2			20:closure 5			ref. to 4		

The example packet contains a complete subgraph of 5 nodes, the graph structure depicted on the right. All packed zeros (pointer fields) will be filled with references to subsequent closures upon unpacking. Back references to closures already packed are stored in the packet with a special info pointer `REF`, followed by the index of the closure in the packet. The code for packing (not shown) is mostly straightforward; it has to copy all relevant data into the storage (a contiguous memory region), which is done by the worker function, and to follow the references, done by the traversal HOF. We have used a mutable array of unboxed values, manipulated in the IO monad, as storage.



Unpacking a subgraph which has been packed in this format merely consists of doing a pass over the contiguous memory region and filling in all the missing pointers. The graph structure is reconstructed in-situ, no data needs to be copied. This is the main difference against the packing algorithm implemented in the RTE of Eden (and GpH), which does not leave space for pointers, but reconstructs the graph structure in newly allocated heap space. The code for unpacking (not shown) does one pass over the whole packet, identifies (closures with) pointer fields, and enqueues their indices in a queue for filling in missing pointers. Recursive functions iterate over the queue and over pointer fields in a dequeued closure. While the packet is analysed, the closures and back references inside are written into the currently filled closure (which has been dequeued, and will be filled to completion before dequeuing the next one).

## Problems

Even though the heap access functionality seems to be realisable, apparent problems arise when applying our concepts in a broader context than tiny test programs. Because of the unsafe type casts involved, developing the methods is cumbersome, and especially packing fails in many test programs (due to wrong assumptions about GHC internals). And it has to be admitted that our functions, despite all efforts, contain various implementation dependencies. While the graph traversal might work correctly, given a correct implementation of the GHC-dependent parts, our packing/unpacking algorithm essentially relies on the GHC standard closure representation.

The truly severe problem we see in this approach is the interaction of heap graph traversal and garbage collection. Graph traversal has to be non-preemptive (no other thread should run in-between) and atomic (no garbage collection should happen during execution). The entire traversal has to happen atomically. The reason is that the traversal accesses the heap by raw pointers, and even stores results with memory addresses as an index in a hash table. So we are accessing the raw heap (GC not transparent), but simultaneously creating new heap cells by our Haskell computation (GC transparent). If garbage collection occurs in the middle of such a heap operation, it will invalidate all “raw data” we obtained from the RTE.

On the other hand, garbage collection cannot be inhibited during traversal. An example can be constructed where the intermediate structures created during traversal require more heap space than available, requiring garbage collection to run. If packing is restarted from the beginning in such a case, packing will end up in the same state again. The only relief would be to treat the pointers in the current closure queue as additional garbage collection roots; but then, these raw memory addresses would have to be replaced by new ones.

---

```

getSparks :: Int -> Bool -> StateIO GumState [GumJob]
getSparks amount forExport = do (GSt ts sparks pes) <- get
    let (sparks,rest) = splitAt amount sparks
        put (GSt ts rest pes)
        if forExport then mapM prepareExport sparks
            else return sparks

prepareExport :: GumJob -> GumJob
prepareExport ptr = do stPtr <- castPtrToStablePtr ptr
    spark <- deRefStablePtr stPtr
    (chan,result) <- createC
    let sendBack = mkGumJob
        (do connectToPort chan
            rnf spark 'seq' -- nf evaluation
            sendData Data spark) -- send back NFData
        replaceBy# spark result
    return sendBack

```

---

Figure 4.15: Spark export preparation for distributed memory

#### 4.7.4 Summary

To conclude, we can state that our Haskell platform GHC provides the tools needed for global address management, and the prototype code for a VSM implementation in Haskell looked promising. However, heap graph traversal from inside Haskell turned out to introduce considerable implementation dependencies and a conceptual obstacle (garbage collection). We have demonstrated that heap graph traversal is a *requirement* for implementing virtual shared memory in Haskell. Altogether, our current opinion is that the virtual shared memory support for GpH *cannot* be encoded in Haskell, at least not with the current assumptions we have made about kernel support. More sophisticated kernel support is needed, decoupling the kernel implementation details from the system level where we wanted to manage global addresses. Our impression is that this issue amounts to implementing global address support in the kernel.

As an alternative, we might consider applying the two-stage task creation of GpH to a *distributed* memory setting. Sparks can easily be managed inside the Haskell scheduling loop, as we have shown previously. In our scheduling and communication framework, it is easy to define a mechanism where idle PEs fetch one or more sparks and return the evaluated data *eagerly*. The code in Fig 4.15, straightforward and without any technical issues, sketches a variant of the helper function `getSparks` for work request handling in `processFish` (see Fig. 4.11). When a spark is exported using this function, its evaluation automatically includes communication of the result back to the source (whereas, in GpH, data resides at the evaluation site until fetched back). It might look potentially dangerous that the spark data is made inaccessible and replaced by a placeholder. Function `prepareExport` may only be called when that export does happen. How-

ever, exactly the same happens in a kernel implementation. A real drawback and potential source of bad performance is that data needed for evaluation of the spark will always be *copied*, potentially duplicating its evaluation. In contrast to the virtual shared memory of GpH, sharing data between two exported sparks is impossible with our version.

We have described how parallel coordination constructs can be realised outside a monolithic kernel implementation, thereby providing much easier access for language designers. While modeling VSM from inside Haskell is attractive, it cannot be safely realised with the kernel support assumed. In contrast, scheduling concepts (crucial in today's heterogeneous architectures) have proven easy to express in our proof-of-concept implementation. Summarising, we can state that the ARTCOP idea is promising and that especially its flexibility makes it attractive for new parallel architectures and adaptive programming models.



# Chapter 5

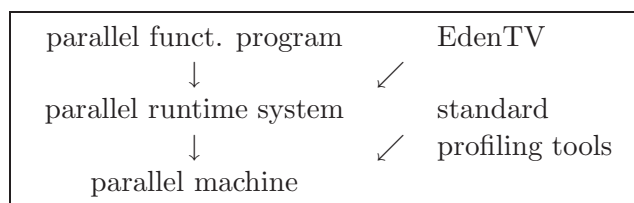
## Visualising Eden program runs: EdenTV

### 5.1 Motivation

Due to the high complexity and nondeterministic interaction in parallel systems, developing, debugging and optimising parallel programs is a very complex process. The parallel functional languages we investigate in our work “eliminate many of the most unpleasant burdens of parallel programming[...]” [HM99], as they offer a conceptual algorithmic view of parallelism and remain abstract about operational properties. This abstraction makes parallel programming much less error-prone. On the other hand, this often hampers program optimisations. The programmer has few possibilities to analyse and optimise the behaviour of a program by only knowing the runtime or speedup achieved. In order to improve the performance of parallel functional programs productively, more information about the execution is needed. For programmers who are not familiar with the language implementation, it is very difficult to know or guess what really happens *inside* the parallel machine during program execution.

Many profiling toolkits for monitoring parallel program executions exist [Fos95, KG96, NAW<sup>+</sup>96], but most of them are less appropriate for high-level languages, as these are generally implemented with the help of a complex parallel runtime system implementing a parallel virtual machine, as we have described earlier for Eden specifically.

Standard profiling tools like `xpvm` [KG96] can only monitor the execution of this runtime system, the activity of the virtual processing elements (PEs or *machines*, usually mapped one-to-one to the



physical ones), the message traffic between these PEs, and the low-level behaviour of the underlying middleware like PVM [PVM] MPI [MPI97]. But the large gap between abstract language concepts and concrete runtime behaviour needs customised *high-level* tools for runtime analysis. The basic runtime and computation units of the *language*, under programmer control and on top of the RTE, are hard to identify from this low-level view.

The Eden trace viewer tool (EdenTV) presented in this chapter visualises the execution of parallel functional Eden programs at a higher level of abstraction. EdenTV shows the activity of the *Eden* threads and processes, their mapping to the machines, stream communication between Eden processes, garbage collection phases, and the process generation tree, i.e. information specific to the Eden programming model. This supports the programmer’s understanding of the parallel behaviour of an Eden program.

EdenTV enables a *post-mortem analysis* of program executions at the level of the Eden RTE. The steps of profiling are *trace generation* and *trace representation*, separated as much as possible in EdenTV, so that single parts can easily be maintained and modified for other purposes.

EdenTV has been developed as a supplement in the context of our work on the Eden implementation. Two versions of the trace representation tool are currently available. A Java implementation has been developed by Pablo Roldán Gómez [RG04]. Björn Struckmeier [Str06] did a re-implementation in Haskell which provides additional features. Both tools had to undergo major revisions after the new implementation of Eden in GHC 6.

In this chapter, we will describe implementation concepts of EdenTV, and present examples of how the high-level EdenTV profiling tool can be profitably used to optimise and spot errors in high-level parallel programs (our related publication [BL07b] contains similar case studies). Typical reasons for bad performance in our lazy functional setting are delayed evaluation or poor load balancing between processes, easily identified by the trace visualisations. In addition, EdenTV works for EDI programs (it is based on the basic runtime computation units) and can thereby help detect bugs and weaknesses of the language implementation itself. EdenTV timeline diagrams will be used throughout the rest of this thesis to support analysis and discussion of our skeleton implementations.

## 5.2 How EdenTV works

To enable the trace generation for EdenTV, the Eden RTE is instrumented with special trace generation commands, activated by a runtime system option. Parts of the well-established Pablo Toolkit [Dan91] and its *Self-Defining Data Format* (SDDF) are used for trace generation. After program termination, the trace files (one per machine) are merged and can be loaded into EdenTV to produce timeline diagrams for threads, processes, and machines.

**Trace generation.** To profile the execution of an Eden program, the RTS continuously writes *events* into a trace file. These trace events, shown in Figure 5.2, indicate the creation or a state transition of a computational unit. Trace events are emitted from the RTE, using the Pablo *Trace Capture Library* [Dan91] and its portable and machine-readable “Self-Defining Data Format”. Eden programs need not be changed to obtain the traces.

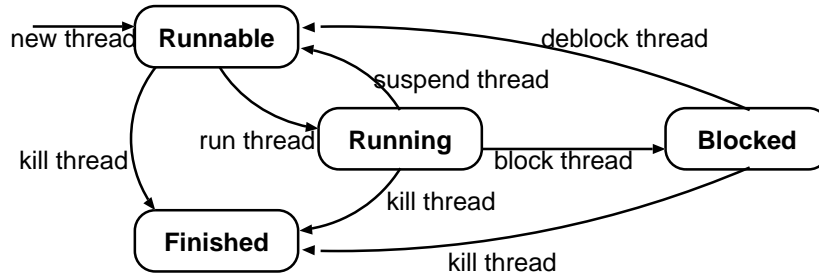


Figure 5.1: Thread state transitions

Start Machine	End Machine
New Process	Kill Process
New Thread	Kill Thread
Run Thread	Suspend Thread
Block Thread	Deblock Thread
Send Message	Receive Message
<i>Start Communication</i>	<i>End Communication</i>
<i>Label Process</i>	<i>GC done</i>

Figure 5.2: EdenTV trace events

The events which are traced during a program run collect information about the behaviour of machines, processes, threads and messages. Concurrent threads are the basic unit for the implementation, so the central task for profiling is to keep track of their execution. Threads are scheduled *round-robin* and run through the straightforward state transitions shown in Figure 5.1. An Eden process, as a purely conceptual unit, consists of a number of concurrent threads which share a common graph heap (as opposed to processes, which communicate via channels). The Eden RTE does not support the migration of threads or processes to other machines during execution, so every thread is located on exactly one machine during its lifetime. The machines form a third category, and a self-suggesting one, since one file per machine is generated by the trace library.

Additional events (written in italics in Fig. 5.2) record underlying RTE actions, garbage collection and communication phases, and allow to label new processes with a name (corresponding trace viewer features are only partially implemented).

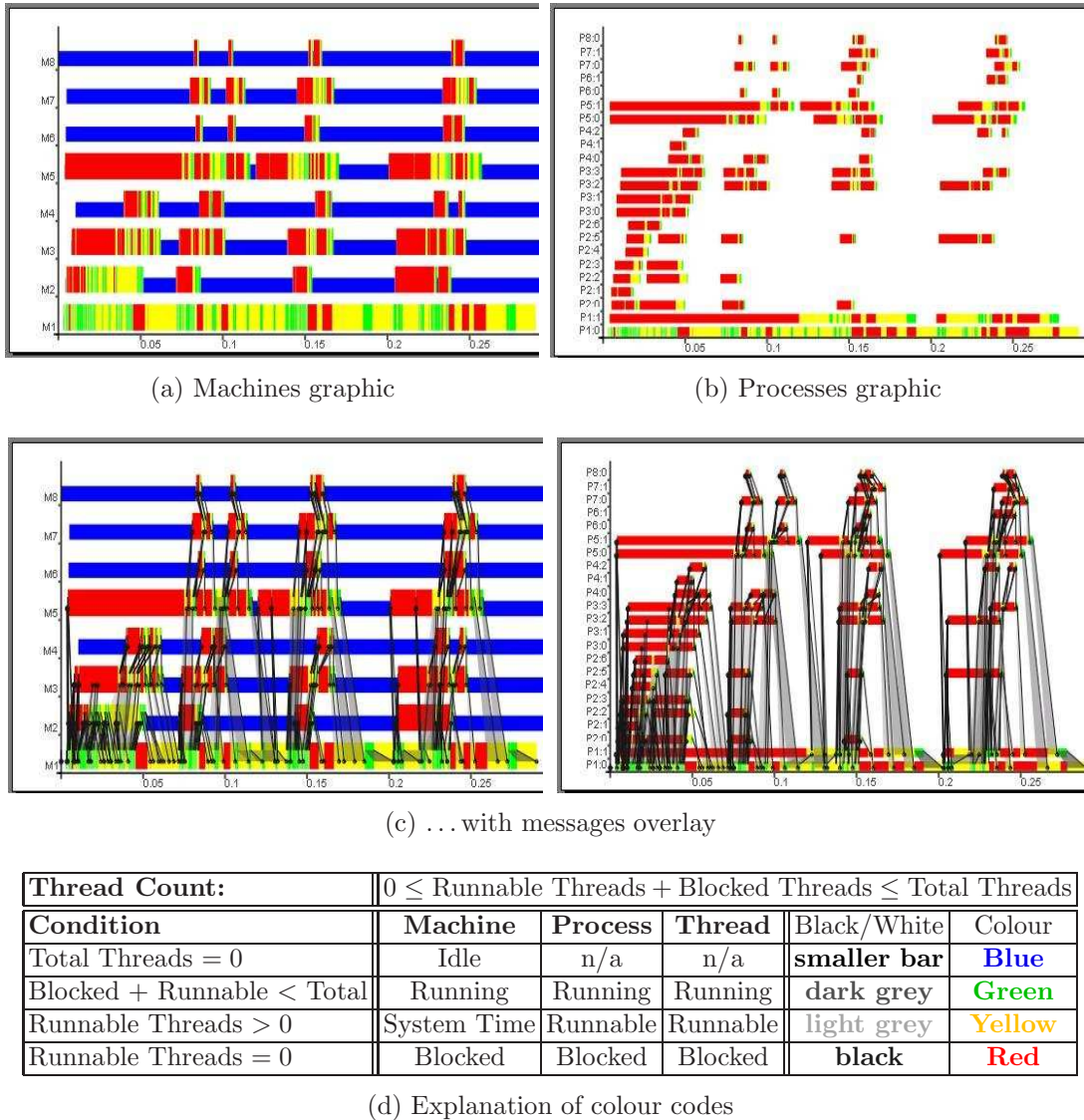


Figure 5.3: Examples of EdenTV diagrams and colour codes table

**Trace representation.** In the timeline diagrams generated by EdenTV, machines, processes, and threads are represented by horizontal bars, with time on the x axis. EdenTV offers separate diagrams for *machines*, *processes*, and *threads*. The machines diagrams correspond to the view of profiling tools observing the parallel machine execution. Figure 5.3 shows examples of the machines and processes diagrams for a parallel divide-and-conquer program with limited recursion-depth. The trace has been generated on 8 Linux workstations connected via fast Ethernet. The diagram lines have segments in different colours, which indicate the activities of the respective logical unit in a period during the execution.

Thread states can be directly concluded from the emitted events. Machine and process states are assigned following a fundamental equation for the thread count inside one process or machine:

$$0 \leq \text{Runnable Threads} + \text{Blocked Threads} \leq \text{Total Threads}$$

Colour codes for processes and machines are assigned as a result of comparing these numbers. In general, the first condition checked by EdenTV is complete equality, i.e. whether the machine or process is in **idle** state (Total Processes resp. Total Threads= 0). Otherwise, a difference between the thread counts (inequality on the right) implies that a thread is **running**. If the thread counts on both sides are equal, the state is either **runnable** or **blocked**, depending on the number of runnable threads (the state is **runnable** if Runnable Threads > 0). This context information for all units is the basis of the graphical representations.

The example diagrams in Figure 5.3 show that the program has been executed on 8 machines (virtual PEs). While there is some continuous activity on machine 1 (the bottom line), where the main program is started, machines 6 to 8 (the upper three lines) are idle most of the time. The corresponding processes graphic (see Figure 5.3(a)) reveals that several Eden processes have been allocated on each machine. The diagrams show that the workload on the parallel machines was low — there were only small periods when threads were running. Messages between processes or machines can optionally be shown by arrows which start from the sending unit line and point at the receiving unit line (see Figure 5.3(c)). The diagrams can be *zoomed* in order to get a closer view on the activities at critical points during execution.

**Additional features.** Several extensions to EdenTV have been made to provide additional information about the program run, beyond the basic machine, process, thread and communication information. All EdenTV versions provide a summary of the messages sent and received by processes and machines (on the right for the trace in

Machine	Runtime (sec)	Processes	Messages	
			sent	received
1	0.287197	4	6132	6166
2	0.361365	18	1224	1206
		⋮		
8	0.362850	6	408	402
Total	0.371875	66	14784	14784

Figure 5.3). In the Haskell version, stream communication is indicated by shading the area between the first and the last message of a stream (see Figure 5.3(c)), garbage collection phases and memory consumption can be shown in the activity diagrams, and the process generation tree can be drawn and labelled with process names. In turn, the Java version offers to indicate the communication phases (when the RTE is receiving messages) by an additional color in the bar. EdenTV is still under active development to add useful information, improve the efficiency of trace analysis, and continuously adapt to new RTE versions (maintaining backward compatibility whenever possible).

## 5.3 Related work

A rather simple (but insufficient) way to obtain information about a program’s parallelism would be to trace the behaviour of the communication subsystem. Tracing PVM-specific and user-defined actions is possible and visualisation can be carried out by `xpvm` [KG96]. Yet, PVM-tracing yields only information about virtual PEs and concrete messages between them. Internal buffering, processes, and threads in the RTE remain invisible, unless user-defined events are used.

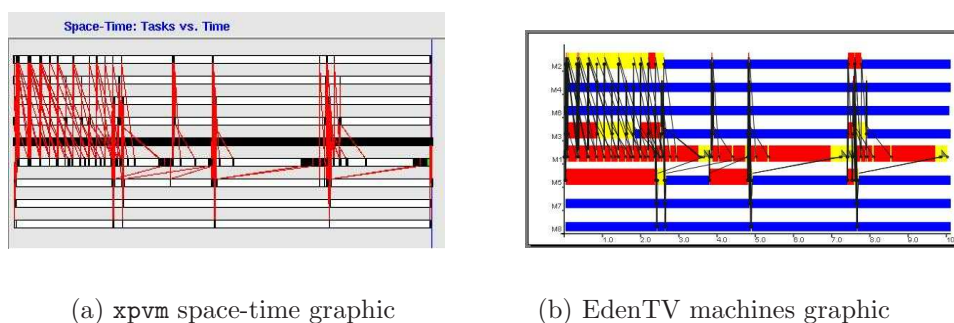


Figure 5.4: `xpvm` vs EdentTV

As a comparison, we show `xpvm` execution traces of our first trace examples. The space-time graphic of `xpvm` shown in Figure 5.4(a) corresponds to the EdentTV machine view of the same run in Figure 5.4(b), if the machines are ordered according to the `xpvm` order. The monitoring by `xpvm` significantly slows down the whole execution (runtime increases from 0.37 seconds to 10.11 seconds). A reason for this unacceptable tracing overhead might be that PVM uses its own communication system for gathering trace information.

Comparable to the tracing included in `xpvm`, many efforts have been made in the past to create standard tools for trace analysis and representation (e.g. Pablo Analysis GUI [Dan91], the ParaGraph Suite [HE91], or the Vampir system [NAW<sup>+</sup>96], to mention only a few essential projects). These tools have interesting features EdentTV does not yet include, like stream-based online trace analysis, execution replay, and a wide range of standard diagrams. The aim of EdentTV, however, is a specific visualisation of logical Eden units, which needed a more customised solution. The EdentTV diagrams have been inspired by the per-processor view of the Granularity Simulator GranSim [Loi96], a profiler for Glasgow parallel Haskell (GpH) [THLP98], which, however, does not trace any kind of communication due to the different language concept. The Eden derivative of GranSim, Paradise [HPR00], equally was a pure *simulator*, but offered the interesting feature to label instantiated processes in the visualisation, thereby linking trace information to the program source code. Last but not least, the direct predecessor of EdentTV was the work by Ralf Freitag [Fre99], which was entirely based on the Pablo Analysis GUI (not available any more).

## 5.4 Simple examples

### 5.4.1 Irregularity and cost of load balancing

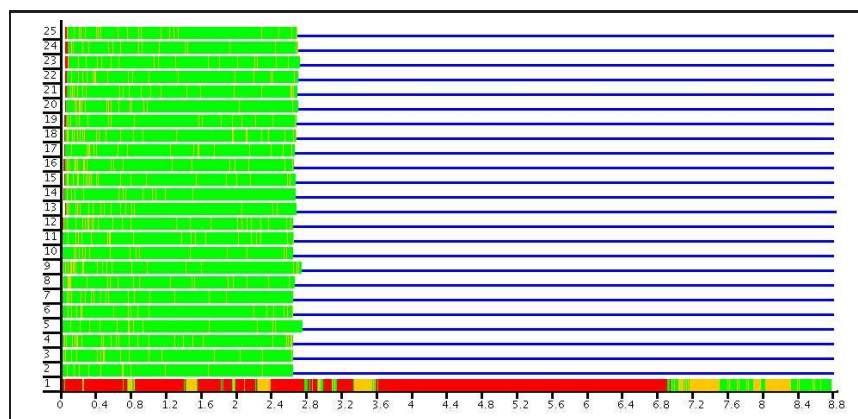
The well-known higher-order function `map`, which applies a function to all elements of a list, can be easily parallelised in different ways. Every list element represents an *independent* task (applying the function), and these may be arbitrarily distributed to a number of sub-processes, given that the initial order is kept or re-established in the result list.

We consider two different versions which can be distinguished by the method of distributing the list elements to different worker processes. Both methods distribute the tasks in a statically determined fashion: For  $n$  processes, we can either cut the input list into  $n$  pieces of regular length, or distribute the single elements round-robin until all tasks are distributed. Problems may arise when tasks have irregular complexity, the subprocesses may expose uneven load distribution, and the machine with the most complex tasks in its (statically determined) task set will dominate the runtime of the parallel computation.

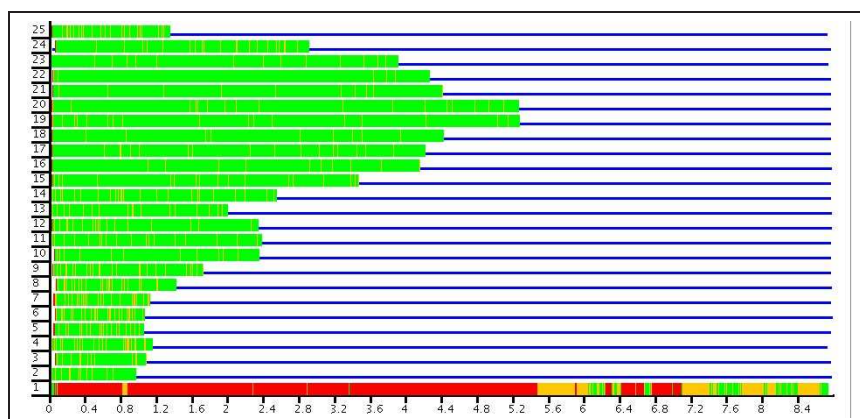
*Example: (Mandelbrot visualisation)* We compare different work distribution schemes using a program which generates Mandelbrot Set visualisations by applying a uniform, but potentially irregular computation to a set of coordinates.

We tested different parallelisations of the underlying computation scheme, `map`, as described above. Parallelisation consists of the parallel computation of pixel rows, and the two versions differ by their task distribution: either round-robin for single elements, or blockwise for the whole list. The Mandelbrot programs ran on 25 machines of a Beowulf-cluster, with a problem size of  $2000 \times 2000$  pixels.

As we can see in the diagrams shown in Fig. 5.5, the round-robin distribution of rows (Fig. 5.5(a)) leads to a well-balanced load in the worker processes, while the blockwise distribution (Fig. 5.5(b)) produces an uneven complexity of tasks, and thus uneven workload of the worker processes, even showing the silhouette of the Mandelbrot graphic (please note that the creation order of the processes is bottom-to-top). Despite the static task distribution, the root process (bottom bar) presents a serious brake, as it merges all results at the end of the computation, leading to a long sequential end phase. Both traces show that the workers can work without being blocked, but they are far below the theoretical optimum of using all processors in parallel all of the time. While from the total runtimes, the difference between the two runs is hardly distinguishable, an analysis using the EdenTV has revealed the load imbalance of the worker processes, and has shown that the bottleneck of result collection in the calling main process dominates runtime and needs optimisation. ◀



(a) Farm skeleton, round-robin task distribution (runtime: 8.78 sec.)



(b) Farm skeleton, static blockwise distribution (runtime: 8.74 sec.)

Figure 5.5: *All Machines* diagrams for different parallel Mandelbrot programs

Eden-5 Beowulf Cluster Heriot-Watt University, Edinburgh, 25 machines  
(Intel P4-SMP@3GHz, 512MB RAM, Fast Ethernet)

### 5.4.2 Lazy evaluation vs. parallelism

When using a lazy computation language, a crucial issue — for Eden as well as other Haskell-based parallelism — is to start the evaluation of needed subexpressions early enough and to fully evaluate them for later use. The basic choice to evaluate a final result either to weak head normal form (wHNF) or to normal form (NF) sometimes does not offer enough control to optimise an algorithm. Strategies [THLP98] forcing additional evaluations must then be applied to certain subresults. On the sparse basis of runtime measurements, such an optimisation would be rather cumbersome. EdenTV, accompanied by code inspection, makes such shortcomings obvious, as in the following example.

*Example: (Warshall’s algorithm)* We compute shortest paths between all nodes of a graph from its adjacency matrix. The program measured here is a



parallel implementation of Warshall’s algorithm with a ring of processes (adapted from [PvE93]). Initialised with a row of the adjacency matrix, each process computes the minimum distances from the corresponding node to every other node by updating its row continuously using the other rows received from, and forwarded to, the ring. With the Eden ring skeleton (discussed in detail later in 8.2, also see [LOMP05]), the task of implementing this algorithm reduces to defining the function `ring_iterate` that each ring process should apply, shown in Fig. 5.6 for the Warshall program.

Each ring process performs `size` (= number of graph nodes) iterations where `k` is the number of its own node, `i` is the iteration counter and `rowk` is the own row. It maps the input from the ring predecessor (`rowi:xs`) to a pair (final result, output to ring successor). The result is the final value of its own row. The output to the successor in the ring consists of the incoming rows to which its own row `rowk` is added in the `k`th iteration. In all but the `k`th iteration, each ring process updates its own row by `updaterow` using the incoming row `rowi`.

The trace visualisations in Fig. 5.7 show the *Processes/Machine* view of EdenTV for two versions of the program on a Beowulf cluster, the input graph consisting of 500 nodes (aggregated on 25 processors). The programs differ by the line in bold face in the function `ring_iterate`, which introduces additional demand for continuously updating the own row in the second version.

---

```

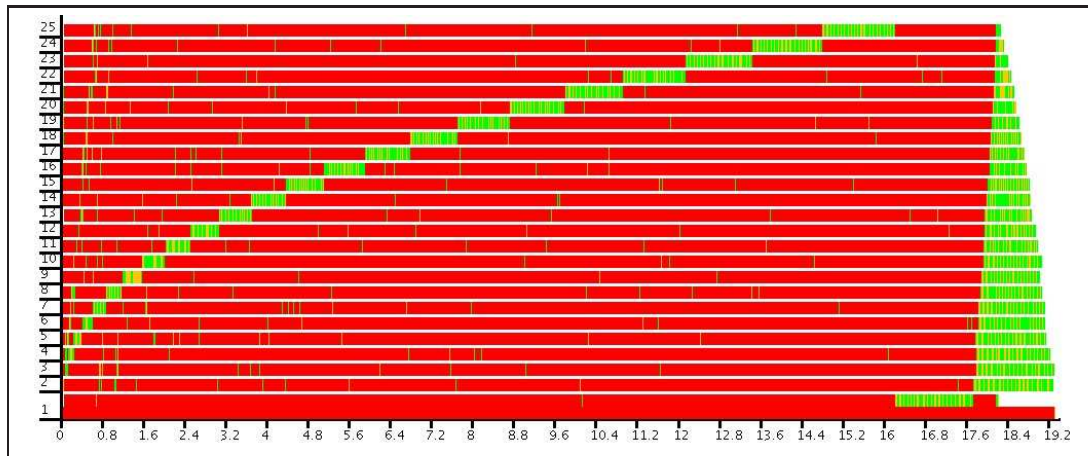
ring_iterate :: Int -> Int -> Int -> [Int] -> [[Int]] -> ([Int],[[Int]])
ring_iterate size k i rowk (rowi:xs)
  | i > size = (rowk, []) -- iterations_finished
  | i == k   = (solution, rowk:restoutput) -- send own updated row
  | otherwise = (solution, rowi:restoutput) -- compute update only
where (solution, restoutput) =
      rnf nextrowk 'seq' -- additional demand control
      ring_iterate size k (i+1) nextrowk xs
nextrowk | i == k   = rowk -- no update, if own row
         | otherwise = updaterow rowk rowi (rowk!!(i-1))

```

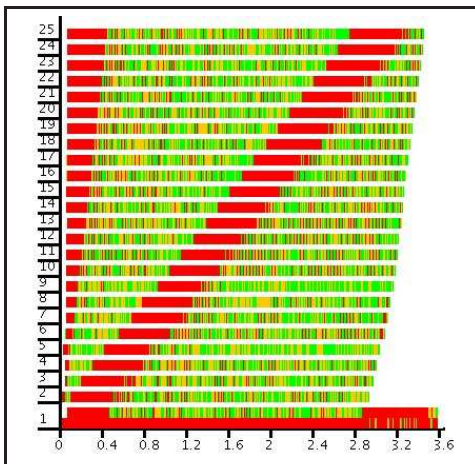
---

Figure 5.6: Ring process functionality for parallel Warshall program

The first program version (without demand control) shows poor performance, due to an inherent data dependence in the algorithm. The trace visualisation clearly shows that the first phase of the algorithm is virtually sequential. A small period of increasing activity (growing segments) between long blocked periods traverses the ring. Note that the first machine executes two processes: the main program and the last ring process. Only the second phase of the algorithm runs in parallel on all machines. The length of this second phase depends on the position in the ring: The first ring process (which started working first) has to do the most work in the end, thereby dominating runtime. The reason for this poor



Runtime: 19.37 sec.



Runtime: 3.62 sec.

above: no additional demand control

left: with additional demand control

Figure 5.7: Warshall's algorithm (500-node graph)

Eden-5 Beowulf Cluster Heriot-Watt University, Edinburgh, 25 machines  
(Intel P4-SMP@3GHz, 512MB RAM, Fast Ethernet)

performance is demand-driven evaluation: Data received from a ring neighbour is passed to the next node *unchanged*, until the node sends its own rows updated with all ring data received before. Only when the nodes' updated own rows are passed to the ring, their evaluation begins. Before this inherent demand takes place, the processes only accumulate data from the ring communication, but do not proceed to updating their rows with the distances received from the ring neighbour. Inserting a normal form evaluation strategy (`rnf`) for the updated row `nextrowk` into the `ring_iterate` function dramatically improves runtime. We still see the impact of data dependence, leading to a short wait phase passing through the ring, but the optimised version shows good speedup and load balance. A crucial issue for such optimisations is to identify the demand problem from the trace visualisation (without a link to the program's source code). ◀

### 5.4.3 Process placement (implementation) bug

As we have described earlier (see 3.1.3), process instantiation in Eden is implemented as a message between PEs with special semantics, in mode `Instantiate`. The primitive operation, as well as the EDI wrapper `spawnProcessAt`, thus allows explicit process placement on particular PEs. The Eden language, by original definition, does not expose placement, but processes are placed either following a local round-robin scheme, or randomly on all available machines. While the support for explicit process placement is crucial for programming efficient skeleton implementations, small prototype programs for EDI and Eden skeletons often rely on the round-robin scheme.

*Example: (Pipeline process placement)* For instance, a pipeline of  $n$  processes can be set up in EDI as  $n$  remote computations spawned by a single caller, and connected via channels.<sup>1</sup> In a small test program, which uses less pipeline stages than available PEs, we can expect the pipeline to use one PE more than the number of stages (say  $k$ ), precisely PEs 1 to  $k + 1$ , since the caller on PE 1 is supposed to place pipeline stages round-robin: on PEs 2, 3 . . . ( $k + 1$ ). However, processes got unexpectedly placed on every *second* PE, as shown by the EdenTV machine view in Fig. 5.8.

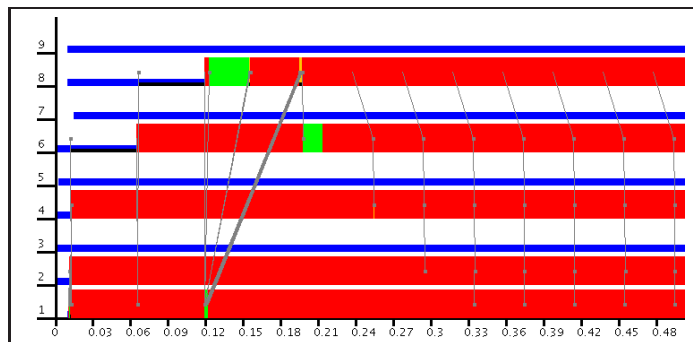


Figure 5.8: Pipeline test program revealing RTE bug (no round-robin placement)

The reason is a subtle bug in the round-robin distribution implementation in the RTE, relevant in a special scenario. A channel for input of a pipeline stage was sent back by each pipeline process right after instantiation, and supplied to the predecessor stage as an embedded part of the remote action to be executed. This means, when instantiating (i.e. packing and sending away) the pipeline processes, input channels of the respective successor stages were not available in the heap, and process instantiation (done in one thread for all stages) was blocked on the respective placeholders every time. When the channel then became available, the

<sup>1</sup>We will discuss variants of pipeline skeletons in more detail later in 8.1.2. The bug explained here was indeed discovered when experimenting with pipeline skeletons.

RTE (erroneously) selected the *next* PE in round-robin manner, not the same as before when the instantiation got blocked. Without EdenTV, errors of this kind, and their cause, are hard to find, the only reliable information being debug messages for developers. ◁

As we have shown, the Eden Trace Viewer provides information at the same level of abstraction as the Eden (and EDI) programming language. It is therefore directly related to the Eden implementation, and provides a useful tool for detecting typical traps of parallelism at the high level: load balancing issues, missing demand in computation or communication, analysis and control of the communication structures, and implementation bugs. In our short and simple examples, EdenTV analysis of program executions led to substantial runtime improvements by additional demand, and to the detection of a bottleneck, which is hard, if not impossible to detect without high-level profiling tools like EdenTV.

## Part III

# Parallel programming with skeletons



# Chapter 6

## Skeleton programming and implementation

### 6.1 Context: High-level parallel programming

We have already argued in our introduction *why* writing efficient and correct parallel programmes is far more complex than sequential programming: Even though research has proposed and developed a range of paradigms which abstract over simple message-passing between processes and memory locks, the latter techniques certainly constitute the predominant and most widely applied model, due to its simplicity and intuitiveness. Methods and tools for explicitly specifying parallel execution at a higher level of abstraction cannot be considered as an established standard in the industry yet. And together with today’s quick advances in parallel hardware architecture, multicore CPUs and Grid technology, there is an immediate need for concepts and sound development paradigms for parallel applications.

As Sergej Gorlatch points out in allusion to a famous Dijkstra paper [Dij68], using bare send and receive operations is “too much an invitation to make a mess of one’s [parallel] program” [Gor04, p.55, quoting Dijkstra], just like the `goto` statement in the 60s. However, message passing obviously is a necessary *implementation tool* for any interaction between machines with distributed memory. A position paper by Phil Trinder [Tri04] points out that, correspondingly, classical process calculi (like CCS, CSP and  $\pi$ -calculus [Mil99]) do not go beyond modeling simple send/receive operations, and therefore are not appropriate tools for modeling complex parallel computation. Simple intuitive theoretical models are needed, going alongside with abstractions geared towards parallel algorithms instead of a low-level machine-oriented view. The desired programming techniques for parallelism should capture algorithms rather than necessary machine interaction, and circumvent the complexity of parallel coordination as far as possible.

## 6.2 Parallel programming with skeletons

### 6.2.1 The skeleton idea

To supply conceptual understanding and abstractions of parallel programming, the notion of *algorithmic skeletons* has been coined by Murray Cole in 1989 [Col89]. An algorithmic skeleton abstractly describes the (parallelisable) structure of an algorithm, but separates specification of the concrete work to do as a parameter function. Skeletons are meant to offer ready-made efficient implementations for common algorithmic patterns, the specification of which remains sequential. Thus, an algorithmic skeleton contains inherent parallelisation potential in the algorithm, but this remains hidden in its implementation.

Standard patterns of parallel execution can be expressed easily by instantiating a suitable skeleton with the right parameter function. A broader research community has quickly adopted and developed the idea further [RG03]. Today, we find a range of languages intended for skeleton programming, with different computation languages and implementation flavours (e.g. [Kuc02, Ben07, Dan07]) and compilation-based approaches [BDO<sup>+</sup>95, MSBK01]. However, algorithmic skeletons still are, more or less, a research subject with little impact on mainstream software engineering. Possible reasons identified by Cole [Col04] are that skeleton approaches did not advertise their pay-back sufficiently, and that it is difficult to integrate existing practices and ad-hoc parallelism – in other words, the ability to customise predefined skeletons. Other models have found far broader acceptance in software engineering, namely MPI and *design patterns*, which we are briefly putting in context to the skeleton approach hereafter.

**Skeletons vs. elementary collective operations.** The low-level message passing standard MPI [MPI97] specifies standard patterns for common tasks of data distribution, exchange and reductions (the latter parameterised with predefined binary operations like sum, max, . . . , or user-defined functions), as “collective operations” (`MPI_Gather`, `MPI_Scatter`, `MPI_Allreduce` and similar).

In the cited position paper [Gor04], Sergej Gorlatch points out the drawbacks of send/receive operations in comparison to these collective operations. In an almost polemic style, Gorlatch argues against so-called “myths” attached to elementary message passing as a programming paradigm, and in favour of collective operations. He appeals for a more systematic development of parallel applications, for which he had provided foundations in earlier, more formal work [Gor00]: a sound theory and cost model for composed collective operations, allowing one to (reliably) optimise parallel programs by rule-based, cost-directed transformations. To a certain extent, collective operations and algorithmic skeletons follow the same philosophy: to specify common patterns found in many applications, and provide optimised implementations that remain hidden in libraries. However, the two models differ substantially in expressive power. While skeletons describe



a whole, potentially complex, algorithm, collective operations only predefine and optimise common standard tasks which are often needed in implementing more complex algorithms. Moreover, collective operations, by their very nature, are explicit about parallelism already in their specification.

**Skeletons vs. parallel “design patterns”.** In the late 90s, the notion of *design patterns* [GHJV00, GHJV93] (common organisational structures for object-oriented applications) set a new trend in software engineering, and a useful new programming methodology. Consequently, the design pattern paradigm has been applied to parallel programming, and has considerable potential in identifying inherent parallelism in common applications. Yet literature about design patterns often targets merely concepts and leaves the implementation to the established low-level libraries (see e.g. textbook [MSM05] for a typical example). Again, design patterns and algorithmic skeletons seem somehow comparable, as they both abstract from the task and identify a common pattern, with a standard implementation. Design patterns for parallel applications capture complex algorithmic structure, provide conceptual insight in parallelisation techniques and problem decomposition, and thereby give guidelines for the *design* of parallel algorithms. However, they cannot provide the programming comfort and abstraction level of algorithmic skeletons, ready-made higher-order functions, which completely hide parallelism issues.

### Functional languages and the skeleton approach.

From the perspective of functional languages, skeletons are specialised higher-order functions with a parallel implementation. Essentially, the skeleton idea applies a functional paradigm (higher-order functions with a parallel implementation) for coordination (in fact independent of the underlying *computation language*). While skeleton libraries for imperative language, e.g. [PK05, Ben07], typically offer a fixed, established set of skeletons (like the more basic collective MPI operations [MPI97]), parallel functional languages are able to express new skeletons, or to easily create them by composition [MSBK01, RG03].

Functional languages where the parallelism is introduced by pre-defined data parallel operations or skeletons, like NESL [Ble96], OCamlP3l [DDL98], or PMLS [MSBK01], have the advantage of providing optimal parallel implementations of their parallel skeletons, and allow skeleton composition. However, the programmer cannot invent entirely new problem-specific skeletons or operations. More explicit functional coordination languages are appropriate tools not only to apply skeletons, but also for their *implementation*, allowing formal analysis and conceptual modeling. Coordination structure, programming model and algorithm structure can be cleanly separated by functional languages, profiting from their abstract, mathematically oriented nature.

We will now concentrate on this implementation work especially for Eden and its implementation language EDI. The skeleton programming paradigm is fully coherent with the implementation structure in layers we have presented earlier for Eden’s high-level coordination features. In particular, skeleton implementation represents an interesting testbed for EDI’s parallel coordination features, since the task of implementing skeletons in a high-level language typically deals with a compromise between high abstraction and specific operational control of the execution, and thereby raises fundamental questions of coordination language design. As we will show subsequently, implementing coordination concepts in a (still declarative) low-level language like EDI is often quite verbose, but the library parts of the implementation retain the aforementioned fundamental advantages, and the explicitness offered by EDI helps produce both efficient and understandable implementations.

In Eden, the programmer can both apply skeletons and express, i.e. implement skeleton, in an arbitrary mixture. Skeletons can themselves be implemented in terms of others (we will see examples later), described as “parallel programming at two levels of abstraction” [KLPR00]: programming at a high level – with skeletons, and programming directly – with process abstractions and instantiations. The lower level, skeleton implementation, is the proper domain of EDI, mainly because its explicit communication allows to specify skeleton-specific communication structures beyond the caller-callee process connections created by Eden’s process instantiation.

## 6.2.2 A skeleton typology

In the original work [Col89], an algorithmic skeleton is specified purely *sequentially*, the *inherent* potential parallelism being exploited inside the implementation only. However, more recent literature uses the term “skeleton” in a more general manner, also including a second kind of higher-order functions. These abstractly describe how the subcomputations of a parallel computation interact, while *parameterising the sequential computation*. The following parts are dealing with and delimiting two different kinds of skeletons with different intentions.

### Problem-oriented, or Algorithmic skeletons

Algorithmic skeletons are *problem-oriented* and capture a common algorithm structure, leaving out parallelism from the specification (verbally following Cole’s original definition). We can distinguish several kinds of algorithmic skeletons.

**Data and task parallelism.** A big class of skeletons exploits the inherent parallelism that is present whenever a uniform transformation operates on a huge data set, often without any dependency between the different items. Pushed to the extreme, this yields the *data parallel* paradigm, where special container data

structures are defined, and operations on them have a completely hidden parallel implementation.

A different kind of inherent parallelism is present when different *tasks* have to be executed. A typical instance of this class, *task parallel* skeletons, is the divide&conquer algorithm scheme, where a problem is either trivial, or decomposed into independent subproblems to be solved and combined. A second example could be pipeline processing, where the data items flow through a chain of transformation stages to finally produce the desired output.

**Algorithm vs. implementation.** Furthermore, since different parallel implementations for the same algorithmic task are possible, we can differentiate between implementations, optimised for different architectures and problem characteristics ( [KLPR00] proposes a mid-level of “implementation skeletons” to remain architecture-independent, but adequately describe the underlying process structure).

An important aspect is that optimised implementations may require the parameter functions to have certain additional non-obvious properties. For instance, in a parallel reduction skeleton, commutativity of the binary operation leaves a lot more room for optimisations, since elements of the input can be combined freely out-of-order.

### Structure-oriented, or topology skeletons

A different class of skeletons we treat is purely *structure-oriented*, and explicit about the parallel execution. Structure-oriented skeletons describe the interaction between the processes of a parallel process network. More specifically, the focus of our research in this area has been to model interconnection networks between nodes of classical parallel machines, for instance process rings, process grids, hypercubes, and also pipelines, when considering their topological character. We use the term *topology skeletons* [BL08,BL05b] to describe this skeleton class. Well understood, these network topologies hardly exist in hardware any more, and are clearly way below the Eden programming level. However, a range of algorithmic work for classical parallel machines exists (see e.g. the algorithm examples in standard text books [Fos95, Qui94]); and algorithm design for a specific network topology, easily realised using topology skeletons, structures the communication pattern.

Usually, classical topologies follow a very regular scheme and can be neatly expressed in a declarative manner [BL08], often being a simple recursion [BL05b]. The implicit parent-child connections of Eden are insufficient. Specific topologies can be constructed using Eden’s dynamic reply channels between arbitrary nodes, in addition to the parent-child connections, or else programmed explicitly right from the beginning in EDI.



# Chapter 7

## Problem-oriented skeletons: Map and reduce

In this chapter, we want to discuss implementation aspects for *problem-oriented* skeletons, i.e. the classical algorithmic skeletons. These can be distinguished as data parallel or task parallel. In the former case, a starting point is provided by classical higher-order functions: `map`, `fold` and `scan` (in both directions), as well as variants and combinations thereof. More oriented towards algorithmic structure and technique are the task-parallel skeletons, such as `divide&conquer`, which we leave out of discussion.

Eden skeletons have been proposed in a variety of older publications ([LOP<sup>+</sup>03] summarises), presenting a range of different implementations and related cost-models, and we do not want to repeat this discussion. The intention of this chapter is not to present new skeletons, but to compare their different implementations in Eden and EDI. We will focus on and restrict the discussion to data parallel implementations (skeletons) of `map` and `reduce` and combinations. Eden versions for these higher-order functions are provided to make the chapter self-contained, sometimes modified to point out crucial properties. We compare them to EDI versions developed by ourselves. A related discussion of skeleton implementation in Eden and EDI can be found in our own work [BL07a].

### 7.1 Data parallel mapping

The higher-order function `map` applies a given function to all elements of a list. Function applications to the list elements are all independent of each other (therefore, parallel `map` problems are sometimes labelled “bag-of-tasks” problems, or “embarrassingly parallel”). Different parallelisations of `map` are discussed in [LOP<sup>+</sup>03], and we discuss implementation aspects for comparable EDI versions.

---

```

-- Eden's parallel map
parMapEden,parMapEden2 :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMapEden f xs = map deLift ([ createProcess (process f) x
                               | x <- xs ] 'using' whnfspine)

-- demand control helper
whnfspine :: Strategy [a]
whnfspine [] = ()
whnfspine (x:xs) = x 'seq' whnfspine xs

-- other version, internally monadic instead of Lift/deLift
parMapEden2 f xs = unsafePerformIO ( mapM (instantiateAt 0 (process f)) xs )

```

---

Figure 7.1: Straightforward implementations for map in Eden

**In a straightforward parallelisation,** one process is created for each element of the resulting list. This can be expressed easily in Eden using process abstraction and instantiation, as shown in Fig. 7.1, or programmed explicitly in EDI (Fig. 7.2).

The first Eden version uses strategy `whnfspine` to create additional demand to force the immediate creation of all processes, thereby requiring `createProcess` instead of `(#)` (strategy `whnfspine` would otherwise wait for the whnf of each process' result prior to creating the next process). As `parMapEden2` shows, our new EDI-based implementation can use the IO monad alternatively, and escapes from the IO monad by `unsafePerformIO` at top level, as the EDI version in Fig. 7.2 does. In the latter version, prior to spawning the child processes, the caller creates a set of channels (by the EDI abstraction `createCs` over the single channel creation `createC`). Each remote computation will receive one of these channels for sending back the result of applying `f` to input `x`. Embedded in this application `f x` is the *input*, potentially unevaluated! Whilst the Eden process instantiation spawns its own concurrent thread in the calling machine to send this input in normal form,

---

```

-- monadic EdI parmap using primitive operations only
parMapEdi :: NFData b => (a -> b) -> [a] -> [b]
parMapEdi f xs = unsafePerformIO (
    do (cs,rs) <- createCs (length xs)
       sequence_ [ spawnProcessAt 0 (sendNF ch (f x))
                  | (x,ch) <- zip xs cs ]
    return rs)

-- Eden version which embeds unevaluated input into the proc. abs.
parMapDM :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMapDM f xs = map deLift ([ createProcess (process (\() -> f x)) ()
                               | x <- xs ] 'using' whnfspine)

```

---

Figure 7.2: EDI and Eden implementations for map with embedded input

the EDI version acts as a *demand-driven* parallel map, useful to avoid bottlenecks in the caller. This can be modelled in Eden as well, by adding a dummy argument to the function applied to the list elements (shown as `parMapDM` in Fig. 7.2), and has sometimes been called *direct-mapping* (hence `dm`). The other way round, an EDI version could be defined which *communicates* input elements, but would look more complex. Essentially, we would need to inline the Eden process instantiation code, which creates input channels and forks input sender threads.

An advantage of EDI and the new implementation is that the `Lift - deLift` trick as well as the explicit demand control using the strategy `whnfspine` is no longer necessary to create a series of processes. However, this is purely an implementation aspect that remains hidden. A far more important difference between the Eden and EDI versions is rather subtle, hidden inside the Eden communication semantics. The presented EDI version `parMapEdi` will *always* send the output as a single data item. In contrast, if the result of applying function `f` is a list (i.e. if `b` is a list type), the Eden versions produce the output, a list of lists, as a list of *streams*. Whether this is desired and advantageous depends on the application. It is possible to construct an example (rather contrived: a backward dependency linking the outputs and the input list) where the EDI version deadlocks, while the Eden version, producing a stream, does not. Of course, an EDI version producing streams is merely a question of replacing `sendNF` by `sendNFStream`, but this will fix the skeleton type to lists. In Eden, the skeleton is polymorphic and uses overloading hidden from the programmer.

**Increasing the granularity** of the parallel processes is crucial and a standard issue. In the typical case, the input list is very long, whereas only few processors are available. Instead of one process per list element, each process (a previously fixed amount) can handle a whole sublist of elements. This is easily implemented in terms of the former `parMap*` versions: The input list is split up into sublists, `map f` is used as the applied function, and the original order is reconstructed by the inverse to the split function.

For this skeleton, the differences between Eden and EDI versions remain as explained before: The input list is either communicated, or unevaluated part of the remote computation – now applying to an input *list* of each worker. Process output is always a list, which will be communicated as a stream in Eden versions.

```
parmapfarm np f xs = unSplit ( parMap* (map f) (split np xs))
-- assuming unSplit . (split n) == id :: [a] -> [a] for every n >= 0
```

This process structure, coined as a `farm`, requires suitable `split` and `unSplit` functions, and takes a parameter determining how many processes should be used. The implicit helper functions, as well as hiding and automatically choosing the `np` parameter (we left it explicit here; it could be `np = noPe`), perhaps do not lead to optimal performance, but this is exactly the original philosophy of skeletons: A

```

parMapMW :: (Trans a, Trans b) => Int -> (a -> b) -> [a] -> [b]
parMapMW np = edenMW np 10 -- arbitrary prefetch 10

edenMW :: (Trans t, Trans r) =>
  Int -> Int -> (t -> r) -> [t] -> [r]
edenMW np prefetch f tasks = results
  where fromWorkers = map deLift
        (zipWith createProcess workerProcs toWorkers)
          'using' whnfspine
  workerProcs = [process (zip [n,n..] . map f) | n<-[1..np]]
  toWorkers   = distribute np tasks requests
  (newReqs, results) = (unzip . merge) fromWorkers
  requests    = initialReqs ++ newReqs
  initialReqs = concat (replicate prefetch [1..np])

distribute :: Int -> [t] -> [Int] -> [[t]]
distribute np tasks reqs = [taskList reqs tasks n | n<-[1..np]]
  where taskList (r:rs) (t:ts) pe
        | pe == r    = t:(taskList rs ts pe)
        | otherwise  =   taskList rs ts pe
  taskList _ _ _ = []

```

---

Figure 7.3: Eden master-worker skeleton and `parMap` implementation

purely sequential specification is denotationally fulfilled by a parallel implementation. We have a typical example of skeleton *implementation* aspects hidden from the user.

**For subtasks of irregular complexity,** or when the number of subtasks may vary depending on the input, *dynamic* load balancing is one of the most desired properties of a parallel `map` skeleton. Up to now, the input list has been distributed statically in advance, and the purely functional coordination constructs of Eden are not sufficient to describe dynamic task distribution. In order to specify a parallel map where the input list is distributed on demand, we need to use the nondeterministic Eden construct `merge`. The `merge` “function” adds data to the output stream as soon as it is available in any of the input streams, in nondeterministic order. As shown in Fig. 7.3, this can be used for a *master-worker* scheme implementing `map`, where a worker process gets a new task every time it returns a result. A `prefetch` parameter determines the number of initial tasks assigned to a worker, a buffer size which prevents workers from running out of work.

In order to indicate which worker has completed a task, every worker tags its results with a fixed number between 1 and `np`. The master process merges result streams `fromWorkers` nondeterministically, and then separates the proper results from these worker numbers, which serve as requests for new work. Results are returned unsorted, in the order in which they have been sent back by the workers.



---

```

ediMW :: (NFData t, NFData r) =>
  Int -> Int -> (t -> r) -> [t] -> IO [r]
ediMW np prefetch f tasks = do
  (wInCCs, wInCs) <- createCs np
  (wOutCs, wOuts) <- createCs np
  sequence_ [ spawnProcessAt 0 (worker f wOutC wInCC)      {- workers -}
             | (wOutC,wInCC) <- zip wOutCs wInCCs ]
  taskChan <- newChan                                     {- task channel -}
  fork (writeList2Chan taskChan
        ((map Just tasks) ++ (replicate np Nothing)))
  sequence_ [ fork (inputSender prefetch inC taskChan answers)
             | (inC,answers) <- zip wInCs wOuts ]      {- input senders -}
  return (concat wOuts)

```

---

Figure 7.4: EDI workpool skeleton, using concurrent `inputSender` threads

Task distribution is specified by the auxiliary function `distribute`, which takes the list of requests and the available tasks as arguments. The function distributes the tasks to `np` sublists as indicated by the requests list. The number of initial requests is determined by the skeleton parameter `prefetch`. A crucial property of the function `distribute` is that it has to be “incremental”, i.e. be able to deliver partial task lists without the need to evaluate requests not yet available.

In its entirety, we prefer to consider master-worker skeletons as topology skeletons, discussed in the next chapter, while [LOP<sup>+</sup>03] applies a different classification and ranges them as “systolic”. More sophisticated versions of the master-worker skeletons can be used for other, more complex algorithm classes, and also allow a whole hierarchy of masters (to avoid bottlenecks) [BDLP08]. What we want to show here, in the context of `map` skeletons, is that a similar workpool skeleton can also be implemented without the need for Eden’s `merge` construct, nor the sophisticated `distribute`. Instead of the single `merged` request list and `distribute`, we use a *Concurrent Haskell channel*<sup>1</sup>, which is read by concurrent sender threads inside the master (in fact, similar to the constructs used to *implement* `nmergeIO` [GHC, library code]). Figure 7.4 shows the resulting EDI workpool skeleton, which returns its result in the IO monad.

The master needs channels not only to receive the results, but also to initiate input communication with the workers, thus two sets of `np` channels are created. A set of worker processes is instantiated with these channels as parameters. As shown in Fig. 7.5, each worker creates a channel to receive input, sends it to the parent, and then connects to the given output channel to send a stream of results.

We use a `Maybe` type in order to indicate termination. The `taskChan` is created

---

<sup>1</sup>A Concurrent Haskell channel (data type `Chan`) models a potentially infinite stream of data which may be written and read concurrently by different threads. Due to nondeterministic scheduling, channel operations are in the IO monad, like the EDI coordination constructs.

```

worker :: (NFData t, NFData r) =>
  (t -> r) -> ChanName' [r] -> ChanName' (ChanName' [t]) -> IO ()
worker f outC inCC
  = do (inC, inTasks) <- createC -- create channel for input
      sendNF inCC inC           -- send channel to parent
      sendNFStream outC        -- send result stream
      ((map f) inTasks)

inputSender :: (NFData t) =>
  Int -> ChanName' [t] -> Chan (Maybe t) -> [r] -> IO ()
inputSender prefetch inC concHsC answers
  = do connectToPort inC
      react ( replicate prefetch undefined ++ answers)
where react :: [r] -> IO ()
  react [] = return ()
  react (_,as) = do
    task <- readChan concHsC -- get a task
    case task of
      (Just t) -> do (rnf t 'seq' sendData Stream t )
                    react as
      Nothing  -> sendData Data [] -- and done.

```

---

Figure 7.5: Worker process and `inputSender` thread for EDI workpool

and (concurrently) filled with the tagged task list (`map Just tasks`), followed by `np` termination signals (`Nothing`). This task channel will be concurrently read by several input senders, one for every worker process, which will be forked next. Every input sender consumes the answers of one worker and emits one new task per answer, after an initial `prefetch` phase (see Fig. 7.5)<sup>2</sup>. The value returned by the master process remains unevaluated. Therefore, results can be combined in various manners. The version presented here collects the results by a simple `concat`, the Haskell prelude function to concatenate a list of lists. Another variant would be the nondeterministic `nmergeIO` from Concurrent Haskell, or we could merge the answers list back into the original task order, using further additional tags added to the tasks and the fact that the order of results is ascending in each worker's output (we will not elaborate this further).

The EDI version of the workpool looks more specialised and seems to use more concurrent threads than the Eden version, which is considerably shorter. Since EDI uses explicit communication, the separate threads to supply the input become obvious. The Eden version works in quite the same way, but the concurrent threads are created implicitly by the process instantiation operation `createProcess`. Apart from one extra thread filling the channel with available tasks, both versions have exactly the same degree of concurrency; it is not surprising that both workpool implementations are similar in runtime and speedup.

---

<sup>2</sup>Note that inside `react`, we resort to the primitives `connectToPort` and `sendData`, to make input senders wait for results without accessing them, and send new tasks as requested.

Once the master process uses concurrent threads and the IO monad, it may easily be extended in different ways. One very useful extension would be to include a state in the master process, e.g. a “current optimal” solution for a branch-and-bound algorithm, or a dynamically increasing task pool, or using a stack instead of a FIFO queue for task management. A recent diploma thesis by Mischa Dieterle [Die07] has shown the feasibility of such skeletons with mostly functional, and purely Eden-based implementations, but implementations become quite cumbersome (see [MP03] for another case study). The explicitness of parallelism, communication and concurrency inflates the EDI code, but makes it more readable and is thus advantageous for later modifications and specialisations.

## 7.2 Parallel map-and-reduce

Another classical example discussed in [LOP<sup>+</sup>03] is a parallel reduction over a list of elements, known as a higher-order function `fold`, and often in combination with a preceding transformation of the list elements (in other words, a `map` again). We will first concentrate on the reduction alone.

Sequentially, the list can be folded in either direction, from the left or from the right, which leads to different types. However, any possible parallelisation has to make use of *associativity*, since the order in which the elements are combined has to be broken up. For parallel versions, a restricted type is common practice, and we can define a parallel reduction using `parMapEden`:

```
parReduce :: (Trans a) => Int -> ( a -> a -> a ) -> a -> [a] -> a
parReduce np f neutral list = foldr f neutral subRs -- requires associativity
  where sublists = splitIntoN np list
        subFold  = foldl' f neutral
-- requires "neutral" to be in fact neutral for f: f x neutral = x
        subRs    = parMapEden subFold sublists
```

The parallel reduction pre-reduces the sublists in several processes (parameter `np` determines how many of them), and the caller only combines the (usually few) sub-results.

Note that we sensibly use a fold-left, and a folding variant `foldl'` which strictly evaluates the intermediate results. The input sublists will be sent as *streams*, and a non-strict `foldl` would accumulate all data in the process' heap before combining it in the last step, whereas the strict variant successively combines to subresults in advance. This is also the essential reason why we see no advantage in using EDI instead of Eden for this skeleton: the strength of EDI is explicit communication. But communication is optimal here, due to the Eden communication semantics: Each of the created lists of lists (by `splitIntoN`) is one task for the `parMap`, and is sent as a stream. This can well be specified in EDI, but we did not present such an EDI-`parMap`; the Eden version will do just well.

```

splitIntoN :: Int -> [a] -> [[a]]
splitIntoN n xs = takeIter parts xs
  where l = length xs
        parts = zipWith (+) ((replicate (l `mod` n) 1) ++ repeat 0)
                      (replicate n (l `div` n))

takeIter :: [Int] -> [a] -> [[a]]
takeIter [] [] = []
takeIter [] _ = error "elements left over"
takeIter (t:ts) xs = hs : takeIter ts rest
  where (hs,rest) = splitAt t xs

unshuffle :: Int -> [a] -> [[a]]
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]
  where takeEach n [] = []
        takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
-- inverse to unshuffle
shuffle :: [[a]] -> [a]
shuffle = concat . transpose

```

Figure 7.6: Helper functions for splitting lists

### Sidestep: A safer interface?

In contrast to the sequential fold operations, this parallel implementation requires the `neutral` element to be indeed *neutral* for  $f$ ,  $f(x, n) = x \forall x$ , because it will be used many times (on every parallel sub-fold) instead of just once. Therefore, implementations sometimes offer an interface where the neutral element is baked into the folding function, instead of passing it as a separate argument, leading to less explicit requirements for the skeleton parameters.

```

parReduceL :: (Trans b) => Int -> ([b] -> b) -> [b] -> b
parReduceL np f list = f subRs
  where sublists = splitIntoN np list
        subRs    = parMapEden f sublists

```

In this case, the requirements for correctness are concentrated only in  $f$ . “Associativity” with lists is required:  $f(f(l_1) : l_2) = f(l_1 ++ [f(l_2)])$ , and implied by the definition, since  $f$  operates on a whole input list and does not specify an order of application. Another assumption is the existence of a neutral element  $n$  for  $f$ :  $f([]) = n$  and  $f([x, n, n...]) = f([x])$ . More generally,  $f(l_1 ++ (n : l_2)) = f(l_1 ++ l_2)$ .

### Optimisation for commutative operations

Additionally, it could be useful to modify the order of the list elements. Up to now, we have used an unspecified `splitIntoN` in the code. If the binary operation  $f$  is not commutative, this helper function must split the input list in `np` parts, but maintain its order, which absolutely requires length information. The respective code is shown in Fig. 7.6.

If  $f$  is commutative, arbitrary reordering of the input list is possible, with vast consequences to the performance of the skeleton. In particular, `splitIntoN` is prohibitive in a context where the input is itself a *stream*: Since the stream length cannot be known in advance, no sublist can be produced until all elements have arrived. With a commutative  $f$ , sublists can instead be *streamed* to the `parMap` processes in *round-robin* manner, as the function `unshuffle` from Fig. 7.6 does. Assuming commutativity, we use this helper function to define **the classical map-and-reduce**:

```
parmapReduceStream :: (Trans a, Trans b) =>
    Int ->          -- no. of processes
    (a -> b) ->     -- mapped on input
    (b -> b -> b) -> -- reduction (assumed commutative)
    b ->           -- neutral element for reduction
    [a] -> b
parmapReduceStream np mapF redF neutral list
= let sublists = unshuffle np list
      subFold  = foldl' redF neutral . (map mapF)
      subRs    = parMapEden subFold sublists
  in foldl redF neutral subRs
```

This final version includes a transformation of the list elements prior to reduction, which is rather simple. We have used `parMapEden` in all previous versions, after breaking up the input in pieces in the style of the `parmapfarm` implementation. All we need to add is an additional `map f`, and the resulting sublists are reduced instead of shuffling.

## Performance

Measurements with the `parMap` variants, including a reduction step, can be found in our publication [BL07a] (not reproduced here). The measured program was computing the sum of Euler Totients,  $\sum_1^n \varphi(k)$  for  $n = 25000$ , using a set of worker processes (`map` as a farm), with numbers distributed evenly among the processes. Since the values are summed up afterwards (`foldl (+) 0`) following the `map`), each `map` process computed the partial sum in parallel as well. The sum is commutative, 0 neutral, and `sum []` is 0; suitable for any of the presented map-and-reduce skeletons. Differences between Eden and EDI versions were almost negligible in these measurements, EDI version showing slightly better speedup. The overhead for the Eden module code is minor, and only the way input data is transmitted is relevant, depending on the concrete application.

## 7.3 The “Google MapReduce” skeleton

Another, more general variant of map-and-reduce has been proposed, as a programming model for processing large datasets, by Google personnel Jeff Dean

and Sanjay Ghemawat. In January 2008, an update of the original publication (OSDI 2004 [DG04]) appeared in the ACM communications [DG08].

The intention to provide a framework which allows one “to express the simple computations [...] but hides the messy details of parallelization, fault-tolerance, data distribution, and load balancing, in a library” [DG04] is precisely the skeleton idea. However, the word “skeleton” does not figure in any of the two publications! But never mind its origin (neither publication claims for the model to be entirely new), its essential merit is that it brought the skeleton approach to industry. The model has found great acceptance as a programming model for parallel data processing (e.g. [CKL<sup>+</sup>07, RRP<sup>+</sup>07]), and recently became very fashionable in higher education, as well as in broader commercial developer communities.<sup>3</sup>

### 7.3.1 MapReduce functionality

The Google-mapReduce<sup>4</sup> skeleton is a generalisation of the classical mapReduce we have presented previously. To describe its functionality, we follow a paper by Ralf Lämmel from Microsoft [Läm06], written in 2006 and published in SCP recently [Läm08] (but not acknowledged by the Google authors in their ACM version). Lämmel aims to deliver an executable (functional) specification of Google-mapReduce, and to “identify and resolve some obscurities in the informal presentation”. His very comprehensive work uses Haskell, and he explains syntax, all required language features and data types he uses from libraries on the way.

The computation scheme of Google-mapReduce is depicted in Fig. 7.7. In a nutshell, a Google-mapReduce instance first transforms key/value pairs to (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 thus has to be grouped by keys, and the whole computation has two logical phases.

To obtain a more thorough and unambiguous specification, Lämmel checks the informal specification of the Google paper and analyses the given examples. Lämmel’s main criticism is that the original paper, at times, confuses lists and sets, and its irritating use of “map” and “reduce”. Both functions are *arguments* to the Google-mapReduce-skeleton written by the user, and not to be confounded with the higher order functions their names might indicate. Additionally, as Lämmel points out, what is called “reduce” in the Google publications is not properly a function which could be the argument to a fold (i.e. reduce) operation, nor is it always a reduction in the narrow sense.

---

<sup>3</sup>This can be confirmed by a web and blog search, which yields a considerable number of posts about the skeleton in developer forums, and a range of educational material online.

<sup>4</sup>To avoid confusion with the skeleton presented previously, we denote the special map-and-reduce in question as Google-mapReduce hereafter.

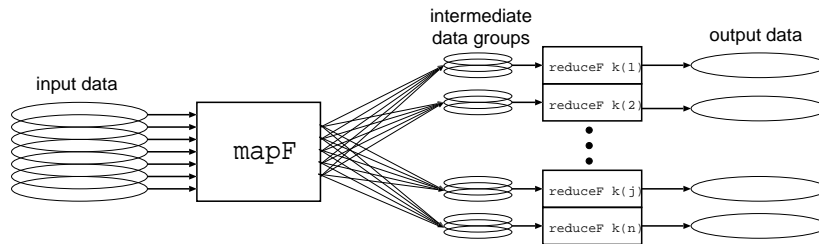


Figure 7.7: Computation scheme of Google-mapReduce

Step by step, Lämmel develops the following general Haskell type for the Google-mapReduce skeleton:<sup>5</sup>

```
gOOGL_E_MapReduce :: forall k1 k2 v1 v2 v3.
  Ord k2 =>                -- Needed for grouping
  (k1 -> v1 -> [(k2,v2)]) -- The 'map' function
-> (k2 -> [v2] -> Maybe v3) -- The 'reduce' function
-> Map k1 v1                -- A key to input-value mapping
-> Map k2 v3                -- A key to output-value mapping
```

As we shall see, the input to a skeleton instance is neither a set, nor a list, but a finite mapping from keys to values, where duplicate values are not allowed for the same key. And likewise, the output of the skeleton conceptually does not allow the same key to appear twice. The ‘reduce’ function is allowed to produce no output for a given input, thereby its type uses `Maybe`. Likewise, its output is not necessarily of the same type as the intermediate value (but it “typically” is,  $v2 == v3$ ).

The ‘map’ function is (uncurried and) applied to all pairs  $(k_1, v_1)$ . It may produce a whole *list* of intermediate pairs from just one application. An implicit grouping step for intermediate pairs follows, which groups intermediate pairs by their key, thereby the ordering constraint on the intermediate keys. In principle, an equality constraint `Eq k2` would suffice, but ordering allows to use more efficient data structures. For each intermediate key, the list of intermediate values is reduced using the supplied ‘reduce’ function, and regrouped with its key, discarding from the output all items which did not reduce to a value.

An example often given in publications on Google-mapReduce is to compute how many times certain words appear in a collection of web pages.

```
wordOccurrence = gOOGL_E_MapReduce toMap forReduction
  where toMap      :: URL -> String -> [(String,Int)]
        toMap url content = zip (words content) (repeat 1)
  forReduction :: String -> [Int] -> Maybe Int
  forReduction word counts = Just (sum counts)
```

<sup>5</sup>The code is provided online by Lämmel, so we do not reproduce it here, see <http://www.cs.vu.nl/~ralf/MapReduce/>

The input is a set of pairs: web page URLs and web page content (and the URL is completely ignored). The ‘map’ part retrieves all words from the content and uses them as intermediate keys, assigning constant 1 as intermediate value to all words. Reduction sums up all these ones to determine how many times a word has been found in the input.

A range of other, more complex applications is possible, for instance, iteratively clustering large data sets by the k-means method (used as a benchmark in two recent publications [CKL<sup>+</sup>07,RRP<sup>+</sup>07]): The input is a collection of data vectors (and arbitrary irrelevant keys). In each iteration, a set of  $k$  cluster centroids are chosen (randomly in the first iteration, from previous results later). Parameterising the function to map with these centroids, the map part computes distances from the input vector to all centroids and yields the ID of the nearest centroid as the key, leaving the data as the value. The reduction, for each centroid, computes the mean vector of all data vectors assigned to the respective cluster to yield a set of  $k$  new cluster centroids, which is used in the next iteration. Figure 7.8 shows the code.

Last but not least, the classical `mapReduce` skeleton we have presented previously can be implemented as a special case. The map function here produces singleton lists and assigns a constant intermediate key 0 to every one. The reduction function ignores these keys, and left-folds the intermediate values as usual.

```
mapReduce :: (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mapReduce mapF redF neutral input = head (map snd (toList gResult))
  where mapF' _ x = [(0,mapF x)]
        redF' _ list = Just (foldl' redF neutral list)
        gResult      = gOOGLEReduce mapF' redF'
                      (fromList (zip (repeat 0) input))
```

### 7.3.2 Parallelisation potential

Both the original description by the Google authors, Dean and Ghemawat, and Ralf Lämmel discuss inherent parallelism of the Google-mapReduce skeleton. While Lämmel presents substantial work for a sound understanding and specification of the skeleton, his parallelisation ideas remain at a high level, at times over-simplified, and he does not discuss any concrete implementation. The original paper by the Google authors describes and quantifies parallelisation and also gives details about the physical setup, the middleware in use, and error recovery strategies.

The skeleton offers different opportunities for parallel execution. First, it is clear that the map function can be applied to all input data independently. Furthermore, since reduction is done for every possible intermediate key, several PEs can be used in parallel to reduce the values for different keys. Additionally, the mapper processes in the implementation perform pre-grouping of intermediate



---

```

type Vector = [Double]

k_means :: Int -> Int -> Int -> [Vector] -> IO [Vector]
k_means iterations inputLength k vs
  = do rndGen <- getStdGen
      let startMap = fromList (zip [1..] start):: Map Int Vector
          start    = map (vs!!) startIndices  :: [Vector]
          startIndices = chooseNDistinct k
                    (randomRs (0,inputLength - 1) rndGen)
          results = iterate (clustering vs) startMap
      return (map snd (toList (results!!iterations)))

clustering :: [Vector]          -- input vectors
            -> Map Int Vector -- k distinct cluster centroids
            -> Map Int Vector -- new centroids
clustering vs csMap = gOOGLE_MapReduce toMap forReduction inputvs
  where inputvs = fromList (zip [1..] vs):: Map Int Vector
        cs      = map snd (toList csMap)
        toMap :: Int -> Vector -> [(Int,Vector)]
        toMap _ vec = [(1 + minIndex (map (distance vec) cs),vec)]
        forReduction :: Int -> [Vector] -> Maybe Vector
        forReduction id vs = Just (center vs)

distance :: Vector -> Vector -> Double          -- a metrics on vectors
distance v1 v2 = sum (map abs (zipWith (-) v1 v2)) -- Here: manh. distance

center :: [Vector] -> Vector                    -- computes the mean vector
minIndex :: Ord a => [a] -> Int                 -- as name suggests
chooseNDistinct :: Eq a => Int -> [a] -> [a] -- as name suggests

```

---

Figure 7.8: k-means clustering implemented by Google-mapReduce

pairs by (a hash function of) intermediate keys. This grouping is done for all data at once, splitting the whole algorithm in two phases. The productive implementation described in [DG08] is based on intermediate files in Google’s own shared file system GFS. Pre-grouped data is periodically written to disk, and later fetched and merged by the reducer tasks before they start reduction of values with the same key. This makes it possible to reassign jobs in case of machine failures, making the system more robust. Furthermore, at the end of the map phase, remaining map tasks are assigned to several machines simultaneously to compensate load imbalances.

### Following the specification by Lämmel

To enable parallel execution, Lämmel proposes the version shown in Fig. 7.10. Interface and functionality of the Google-mapReduce skeleton are extended in two places:

First, input to the map function is grouped in bigger “map jobs”, which allows

to adapt task size to the resources available. For instance, the job size can be chosen appropriately to fit the block size of the file system. For this purpose, the proposed outer interface includes a `size` parameter and an estimation function `estSize`. The skeleton input is sequentially traversed and partitioned in tasks with estimated size close (but less than) the desired task size.

Second, *two* additional pre-groupings of equal keys are introduced. The map operation can produce any number of intermediate output for one input. Assuming commutativity of the reduction in use, the map processes hold on to all data they produce, and pre-group output with the same intermediate key, using the `COMBINER` parameter function. In many cases, this combiner will be the same function as the one used for reduction, but in the general case, its type differs from the `REDUCE` function type. Furthermore, both the outer and the inner interface include two parameters for partitioning (possibly many) different intermediate keys into a (smaller) number of key groups. The parameter `parts` indicates how many partitions (and parallel reducer processes) to use, and the function `keycode` maps (or: is *expected* to map; the code in [Läm06] does not check this property) each possible intermediate key to a value between 1 and `parts`. This mimics the behaviour of the productive Google implementation, which saves partitioned data into  $n$  intermediate files per mapper.

Our parallel straightforward implementation of the skeleton consists of replacing the `map` calls in the code (see Fig. 7.10) by appropriate map skeletons. An implementation which verbally follows the description should create  $m$  mapper processes, which is best done using the farm skeleton presented previously. However, the interface proposed by Lämmel lacks the  $m$  parameter, thus our par-

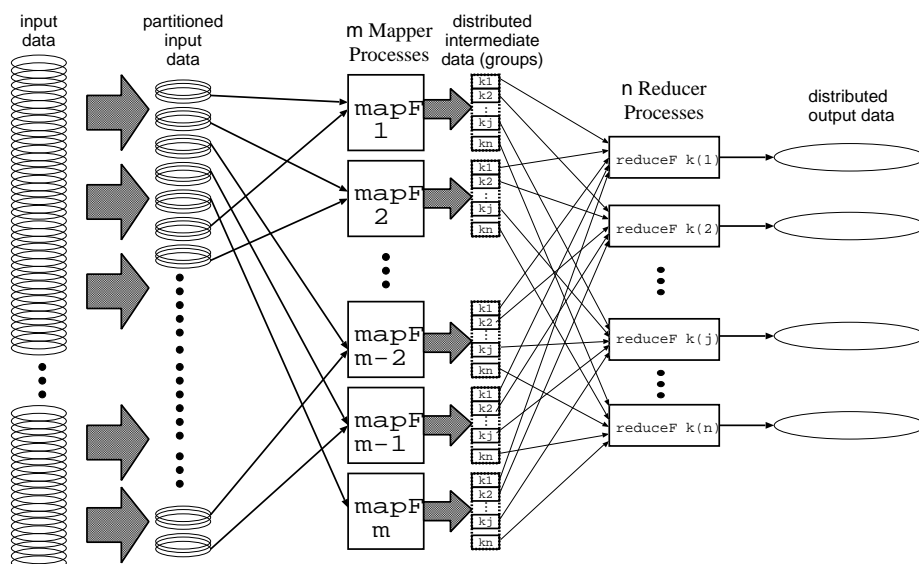


Figure 7.9: Parallel Google-mapReduce, parallelisation as described in papers

```

-- outer interface
parMapReduce' :: (Ord k1, Ord k2) =>
  (v1 -> Int) -> Int          -- Size estimation on input, desired task size
-> Int -> (k2 -> Int)         -- Number of partitions, key partitioning
-> (k1 -> v1 -> [(k2,v2)])   -- 'map' function
-> (k2 -> [v2] -> Maybe v3)   -- 'combiner' function
-> (k2 -> [v3] -> Maybe v4)   -- 'reduce' function
-> Map k1 v1 -> Map k2 v4     -- Input and output
parMapReduce estSize size parts keycode mAP cCOMBINER rREDUCE
=  concatOutput
  .  parMapReduce parts keycode mAP cCOMBINER rREDUCE
  .  splitInput estSize size

-- inner interface
parMapReduce :: Ord k2 =>
  Int -> (k2 -> Int)         -- Number of partitions, key partitioning
-> (k1 -> v1 -> [(k2,v2)])   -- 'map' function
-> (k2 -> [v2] -> Maybe v3)   -- 'combiner' function
-> (k2 -> [v3] -> Maybe v4)   -- 'reduce' function
-> [Map k1 v1]               -- Distributed input data
-> [Map k2 v4]               -- Distributed output data
parMapReduce parts keycode mAP cCOMBINER rREDUCE
=  map ( -- parallelise! n reducers
      reducePerKey rREDUCE    -- 7. Apply 'reduce' to each partition
      . mergeByKey )         -- 6. Merge scattered intermediate data
  . transpose                 -- 5. Transpose scattered partitions
  . map ( -- parallelise! m mappers
      map (
        reducePerKey cCOMBINER -- 4. Apply 'combiner' locally
        . groupByKey )         -- 3. Group local intermediate data
      . partition parts keycode -- 2. Partition local intermediate data
      . mapPerKey mAP )         -- 1. Apply 'map' locally to each piece

```

Figure 7.10: Parallel Google-mapReduce skeleton, following Lämmel [Läm06] (we have added the parallelisation annotations in bold face)

allelisation might simply use as many mappers as reducer processes,  $n = m$  (using the `parmapfarm` skeleton). The number of reducers,  $n$ , is given as a parameter: the number of `parts` in which the hash function `keycode` partitions the intermediate keys. Since the intermediate outputs of the  $m$  mapper processes are partitioned by `keycode`, any of the `parMap` skeletons we have presented earlier can be used to create these  $n$  reducer processes.

### EDI implementation

A major drawback of this straightforward version, directly derived from Lämmel’s code [Läm06], is its strict partitioning into the map phase and the reduce phase, and the call to `transpose` in between. In our implementation, all intermediate

data produced by the mapper processes is sent back to the caller, to be reordered (by `transpose`) and sent further on to the reducer processes. We have therefore developed a more realistic version with direct stream communication between mappers and reducers.

In the optimised EDI version, we keep creating as many mappers as reducers,  $n$  (a larger number of mapper processes could easily be created). 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  $n : n$  communication must be set up. Each process creates a list of  $n$  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. Due to the complex communication structure, we have preferred to use EDI for the implementation (using dynamic channels, an Eden implementation is possible as well). The full code of this EDI implementation is included in the appendix (part B.2.1), and uses some more internal channels, similar to the process creation function in the Eden implementation.

From the algorithmic perspective, the implementation we propose deviates from the originally presented skeleton in a subtle way, profiting from the streaming features in EDI: The productive implementation uses an *overall* pre-grouping of intermediate data by keys (using a hash function which assigns each possible key to one of a given number of buckets). The whole data subset processed by one mapper is pre-grouped into buckets, each for one reducer process. In Google's productive implementation, the buckets are written to a distributed mass storage system (GFS) and later fetched by reducer processes. While this is clearly essential for fault tolerance (in order to restart computations without data being lost in failing machines), we consider accumulating all intermediate data on mass storage a certain disadvantage in performance and infrastructure requirements. Using stream processing for the intermediate data could be an advantage, since it avoids the file system overhead for long-running data-intensive jobs: Data is processed in a pipeline and held in memory, no intermediate files exist. However, this might require restructuring parts of the algorithm, due to the intermediate grouping being slightly different.

### 7.3.3 Example applications

Our first example illustrates that, in both skeleton versions, the additional reduction using the `cOMBINER` function (as opposed to simply applying `rEDUCE` twice) is necessary, but might render algorithms more complicated.

*Example: (Parallel  $k$ -means)* In a parallel  $k$ -means implementation, computing

**Centroid of vectors**  $\{v_1, \dots, v_n\}$

Let  $n = k + l$ , then  $\frac{1}{k+l} \cdot \sum_1^{k+l} \vec{v}_i = \frac{\sum_1^k \vec{v}_i}{k+l} + \frac{\sum_1^l \vec{v}_{k+j}}{k+l}$

$$= \frac{k}{k+l} \cdot \underbrace{\frac{\sum_1^k \vec{v}_i}{k}}_{\text{pre-grouped! } S_k} + \frac{l}{k+l} \cdot \underbrace{\frac{\sum_1^l \vec{v}_{k+j}}{l}}_{\text{pre-grouped! } S_l}$$

Defining  $\mu_{kl} := \frac{k}{k+l} = \left(\frac{k+l}{k}\right)^{-1} = \left(1 + \frac{l}{k}\right)^{-1}$

we can also write  $\frac{l}{k+l} = \frac{k+l-k}{k+l} = 1 - \left(1 + \frac{l}{k}\right)^{-1} = 1 - \mu_{kl}$

$$\Rightarrow \frac{1}{k+l} \cdot \sum_1^{k+l} \vec{v}_i = \mu_{kl} \cdot S_k + (1 - \mu_{kl}) \cdot S_l$$

Figure 7.11: Combining pre-grouped sub-centroids for  $k$ -means

```
-- precombine a collection of vectors, include their number
COMBINER :: Int -> [Vector] -> Maybe (Int,Vector)
COMBINER _ [] = Nothing -- should not happen
COMBINER _ vs = Just (length vs, center vs)

rREDUCE :: Int -> [(Int,Vector)] -> Maybe (Int,Vector)
rREDUCE _ [] = Nothing -- should not happen
rREDUCE _ vs = Just (round w,v)
  where vs' = map (\(k,v) -> (fromIntegral k,v)) vs :: [(Double,Vector)]
        (w,v) = foldl1' combineWeighted vs'

combineWeighted :: (Double,Vector) -> (Double,Vector) -> (Double,Vector)
combineWeighted (k1,v1) (k2,v2) = (k1+k2,zipWith (+) v1' v2')
  where f = 1/(1+(k2/k1))
        v1' = map (*f) v1
        v2' = map *(1-f) v2
```

Figure 7.12: Parameter functions to be used for parallel  $k$ -means algorithm

new centroids by simply summing up *all* vectors clearly bears the risk of numeric overflows (unless arbitrary-precision libraries are used). It is a good idea to precombine subsets of vectors to a sub-centroid but it must be taken into account that this precombination might be based on an *arbitrary* number of vectors, and thus pre-computed sub-centroids should not be combined by a simple sum.

Once the problem is spotted, combining the obtained sub-centroids is easy if the number of pre-grouped vectors is known. Fig. 7.11 shows how values can

be accurately combined without risk of overflow. Thus, the pre-grouping, or `cOMBINER`, function yields not only the centroid  $S_i$  of a subset of vectors, but also  $k_i$ , the number of vectors used to obtain it. The final reduction `rEDUCE` can then compute the overall centroid as shown, and avoids summing and multiplying numbers to astronomic height. In the version which we derived from Lämmel's code (Fig. 7.10), the `rEDUCE` function should also be used to pre-group vectors, and consequently needs a slightly modified `mAP` function.

However, parallel  $k$ -means is not suitable to show the effects of the suggested pre-grouping modification. The problem is, the algorithm works iteratively in fixed steps, and amounts to setting up a new `Google-mapReduce` instance for each iteration. These globally synchronised iteration steps and the skeleton setup overhead dominate the runtime behaviour. ◀

*Example: (A micro-benchmark)* In order to spot differences between the two `Google-mapReduce` implementations, an artificial micro-benchmark is more practical than a real program. For such an overhead test, we have instantiated the skeleton with the following worker functions:

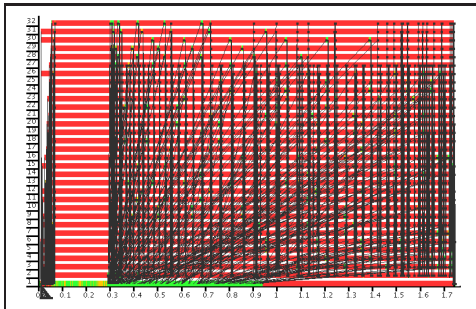
```
mapStage :: Int -> Int -> Int -> String -> [(Int,Char)]
mapStage buckets delay n "" = mapStage buckets delay n (show n)
mapStage buckets delay _ input = let key = (read input 'mod' buckets)
                                   in spendTime delay [(key,c) | c <- input]

reduceStage :: Int -> Int -> [Char] -> Maybe Char
reduceStage delay c [] = spendTime delay Nothing
reduceStage delay c str = spendTime delay (Just (['a'..'z']!!c))
```

The input for the `mapStage` of the skeleton are strings which represent natural numbers. The function simply reads the number and assigns a key in the expected range. The reduce stage completely ignores the data and only yields a letter which indicates the bucket. Both functions have a `delay` parameter which may add some artificial workload to the computation.

Fig. 7.13 shows `EdenTV` process views of program runs for this program with the two `Google-mapReduce` implementations, with and without additional workload. An input of 100000 strings, representing numbers from 1 to 100000, was grouped into 26 buckets (by their modulus). Data was processed in chunks of 500 strings for each task, by 31 workers (main PE reserved for the master).

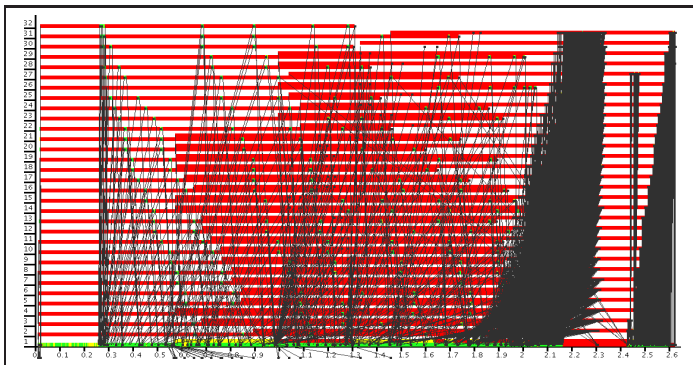
As explained, the version derived from R.Lämmel's code uses separate processes for the map phase and the reduce phases, and collects intermediate results in the master process, which leads to poor performance in the run without workload. The optimised `EDI` version is much faster and seems to have considerably lower overhead altogether. On the other hand, please note that the `EDI` version uses considerably more threads, and sends almost as many messages. The difference is that data is directly sent to the respective reducer process, whereas in the other version, the single master process quickly becomes a bottleneck.



(a) Without workload:  
(process view including messages)

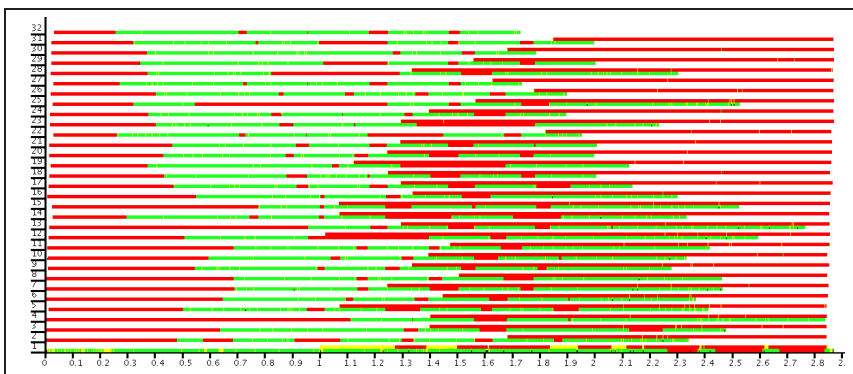
left: EDI version

below: Version derived from Lämmel

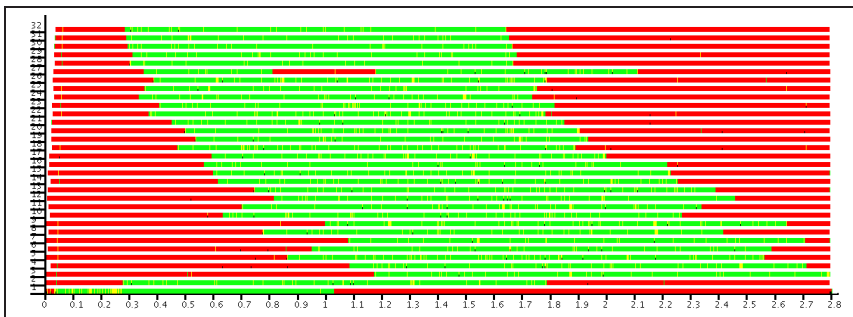


Beowulf Cluster  
Heriot-Watt University,  
Edinburgh, 32 machines  
(Intel P4-SMP@3GHz,  
512MB RAM, Fast  
Ethernet)

(b) Measurement with workload (process view)



(from Lämmel)



(EDI version)

Figure 7.13: Google-mapReduce micro-benchmark, with and without workload

As the traces with increased workload show, performance is much better as soon as the workers really *do* something. The EDI version, which we optimised for *streaming* the data through the process network, does not have a real advantage for the trivial reduction step we have used here, and takes the same time when creating enough additional workload. <

In summary, a broad variety of functional code in this chapter shows, more or less, that Eden and EDI provide comparable mechanisms for implementing problem-oriented skeletons. EDI code is more directly accessible to reading since no special communication semantics and implicit mechanisms come into play. On the other hand, the EDI code is longer and often specifies standard tasks, such as channel creation between a caller and a new process.

Both Eden and EDI are languages which offer explicit communication channels to the programmer: as a side-effecting non-functional extension in Eden, encapsulated in the IO monad and essential for programming in EDI. When complex process topologies like Google-mapReduce need to be set up, the implementation can profit from EDI's more explicit constructs for direct communication. In the next chapter, we will concentrate on these *process topologies* and present skeletons in a different sense, where explicit communication is even more important.



# Chapter 8

## Structure-oriented skeletons

While in the previous chapter, we have described examples of (problem-oriented) data processing skeletons, we are now concentrating on skeletons in a different sense. The problem-oriented, “algorithmic” skeleton is completely implicit about its parallel implementation, while the structure-oriented skeletons to be discussed now describe nothing but the “topology”: process interaction and communication patterns. And, well understood, the latter can be used to implement the former.

To functionally describe patterns of interaction between parallel processes, we have introduced the term *structure-oriented skeletons* and want to specify them more precisely now. We will concentrate on functional specification and implementation of *topology skeletons* and show how they substantially use explicit communication, provided either by the *dynamic reply channels* in Eden, or by EDI’s explicit communication.

**Topology skeletons** define parallel schemes in which a number of parallel processes interact with each other using a fixed *communication scheme*, to produce an output to a calling parent process. The underlying communication scheme of a topology skeleton is a regular, and often recursively described, interconnection pattern, for instance, a pipeline, a torus, or a hypercube.

Processes in the network may either be all identical (or provide highly similar functionality), or we may find different functionalities, in which case several interconnected instances of each are used in one topology skeleton. Process functionality is specified as a function which maps all available inputs (from the caller and topological neighbours) to respective outputs (to parent and neighbours).

Classical parallel hardware often used topologies like  $n$ -dimensional grids or hypercubes as an internal interconnection between their compute nodes, and these process interconnection schemes are thus a well-studied area. A range of parallel algorithms [Fos95, Qui94] rely on such process topologies, and their regularity

structures the process interaction beyond completely free message-passing, which is generally a win for systematic algorithm design, implementation and debugging.

Topology skeletons for regular topologies can easily be expressed in a lazy functional language. Similar to the algorithmic skeletons, they are essentially higher-order functions which take a functional specification for the node behaviour as an argument and correctly interconnect the local computation input and output in the intended way. For example, a simple unidirectional ring can be defined in Haskell as follows:

```
ring :: ((i,[r]) -> (o,[r])) -- ring process mapping
      -> [i] -> [o]         -- input-output mapping
ring f inputs = outputs
  where (outputs, ringOuts) = foldl fld ([],ringOuts) inputs
        fld :: ([o],[r]) -> i -> ([o],[r])
        fld (outs,ringIn) inp = let (out,ringOut) = f (inp,ringIn)
                                in (out:outs, ringOut)
```

A set of identical ring processes is described functionally as  $(i, [r]) \mapsto (o, [r])$ : Each process receives input from the caller (type  $i$ ) and a stream from the ring predecessor (type  $[r]$ ), and produces respective output ( $o$  to the caller, stream type  $[r]$  to the successor in the ring). Each ring processes is one operation inside a list fold over the list of initial inputs. The ring is closed by using the output part `ringOuts` as an initial input to the `foldl`, and ring size is determined by the length of the `input`.

The master-worker scheme, which we have already used in Sec. 7.1 to create a parallel `map` implementation with dynamic load balancing (see Fig. 7.3 and 7.4), is another example of a structure-oriented, or topology, skeleton (with a simple star topology). It specifies an abstract interaction between a master process and a set of workers, where non-deterministic behaviour and  $n : 1$  communication realise dynamic load balancing. Our publication [BDLP08], as well as [Die07], are presenting more advanced variants, in which worker processes may add new tasks during runtime and use a global state in all workers. These versions rely on the same basic interaction pattern (a simple star), and they can implement more complex algorithmic schemes beyond the simple `map` implementation.

Topology skeletons provide means to structure and systemise communication of parallel processes, and are therefore a useful tool for parallel algorithm implementation. Realistic and efficient implementations of topology skeletons will use the explicit communication mechanisms provided by both Eden and EDI, for instance, to make the ring processes communicate directly (our publication [BL05a] quantifies the message reduction for rings and toroids). Additionally, *recursion* can be used to specify the regular topology more elegantly and speed up the process creation. We will present selected topology skeletons (process pipeline, process ring, process toroid, and hypercube), and investigate different implementations for them using Eden and EDI, again comparing the expressiveness of

the two languages. Furthermore, we will compare recursive and non-recursive skeleton implementations and discuss the aspect of *nesting* topology skeleton.

The strength of EDI is its explicit communication control, bought at the price of excess explicit evaluation control, reduced safety, and lengthy code. Explicit communication is essential for tasks like implementing topology skeletons. This is why the explicit communication features of Eden (`new` and `parfill`) have been added to the (otherwise functional) language, to allow performance optimisation. Eden's dynamic channels are a debatable concept because, when used inappropriately, they run the risk of losing referential transparency in programs. On the other hand, they are apparently an enormous benefit, if not a requirement, in reasonably implementing topology skeletons. Unless the tasks of a parallel algorithm interact in a tree shape (the natural process topology created by the call hierarchy), programmers will surely wish for more control over inter-process communication. On the other hand, by allowing completely free communication structures between parallel processes, e.g. in the style of MPI [MPI97], programming comfort and security are abandoned. The liberty offered by free communication can lead to hard-to detect race-conditions when new programs are developed, and is opposed to the general aim of parallel functional programming: reliability, readability, and provable soundness. This is why only few parallel functional languages support arbitrary connections between their units of computation at all. Examples are the channel concept of Clean [SP99], as well as communication features of Facile [GMP89] and Concurrent ML [Rep99]. Even more general and powerful than the dynamic channels in Eden and EDI, they relax the type safety restrictions and one-to-one restrictions.

## 8.1 Process pipelines

### 8.1.1 Uniform type, implementation variants

A simple parallelising strategy is to distribute steps of a multi-step algorithm in a pipeline of processes. A pipeline is composed of stages which provide similar functionality and, in the simple case, uniform type. As a higher-order function, the pipeline skeleton takes a *list of functions* with type `[a] -> [a]` and generates a communication-optimal pipeline by interconnecting the pipeline stage processes.

```
type Pipe a = [ [a] -> [a] ] -> [a] -> [a]
```

Creating pipelines with heterogeneous stage types needs greater effort in the strongly typed Haskell language, and we will come back to this shortly.

Please note that the given type is slightly more general than one might expect: The type we give allows a pipeline stage to produce no output or more than one output for one input. There are a number of cases where this is useful, or even required, and this would not be possible with pipeline stage specifications of type `a -> a` (which is the straightforward conception of a pipeline).

**History.** In the early days of the Eden language, Galán et al., in [GPP96], defined pipelines by folding a homogeneous process abstraction list with a suitable process composition operator. The respective code is shown in Fig. 8.1 (we changed the ( $\gg$ ) operator from the original paper to ( $\gg->$ ), since its form conflicts with a standard monad operation). This definition takes a list of process abstractions of uniform type and creates a process abstraction which will unfold a pipeline with the intended semantics – however this is inefficient, since all pipeline processes are connected through their respective parent! And another, less obvious, drawback is causing a lot more parallelisation overhead. Please note that the composition of two processes by ( $\gg->$ ) yields a (composite) *process*, not a function. Consequently, the created pipeline has a spine of unnecessary intermediate “detour” processes (see depicted communication structure on the right of Fig. 8.1). This drawback is, however, relative when classified in the broader development context of Eden: Extended static analysis (and a suitable runtime support) was supposed to ‘shortcut’ connections over intermediate processes, to send data directly to the consuming process [KPS99, PPRS00].

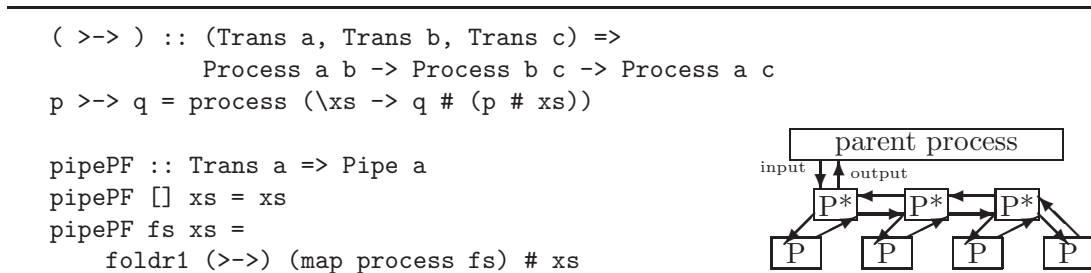


Figure 8.1: Anno 1997: Pipeline by a fold

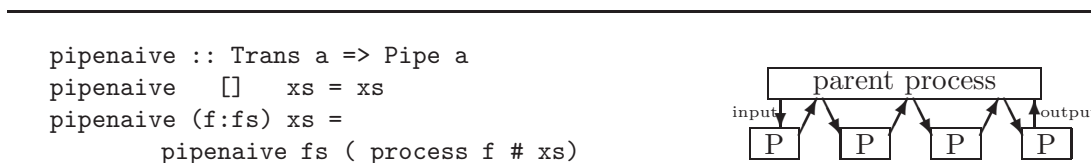


Figure 8.2: Naïve tail-recursive pipeline

**Recursive creation, variants.** A slightly better, and more intuitive, way to create a pipeline is to use *recursion* in the skeleton. However, a naïve tail-recursive scheme for the pipeline stages creates a purely hierarchical communication structure as well, as shown in Fig. 8.2: All pipeline stages created by `pipenaive` send back their results to the caller, which forwards them to the next stage. Figure 8.3 (`pipeR`) shows how inner recursion in the process abstraction leads to a direct connection. Each process creates its pipeline successor and forwards data directly to it as an input. But still, the final *result* will flow back through all pipeline stages before reaching the caller. In order to establish direct communication between

```

pipeR :: Trans a => Pipe a
pipeR [] vals = vals
pipeR ps vals = (process (generatePipe ps)) # vals
generatePipe [p] vals = p vals
generatePipe (p:ps) vals =
    (process (generatePipe ps)) # (p vals)

```

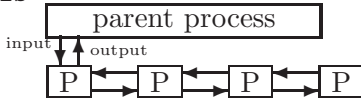


Figure 8.3: Pipeline by inner recursion

```

pipeC :: Trans a => Pipe a
pipeC [] vals = vals
pipeC ps vals = new (\chan res ->
    (process (generatePipeC ps chan)) # vals 'seq' res)
generatePipeC [f] c vals =
    parfill c (f vals) ()
generatePipeC (f:fs) c vals =
    (process (generatePipeC fs c)) # (f vals)

```

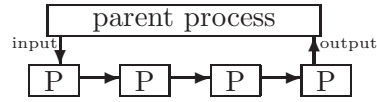


Figure 8.4: Pipeline with dynamic reply channel

the last pipeline stage and the caller, Eden's dynamic channels must be used, as shown in `pipeC` (Fig. 8.4, also presented in [PRS01]). In this version, each process creates its successor in the pipeline recursively, but the results are sent back to the first process as a side effect, via the dynamic channel `c` which is embedded in the process abstraction.

**Single-source versions.** The pipeline can also be created directly from the parent, which then collects and distributes dynamic channels appropriately to connect the stages. This version requires additional demand control, and more communication, since channel names are communicated twice instead of being embedded in the process abstraction.

```

pipeParent :: Trans a => [[a]->[a]] -> [a] -> [a]
pipeParent [] xs = xs
pipeParent fs xs = new (\c_res res ->
    let clistL = [ createProcess (chproc f) # c
                  | (c,f) <- zip (c_res:(map deLift clistL) (reverse fs))]
              'using' seqList rwhnf
    in clistL 'seq' parfill (deLift (last clistL)) xs res)
-- helper function: "channel process"
chproc :: (Trans a, Trans b) => (a -> b) -> Process (ChanName b) (ChanName a)
chproc f = process ch_f
    where ch_f c_out = new (\c_inp inp -> parfill c_out (f inp) c_inp)

```

Figure 8.5: Eden pipeline skeleton where caller creates all processes

```

ediPipeFold :: (NFData a) =>
    [[a] -> [a]] -> -- all stages as a list (same type)
    [a] -> IO [a]
ediPipeFold stages input
    = do (outC, out) <- createC      -- create last out-channel
        inC <- foldM spawnWithChans outC (reverse stages)
            -- create stages right-to-left
        fork (sendNFStream inC input) -- send input to first stage
        return out -- return result (to be received)

spawnWithChans :: (NFData a, NFData b) =>
    ChanName' [b] -> -- strict in result channel
    ([a] -> [b]) -> -- the stage
    IO (ChanName' [a]) -- input channel returned
                        -- (goes to previous stage)
spawnWithChans outC stage
    = do (inCC,inC) <- createC -- create back-channel for input
        spawnProcessAt 0 (pipestage inCC outC stage)
        return inC -- return in-channel (sent back by pl stage)

pipestage :: (NFData a, NFData b) =>
    ChanName' (ChanName' [a]) -> -- input back channel
    ChanName' [b] -> -- output channel
    ([a] -> [b]) -> IO () -- functionality
pipestage inCC outC f
    = do (inC,input) <- createC -- create in-channel
        sendWith rwhnf inCC inC -- and send it back
        sendNFStream outC (f input) -- send results

```

---

Figure 8.6: Pipeline skeleton in EDI, single-source variant

---

```

ediRecPipe :: NFData a => [[a] -> [a]] -> [a] -> IO [a]
ediRecPipe [] input= return input
ediRecPipe fs input = do (inCC,inC) <- createC
                        (resC,res) <- createC
                        spawnProcessAt 0 (doPipe inCC resC (reverse fs))
                        fork (sendNFStream inC input)
                        return res

doPipe :: NFData a =>
    ChanName' (ChanName' [a]) -> ChanName' [a] -> [[a] -> [a]] -> IO ()
doPipe incc resC [f] = do (inC,input) <- createC
                        sendNF incc inC
                        sendNFStream resC (f input)
doPipe incc resC (f:fs) = do (myInC,myIn) <- createC
                        spawnProcessAt 0 (doPipe incc myInC fs)
                        sendNFStream resC (f myIn)

```

---

Figure 8.7: Pipeline skeleton in EDI, recursive variant

---

**An EDI version of the pipeline**, shown in Fig. 8.6, can proceed exactly in the same way and saves some of the Eden process creation overhead: As shown previously, the first action of a newly instantiated process is to send an input channel to its parent. In the fold operation which unfolds the pipeline, this channel is directly forwarded to the preceding pipeline stage instead of being used to send yet another channel as the parent input.

EDI can also use recursion to unfold the process pipeline. The code for the recursive version (shown in Fig. 8.7) looks very similar to the recursive Eden version with a channel at first sight. However, the pipeline is unfolded the other way round: The first process spawned by `doPipe` applies the *last* function.

### 8.1.2 Heterogeneous pipeline stages, and including I/O

An interesting extension – easily implemented in EDI – is to allow side-effecting pipeline stages. The processes can, for instance, be explicitly placed on particular PEs, and perform some I/O action that depends on local resources at this PE.

This behaviour can be integrated in the EDI versions without problems, by small modifications in the helper functions. Most importantly, the pipeline stage is naturally a stream-processing unit and must be enabled to perform repeated I/O actions each time a stream element is processed. To achieve this, we need to drop the generalisation we have made before, and to suppose a 1 : 1 correspondence between input and output stream elements. Triggering the I/O actions is then possible by a `mapM` on the input stream inside the pipeline stage.

---

```

pipestageIO :: (NFData a, NFData b) =>
  ChanName' (ChanName' [a]) -> -- input channel, first stage
  ChanName' [b] ->           -- output channel
  (a -> IO b) -> IO ()      -- functionality
pipestageIO inCC outC f
  = do (inC,input) <- createC
      sendWith rwhnf inCC inC -- send input reply channel
      connectToPort outC    -- process I/O stream, using ParPrim directly,
      mapM_ sendIONF input  -- triggering one action per input element
      sendData Data []      -- close stream when finished
  where sendIONF x = do b <- f x -- I/O action
                    rnf b 'seq' sendData Stream b

```

---

Figure 8.8: Helper functions for pipeline including I/O

Another issue which can be expected in most cases where such an I/O-performing skeleton is useful, is that the pipeline stages will not be of uniform type. In this case, the pipeline cannot be constructed recursively from a function list, because the list cannot contain functions of different types. It is possible to define skeletons for pipelines with a fixed number of stages:

```

pipe2IO :: (NFData a, NFData b, NFData c) =>
  (a -> IO b) -> (b -> IO c) -> [a] -> IO [c]
pipe2IO st1 st2 input
  = do (outC,out) <- createC
      fork ( -- separate thread to spawn the pipeline stages:
            do (in2CC,in2C) <- createC
                spawnProcessAt 0 (pipestageIO in2CC outC st2)
                (inCC,inC) <- createC
                -- will block until in2C available from stage 2
                spawnProcessAt 0 (pipestageIO inCC in2C st1)
                sendNFStream inC input)
      return out

pipe3IO :: (NFData a, NFData b, NFData c, NFData d) =>
  (a -> IO b) -> (b -> IO c) -> (c -> IO d) ->
  [a] -> IO [d]
pipe3IO st1 st2 st3 input
  = do (outC,out) <- createC
      fork ( do (in3CC,in3C) <- createC
                spawnProcessAt 0 (pipestageIO in3CC outC st3) -- (1)
                (in2CC,in2C) <- createC
                -- and block in in3C, then
                spawnProcessAt 0 (pipestageIO in2CC in3C st2) -- (2)
                (inCC,inC) <- createC
                -- and block on in2C, then
                spawnProcessAt 0 (pipestageIO inCC in2C st1) -- (3)
                -- and block on inC, then
                sendNFStream inC input -- (4)
            )
      return out
-- ...etc

```

---

Figure 8.9: EDI pipeline skeletons, fixed no. of stages

These versions follow one and the same pattern of execution: After creating a channel for the final output, a separate thread is forked to create and interconnect all pipeline stages (in reverse order) and then send input, and the caller returns the result placeholder. Apart from the separate thread which spawns the stages, this is the same monadic `foldM` we have seen before, with channel creation and process instantiation (`spawnProcess`) inside the fold function. For instance, in the version for three stages, the forked thread will successively:

1. spawn stage 3,
2. block on `in3C`, until it is received from stage 3, then spawn stage 2,
3. block on `in2C`, until it is received from stage 2, then spawn stage 1,
4. send input to stage 1 (after receiving `inC`).



The types for different stages do not match, and we cannot pass a list of stages. Skeleton code for a known number of stages can, however, be generated *automatically*, using Template Haskell [SJ02]. The full code is shown in Appendix B.2.2; Fig. 8.10 illustrates the concepts of systematically generating a lambda-abstraction for the entire skeleton, essentially the forked thread which spawns and links all stages.

---

<b>Code generation for <math>n</math> pipeline stages:</b>		
1. <b>Generate new names:</b>	<i>outCN</i>	name of result channel
	<i>outN</i>	name of result placeholder
(for $k \in \{1..n\}$ )	<i>st<sub>k</sub></i>	pipeline stage function names
	<i>inN</i>	name of input
2. <b>Generate code:</b>		
a) creating result channel	<i>chanCreate</i>	<code>(outCN,outN) &lt;- createC</code>
b) Helper function <code>spawnCode</code> for each pipeline stage $k$ (parameters <i>chan</i> , <i>st<sub>k</sub></i> , returns code and placeholder name <i>r</i> )	<i>spawn(k)</i>	<code>(rC,r) &lt;- createC spawnProcessAt 0     (pipestageIO rC chan st<sub>k</sub>)</code>
Sequence of these alternating <code>createC</code> , <code>spawnProcessAt</code> calls created by a list fold over the (reversed) list of stage names. Empty code and <i>outCN</i> as initial values.		
c) sending input	<i>sendIn</i>	<code>sendNFStream inCN inN</code>
d) returning result	<i>ret</i>	<code>return outN</code>
3. <b>Build Lambda-Abstr.:</b>		
	$\lambda st_1 \dots st_n inN \mapsto$	
		do <i>chanCreate</i>
		fork (do <i>spawn(n)</i>
		...
		<i>spawn(1)</i>
		<i>sendIn</i> )
		<i>ret</i>

---

Figure 8.10: Template Haskell scheme to create pipeline skeleton

At first, names for all pipeline stages and the input data are generated, and will become bound variables in a generated lambda expression. The first generated statement (`createC`) on the right-hand side creates a new channel to receive the final result from the last pipeline stage (return value is the created placeholder). The helper function `spawnCode` takes a channel name *c* and a function name *f* (both just names created in Template Haskell). Code is generated which creates a new channel (to receive input) and spawns a `pipestageIO` (see Fig. 8.8). The helper function then returns the newly created placeholder name and the code. Starting with an empty statement list and the result channel, a list fold over the (reversed) list of stage names creates a sequence of these alternating `createC`, `spawnProcessAt` statements, where all statements are concatenated and each newly created channel is used in the next call.

After spawning all pipeline stages, a statement to send the input to the first stage (the last channel returned by the fold operation) is added. The resulting statement list is executed in a forked thread, as in the two fixed-stage versions shown in Fig. 8.9. Using this Template Haskell function, code for pipelines of any length can be created on demand upon compilation<sup>1</sup>, e.g.

```
$(mkPipe 6) f_1 f_2 f_3 f_4 f_5 f_6 input
```

## 8.2 Process ring skeletons

A range of parallel algorithms are structured in such a way that all processes communicate in a logical ring, for instance to circulate global data piecewise. As an introductory example, we have already shown a sequential ring skeleton on page 112, and mentioned that explicit channel communication is needed to interconnect the ring processes directly. Indeed, a straightforward parallelisation (using process instantiation instead of application for the node function) creates an unwanted communication structure, as depicted here.

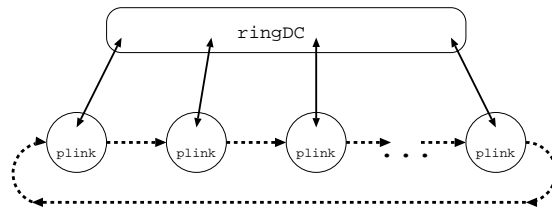
```
badRing f inputs = outputs
  where (outputs, ringOuts) = foldl' fld ([], ringOuts) inputs
  --   fld :: (Trans i, Trans o) => ([o],[r]) -> i -> ([o],[r])
        fld (outs,rIn) inp = let (out,rOut) = (process f) # (inp,rIn)
                               in (out:outs, rOut)
```



Figure 8.11: Edens ring skeleton without dynamic channels

**Implementation with channel communication** Figure 8.12 shows the definition of a ring skeleton in Eden which uses dynamic channels. All processes are created by the process evaluating the function `ring` and communicate in a unidirectional way. Additionally, we propose a more convenient interface for the ring skeleton: The number of ring processes is given by the first parameter. Parameter functions `split` and `combine` specify how to distribute the input to the ring processes and how to combine their subresults to yield an overall result.

<sup>1</sup>The \$ sign in Template Haskell stands for a function call at compile time, which produces code to splice into the program. In the example, `mkPipe` is called with argument 6.



```

ring :: (Trans ri,Trans ro,Trans r) =>
  Int          -- ring size
-> (Int -> i -> [ri]) -- input split function
-> ([ro] -> o)      -- output combine function
-> ((ri,[r]) -> (ro,[r]))-- ring process mapping
-> i -> o          -- input-output mapping
ring n split combine f input = combine toParent
  where
    (toParent,ringOuts) = unzip [plink f # inp | inp <- nodeInputs]
    inputs              = split n input
    nodeInputs          = mzip inputs ringIns
    ringIns              = leftRotate ringOuts
    leftRotate xs       = tail xs ++ [head xs]

plink :: (Trans ri,Trans ro,Trans r) =>
  ((ri,[r]) -> (ro,[r])) -> Process (ri,ChanName [r]) (ro,ChanName [r])
plink f = process fun_link
  where fun_link (fromParent,nextChan) = new (\ prevChan prev ->
    let (toParent,next) = f (fromParent,prev)
        in parfill nextChan next (toParent,prevChan))

```

Figure 8.12: Eden Ring Skeleton

Node function `f` determines the behaviour of each ring process. It is applied to the corresponding part of the `input` and the stream received from its ring predecessor, yielding an element of the list `toParent` which is part of the overall result, and a stream sent to its ring successor. Note that the ring is closed by using the list of ring outputs `ringOuts` rotated by one position (by `leftRotate`) as inputs `ringIns` in the node function applications.

The function `plink` establishes direct channel connections between the ring processes. It embeds the node function `f` into a process which creates a new input channel `prevChan` that is passed to the neighbour ring process via the parent. The ring output `next` is sent via the received channel `nextChan`, while the ring input `prev` is received via its newly created input channel `prevChan`. The ring input/output from/to the parent is received and sent on static channel connections while communication between ring processes occurs on dynamic reply channels. As all processes are created by a single parent process, the default round-robin placement policy of Eden is sufficient to guarantee an even distribution of processes on processors.

Laziness is essential in this example - a corresponding definition is not possible in an eager language. The second component of the ring node input is recursively defined from the list of output, and therefore not present when the processes are created. We have to use `mzip`, a variant of Haskell's `zip` function (converting a list of pairs into a pair of lists) which uses a *lazy* pattern to match the second argument.

**Analysis.** We can exactly quantify the amount of messages saved by using the skeleton version with dynamic channels (also see [BL05a]). In general, a process instantiation needs one system message from the parent for process creation. *Tuple inputs and outputs* of a process are evaluated componentwise by independent concurrent threads. Communicating input channels (destination of input data `ri` from the parent) requires  $tsize(ri) + 1$  administrative messages from the child, where  $tsize(a)$  is the number of top level tuple components for a tuple type `a`, and 1 otherwise, and "+1" accounts for the closing message.

Let  $n$  denote the ring size,  $i_k$  and  $o_k$  be the number of input and output items for process  $k$ , and  $r_k$  the amount of data items which process  $k$  passes to its neighbour in the ring. Input data for the ring process is a pair, thus  $3 = tsize((ri, [r])) + 1$  administrative messages from each ring process install the static channels. In case of the ring without dynamic channels, the total number of messages is:

$$Total_{noDC} = \underbrace{\sum_{k=1}^n (1 + i_k + r_k)}_{\text{sent by parent}} + \sum_{k=1}^n \underbrace{(3 + o_k + r_k)}_{\text{sent by child k}}$$

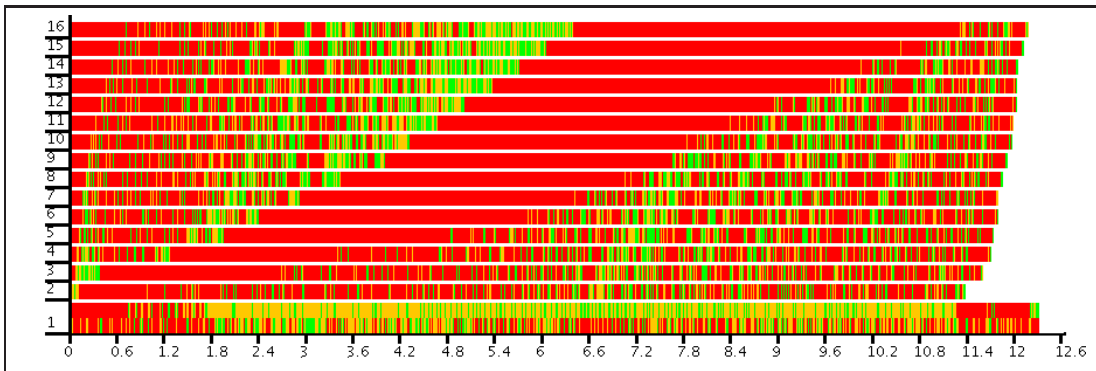
As seen in Fig. 8.11, ring data is communicated twice, via the parent. Thus the parent either sends or receives every message counted here!

Using dynamic channels, each ring process communicates one channel name via the parent (2 messages) and communicates directly afterwards:

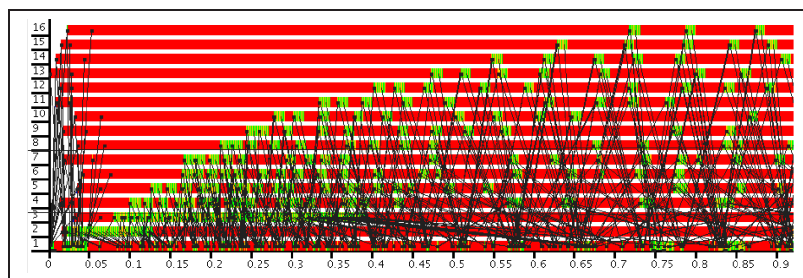
$$Total_{DC} = \underbrace{\sum_{k=1}^n (1 + i_k + 2)}_{\text{sent by parent}} + \sum_{k=1}^n \underbrace{(3 + o_k + 2 + r_k)}_{\text{sent by child k}}$$

It follows that using dynamic channels saves  $(\sum_{k=1}^n r_k) - 4n$  messages, and we avoid the communication bottleneck in the parent process.

**Traces: Impact of dynamic channels.** As an example for a ring-structured algorithm, we again use the parallel Warshall's algorithm which we have described in Section 5.4. The trace visualisations of Fig. 8.13 and 8.14 show the *Processes per Machine* view of EdenTV for an execution of the Warshall program on 16 processors of a Beowulf cluster, with an input graph of 500 nodes. The dynamic channel version uses about 50% of the messages of the static version (8676 instead

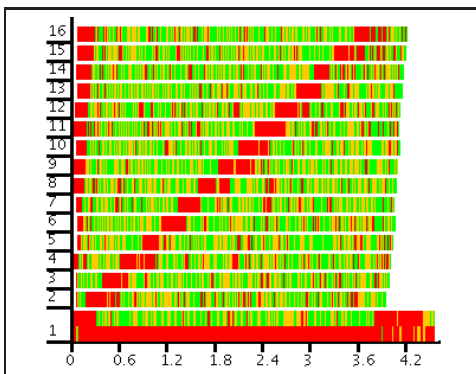


Runtime: 12.33 sec.

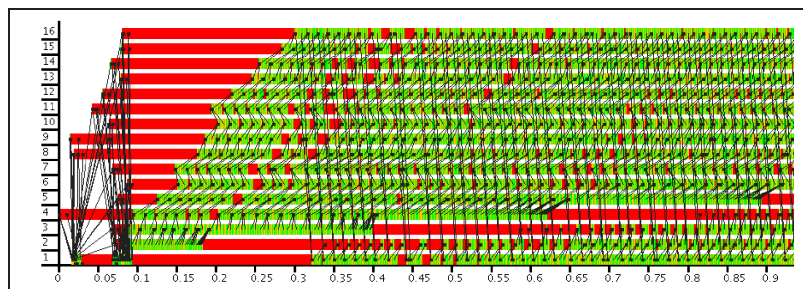


Zoom of the initial second with message traffic (black arrows)

Figure 8.13: Warshall's algorithm (500 nodes) using **static connections** in ring



Runtime: 4.56 sec.



Zoom of the initial second with message traffic (black arrows)

Figure 8.14: Warshall's algorithm (500 nodes) using **dynamic channels** in ring

Eden-5 Beowulf Cluster Heriot-Watt  
University, Edinburgh, 16 machines  
(Intel P4-SMP@3GHz, 512MB RAM, Fast  
Ethernet)

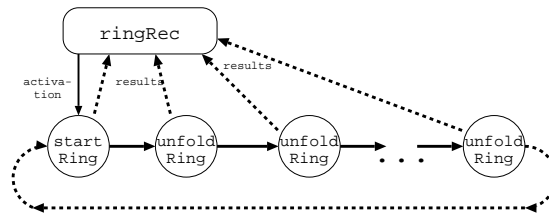
of 16629) – network traffic is considerably reduced. The figures also show zooms of the initial second of both traces, with messages between processes drawn as lines between the horizontal bars. The static version shows the massive bottleneck on the main machine (bottom bar): Worker processes often block waiting for data. The trace of the dynamic version nicely shows the intended ring structure and far less blocked phases.

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

## Recursively unfolding rings

Figure 8.15 shows an alternative definition of the ring skeleton, which uses recursion to unfold the ring. The explicit demand on the unit value `plist` by `plist 'seq'` leads to immediate creation of the ring processes when the ring skeleton is called. The first process evaluates the `startRing` function. It creates a dynamic reply channel which is passed through the sequence of ring processes and will be used by the last process to close the ring connection. It is assumed that the number of ring processes is at least two. Thus, the functions `startRing` and `unfoldRing` are never called with an empty input list. The initial input to the ring processes is now passed as a parameter and thus will be communicated together with the process instantiation. As it is passed through the sequence of ring processes, each ring process takes (and evaluates) its part of the input and passes the remaining list to its successor process. The *static* output of the ring processes is merely the unit value `()`, and their real output is returned to the originator process via initially created dynamic reply channels `pChans`, which are communicated to the ring processes.

As we see, the roles of static and dynamic channel connections are interchanged in the two ring skeleton versions. The previously static output connections to the parent are now modelled by dynamic reply channels, while the previously dynamic ring connections can now be realised as static connections, except that the connection from the last to the first ring process is still implemented by a dynamic reply channel.



```

ringRec n split combine f input = plist 'seq' combine toParent
  where (pChans, toParent) = createChans n -- result channels
        plist = (process (startRing f (split n input))) # pChans

startRing :: (Trans ri, Trans ro, Trans r) =>
  ((ri,[r]) -> (ro,[r])) -> [ri] -> [ChanName ro] -> ()
startRing f (i:is) (c:cs)
  = new (\ firstChan firstIns -> -- channel to close the ring
        let (result,ringOut) = f (i,firstIns)
            recCall    = unfoldRing firstChan f is
            next      = (process recCall) # (cs,ringOut)
        in parfill c result next )

unfoldRing :: (Trans ri, Trans ro, Trans r) =>
  ChanName [r] -> ((ri,[r]) -> (ro,[r])) -> [ri] ->
  ([ChanName ro],[r]) -> ()
unfoldRing firstChan f (i:is) ((c:cs),ringIn) = parfill c result next
  where (result, ringOut) = f (i,ringIn)
        recCall          = unfoldRing firstChan f is
        next | null is   = parfill firstChan ringOut ()
              | otherwise = (process recCall) # (cs,ringOut)

createChans :: Trans a => Int -> ([ChanName a],[a])
createChans 0 = ([],[a])
createChans n = new (\chX valX -> let (cs,xs) = createChans (n-1)
                                   in (chX:cs,valX:xs))

```

Figure 8.15: Recursively unfolding ring skeleton

## Other variants

**EDI versions.** Analogous ring skeletons may also be specified at a lower level in EDI. Appendix B.2.3 contains the respective code, which is straightforward. The ring example shows once more how Eden and EDI are interchangeable and comparable in performance. There are, however, situations where Eden's implicit concurrency and eagerness lead to unwanted behaviour, and the source code usually does not clearly indicate the errors – which we will exemplarily illustrate for the ring skeletons. While the skeleton description is coherent at first sight, some questions may arise when using it in different settings. The given type restricts the ring communication to a stream. This is a sensible restriction since,

with a non-stream type, the ring necessarily degenerates to a pipeline, or simply deadlocks. Likewise, the recursive Eden version shows the case where the initial input (of type `a`) with the ring processes is static and thus embeddable into the process abstraction.

A more subtle detail can lead to problems when the general ring skeleton is used in a special context: If the initial ring process input (or output) happens to be a tuple, the programmer might expect that each component will be evaluated and sent concurrently, as usual in Eden. However, all our ring implementations add additional parameters to the input, in order to exchange channels to the ring neighbours prior to computation. The ring process abstraction in the single-source version internally is of type `Process (a,ChanName [r]) (b,ChanName [r])` and, thus, does *not* use concurrency for components of their external input and output – the ring will immediately deadlock if the components of type `a` expose non-local data dependencies.<sup>2</sup> Different implementations, specialised to avoid this problem, are possible, but the difficulty is to find out the reason for the deadlock. Neither the calling program, nor the skeleton source code will clearly indicate the problem; it will remain hidden in the overloaded communication inside the Eden module. And the other way round, any EDI version of the skeleton will have the drawback that the output to the caller is never sent as a stream, unless a special “stream” version is used (already mentioned and discussed for the `map` skeletons).

**Ring definition by pipeline skeletons.** A simple idea for a ring skeleton is to use a pipeline skeleton and a back-reference from output to input as a

---

<sup>2</sup>The constraint is even stronger for the recursive versions: Parent input is embedded into the process abstraction and has to be available upon process creation.

---

```

closePipe :: (Trans i,Trans a,Trans o) =>
  ([ [a]->[a] ] -> [a] -> [a]) -> -- a pipeline skeleton
  ((i,[a]) -> (o,[a])) -> [i] -> [o]
  -- resulting ring skeleton (inner interface)
closePipe pipeSkel ringF ringIns
  = let rComm = pipeSkel ringNodes rComm          -- pipeline
      ringNodes = zipWith (pipeRingNode ringF) ringIns rOutCs
      (rOutCs,rOuts) = createChans (length ringIns) -- result channels
      in rnf rOutCs 'seq' rComm 'seq' rOuts -- force channels,activate system

pipeRingNode :: (Trans i,Trans a,Trans o) =>
  ((i,[a]) -> (o,[a])) -> i -> ChanName o -> [a] -> [a]
pipeRingNode ringF rIn rOutC rCommIn
  = let (rOut,rCommOut) = ringF (rIn,rCommIn) -- apply ring function
      in parfill rOutC rOut rCommOut -- concurrently send parent output

```

---

Figure 8.16: Function to derive a ring skeleton from a pipeline skeleton



ring skeleton. Yet, as we have discussed earlier (in conjunction with `mzip`), this back-reference might as well lead to deadlock and render ring creation impossible, depending on the pipeline implementation. For instance, the single-source variant in Fig. 8.5 cannot be used, while the recursive pipeline skeletons work allright.

Figure 8.16 shows the code to (generally) derive a ring skeleton from a pipeline skeleton. Please note that the applied technique is very similar to the one in the recursive version before: The pipeline is closed to a ring and used for ring communication, and output to the parent is sent as a side effect, via previously created channels. In addition to this output channel, the whole node input from the parent is embedded in the process abstraction. With the code shown here, ring communication takes a detour via the caller (which feeds pipeline output back into the pipeline). Another variant is possible, where the caller itself is one stage of the created pipeline. We do not digress further; Appendix B.2.4 contains the full code.

### Speedup experiments

Experiments with application programs using Eden and EDI ring skeletons show that the recursive ring creation is slightly advantageous as the number of ring processes increases. Fig. 8.17 shows speedups for the Warshall example program using the two Eden and EDI ring skeletons. We also tested two versions derived from pipeline skeletons (`pipeC` and the recursive EDI version). For a small number of processes, using recursion has almost no impact on performance. The number of messages sent and received by the parent process is slightly reduced while the overall amount of messages remains almost the same.

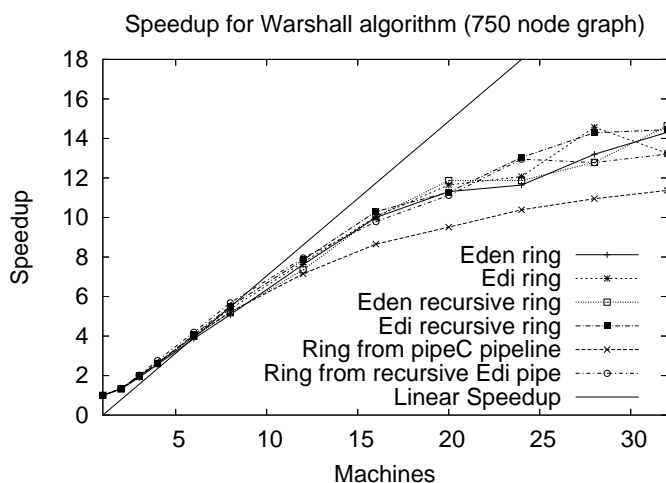


Figure 8.17: Speedup of Warshall program (750 nodes) using different ring skeletons (Speedups based on runtime 76.75sec for the sequential ring from the introduction)

Altogether, speedup differences between the different version versions are mi-

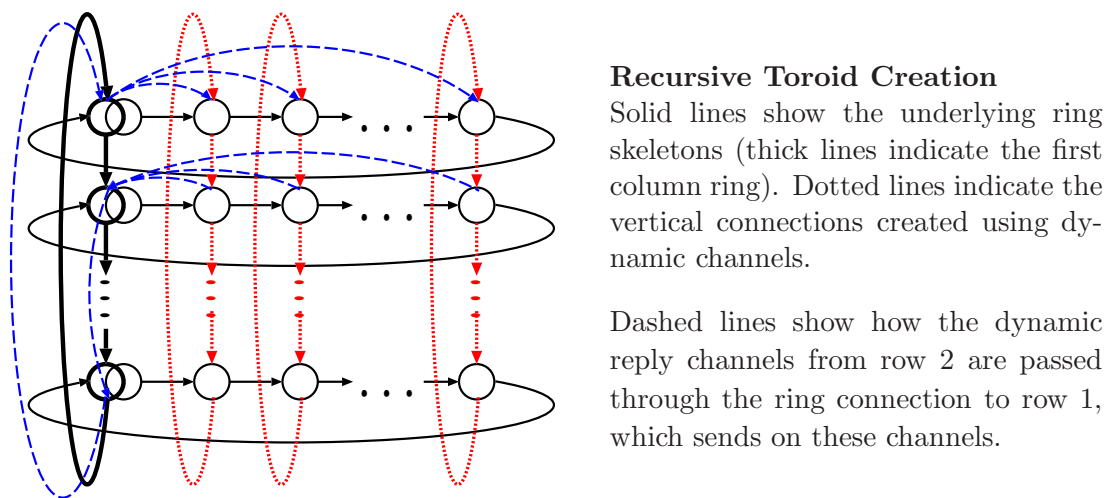


Figure 8.18: Creation scheme of a torus topology using ring skeletons

nor. Interestingly, the version derived from `pipeC` shows much worse performance than the one using the recursive EDI pipeline. This difference in speedup can be attributed to the fact that the EDI version saves process creation overhead (as no output communication is necessary).

## 8.3 Nesting skeletons

### 8.3.1 A toroid topology created as a nested ring

As we have seen previously, process topologies can often be unfolded recursively, whereby the process creation overhead is distributed over several nodes. While it is less relevant for one-dimensional topologies, pipelines and rings discussed before, the distributed startup is a performance factor for higher dimensions. For instance, the Eden toroid skeleton defined in [LOP<sup>+</sup>03] creates all processes by a single process and establishes dynamic interconnection channels between them. This single-source creation of process systems may lead to a serious bottleneck in the creator process when the number of processes increases.

We have presented and discussed recursive process creation for rings and toroids in our work [BL05b], from which we will now present and discuss toroid skeleton definitions which unfold by means of recursive ring skeletons, since a toroid is nothing but a two-dimensional grid with ring connections in each dimension. Figure 8.18 depicts the generation scheme for the torus topology used in our definitions. The first column and all rows are created as unidirectional rings. The other column rings must be installed using dynamic channels.

Figure 8.19 shows the core of the recursively unfolding torus skeleton, following [BL05b]. The `toroidRec` function describes the toroid by its dimensions (number of rows and columns) and the functionality of each node. In order to

---

```

toroideRec :: (Trans input, Trans output, Trans horiz, Trans vert) =>
  Int -> Int -> -- dimensions
  ((input,[horiz],[vert]) -> (output,[horiz],[vert])) -> -- node function
  [[input]] -> [[output]] -- resulting mapping
toroideRec dim1 dim2 f rows
  = rnf outChans 'seq' start_it 'seq' -- force channel & ring creation
    list2matrix dim2 outs -- re-structure output
  where (outChans,outs) = createChans (dim1*dim2)
        ringInput = (list2matrix dim2 outChans, rows)
        -- creating first column ring
        start_it = ringP dim1 dim2 (\_ -> uncurry zip ) spine
                  (gridRow dim1 dim2 f) ringInput

-- ring function for 1st column ring
gridRow :: (Trans i, Trans o, Trans h, Trans v) =>
  Int -> Int -> -- dimensions
  ((i,[h],[v]) -> (o,[h],[v])) -> -- node function
  ([[ChanName o], [i]), [[ChanName [v]]]) -> (((), ([[ChanName [v]]]))
gridRow dim1 dim2 f ((ocs, row), allnextRowChans) =
  let (cChanNamevs, rowChans) = createChans dim2
      -- creating row ring
      start = startRingDI staticIn (gridNode f) dummyCs mynextRowChans
      staticIn = mzip3 row ocs cChanNamevs
      mynextRowChans = allnextRowChans!!(dim1-2)
      (dummyCs, _ ) = createChans dim2
  in rnf cChanNamevs 'seq' rnf dummyCs 'seq' start 'seq'
    ((), rowChans:take (dim1-2) allnextRowChans)

-- ring function for row rings
gridNode :: (Trans i, Trans o, Trans h, Trans v) =>
  ((i,[h],[v]) -> (o,[h],[v])) ->
  ((i,ChanName o, ChanName (ChanName [v])),ChanName [v],[h]) -> (((),[h])
gridNode f ((a,cResult,cv),cToBottom,fromLeft) =
  new ( \ cFromAbove fromAbove ->
    let (out,toRight,toBottom) = f (a,fromLeft,fromAbove)
    in parfill cv cFromAbove -- send vertical input channel
      (parfill cResult out -- send result for parent
        (parfill cToBottom toBottom -- send data on column ring
          ((), toRight))) -- result and data on row ring

```

---

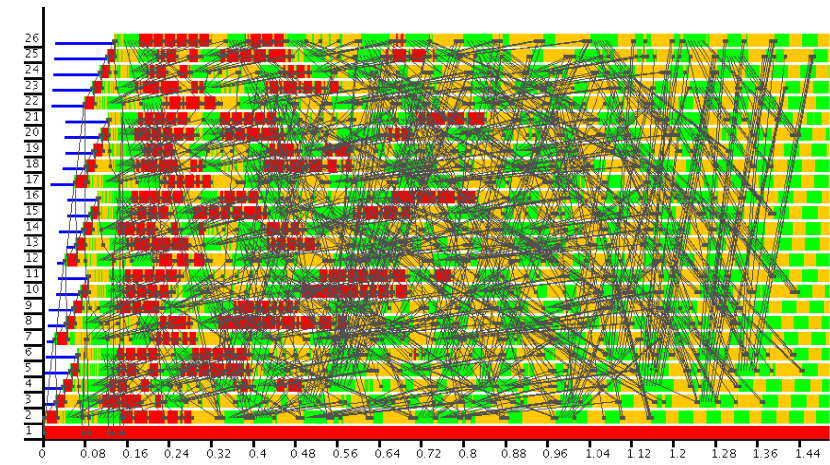
Figure 8.19: Core of recursively unfolding toroid skeleton

place all processes on different processor elements, the first column of the torus structure is created with a variant `ringP` of the recursively unfolding ring skeleton, which allows for placing ring processes with a constant stride. To place processes row by row, the first column is placed with stride `dim2`, i.e. the length of the rows.

The ring function `gridRow` for the first column ring creates a ring for each row. Instead of using the normal interface of the ring skeleton, we use the internal `startRing` function because we want to embed the column processes into the row

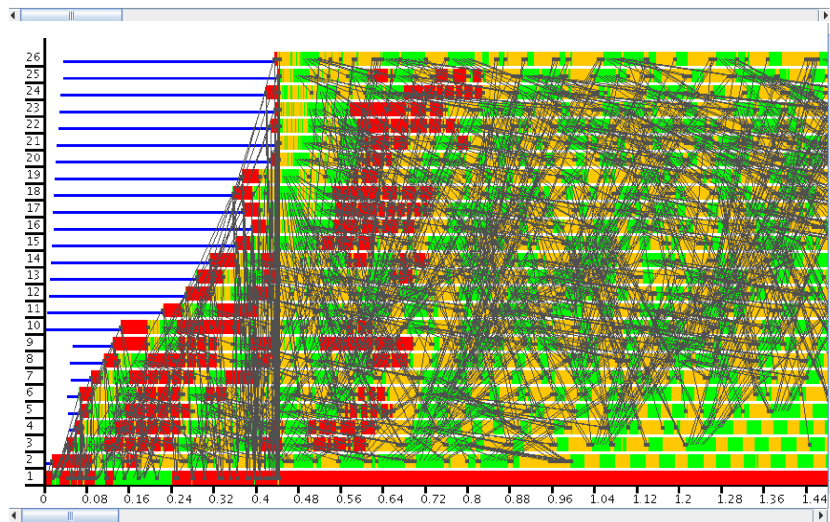
Multiplication  
of dense ran-  
dom  $1000 \times 1000$   
matrices with  
**Recursive Toroid**

Overall Runtime:  
 $12.1sec.$



Multiplication  
of dense ran-  
dom  $1000 \times 1000$   
matrices with  
**Single-Source  
Toroid**

Overall Runtime:  
 $12.4sec.$



Eden-5, Beowulf Cluster Heriot-Watt University, Edinburgh, 26 machines  
(Intel P4-SMP@3GHz, 512MB RAM, Fast Ethernet)

Figure 8.20: Start phase of matrix multiplication traces using toroid skeletons

rings. A subtlety of the inner rings is the circular dependency of their dynamic input, i.e. the dynamic channels to establish the additional column rings. It is necessary to use a variant `startRingDI` which decouples static input (which is available at instantiation time) from dynamic input (not produced until after process instantiation). Otherwise, the inner rings would immediately deadlock on process instantiation. Each row ring process returns a channel name for its vertical input, which must be collected and passed to the previous row through the first column ring (as indicated in Figure 8.18 for the second row).

Measurements with a toroid-based matrix multiplication algorithm (Cannon's algorithm, see [Qui94]) show that runtimes are slightly better for the recursive version, due to a distributed startup sequence. Figure 8.20 shows EdenTV *All*

*machines* diagrams<sup>3</sup> of the start phase, executed on 26 nodes of a Beowulf cluster using either a single-source or a recursive toroid skeleton.

While runtime is only slightly improved, the traces show the expected improvement in startup: Process creation is carried out by different processors in a hierarchical fashion in the recursive skeleton implementation. One can observe how the first column unfolds, starting at processor 2 with stride 5, and how each of these processes unrolls one row. Process creation takes about 0.15 sec. in this version, whereas the single-source version below needs 0.4 sec. until all processes start to work (explaining the difference in runtime).

The improvement in startup pays especially for skeletons with a big number of processes. In any case, it substantially reduces the network traffic. The program investigated here already includes the input matrices in the process abstraction instead of communicating these big data structures via channels (which would be more time-consuming). However, the parent process in the single-source version has to send the channel names to each toroid process, which requires 125 messages. The parent process in the recursive version only sends 2 messages – creation and input to the `startring` process of the first column ring.

### An EDI version

The toroid skeleton presented makes massive use of explicit communication and, especially here, the question arises whether a more explicit approach could be profitable. We have implemented an alternative EDI version, shown in Fig. 8.21, which equally uses nested skeletons to create the toroid structure.

This version unfolds the toroid in the same way as the one described before: The first column is created as a process ring, in which each node creates one row of processes connected in a horizontal ring. The vertical ring (first column) is created by `ediRecRingIO`, an EDI ring skeleton which returns its result in the IO monad. The horizontal rings are created using `embeddedRing`, a ring skeleton derived from a pipeline skeleton, which embeds the caller as one of the ring nodes. As mentioned, this is easily done and actually an optimisation when a ring skeleton is derived from a pipeline skeleton.

Horizontal rings are used directly for toroid communication. Vertical communication is done as a side-effect in each node, using stream channels which are created on process startup. The output of each horizontal ring node is a pair of two channels: one to receive a channel for vertical toroid communication (used by one node of the outer ring), the other to receive vertical input data for the computation (passed one step through the outer ring, then sent to the next row by its respective parent). Vertical output will then be sent from the node by a forked thread; another forked thread will send back the final results to the caller via channels explicitly created in advance. Thus, all nodes perform quite a few

---

<sup>3</sup>Every processor executes exactly one process, so we identify nodes and processes.

```

embeddedRing = embedEdiPipe (\fs xs -> unsafePerformIO (ediRecPipe fs xs))

toroideRecIO :: (NFData i, NFData o, NFData horiz, NFData vert) =>
  Int -> Int -> -- dimensions
  ((i,[horiz],[vert]) -> (o,[horiz],[vert])) -> -- node function
  [[i]] -> IO [[o]] -- resulting mapping
toroideRecIO dim1 dim2 f inRows
  = do (outCs,outs) <- createCs (dim1 * dim2) -- result channels (all nodes)
      let inputss = reverse (zipWith zip inRows (list2matrix dim2 outCs))
          ediRingRecIO dim1 dim2 (\_ -> id) spine (gridRowE f ) inputss
      return (list2matrix dim2 outs) -- unevaluated!

-- node function for vertical ring (1st column).
gridRowE :: (NFData i, NFData o, NFData horiz, NFData vert) =>
  ((i,[horiz],[vert]) -> (o,[horiz],[vert])) -> -- node function
  -- inputs,back channels for output
  ((i,ChanName' o),[ChanName' [vert] ]) ->
  ((),[ ChanName' [vert] ])
  -- no output, channels take one step in column ring
gridRowE gridF (nodeInputs,fromBelow) = unsafePerformIO $
  do let (vCCs,vDataCs) = unzip (embeddedRing (gridNodeE gridF) nodeInputs)
      fork ( sequence_ [sendNF cc c
                       | (cc,c) <- mzip vCCs fromBelow ] )
      return ((),vDataCs)

-- node function for horizontal ring
gridNodeE :: (NFData i, NFData o, NFData horiz, NFData vert) =>
  ((i,[horiz],[vert]) -> (o,[horiz],[vert])) -> -- node function
  ((i,ChanName' o),[horiz]) -> -- input,back channel for output
  ((ChanName' (ChanName' [vert]),ChanName' [vert]),[horiz])
  -- parent output: channels to send/receive vData
  -- ring node function: (i,[h]) -> ((ch(ch [v]),ch[v]),[h])
gridNodeE gridF ((input,outC),hData) = unsafePerformIO $
  do (vDataC,vData) <- createC -- to receive vertical input data
      (vCC, vChan) <- createC -- where to send vertical output data
      let (out,hOut,vOut) = gridF (input,hData,vData)
          fork (sendNF outC out) -- output not a stream...
          fork (sendNFStream vChan vOut)
      return ((vCC,vDataC),hOut)

```

Figure 8.21: EDI toroid skeleton, using ring and pipeline

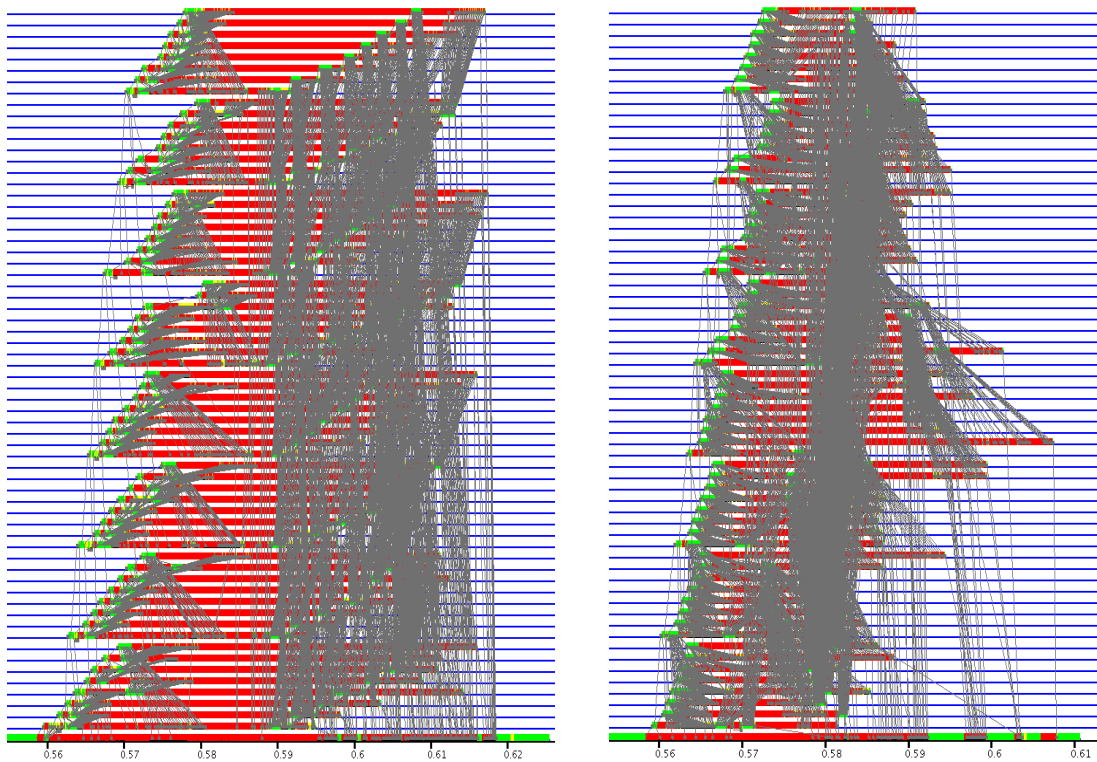
side-effects which should, in principle, be encapsulated in the IO monad. The ring skeletons, and the proposed interface, use *pure* node *functions*, which do not allow side-effects. One could equally use Eden’s side-effecting constructs `parfill` and `new` (resp. `createChans`), but this would only hide the side-effects “cosmetically”. While we abstain from presenting yet more ring skeletons for this special purpose (with monadic node function type, which is a very easy modification in any EDI version), the node functions for outer and inner ring explicitly show all

side-effects, wrapped into a single `unsafePerformIO` call. The better and cleaner solution would, of course, be a ring skeleton with monadic node function type.

Most notably, the communication inside the ring skeletons is reduced to data which is guaranteed to be available: parent input to each node, and the result channels created by the caller. These inputs are paired by the caller already, thus the input split function becomes trivial (`\_ -> id`). The dynamic input that had to be separated in the previous version is now explicitly communicated in a forked thread by the caller.

### Comparing startup and performance (toy program)

We have tested the two toroid skeletons with a “toy program”, which merely checks the correctness of the toroid communication structure. The toroid nodes send a node identification to the neighbours over the toroid connections and return the received data as a result, which the caller checks and outputs for all nodes. The traces in Fig. 8.22 show the machine view for program runs which unfold a toroid of  $8 \times 8$  nodes, including message traffic.



(a) Eden toroid skeleton: 0.065sec.

(b) EDI toroid skeleton: 0.05sec.

Figure 8.22: Toroid skeleton comparison, using a toy program  
(using 65 virtual PEs, executed on 32 physical PEs – Heriot-Watt Beowulf Cluster)

Both program runs are very short for this small toy program, only showing the differences and *overhead* of the skeleton versions and their communication, because no computation is done in the toroid nodes. As the traces show, both toroid skeletons unfold the toroid recursively as two nested rings.

Eden process instantiation implies channel exchange between parent and child, which is partially unnecessary here and left out from the EDI version. This process instantiation protocol overhead has several effects: It leads to a slightly increased total amount of messages (5,089 messages as against 4,168 for the EDI version). Additionally, as every such communication is done in a separate thread, the Eden version uses many more threads (458, as against 273 for EDI). These additional threads are also the reason why the toroid processes in the Eden version need more time to run to completion.

Well-understood, the mere startup time and the redundant messages in the Eden version will not have much impact on runtime in a real application: The “real” computation data will usually be much bigger than just a number, and the computation time of the toroid nodes will compensate for the startup overhead and post-processing phase. Performance differences between the two versions are minor and only show up in this minimal micro-benchmark.

### 8.3.2 Two versions of a parallel pipeline

Another parallel computation scheme can give rise to nesting skeletons we have presented: a pipelined computation  $((f_n \circ f_{n-1} \circ \dots \circ f_1)(x))$  applied to a large set of data  $(x \in)X$ . The pipeline computation (as the inner skeleton) can easily be parallelised by using several pipeline instances in parallel, a parallel `map` of a pipeline, as depicted in Fig. 8.23. For applications where the tasks for the pipeline (as a whole) have irregular complexity, a workpool implementation can be used as the parallel `map`, which allows for dynamic load balancing *in the large*. A new task is fed into the pipeline every time a result appears at the other end and, ideally, the prefetch may be adjusted in such a way that all pipeline stages

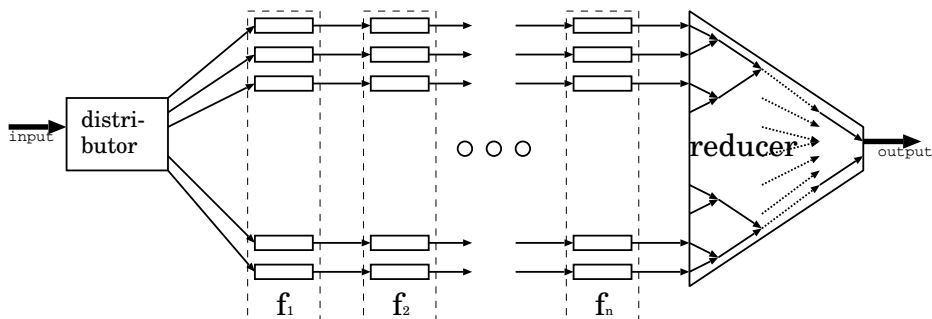


Figure 8.23: `parpipeWhole`, parallel pipeline, created as `parMap(pipe ...)`



work continuously (at least one task should be held in every stage).

On the other hand, these pipeline *stages* may have considerably varying complexity, leading to task congestion at the most complex stage. For applications of this kind, a different implementation scheme is favourable: A pipeline of parallelised map stages may be used<sup>4</sup>, as depicted in Fig. 8.24. Both implementation schemes may be combined with, and connected directly to, a (separate) reduction network (see Figures), which we leave out of discussion.

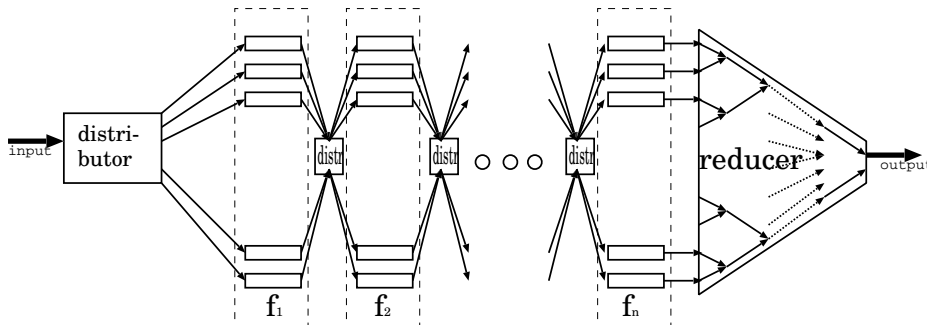


Figure 8.24: `parpipeStages`, Parallel pipeline, created as `pipe(parmap ...)`

The decision which skeleton to use strongly depends on the concrete computation and its characteristics. A parallel `map` of pipelines appears sensible for large data sets and pipeline stages of regular and comparable complexity, whereas heterogenous pipeline stage complexity requires load balancing at the inner level.

Well-understood, both computation schemes are *equivalent* if we use a parallel `map` skeleton with static task distribution. Only if we allow dynamic task distribution (and reordering the tasks and results), the second variant makes more sense. Using dynamic load balancing implies that tasks are processed out-of-order, either passing through the entire pipeline or reordered at every stage. The second variant also allows to instantiate a different number of processes for more complex pipeline stages, which is not possible in the first variant. An essential drawback of the second variant is its increased parallelism overhead. The picture is misleading: Not the *workers* are pipelined, but only the master nodes of a succession of workpool skeletons. Redistributing the tasks at every stage implies two additional communications per task and pipeline stage, which will be saved when tasks pass the entire (communication-optimised) pipeline at once. The second variant absolutely has to use additional master nodes for the master-worker skeletons at each pipeline stage (depicted as “distributors”).

However, to implement either variant requires more than the simple composition of two skeletons discussed previously. Both the `map` and the `pipe` skeletons we have presented expect worker functions which operate on a single task, to yield

<sup>4</sup>We have motivated the multi-type I/O-pipeline in 8.1.2 by possible side-effects at different stages, so a `map` skeleton which embeds I/O actions into its computation is required.

```

parpipeWhole3, parpipeStages3 :: (NFData a, NFData b, NFData c, NFData d) =>
  Int -> -- parallelism degree
  -- exemplarily: 3 stages
  ([a] -> IO [b]) -> ([b] -> IO [c]) -> ([c] -> IO [d]) ->
  [a] -> IO [d]
parpipeWhole3 n f1 f2 f3 xs = mw n (n * prefetch) -- try to fill all stages
  3 -- leave room for 2 pipeline sub-processes
  pipe xs
  where pipe xs = pipe2 f1 f2 xs >>= f3 -- last stage remains local!
  -- last stage local, 2 more pipeline stages
  prefetch = arbitrary

parpipeStages3 n f1 f2 f3 = pipe3 f1' f2' f3'
  where f1' = parallelStage f1
  f2' = parallelStage f2
  f3' = parallelStage f3
  pf = arbitrary
  parallelStage f = mw n pf 3 f -- n workers, placed on each 3rd PE
-- assuming:
-- workpool-using parmap skeleton with placement stride and list interface
mw :: (NFData a, NFData b) =>
  Int -> Int -> Int -> -- no. of proc.s, prefetch, stride
  ([a] -> IO [b]) -> [a] -> IO [b] -- list worker fct., input, output

-- (template-haskell generated) multitype pipelines with list interface
pipe2 :: (NFData a, NFData b, NFData c) =>
  ([a] -> IO [b]) -> ([b] -> IO [c]) -- 2 stages incl. I/O
  -> [a] -> IO [c]
pipe3 :: (NFData a, NFData b, NFData c, NFData d) =>
  ([a] -> IO [b]) -> ([b] -> IO [c]) -> ([c] -> IO [d]) -- 3 stages incl. I/O
  -> [a] -> IO [d]

```

---

Figure 8.25: Two parallel pipeline skeleton versions (sketch)

a resulting skeleton which processes a whole task list (and returns results in the IO monad). As we want to include one skeleton call as the worker function of the second, our desired skeleton composition requires *list* interfaces (but IO actions should, even so, be triggered upon the single inputs and not on the whole list).

Code for both variants is sketched in Fig. 8.25 (exemplarily for a 3-stage pipeline), assuming the respective changes in the skeletons to be composed. The proposed interface allows to specify a *parallelism degree*, the number of parallel pipelines or, in the second variant, the multiplicity of every stage. Processes are *placed* explicitly following this information, in order to effectively use all available machines and to not place several processes on the same machine unnecessarily.

For placement, we assume a master-worker `map` implementation which instantiates worker processes round-robin with a constant *stride* (starting with the next PE), thereby leaving PEs free for additional sub-processes. In the first variant,

if  $n$  parallel pipelines of length  $k$  are requested, each pipeline will require  $k$  PEs, thus the stride  $k$ . One of the pipeline stages has to be computed locally (we opted for the last stage); otherwise, the caller of the pipeline skeleton (each worker in the master-worker skeleton) will unnecessarily create sub-processes for *all* stages (and would need a stride of  $k + 1$ ), acting as a mere interstation for the tasks.

In the second variant, with parallelism degree  $n$  in every (master-worker parallelised) stage, the stride should equally be  $k$ . This placement logically splits up the PEs in groups (assuming a sufficient number of PEs) by the remainder of their ID divided by  $k$ , and each group interacts in master-worker fashion to compute one pipeline stage, using one master and a set of  $n$  workers.

The interface we propose assumes IO-monadic worker functions, which operate on whole lists. Nevertheless, we have to suppose that these skeletons deliver results as a *stream* of single elements, and not as one single result in the IO monad (which requires an `unsafeInterleaveIO`). Rather complicated additional requirements have to be met by the worker functions when their type is (necessarily) liberalised for this special purpose:

- The master-worker implementation `mw` must be able to handle its input of tasks as a *stream*, which may not be taken for granted (some of the nested implementations we have developed and discussed in our work [BDLP08] require the task list length to be known in advance).
- The workers of the master-worker `map` implementation `mw` are required to deliver exactly one result for each task. This is trivially fulfilled with a worker function of type `a -> b`, but becomes an additional non-trivial requirement with the list type we need.
- From the first version, a similar requirement holds for the pipeline skeleton: The stages of the (multi-type, I/O-including) pipeline need to yield *exactly one result* for each task. In Section 8.1.1, we started by pipeline skeletons with the more liberal list type; then, in Section 8.1.2, we discussed and introduced an implied 1:1 correspondence between input and output (enforced by the single-item type). For the present purpose, the more liberal list type is needed, while the implied 1:1 correspondence has to be kept, since feeding new tasks into the pipeline is triggered by result arrival (in the outer master-worker skeleton). Type checking will not discover such errors (which lead to deadlocks) any more.
- Moreover, as the pipeline stages of the presented skeleton include IO actions, these actions are supposed to happen on every single input, and not on a whole list. Result elements have to be sent as a stream, which keeps parallel data processing continuously at work. However, such worker action and communication cannot be set up by EDI communication constructs alone – the worker functions have to guarantee that they are *incremental*, i.e. may produce result prefixes from input prefixes.

These various non-trivial constraints apply when using either of the parallel pipelines obtained by skeleton nesting. A similar diagnosis holds for our first example, the toroid skeleton, where ring input had to be splitted into a static and a dynamic part to avoid a deadlock. Nesting skeletons introduces additional complexity, as it implies complex combinations of implicit skeleton properties which, while harmless in the basic versions, might render the nesting solution unusable. For the parallel pipeline, the overall judgement is that a direct, specialised implementation (using EDI or relying on dynamic channels) is favourable – however elegant and esthetic the nested variants may be. With the explicitness of EDI, no implicit side-conditions and effects complicate the implementation.

In summary, Eden and EDI are both suitable to implement efficient topology skeletons. While Eden implementations are generally shorter and more elegant, they introduce the (necessary) concurrency by implicit side-effects, and are otherwise comparable to more explicit implementations expressible in EDI. EDI is favourable for quick development of specialised versions, and easier to debug. The example of parallel pipelines shows that skeleton nesting has its limits; it either introduces various side-conditions on skeleton usage, or amounts to using special skeleton versions to work around the technical issues related to the nesting.

**Part IV**  
**Conclusion**



# Chapter 9

## Conclusions and future work

Parallel programming is hard – because of the nondeterministic nature and complexity of interaction between parallel (or concurrent) processes, but also because parallel programming models often do not provide sufficient abstraction from the low-level, machine-oriented view. With this thesis, we hope we have been able to demonstrate that parallel functional languages offer a suitable abstraction level and useful tools to capture parallelism concepts, without getting lost in technical details. On the other hand, as the presented work explicitly tackles implementation issues, we have pointed out that the desired abstraction level varies depending on the task at hand, and that parallel implementations always deal with a compromise between explicit control and abstraction.

### 9.1 Summary of contributions

- We have defined the low-level parallel functional language EDI, identifying the most basic and orthogonal control constructs needed to implement parallel coordination.

EDI provides basic data communication as well as system information about location and resources, and it enables programmers to explicitly control the evaluation degree and to create new local (concurrent) and remote (parallel) tasks. Concurrency is already included in our base platform GHC [GHC], and evaluation control is provided by evaluation strategies [THLP98]. Data communication, remote task creation, and system information, not to mention the basic parallel setup and infrastructure, have been implemented as our own work.

- In Chapter 3, we have presented in detail an implementation for the language Eden, which is structured in layers and based on the concepts identified and addressed by EDI. We pointed out the advantages of such a structured implementation.

Furthermore, general applicability and advantages of the layer concept and EDI have been shown by the design and prototype implementation of the more general ARTCOP system (using a slightly modified and extended kernel support) in Chapter 4. Our feasibility study of VSM shows that the limits of system programming using our functional approach cannot be pushed further easily.

- As a second evaluation for the EDI concepts, the entire Part III was devoted to skeleton implementations in EDI, in comparison to Eden implementations. In contrast to the implementation concepts, this investigation addresses advantages and drawbacks of EDI as a *language*.

We have discussed a range of known Eden skeleton implementations, and provided EDI versions for a comparison when applicable. Throughout the discussion of our skeleton implementations, we pointed out that EDI versions provide easier access than Eden implementations, which, at times, obfuscate runtime behaviour and potential problems. On the other hand, a number of Eden provides more programming comfort, and implementations can profit from the implicit communication modes and process interconnections. A new skeleton, Google-mapReduce, has been investigated for the first time in a really parallel functional context.

- Our comparison and analysis shows that especially *topology skeletons* (a notion which we have coined in previous publications), are the core domain of EDI and its support for explicit communication.
- The layered implementation concept and the comparative discussion of skeleton implementations substantiate our general claim that functional languages provide a suitable abstraction level for reasoning about parallel programming in an abstract manner. Crucial system properties, e.g. adaptive scheduling policies and task creation, as well as skeleton implementation details, are exposed in easily readable concise code.

## 9.2 Discussion and future work

Our implementation of the parallel functional language Eden, which constitutes the starting point, has been organised in several logical layers, and the upper parts are encoded in (concurrent) Haskell with only few and simple extensions. Starting from a low-level implementation perspective, levels of increasing abstraction and decreasing explicitness have been constructed. Aiming at a strict separation between these levels, we have obtained an implementation of Eden which is easy to maintain and accessible to rapidly prototyping modifications and extensions of its coordination constructs.



Aside from their original purpose of implementing a more abstract language, the implementation primitives help clarify the general requirements of parallel coordination, and can even directly serve as an explicit parallel functional language. However, without a protecting additional layer, the primitives (due to their pragmatic design) offer too much liberty to specify erroneous operations. The language EDI, presented in Section 3.2, constitutes a reasonable compromise, while maintaining the principle of *completely explicit* execution control and orthogonality (in contrast to the Eden language). As we have also illustrated by a simulation based on Concurrent Haskell, EDI may also be considered as extending the Concurrent Haskell programming model to distributed machines. Communication via shared (mutable) variables is replaced by transmission of (immutable) data, and thread concurrency is extended by process concurrency in a distributed machine setup. Consequently, EDI can be used as a functional language for *distributed* programming.

In the near future, we plan to extend and consolidate our Eden implementation in cooperation with GHC experts. The sequential base system of Eden, GHC, supports multicore CPUs and shared memory, but the support is specialised towards a semi-implicit programming model (similar to GpH) [HMJ05, HS07], while the current Eden implementation uses concurrent Haskell, but is limited to single OS threads. Therefore, combining the two approaches in the implementation is self-suggesting future work and has a high potential for performance gain. This will constitute the first Haskell implementation which combines cluster and multicore support.

In Chapter 4, we have described our results regarding an even more radical approach towards implementation techniques. We have investigated to which extent we can generalise the Eden implementation concept and express general concepts of parallel coordination in a functional language. In a working prototype implementation, we have managed to implement complex adaptive scheduling techniques, load-balancing, and thread management in Haskell (at system level), with the minimal kernel support assumed for EDI. Potential benefits and obstacles related to implementing another kind of parallel programming support, virtual shared memory (VSM), at system level have been investigated to sound the limits of our approach. Though we have been able to program a prototype in Haskell, the results obtained indicate that such an implementation would require substantially extended kernel support, not mentioning any consequences for performance. However, suitable abstract memory models for modern multicore architectures are a crucial component of a generalised runtime environment like the projected ARTCOP. Aiming to exploit its potential for modeling and formal reasoning about implementation properties, we plan to continue our collaborative ARTCOP research [HL08], which targets abstract process and memory models as well as a flexible implementation for managing parallel computations, based on an architectural machine model and implemented largely at a functional level, as presented in Chapter 4. A first stepping stone, and a separable task, is to

investigate, develop, and test the idea of applying the two-stage task creation of GpH to a distributed memory setup, outlined in Sec. 4.7.4.

Aside from the realisation of parallel Haskell dialects in a GHC setting, the work presented also provides an even more general insight: the implementation concept can be generalised to *arbitrary languages and systems*. e.g. by replacing Haskell libraries with something else. The general requirements addressed by EDI (local and remote task creation, data transfer, evaluation control, and system information) do not depend on the computation language in use, nor do they fix the programming model of implemented languages.

Part III of the thesis has investigated parallel skeletons as a testbed for assessing usage, advantages and drawbacks of EDI and Eden in comparison. Non-apparent effects of the implementation language and implementation variants have been illustrated by a range of different implementations for data parallel transformations. Well-known implementation concepts in Eden have been contrasted to EDI versions, pointing out subtle differences. We can state that the explicitness of EDI renders apparent what the more complex Eden communication semantics might hide. Especially for the question of stream communication, we can state from our results that Eden obviously provides more programming comfort: Stream communication is automatically selected by overloading, and many applications exist where an Eden version works alright, while an otherwise equivalent EDI version would deadlock. Therefore, a type class providing overloaded communication similar to `Trans` is a sensible extension to EDI as a language (and may be added instantly without problems). In EDI, however, it should be the programmer's choice whether to use these class functions. On the other hand, the workpool implementation presented in Sec. 7.2 is a case where even the thin safety wrapper of EDI communication functions turned out to be too restrictive.

Our thesis constitutes the first investigation of the Google-mapReduce skeleton in a *really* parallel functional context, for which we have proposed two implementations, one especially geared towards stream processing. We plan to further develop and test our implementations in order to obtain a more precise statement about optimisation potential, performance differences and drawbacks of the proposed versions.

In close connection with the strengths of EDI, we have introduced the concept of *topology skeletons*, which describe regular process structures and their interaction. Discussing pipelines, rings and toroids, we have illustrated that EDI allows one to define arbitrary communication structures and provides directly accessible implementations. The non-functional extensions of Eden provide similar expressiveness, but hide the (necessary) side-effects behind its purely functional face, which can lead to problems. To make this evident, issues related to skeleton nesting have been briefly discussed for a toroid skeleton and for parallel pipelines. We have pointed out that skeleton nesting is a nontrivial issue. Implicit properties of a skeleton implementation, harmless for direct applications, might combine

to severe usage restrictions when nesting skeletons. The bottom line is that a specialised implementation for the concrete nested skeleton should be considered in competition, and both Eden and EDI provide the necessary tools to define that custom skeleton. As we have announced in the introduction, skeleton *implementation* necessitates more explicit control, at a lower level of abstraction than skeleton-based *programming*.

Finally, aside from the language comparison we have aimed at, all our studies substantiate the claim that the skeleton approach and, more generally, the *functional* paradigm provides an abstract perspective on parallelism concepts and clearly indicates fundamental properties of different implementations, which would otherwise be obfuscated by algorithmic particularities and implementation details. This brings us back to our introductory statement on page 3:

There is a pressing need to investigate and establish new parallel programming paradigms in the mainstream. Parallel programming needs more conceptual understanding – which functional coordination concepts make accessible.



# Appendix A

## Bibliography

- [AAA<sup>+</sup>95] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of haskell. In Paul Hudak, editor, *Proceedings of the Haskell Workshop*, La Jolla, USA, June 1995.
- [All07] Frances E. Allen. Compiling for Performance — A Personal Tour. Turing Award Lecture, June 2007. URL: [http://awards.acm.org/turing/addl\\_info/vstream/2006/turingaward2006.mov](http://awards.acm.org/turing/addl_info/vstream/2006/turingaward2006.mov).
- [ATLM06] A.D. Al Zain, P.W. Trinder, H-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *Scalable Computing: Practice and Experience*, 7(3):9–26, 2006.
- [BDLP08] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical Master-Worker Skeletons. In Paul Hudak and David Warren, editors, *PADL'08 — Practical Aspects of Declarative Languages*, Springer LNCS 4902, San Francisco, USA, January 2008.
- [BDO<sup>+</sup>95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vaneschi. P<sup>3</sup>L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.
- [Ben07] Anne Benoit. ESkel — The Edinburgh Skeleton Library. University of Edinburgh, URL: <http://homepages.inf.ed.ac.uk/abenoit1/eSkel/>, 2007.
- [Ber03] Jost Berthold. Dynamic Chunking in Eden. In Phil Trinder, Greg Michaelson, and Ricardo Peña, editors, *Intl. Workshop on the Implementation of Functional Languages (IFL 2003)*, Springer LNCS 3145, Edinburgh, UK, 2003.
- [Ber04] Jost Berthold. Towards a Generalised Runtime Environment for Parallel Haskell. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science*

- *ICCS'04*, number 3 in Springer LNCS 3038, Krakow, Poland, 2004. (Workshop on Practical Aspects of High-level Parallel Programming — PAPP04).
- [Bev89] D. I. Bevan. An efficient reference counting solution to the distributed garbage collection problem. *Parallel Computing*, 9(2):179–192, 1989.
- [BF] E. Biagioni and G. Fu. The Hello Operating System. Information at <http://www2.ics.hawaii.edu/~esb/prof/proj/hello/>.
- [BKL<sup>+</sup>03] Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp. High-level Process Control in Eden. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *EuroPar 2003 – Parallel Processing*, Springer LNCS 2790, Klagenfurt, Austria, 2003.
- [BKPS03] Paul A. Bailes, Colin J. M. Kemp, Ian Peake, and Sean Seefried. Why functional programming really matters. In M. H. Hamza, editor, *The 21st IASTED International Multi-Conference on Applied Informatics (AI 2003), February 10-13, 2003, Innsbruck, Austria*. IASTED/ACTA Press, 2003.
- [BL05a] Jost Berthold and Rita Loogen. The impact of dynamic channels on functional topology skeletons. In Alexander Tiskin and Frederic Loulergue, editors, *HLPP 2005 — 3rd International Workshop on High-level Parallel Programming and Applications*, Coventry, UK, 2005. Proceedings.
- [BL05b] Jost Berthold and Rita Loogen. Skeletons for recursively unfolding process topologies. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *Proceedings of ParCo 2005*, volume 33 of *NIC*, 2005.
- [BL07a] Jost Berthold and Rita Loogen. Parallel coordination made explicit in a functional setting. In Zoltán Horváth and Viktória Zsóka, editors, *18th Intl. Symposium on the Implementation of Functional Languages (IFL 2006)*, Springer LNCS 4449, Budapest, Hungary, 2007. (**awarded best paper** of IFL'06).
- [BL07b] Jost Berthold and Rita Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In G.R. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bücker, P. Gibbon, and B. Mohr, editors, *Proceedings of ParCo 2007*, volume 38 of *NIC*, Jülich, Germany, September 2007.
- [BL08] Jost Berthold and Rita Loogen. The Impact of Dynamic Channels on Functional Topology Skeletons. *Parallel Processing Letters (World Scientific Publishing Company)*, 18(1):101–115, 2008.
- [BLAZ08] Jost Berthold, Hans-Wolfgang Loidl, and Abyd Al-Zain. Scheduling Light-Weight Parallelism in ARTCoP. In Paul Hudak and David Warren, editors, *PADL'08 — Practical Aspects of Declarative Languages*, Springer LNCS 4902, San Francisco, USA, January 2008.

- 
- [Ble96] G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
- [BLO95] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for parallel and concurrent programming. In D.N. Turner, editor, *Functional Programming, Workshops in Computing*, Glasgow, 1995. Springer.
- [BLOMP97] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Geneva, Switzerland, 1997. IEEE Press.
- [BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report 10, Philipps-University of Marburg, 1996. Available at <http://www.mathematik.uni-marburg.de/fb12/bfi/bfi10.ps>.
- [BLPW02] Jost Berthold, Rita Loogen, Steffen Priebe, and Nils Weskamp. Porting the Eden System to GHC 5.00.2. In *14th Intl. Workshop on the Implementation of Functional Languages (IFL 2002)*. Univ. Complutense de Madrid, Sept. 2002. Workshop version only, not submitted.
- [Bre98] Sylvia 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/>.
- [CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, Cambridge(MA), USA, 1990.
- [CG92] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [Chine] Andrew Chien. Parallelism Drives Computing. Talk given at Manycore Computing Workshop, 2007 June. <http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/Andrew%20Chien.pdf>.
- [CKL<sup>+</sup>07] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*. MIT Press, Cambridge(MA), USA, 2007.
- [CKLP01] M.M.T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal — Nested Data-Parallelism in Haskell. In *EuroPar’01 — European Conf. on Parallel Processing*, Springer LNCS 2150, Manchester, UK, August 2001.

- [CLJ<sup>+</sup>07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming*, Nice, France, 2007. ACM Press.
- [Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge(MA), USA, 1989.
- [Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
- [Dan91] Daniel A. Reed and Robert D. Olson and Ruth A. Aydt et al. Scalable performance environments for parallel systems. In *Sixth Distributed Memory Computing Conference Proceedings (6th DMCC'91)*, Portland(OR), USA, 1991. IEEE Press.
- [Dan07] Marco Danelutto. The parallel programming library Muskel. Università di Pisa, <http://www.di.unipi.it/~marcod/Muskel/Home.html>, 2007.
- [DBL08] Mischa Dieterle, Jost Berthold, and Rita Loogen. Implementing a Distributed Work Pool Skeleton in a Parallel Functional Language. 2008. In preparation.
- [DDL98] Marco Danelutto, Roberto DiCosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *ACM workshop on ML and its applications*, 1998.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04, Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004*, 2004.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [Die07] Mischa Dieterle. Parallel functional implementation of master worker skeletons. Diploma Thesis, Philipps-Universität Marburg, October 2007. In German.
- [Dij68] E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11:147–148, 1968.
- [DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.
- [FFR<sup>+</sup>07] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In Claudio Russo and Derek Dreyer, editors, *Proceedings of the ACM SIGPLAN Workshop on ML*, Freiburg, Germany, October 2007. ACM.



- [FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
- [Fly66] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.
- [Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. <http://www.mcs.anl.gov/dbpp/>.
- [Fre99] Ralf Freitag. Entwicklung eines Werkzeugs zur Analyse des Laufzeitverhaltens paralleler funktionaler Programme. Diploma Thesis, Philipps-Universität Marburg, 1999. In German.
- [GHC] The Glasgow Haskell Compiler. <http://www.haskell.org/ghc>.
- [GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, LNCS 707, Kaiserslautern, Germany, 1993. Springer.
- [GHJV00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.
- [GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: a Symmetric Integration of Concurrent and Functional Programming. In Josep Díaz and Fernando Orejas, editors, *Tapsoft'89 - Int. Joint Conf. on Theory and Practice of Software Development*, Springer LNCS 352, 1989.
- [Gor00] Sergei Gorlatch. Toward formally-based design of message passing programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, 2000.
- [Gor04] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.
- [GPP96] L. A. Galán, C. Pareja, and R. Peña. Functional skeletons generate process topologies in Eden. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP'96 - Programming Languages: Implementations, Logics, and Programs*, LNCS 1140. Springer, 1996.
- [HBL03a] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters (World Scientific Publishing Company)*, 13(3):413–424, 2003.
- [HBL03b] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic Skeletons in Template Haskell. In F. Loulergue, editor, *HLPP 2003 - 2nd International Workshop on High-level Parallel Programming and Applications, Paris, France, 2003*.

- [HDD95] I. Holyer, N. Davies, and C. Dornan. The Brisk Project: Concurrent and Distributed Functional Systems. In *Proceedings of the Glasgow-Workshop on Functional Programming*, Electronic Workshops in Computing, Ullapool, Scotland, July 1995. Springer. [http://www.bcs.org/upload/pdf/ewic\\_fp95\\_paper10.pdf](http://www.bcs.org/upload/pdf/ewic_fp95_paper10.pdf).
- [HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.
- [Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.
- [HHJW07] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego (CA), USA, June 2007. ACM.
- [HJLT05] T. Hallgren, M.P. Jones, R. Leslie, and A.P. Tolmach. A Principled Approach to Operating System Construction in Haskell. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, (ICFP'05)*, Tallinn, Estonia, September 26–28, 2005. ACM.
- [HL00] C. Herrmann and C. Lengauer. *HDC*: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.
- [HL08] Martin Hofmann and Rita Loogen. Automatisierte modellgestützte Verwaltung von Parallelität auf heterogenen Architekturen. DFG project application, under consideration, March 2008. Includes further ARTCoP research. Contributions by Hans-Wolfgang Loidl and Jost Berthold.
- [HM99] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999.
- [HMJ05] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In Daan Leijen, editor, *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*. ACM Press, September 2005.
- [Hoe06] Jay P. Hoeflinger. Extending OpenMP to Clusters. Intel White Paper, 2006. <http://www.intel.com/cd/software/products/asmo-na/eng/329076.htm>.
- [HPR00] F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming 1 (SFP'99)*. Intellect, 2000.

- [HS07] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, (ICFP'07)*, Freiburg, Germany, October 2007. ACM.
- [Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, February 1989.
- [Ibo06] Pepe Iborra. GHCI debugger. Presentation, design and implementation notes. Web pages in Haskell Wiki, 2006. <http://haskell.org/haskellwiki/GHC/GHCiDebugger> and <http://cvs.haskell.org/trac/ghc/wiki/GhciDebugger>.
- [IEE92] IEEE. *POSIX P1003.4a — Threads Extension for Portable Operating Systems*, 1992.
- [IM07] José Iborra and Simon Marlow. Examine your Laziness. A lightweight procedural debugging technique for Haskell. Technical report, DSIC, Universidad Politécnic de Valencia, April 2007. <http://www.dsic.upv.es/docs/bib-dig/informes/etd-04042007-111431/papernew2.pdf>.
- [Int] Intel Corp. Threading Analysis Tools. Product Line. Information at <http://www.intel.com/cd/software/products/asmona/eng/threading//>.
- [JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of POPL '96*, New York, NY, USA, 1996. ACM Press.
- [JL92] Richard Jones and Rafael D Lins. Cyclic weighted reference counting without delay. Technical Report 28-92\*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1992.
- [JME99] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in haskell. In Pieter W. M. Koopman and Chris Clack, editors, *Implementation of Functional Languages, 11th International Workshop, IFL'99, Lochem, The Netherlands, September 7-10, 1999, Selected Papers*, LNCS 1868. Springer, 1999.
- [JP92] S. Jagannathan and J. Philbin. A Customizable Substrate for Concurrent Languages. In *Conf. on Programming Language Design and Implementation (PLDI'92)*. ACM Press, July 1992. ACM SIGPLAN Notices 27(7).
- [Kar81] K. Karlsson. Nebula, a Functional Operating System. Technical report, Chalmers Univ, 1981.
- [KG96] James Arthur Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proceedings of HICSS-29*. IEEE Computer Society Press, 1996.

- [KLPR00] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden — Low-Effort Parallel Programming. In *IFL'00 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 2011, Aachen, Germany, September 2000. Springer.
- [Klu] U. Klusik. An Efficient Implementation of the Parallel Functional Language Eden on Distributed-Memory System. unpublished Ph.D work.
- [KPS99] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *SFP'99 — Scottish Functional Programming Workshop*, volume 1 of *Trends in Functional Programming*, pages 2–11, Stirling, Scotland, 1999. Intellect. <http://www.cee.hw.ac.uk/~greg/sfp/>.
- [Kuc02] H. Kuchen. The Münster Skeleton Library Muesli, University of Münster 2002. <http://www.wi.uni-muenster.de/PI/forschung/Skeletons/>.
- [KW93] D.J. King and P. Wadler. Combining Monads. Technical report, University of Glasgow, 1993.
- [Läm06] Ralf Lämmel. Google's MapReduce Programming Model – Revisited. <http://www.cs.vu.nl/~ralf/MapReduce/>; Online since 2 January, 2006; 42 pages, 2006.
- [Läm08] Ralf Lämmel. Google's mapreduce programming model - revisited. *Sci. Comput. Program*, 70(1):1–30, 2008.
- [LMJT07] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for ghc. In Gabriele Keller, editor, *Haskell 2007: Proceedings of the ACM SIGPLAN workshop on Haskell, Freiburg, Germany, September 30, 2007*. ACM, Sep 2007.
- [Loi96] H. W. Loidl. GranSim User's Guide. Technical report, University of Glasgow. Department of Computer Science, 1996.
- [Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, 1998.
- [LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
- [Loo99] Rita Loogen. Programming language constructs. In Hammond and Michaelson [HM99], chapter 3, pages 63–92.
- [LOP<sup>+</sup>03] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
- [MH04] Rajiv Mirani and Paul Hudak. First-class monadic schedules. *ACM TOPLAS*, 26(4):609–651, July 2004.

- [Mil99] Robin Milner. *Communicating and Mobile Systems: The  $\pi$  Calculus*. Cambridge Univ. Press, 1999.
- [MIPG07] Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. A lightweight interactive debugger for Haskell. In Gabriele Keller, editor, *Haskell 2007: Proceedings of the ACM SIGPLAN workshop on Haskell*. ACM, Sep 2007.
- [MP03] Rafael Martínez and Ricardo Pena. Building an Interface Between Eden and Maple: A Way of Parallelizing Computer Algebra Algorithms. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *IFL'03: Implementation of Functional Languages, 15th International Workshop, Selected Papers*, LNCS 3145, Edinburgh, UK, 2003. Springer.
- [MPI97] MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.
- [MSBK01] Greg Michaelson, Norman Scaife, Paul Bristow, and Peter King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.*, 16:181–206, 2001.
- [MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. SPS. Addison-Wesley, Boston, 2005.
- [NAH<sup>+</sup>95] R.S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH Language Reference Manual. Technical Report CSG Memo 369, Laboratory for Computer Science, MIT, January 1995.
- [NAW<sup>+</sup>96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.
- [Per88] N. Perry. Towards a Functional Operating System. Technical report, Dept. of Computing, Imperial College, London, UK, 1988.
- [Pey00] Simon Peyton-Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. Presented at the 2000 Marktoberdorf Summer School., 2000. Available online: <http://research.microsoft.com/~simonpj/papers/marktoberdorf>.
- [PH99] S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at <http://www.haskell.org/>.
- [PK05] Michael Poldner and Herbert Kuchen. Scalable farms. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of ParCo 2005*, NIC Series, vol.33, Malaga, Spain, 2005. Central Institute for Applied Mathematics, Jülich, Germany.

- [PPRS00] C. Pareja, R. Pena, F. Rubio, and C. Segura. Optimising Eden by Transformation. In Stephen Gilmore, editor, *SFP'00 — 1st Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, University of St Andrews, Scotland, 2000. Intellect.
- [PRS01] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In Kevin Hammond and Sharon Curtis, editors, *3rd Scottish Functional Programming Workshop (SFP01), selected papers*, volume 3 of *Trends in Functional Programming*. Intellect, 2001.
- [PTL01] R.F. Pointon, P.W. Trinder, and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In Markus Mohnen and Pieter W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, LNCS 2011, page 53ff, Aachen, Germany, 2001. Springer.
- [PvE93] M.J. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, Massachusetts, USA, 1993.
- [PVM] PVM: Parallel Virtual Machine. Website <http://www.csm.ornl.gov/pvm/>.
- [Qui94] M.J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.
- [Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ. Press, 1999.
- [RG03] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
- [RG04] Pablo Roldán Gómez. Eden Trace Viewer: Ein Werkzeug zur Visualisierung paralleler funktionaler Programme. Diploma Thesis, Philipps-Universität Marburg, 2004. In German.
- [RRP<sup>+</sup>07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyraki. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.
- [Sha84] E. Shapiro. Systems Programming in Concurrent Prolog. In *Symp. on Principles of Programming Languages (POPL'84)*, pages 93–105, Salt Lake City, Utah, 1984.
- [SJ02] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

- 
- [SP99] P. R. Serrarens and R. Plasmeijer. Explicit message passing for concurrent clean. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *IFL'98 — Implementation of Functional Languages*, LNCS 1595. Springer, 1999.
- [ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.
- [Sto84] W.R. Stoye. A New Scheme for Writing Functional Operating Systems. Technical Report 56, Computer Lab, Cambridge University, 1984.
- [Str06] Björn Struckmeier. Implementierung eines Werkzeugs zur Visualisierung und Analyse paralleler Programmläufe in Haskell. Diploma Thesis, Philipps-Universität Marburg, 2006. In German.
- [Tay97] F.S. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Department of Computing, Imperial College, London, 1997.  
<http://www.lieder.demon.co.uk/thesis/thesis.ps.gz>.
- [THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, 1998.
- [THM<sup>+</sup>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 Press, 1996.
- [TLP02] P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and distributed haskells. *J. of Functional Prog.*, 12(4, 5):469–510, 2002.
- [Top] Top 500 Supercomputer Sites. WWW. <http://www.top500.org>.
- [Tri04] Phil Trinder. Send statements considered harmful, or high level coordination constructs. BCS “Grand Challenges in Computing” Conference contribution, 2004. (available online).
- [Tur87] D. Turner. Functional Programming and Communicating Processes. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE II*, LNCS 259, Eindhoven, The Netherlands, 1987. Springer.
- [vP03] A. van Weelden and R. Plasmeijer. Towards a Strongly Typed Functional Operating System. In Ricardo Pena and Thomas Arts, editors, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers*, LNCS 2670. Springer, 2003.

# List of Figures

3.1	Layered Eden implementation . . . . .	23
3.2	RTE message-passing module (interface <i>MPSystem.h</i> ) . . . . .	24
3.3	Primitive operations to implement Eden (module <i>ParPrim.hs</i> ) . . . . .	27
3.4	Type class <code>Trans</code> of transmissible data . . . . .	28
3.5	Eden module: Overloading for communication . . . . .	28
3.6	Eden module: Process abstraction and instantiation . . . . .	29
3.7	Implementation of Eden’s advanced, non-functional features . . . . .	31
3.8	<i>Inadequate</i> simulation for Eden coordination constructs . . . . .	32
3.9	EDI, IO-monadic coordination features for parallel Haskell . . . . .	37
3.10	Layer view applied to GpH implementation . . . . .	39
3.11	Layered implementation of a data-parallel language . . . . .	40
4.1	Layer view of ARTCoP . . . . .	45
4.2	Component view of ARTCoP . . . . .	45
Table 4.1	Haskell Scheduler classes (implemented at system level) . . . . .	47
4.3	Default scheduler . . . . .	47
Table 4.2	ARTCoP primitives (provided by kernel) . . . . .	48
4.4	A simple parallel scheduler . . . . .	48
4.5	Kernel operations, as far as implemented by EDI primitives . . . . .	49
4.6	Scheduler for a parallel job-pool . . . . .	50
4.7	System level code related to load information . . . . .	51
4.8	Types for modelling the GUM system . . . . .	53
4.9	GUM scheduler . . . . .	53
4.10	GRIDGUM 2 work location algorithm . . . . .	55
4.11	GRIDGUM 2 work request handling algorithm . . . . .	56
4.12	Sketch: Replacing heap data by a fetching node . . . . .	62
4.13	Heap graph traversal (breadth-first/depth-first) . . . . .	65
4.14	Worker functions to pretty-print a heap graph structure . . . . .	66
4.15	Spark export preparation for distributed memory . . . . .	69
5.1	Thread state transitions . . . . .	73
5.2	EdenTV trace events . . . . .	73
5.3	Examples of EdenTV diagrams and colour codes table . . . . .	74
5.4	<code>xpvm</code> vs EdenTV . . . . .	76
5.5	<i>All Machines</i> diagrams for different parallel Mandelbrot programs . . . . .	78
5.6	Ring process functionality for parallel Warshall program . . . . .	79
5.7	Warshall’s algorithm (500-node graph) . . . . .	80



---

5.8	Pipeline test program revealing RTE bug (no round-robin placement)	81
7.1	Straightforward implementations for <code>map</code> in Eden	92
7.2	EDI and Eden implementations for <code>map</code> with embedded input	92
7.3	Eden master-worker skeleton and <code>parMap</code> implementation	94
7.4	EDI workpool skeleton, using concurrent <code>inputSender</code> threads	95
7.5	Worker process and <code>inputSender</code> thread for EDI workpool	96
7.6	Helper functions for splitting lists	98
7.7	Computation scheme of Google-mapReduce	101
7.8	$k$ -means clustering implemented by Google-mapReduce	103
7.9	Parallel Google-mapReduce, parallelisation as described in papers	104
7.10	Parallel Google-mapReduce skeleton, following Lämmel [Läm06]	105
7.11	Combining pre-grouped sub-centroids for $k$ -means	107
7.12	Parameter functions to be used for parallel $k$ -means algorithm	107
7.13	Google-mapReduce micro-benchmark, with and without workload	109
8.1	Anno 1997: Pipeline by a fold	114
8.2	Naïve tail-recursive pipeline	114
8.3	Pipeline by inner recursion	115
8.4	Pipeline with dynamic reply channel	115
8.5	Eden pipeline skeleton where caller creates all processes	115
8.6	Pipeline skeleton in EDI, single-source variant	116
8.7	Pipeline skeleton in EDI, recursive variant	116
8.8	Helper functions for pipeline including I/O	117
8.9	EDI pipeline skeletons, fixed no. of stages	118
8.10	Template Haskell scheme to create pipeline skeleton	119
8.11	Eden ring skeleton without dynamic channels	120
8.12	Eden Ring Skeleton	121
8.13	Warshall's algorithm (500 nodes) using <b>static connections</b> in ring	123
8.14	Warshall's algorithm (500 nodes) using <b>dynamic channels</b> in ring	123
8.15	Recursively unfolding ring skeleton	125
8.16	Function to derive a ring skeleton from a pipeline skeleton	126
8.17	Speedup of Warshall program (750 nodes) using different ring skeletons	127
8.18	Creation scheme of a torus topology using ring skeletons	128
8.19	Core of recursively unfolding toroid skeleton	129
8.20	Start phase of matrix multiplication traces using toroid skeletons	130
8.21	EDI toroid skeleton, using ring and pipeline	132
8.22	Toroid skeleton comparison, using a toy program	133
8.23	<code>parpipeWhole</code> , parallel pipeline, created as <code>parMap(pipe ...)</code>	134
8.24	<code>parpipeStages</code> , Parallel pipeline, created as <code>pipe(parmap ...)</code>	135
8.25	Two parallel pipeline skeleton versions (sketch)	136



# Appendix B

## Code collection

### B.1 Implementation

#### B.1.1 Eden module: Eden.hs

```
{-# OPTIONS -cpp -fglasgow-exts #-}
-- Eden Project, JB
-- Dissertation Jost Berthold
--
-- Eden module, defining high-level coordination concepts via
-- Prim.Op.s (which are wrapped inside ParPrim.hs).
--
-- This version: working version 03/2008 from Eden group.
-- Includes a primitives simulation using concurrent Haskell.
-- (compile with -DSIMUL)

module Eden(
  -- reexported from Strategies ---
  NFDData(..), using, r0
  ----- basic Eden -----
  , noPe, selfPe
  , Process, process, ( # )
  , instantiate
  , instantiateAt      -- explicit placement
  , Trans(..)
  ----- dynamic channels -----
  , ChanName          -- Communicator a -> IO(), abstract outside
  , new, parfill      -- using unsafePerformIO
  -----
  , merge, mergeProc -- merge, as specified in Eden language, but function!
  -----
  , Lift(..), deLift, createProcess, cpAt -- deprecated legacy code for Eden 5
)
where

#ifdef SIMUL
#define __PARALLEL_HASKELL__
#endif

#ifdef __PARALLEL_HASKELL__
#warning Compiling a sequential version of Eden.hs
#endif

import Control.Concurrent      -- Instances only
import System.IO.Unsafe(unsafePerformIO) -- for functional face

import qualified ParPrim
import ParPrim hiding(noPe,selfPe)

import Control.Parallel.Strategies -- reexported!
```

```

(NFData(..),using, r0)

-----
-- legacy code for Eden 5:

{-# DEPRECATED deLift, Lift "Lift data type not needed in Eden 6 implementation" #-}
data Lift a = Lift a
deLift :: Lift a -> a
deLift (Lift x) = x

{-# DEPRECATED createProcess "better use instantiate :: Process a b -> a -> IO b instead" #-}
createProcess :: (Trans a, Trans b)
               => Process a b -> a -> Lift b
createProcess p i = unsafePerformIO (instantiate p i >>= \x -> return (Lift x))

cpAt :: (Trans a, Trans b)
      => Int -> Process a b -> a -> Lift b
cpAt pe p i = unsafePerformIO (instantiateAt pe p i >>= \x -> return (Lift x))

----- Eden constructs, also available in seq. version -----

-- system information
noPe, selfPe :: Int
#if defined(__PARALLEL_HASKELL__)
noPe = unsafePerformIO ParPrim.noPe
selfPe = unsafePerformIO ParPrim.selfPe
#else
noPe = 1
selfPe = 1
#endif

-- processes and instantiation
process :: (Trans a, Trans b) => (a -> b) -> Process a b
instantiate :: (Trans a, Trans b) => Process a b -> a -> IO b
instantiateAt :: (Trans a, Trans b) => Int -> Process a b -> a -> IO b
( # ) :: (Trans a, Trans b) => Process a b -> a -> b

#if defined(__PARALLEL_HASKELL__)
data Process a b
  = Proc (ChanName b -> -- send back result, overloaded
         ChanName' (ChanName a) -> -- send input Comm., not overloaded
         IO ()
        )

process f = Proc f_remote
  where f_remote (Comm sendResult) inCC
        = do (sendInput, input) <- createComm
              connectToPort inCC
              sendData Data sendInput
              sendResult (f input)

instantiate = instantiateAt 0

instantiateAt p (Proc f_remote) procInput
  = 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

{-# NOINLINE ( # ) #-}
p # x = unsafePerformIO (instantiateAt 0 p x)

#else
-- sequential simulation:
data Process a b = Proc (a -> b)

process f = Proc f
instantiate (Proc f) x

```

```

    = -- rnf fx 'seq' -- WRONG: can be tuple with infinite parts. NO EASY WAY TO DO IT!
      return fx
    where fx = f x
  instantiateAt _ = instantiate
  (Proc f) # x = f x
#endif

----- merge function, borrowed from Concurrent Haskell -----
merge :: [[a]] -> [a]
merge xss = unsafePerformIO (nmergeIO xss)

mergeProc = merge

-----
-- overloading trick: a "communicator" provides a suitable
-- communication function for the overloaded type

-- type Comm a = (a -> IO())
-- JB20061017: leads to obscure runtime errors
-- Must use an own data type like this:

newtype Comm a = Comm (a -> IO())
-- assumed: contained function sends a over a (previously wired-in) channel
instance NFData (Comm a)

type ChanName a = Comm a -- provide old Eden interface to the outside world

-----
-- Eden-specific operations new/parfill for dynamic channels:

{-# NOINLINE new #-}
new :: Trans a => (ChanName a -> a -> b) -> b
{-# NOINLINE parfill #-}
parfill :: Trans a => ChanName a -> a -> b -> b

#if defined(__PARALLEL_HASKELL__)
parfill (Comm sendVal) val cont
  = unsafePerformIO (fork (sendVal val) >> return cont)
new chanValCont = unsafePerformIO $ do
  (chan , val) <- createComm
  return (chanValCont chan val)
#else
-- no channel support in seq. version
new _ = error "new: channels not supported"
parfill _ _ v = error "parfill: channels not supported"
#endif

-----
-- Trans class: overloading communication for streams and tuples

#if defined(__PARALLEL_HASKELL__)
class NFData a => Trans a where
  -- lists/streams written element by element, other types as single
  -- values. All data is evaluated to NF prior to communication
  write :: a -> IO ()
  write x = rnf x 'seq' sendData Data x
  -- produce suitable communicator for tuple types:
  createComm :: IO (ChanName a, a)
  createComm = do (cx,x) <- createC
                 return (Comm (sendVia cx) , x)
-----
-- Trans Instances:
-----

-- "standard types" from Prelude are Transmissible with default
-- communication
instance Trans Int
instance Trans Float
instance Trans Double
instance Trans Char

```

```

instance Trans Integer
instance Trans Bool

-- maybe instance, no NFData in GHC < 6.8
# if __GLASGOW_HASKELL__ < 608
instance NFData a => NFData (Maybe a)
  where rnf Nothing = ()
        rnf (Just x) = rnf x
#endif
instance Trans a => Trans (Maybe a)

instance Trans ()
-- unit: no communication desired? BREAKS OLD PROGRAMS
-- where
--   write () = error "Eden.lhs: writing unit value"
--   createComm = return (Comm (\_ -> return ()), ())

-- stream communication:
instance (Trans a) => Trans [a] where
  write l@[ ] = sendData Data l
  write (x:xs) = (rnf x 'seq' sendData Stream x) >>
    write xs

-- "higher-order channels"
instance (NFData a, Trans a) => Trans (ChanName' a)
instance (NFData a, Trans a) => Trans (Comm a)

-- tuple instances:
instance (Trans a, Trans b) => Trans (a,b)
  where createComm = do (cx,x) <- createC
                        (cy,y) <- createC
                        return (Comm (write2 (cx,cy)),(x,y))
instance (Trans a, Trans b, Trans c) => Trans (a,b,c)
  where createComm = do (cx,x) <- createC
                        (cy,y) <- createC
                        (cz,z) <- createC
                        return (Comm (write3 (cx,cy,cz)),(x,y,z))
instance (Trans a, Trans b, Trans c, Trans d) => Trans (a,b,c,d)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        return (Comm (write4 (ca,cb,cc,cd)),
                                (a,b,c,d))
instance (Trans a, Trans b, Trans c, Trans d, Trans e)
=> Trans (a,b,c,d,e)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        (ce,e) <- createC
                        return (Comm (write5 (ca,cb,cc,cd,ce)),
                                (a,b,c,d,e))
instance (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f)
=> Trans (a,b,c,d,e,f)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        (ce,e) <- createC
                        (cf,f) <- createC
                        return (Comm (write6 (ca,cb,cc,cd,ce,cf)),
                                (a,b,c,d,e,f))
instance (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f, Trans g)
=> Trans (a,b,c,d,e,f,g)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        (ce,e) <- createC
                        (cf,f) <- createC
                        (cg,g) <- createC

```

```

        return (Comm (write7 (ca,cb,cc,cd,ce,cf,cg)),
               (a,b,c,d,e,f,g))

instance (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f, Trans g, Trans h)
  => Trans (a,b,c,d,e,f,g,h)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        (ce,e) <- createC
                        (cf,f) <- createC
                        (cg,g) <- createC
                        (ch,h) <- createC
                        return (Comm (write8 (ca,cb,cc,cd,ce,cf,cg,ch)),
                                (a,b,c,d,e,f,g,h))

instance (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f, Trans g, Trans h, Trans i)
  => Trans (a,b,c,d,e,f,g,h,i)
  where createComm = do (ca,a) <- createC
                        (cb,b) <- createC
                        (cc,c) <- createC
                        (cd,d) <- createC
                        (ce,e) <- createC
                        (cf,f) <- createC
                        (cg,g) <- createC
                        (ch,h) <- createC
                        (ci,i) <- createC
                        return (Comm (write9 (ca,cb,cc,cd,ce,cf,cg,ch,ci)),
                                (a,b,c,d,e,f,g,h,i))

-- bigger tuples use standard communication

-----
-- helper functions for Trans class:

-- send function for a single data type (no tuple, non-concurrent)
sendVia :: (NFData a,
            Trans a)
        => (ChanName' a) -> a -> IO()
sendVia c d = connectToPort c >>
              (sendData Connect d) >> -- optional: connect before evaluation
              write d

-----
-- send functions for tuples...
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
write3 :: (Trans a, Trans b, Trans c)
        => (ChanName' a, ChanName' b, ChanName' c) -> (a,b,c) -> IO ()
write3 (c1,c2,c3) (x1,x2,x3) = do
  fork (sendVia c1 x1)
  fork (sendVia c2 x2)
  sendVia c3 x3
write4 :: (Trans a, Trans b, Trans c, Trans d)
        => (ChanName' a, ChanName' b, ChanName' c, ChanName' d)
        -> (a,b,c,d) -> IO ()
write4 (c1,c2,c3,c4) (x1,x2,x3,x4) = do
  fork (sendVia c1 x1)
  fork (sendVia c2 x2)
  fork (sendVia c3 x3)
  sendVia c4 x4
write5 :: (Trans a, Trans b, Trans c, Trans d, Trans e)
        => (ChanName' a, ChanName' b, ChanName' c, ChanName' d, ChanName' e)
        -> (a,b,c,d,e) -> IO ()
write5 (c1,c2,c3,c4,c5) (x1,x2,x3,x4,x5) = do
  fork (sendVia c1 x1)
  fork (sendVia c2 x2)
  fork (sendVia c3 x3)
  fork (sendVia c4 x4)
  sendVia c5 x5
write6 :: (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f)

```

```

=> (ChanName' a, ChanName' b, ChanName' c, ChanName' d,
    ChanName' e, ChanName' f
    ) -> (a,b,c,d,e,f) -> IO ()
write6 (c1,c2,c3,c4,c5,c6) (x1,x2,x3,x4,x5,x6) = do
    fork (sendVia c1 x1)
    fork (sendVia c2 x2)
    fork (sendVia c3 x3)
    fork (sendVia c4 x4)
    fork (sendVia c5 x5)
    sendVia c6 x6
write7 :: (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f, Trans g)
=> (ChanName' a, ChanName' b, ChanName' c, ChanName' d,
    ChanName' e, ChanName' f, ChanName' g
    ) -> (a,b,c,d,e,f,g) -> IO ()
write7 (c1,c2,c3,c4,c5,c6,c7) (x1,x2,x3,x4,x5,x6,x7) = do
    fork (sendVia c1 x1)
    fork (sendVia c2 x2)
    fork (sendVia c3 x3)
    fork (sendVia c4 x4)
    fork (sendVia c5 x5)
    fork (sendVia c6 x6)
    sendVia c7 x7
write8 :: (Trans a, Trans b, Trans c, Trans d, Trans e, Trans f, Trans g, Trans h)
=> (ChanName' a, ChanName' b, ChanName' c, ChanName' d,
    ChanName' e, ChanName' f, ChanName' g, ChanName' h
    ) -> (a,b,c,d,e,f,g,h) -> IO ()
write8 (c1,c2,c3,c4,c5,c6,c7,c8) (x1,x2,x3,x4,x5,x6,x7,x8) = do
    fork (sendVia c1 x1)
    fork (sendVia c2 x2)
    fork (sendVia c3 x3)
    fork (sendVia c4 x4)
    fork (sendVia c5 x5)
    fork (sendVia c6 x6)
    fork (sendVia c7 x7)
    sendVia c8 x8
write9 :: (Trans a,Trans b,Trans c,Trans d,Trans e,Trans f,Trans g,Trans h,Trans i)
=> (ChanName' a, ChanName' b, ChanName' c, ChanName' d,
    ChanName' e, ChanName' f, ChanName' g, ChanName' h, ChanName' i
    ) -> (a,b,c,d,e,f,g,h,i) -> IO ()
write9 (c1,c2,c3,c4,c5,c6,c7,c8,c9) (x1,x2,x3,x4,x5,x6,x7,x8,x9) = do
    fork (sendVia c1 x1)
    fork (sendVia c2 x2)
    fork (sendVia c3 x3)
    fork (sendVia c4 x4)
    fork (sendVia c5 x5)
    fork (sendVia c6 x6)
    fork (sendVia c7 x7)
    fork (sendVia c8 x8)
    sendVia c9 x9

#else
class NFData a => Trans a -- where nothing happens
  where dummyMethod :: a -> a -- dummy method avoids GHC warning
        dummyMethod _ = error "dummyMethod"

instance (Trans a) => Trans [a]
instance Trans ()
instance Trans Int
instance Trans Float
instance Trans Double
instance Trans Char
instance Trans Integer
instance Trans Bool

-- maybe instance using default
instance Trans a => Trans (Maybe a)

-- Trans instances for tuples
instance (Trans x, Trans y)
=> Trans (x,y)
instance (Trans a, Trans b, Trans c)
=> Trans (a,b,c)

```



```
instance (Trans a, Trans b, Trans c, Trans d)
  => Trans (a,b,c,d)
instance (Trans a, Trans b, Trans c,
  Trans d, Trans e) => Trans (a,b,c,d,e)
instance (Trans a, Trans b, Trans c,
  Trans d, Trans e,Trans f) => Trans (a,b,c,d,e,f)
instance (Trans a, Trans b, Trans c, Trans d,
  Trans e,Trans f,Trans g) => Trans (a,b,c,d,e,f,g)
instance (Trans a, Trans b, Trans c, Trans d,
  Trans e,Trans f,Trans g,Trans h)
  => Trans (a,b,c,d,e,f,g,h)
instance (Trans a, Trans b, Trans c, Trans d,
  Trans e,Trans f,Trans g,Trans h, Trans i)
  => Trans (a,b,c,d,e,f,g,h,i)
#endif
```

## B.1.2 Primitives wrapper: ParPrim.hs

```

{-# OPTIONS -fglasgow-exts -cpp #-}
-- Eden Project, JB

-- Base module, importing PrimOps => exporting IO actions
--
-----

module ParPrim(
  noPe, selfPe      -- system information      :: Int
  , ChanName'      -- primitive channels (abstract in Eden module and outside)
  , fork           -- forking conc. threads :: IO () -> IO ()
  , createC       -- creating placeholders :: IO (ChanName' a, a)
  , connectToPort -- set thread's receiver :: ChanName' a -> IO ()
  , sendData      -- sending data to recv. :: Mode -> a -> IO ()
  , Mode(..)      -- send modes: implemented:
                  -- 1 - connect (no graph needed)
                  -- 2 - stream (list element)
                  -- 3 - single (single value)
                  -- 4 - rFork (receiver creates a thread, different ports)
                  -- additional payload (currently only for rFork) in high bits
)
  where

#ifdef SIMUL
-- simulation: all functionality is imported
import ParPrimConcHs
#else
-- whole rest of file

import GHC.IOBase(IO(..))
import GHC.Base(error, Int#, Int(..), (+#),
  fork#, expectData#, noPe#, selfPe#,
  connectToPort#, sendData#
)
import Control.Parallel.Strategies(NFData(..))

-----
-- IO wrappers for primitive operations:
--
-- all primitives are implemented out-of-line,
-- wrappers should all be of type * -> IO (...)
--
-- (eden implementation can work with unsafePerformIO)
-----

-- system information
{-# NOINLINE noPe #-}
noPe :: IO Int
noPe = IO ( \s -> case (noPe# s) of
  (# s',r #) -> (# s',I# r #)
)
{-# NOINLINE selfPe #-}
selfPe :: IO Int
selfPe = IO ( \s -> case (selfPe# s) of
  (# s',r #) -> (# s',I# r #)
)

-- not for export, only abstract type visible outside
data ChanName' a = Chan Int# Int# Int#
  deriving Show

instance NFData a => NFData (ChanName' a)
  where rnf (Chan pe proc i) = rnf (I# (pe +# proc +# i))

-- tweaking fork primop from concurrent haskell... (not returning threadID)
{-# NOINLINE fork #-}
fork :: IO () -> IO ()
fork action = IO ( \s -> case (fork# action s) of
  (# s' , _ #) -> (# s' , () #)
)

```

```

    )

-- creation of one placeholder and one new inport
{-# NOINLINE createC #-}
-- returns consistent channel type (channel of same type as data)
createC :: IO ( ChanName' a, a )
createC = IO (\s -> case (expectData# s) of
    (# s',i,p, bh #) -> case selfPe# s' of
        (# s'', pe #) ->
            (# s'',(Chan pe p i, bh) #)
    )

-- TODO: wrap creation of several channels in RTS? (see eden5::createDC# )
--      (would save foreign call overhead, but hard-wire more into RTS)

{-# NOINLINE connectToPort #-}
connectToPort_ :: Int# -> Int# -> Int# -> IO ()
connectToPort_ pe proc i
    = IO (\s -> case (connectToPort# pe proc i s) of
        s' -> (# s', () #)
    )

connectToPort :: ChanName' a -> IO ()
connectToPort (Chan p proc i) = connectToPort_ p proc i

-- send modes for sendData
data Mode = Connect -- announce sender at receiver side (no graph needed)
          | 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
decodeMode :: Mode -> Int
decodeMode Connect      = 1
decodeMode Stream       = 2
decodeMode Data         = 3
decodeMode (Instantiate n) = let k = 4 + n*8
                              in -- k 'seq' -- needed to pass NF to PrimOp?
                                 k
-- decodeMode other = error "sendData: no such mode"

{-# NOINLINE sendData #-}
sendData :: Mode -> a -> IO ()
sendData mode d
    = IO (\s -> case (sendData# m d s) of
        s' -> (# s', () #)
    )
    where (I# m) = decodeMode mode
#endif

```

## B.1.3 Primitives simulation using Concurrent Haskell

```

{-# OPTIONS -fglasgow-exts -cpp -fscoped-type-variables #-}
-- Eden Project, JB

-- Base module, importing PrimOps => exporting IO actions
--
-- This version: simulates primitives by Concurrent Haskell
-- (can serve as specification of primitives semantics)
-----

module ParPrimConcHs
  (noPe, selfPe -- system information      :: Int
  , ChanName'   -- primitive channels (abstract in Eden module and outside)
  , fork        -- forking conc. threads :: IO () -> IO ()
  , createC     -- creating placeholders :: IO (ChanName' a, a)
  , connectToPort -- set thread's receiver :: ChanName' a -> IO ()
  , sendData    -- sending data to recv. :: Mode -> a -> IO ()
  , Mode(..)    -- send modes: implemented:
                -- 1 - connect (no graph needed)
                -- 2 - stream (list element)
                -- 3 - single (single value)
                -- 4 - rFork (receiver creates a thread, different ports)
                -- additional payload (currently only for rFork) in high bits
  , simInitPes
  )
  where

#warning Concurrent Haskell Simulation of Primitives
#ifndef __GLASGOW_HASKELL__
#error Need GHC to compile this simulation.
#endif

import GHC.Base(unsafeCoerce# )

import qualified Data.Map as Map -- collides with prelude functions
import Data.Map(Map)

import System.IO.Unsafe
import Control.Concurrent
import Control.Parallel.Strategies

-- Concurrent-Haskell simulation of Eden PrimOps
-----

-- tracing
trace :: String -> IO ()
#ifdef TRACE
trace msg = do me <- myThreadId
              (pe,p,_) <- myInfo
              putStrLn (show (pe,p,me) ++ msg)
#else
trace _ = return ()
#endif

-----

-- (*) unsafe type casts. cannot use dynamics, missing type context.
-- THIS IS A HAAAAAAAAAAAAACK!!!!!!!!!! (*)

toIO :: a -> IO ()
toIO x = case cast x of
  Nothing -> error "IO? wrong cast"
  Just io -> io

cast :: a -> Maybe b
cast x = Just (unsafeCoerce# x)
toDyn :: a -> Untyped
toDyn x = unsafeCoerce# x
fromDyn :: a -> Untyped -> a
fromDyn _ unit = unsafeCoerce# unit
-----

```

```

---- Simulation specials ----

-- global ID supply for process IDs and Channel IDs:
-- (CAF trick, evaluated, i.e. created, by first usage)
{-# NOINLINE idSupply #-}
idSupply :: MVar Int
idSupply = unsafePerformIO (newMVar 1)

-- pulling a fresh channel/process ID:
freshId :: IO Int
freshId = do i <- takeMVar idSupply
            putMVar idSupply (i+1)
            return i

-- process and thread book-keeping:

-- (PE, processID, Maybe connected channel)
type ThreadInfo = (Int,Int,Maybe Int)

-- global thread table: ID -> ThreadInfo
-- (first time called: created with first thread as an entry)
{-# NOINLINE thrs #-}
thrs :: MVar (Map ThreadId (Int,Int,Maybe Int))
thrs = unsafePerformIO (myThreadId >>= \id ->
    newMVar (Map.insert id (1,1,Nothing) Map.empty ))

-- retrieving own thread information
myInfo :: IO ThreadInfo
myInfo = do tid <- myThreadId
            thrMap <- readMVar thrs
            case Map.lookup tid thrMap of
                Nothing -> error (show tid ++ " not found!")
                Just x -> return x

-- retrieving the channel a thread has connected to
myChan :: IO Int
myChan = do (_,_,c) <- myInfo
            case c of
                Nothing -> do tid <- myThreadId
                             error (show tid ++ " not connected!")
                Just x -> return x

-- when thread finished:
removeThread :: ThreadId -> IO ()
removeThread id = do trace ("Kill " ++ show id)
                    thrMap <- takeMVar thrs
                    putMVar thrs (Map.delete id thrMap)

-- table of open channels, and channel lookup
-- (channels are MVars, but for values of various types, we use unsafeCoerce)
-- ( to test the 1:1 restriction, we save past senders for stream comm.)
type Untyped = ()

{-# NOINLINE chs #-}
chs :: MVar (Map Int (Maybe ThreadId, MVar Untyped))
chs = unsafePerformIO (newMVar Map.empty)

-- for Connect messages: only register the calling thread as the sender
registerSender :: Int -> IO ()
registerSender id
    = do cMap <- takeMVar chs
        tid <- myThreadId
        case Map.lookup id cMap of
            Nothing -> error $ "missing MVar for Id " ++ show id
            Just (t,var) -> if (t == Nothing || t == Just tid)
                then do putMVar chs
                        (Map.insert id (Just tid,var) cMap)
                else error ("duplicate connect message: "
                    ++ show tid ++ "->"
                    ++ show id)

```

```

-- for receiving messages, removes the channel (Data message)
getRemoveCVar :: Int -> IO (MVar Untyped)
getRemoveCVar id = do cMap <- takeMVar chs
  case Map.lookup id cMap of
    Nothing      -> error ("missing MVar for Id "
                          ++ show id)
    Just (_,var) -> do putMVar chs (Map.delete id cMap)
                      return var

-- for receiving stream messages, updates the channel, checks the sender
updateGetCVar :: MVar Untyped -> Int -> IO (MVar Untyped)
updateGetCVar newVar id
  = do cMap <- takeMVar chs
      tid <- myThreadId
      case Map.lookup id cMap of
        Nothing      -> error $ "missing MVar for Id " ++ show id
        Just (t,var) -> if (t == Nothing || t == Just tid)
            then do putMVar chs
                    (Map.insert id (Just tid,newVar) cMap)
                    return var
            else error "1:1 restriction violated"

-- holds number of PEs simulated (can be changed using simInitPes function)
{-# NOINLINE pesVar #-}
pesVar :: MVar ([Int],())
pesVar = unsafePerformIO (newMVar ([2,3,4,1],())) -- arbitrary default: 4 PEs

simInitPes :: Int -> IO ()
simInitPes pes | pes < 1 = error "invalid number of PEs requested"
               | otherwise = do (_,test) <- takeMVar pesVar
                                trace ("Init. with " ++ show pes ++ " PEs.")
                                test 'seq' -- protect against double init.
                                putMVar pesVar
                                ([2..pes+1],error "double simInitPes")

-- round-robin placement:
choosePe :: IO Int
choosePe = do pe <- selfPe
  trace "choosing PE"
  (list,test) <- takeMVar pesVar
  let place = list!!(pe-1)
      pes = length list
      new = if place == pes then 1 else place+1
      newList = take (pe-1) list ++ new:drop pe list
  putMVar pesVar (newList,test)
  trace "chosen"
  return place

----- Primitives simulation -----

-- the following is exported:

-- system information
{-# NOINLINE noPe #-}
noPe :: IO Int
noPe = do (p,_) <- readMVar pesVar
  return (length p)

-- place processes in round-robin manner
{-# NOINLINE selfPe #-}
selfPe :: IO Int
selfPe = do (pe,_,_) <- myInfo
  return pe

-- abstract outside!
data ChanName' a = Chan Int Int Int
  deriving (Show)

instance NFDData a => NFDData (ChanName' a)
  where rnf (Chan pe proc i) = rnf (pe + proc + i)

```

```

-- tweaking fork primop from concurrent haskell... (not returning threadID)
{-# NOINLINE fork #-}
fork :: IO () -> IO ()
fork action = do (pe,p,_) <- myInfo
  trace ("new thread")
  tMap <- takeMVar thrs
  tid <- forkIO action'
  putMVar thrs (Map.insert tid (pe,p,Nothing) tMap)
  trace ("forked! ID=" ++ show tid)
  where action' = do id <- myThreadId
    trace ("run thread " ++ show id)
    action
    removeThread id

-- creation of one placeholder and one new inport
-- returns consistent channel type (channel of same type as data)
createC :: IO ( ChanName' a, a )
createC = do (pe,p,_) <- myInfo
  id <- freshId
  var <- newEmptyMVar
  trace ("new channel in " ++ show (pe,p) ++ ", ID=" ++ show id)
  cList <- takeMVar chs
  let c = Chan pe p id
      x = unsafePerformIO $ readMVar var
      x' = fromDyn (error "createC cast") x
  putMVar chs (Map.insert id (Nothing,var) cList)
  trace "channel created!"
  return (c, x' )

-- connect a thread to a channel
connectToPort :: ChanName' a -> IO ()
connectToPort (Chan pe p cid)
  = do id <- myThreadId
    tlist <- takeMVar thrs
    putMVar thrs (Map.updateWithKey newChan id tlist)
  where newChan _ (pe,proc,_) = Just (pe,proc, Just cid)

-- send modes for sendData
data Mode = Connect -- announce sender at receiver side (no graph needed)
  | 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

sendData :: Mode -> a -> IO ()
sendData Connect _ = do ch <- myChan
  registerSender ch

sendData Data d = do cd <- myChan
  var <- getRemoveCVar cd
  putMVar var $ toDyn d

sendData Stream d = do cd <- myChan
  v2 <- newEmptyMVar
  var <- updateGetCVar v2 cd
  let x = unsafePerformIO $ readMVar v2
      newList = d: fromDyn undefined x
  putMVar var $ toDyn newList

sendData (Instantiate maybePe) d
  = do newPid <- freshId
    pes <- noPe
    pe <- if maybePe == 0 then choosePe
      else return (1+((maybePe-1) `mod` pes))
    trace ("new process on PE " ++ show pe)
    tlist <- takeMVar thrs
    id <- forkIO action
    putMVar thrs (Map.insert id (pe,newPid,Nothing) tlist)
    trace ("process,thread: " ++ show (newPid,id))
  where action = do id <- myThreadId
    trace ("process starting")
    toIO d
    removeThread id

```

## B.2 Skeletons

### B.2.1 Google MapReduce Skeleton, optimised EDI version

```
{- Google's MapReduce programming model revisited

(C) Ralf Laemmel, 2006--2007

(C) Parallelisation by JB (20080220ff), as marked:

We depart from the MapReduce skeleton described in the Google paper.
Directly taking parallelism into account, we mimic the communication
via intermediate files by channel communication: the mappers deliver
(pre-grouped) intermediate data directly to responsible reducers via
channels.

As in the first implementation, we use an equal number of map and
reduce processes. In this implementation, mapper and reducer are
gathered in one process, avoiding unnecessary communication.

Additionally, pre-grouping is not done for all intermediate data at
once, but separately for the intermediate results of each task (input
is bundled into bigger tasks by the outer interface).

This module: using lists and stream communication between processes.
-}
```

```
module MapReduce.OptEdiLists
  ( mapReduceList, mapReduce'
  ) where

import Data.Map (Map,empty,insertWith,mapWithKey,filterWithKey,
               toList,fromList,unionsWith,unions, fromListWith)

-- parallelisation
import ParMap(unshuffle)
import Edi
import Control.Concurrent
import System.IO.Unsafe
import Data.List hiding (partition)

-----

mapReduce' :: (Ord k1, Ord k2,
              NFDData k1, NFDData k2,
              NFDData v1, NFDData v2, NFDData v3, NFDData v4)
=> (v1 -> Int) -- Size of input values
-> Int -- Split size for map tasks
-> Int -- Number of partitions
-> (k2 -> Int) -- Partitioning for keys
-> (k1 -> v1 -> [(k2,v2)]) -- The *'\map'* function
-> (k2 -> [v2] -> Maybe v3) -- The *'\combiner'* function
-> (k2 -> [v3] -> Maybe v4) -- The *'\reduce'* function
-> Map k1 v1 -- Input data
-> Map k2 v4 -- Output data

mapReduce' size split parts keycode mAP cCOMBINER rREDUCE input
  = unsafePerformIO $ do output <- mapReduceList parts keycode mAP cCOMBINER rREDUCE
                        (splitInputList size split input)
                        return (concatOutputList output)

-- internally using no maps, but only lists... different input/output conversion
splitInputList :: Ord k1 => (v1 -> Int) -> Int -> Map k1 v1 -> [(k1,v1)]
splitInputList = fst
  . foldl splitHelper ([[]],0) -- 2. Splitting as a list fold
  . toList -- 1. Access dictionary as list of pairs
  where splitHelper (ps,s) x@(k1,v1)
```



```

= if getSize v1 + s < desiredSize || null (head ps)
  then ((x:head ps):tail ps), getSize v1 + s)
  else ([x]:ps,getSize v1)

concatOutputList :: Ord k2 => [(k2,v4)] -> Map k2 v4
concatOutputList rss = unionsWith undefined (map fromList rss)

mapReduceList :: (NFData k1, NFData k2,
  NFData v1, NFData v2, NFData v3, NFData v4,
  Ord k2) =>
  Int -- Number of partitions
-> (k2 -> Int) -- Partitioning for keys
-> (k1 -> v1 -> [(k2,v2)]) -- The *\map* function
-> (k2 -> [v2] -> Maybe v3) -- The *\combiner* function
-> (k2 -> [v3] -> Maybe v4) -- The *\reduce* function
-> [(k1,v1)] -- Distributed input data (list)
-> IO [(k2,v4)] -- Distributed output data
-----
mapReduceList parts keycode mAP cOMBINER rEDUCE input
= do (partCCs,partCss) <- createCs parts -- ChanName' [ChanName' Interim]
  (toXCCs,toXCs) <- createCs parts -- ChanName' (ChanName' [ChanName' Interim])
  (inCCs,inCs) <- createCs parts -- in-channels for map workers
  (resCs,ress) <- createCs parts -- results
-- instantiate workers:
sequence_ [ spawnProcessAt 0 (workerProc part inCC toXCC partCC resC) |
  (part,inCC,toXCC,partCC,resC) <-
  zip5 [1..parts] inCCs toXCCs partCCs resCs ]
sequence_ [ fork (inputSender ch cList inC wIn ) |
  ( ch,cList, inC, wIn ) <-
  zip4 toXCs (transpose partCss) inCs wIns ]
return ress
where
wIns = unshuffle parts input --static input distribution!
workerProc myPart -- parameter defines own partition to reduce
  inChanC -- :: ChanName' (ChanName' [(k1,v1)])
  -- to receive from P where to send data to peers (parts-1 elements)
  peerCCC -- :: ChanName' (ChanName' [ChanName' Interim])
  -- where to send created peerCs
  peerCC -- :: ChanName' [ChanName' Interim]
  -- where to send results
  resChan -- :: ChanName' [(k2,v4)]
= do
-- create and communicate peer channels (one too many)
(fromPeerCs,fromPeers_) <- createCs parts -- (parts - 1) peers
let (_,fromPeers) = takeOutN myPart fromPeers_
  sendNF peerCC fromPeerCs
-- create back channel to receive other peerCs (ignore own)
(toPeerCC,toPeerCs_) <- createC
let (_,toPeerCs) = takeOutN myPart toPeerCs_
  sendNF peerCCC toPeerCC
-- create and communicate input channel
(inC,input) <- createC
sendNF inChanC inC
-- do the work.
-- map mAP on input
let mapped = mapCombine input -- :: [(k2,v3)]
  mapCombine = map (reducePerKeyList cOMBINER)
    . map groupByKeyList
    . map (concatMap (uncurry mAP))
-- partition by keycode
let rData = map (partition parts keycode) mapped -- :: [[[(k2,v3)]]]
  selectFor n = concatMap (!!n-1) rData -- :: Int -> [(k2,v3)]
  rData' = map selectFor [1..parts] -- ???
  (ownRData,toPeers) = takeOutN myPart rData'
-- send other parts to others, split off comm. jobs
sequence_ [ fork (sendNFStream ch dat) |
  (ch,dat) <- zip toPeerCs toPeers ]
-- merge own and remote data, reduce
redData <- nmergeIO (ownRData:fromPeers) -- :: [(k2,v3)]
let res = reducePerKeyList rEDUCE (mergeByKeyList redData)
  sendNFStream resChan res

```

```

inputSender :: (NFData interim, NFData k1, NFData v1) =>
  ChanName' [ ChanName' interim] ->
  [ChanName' interim] ->
  ChanName' [[(k1,v1)]] ->
  [[(k1,v1)]] ->
  IO ()
inputSender peerCC peerCList inC workerIn
  = do
    sendNF peerCC peerCList
    sendNFStream inC workerIn

-- takeOutN takes n and a list, returns list!!(n-1) and the list without it
takeOutN :: Int -> [a] -> (a,[a])
takeOutN n [] = (undefined,[])
takeOutN 1 (x:xs) = (x,xs)
takeOutN n (x:xs) = (y,x:ys)
  where (y,ys) = takeOutN (n-1) xs

-----
mapPerKey :: (k1 -> v1 -> [(k2,v2)]) -> Map k1 v1 -> [(k2,v2)]
mapPerKey mAP =
  concat          -- 3. Concatenate per-key lists
  . map (uncurry mAP) -- 2. Map *'\map'* over list of pairs
  . toList        -- 1. Turn dictionary into list

-- Partition intermediate data
partition :: Int -> (k2 -> Int) -> [(k2,v2)] -> [[(k2,v2)]]
partition parts keycode pairs = map select keys
  where
    keys      = [1..parts]          -- the list 1, .., parts
    select part = filter pred pairs -- filter pairs by key
    where
      pred (k,_) = keycode k == part

mergeByKeyList :: Ord k2 => [(k2,v3)] -> [(k2, [v3])]
--
-- not distinct      distinct
mergeByKeyList =
  toList          -- 3. Convert back to list
  . fromListWith (++) -- 2. construct a Map
  -- rely on efficient Map implementation
  . map singleton -- 1. convert entries to singleton list
  where
    singleton (k,x) = (k,[x])

groupByKey :: Ord k2 => [(k2,v2)] -> Map k2 [v2]
groupByKey = foldl insert empty
  where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict

groupByKeyList :: Ord k2 => [(k2,v2)] -> [(k2,[v2])]
groupByKeyList = toList . groupByKey -- rely on efficient Map implementation

reducePerKeyList :: Ord k2 =>
  (k2 -> [v23] -> Maybe v34) -> [(k2,[v23])] -> [(k2,v34)]
reducePerKeyList f =
  mapWithKey_ unJust -- 3. Eliminate Justs
  . filter (isJust . snd) -- 2. Filter non-Nothings
  . mapWithKey_ f -- 1. Apply reduce/combiner per key
  where
    isJust (Just _) = True -- Keep entries of this form
    isJust Nothing = False -- Remove entries of this form
    unJust _ (Just x) = x -- Transforms type Maybe a into a
    mapWithKey_ f = map (\(k,v) -> (k,f k v))

```

## B.2.2 PipeIO.hs, implementation of multi-type IO-pipelines

```

--
-- Ph.D work Jost Berthold, 12/2007
--
-- pipe using IO actions as its stages, Edi version:
--
-----
-- question is, how to specify type changes in the stages, while
-- maintaining the stages as a list...
--
-- Could use TH to build the pipeline, try it NOW!
{-# OPTIONS_GHC -fth #-}
-- for template haskell code in mkPipe

module PipeIO where

import Edi
import Control.Monad

import ParPrim -- see (*)

import Language.Haskell.TH -- for generating multi-type pipelines
import Language.Haskell.TH.Syntax

pipestageIO :: (NFData a, NFData b) =>
  ChanName' (ChanName' [a]) -> -- input channel, first stage
  ChanName' [b] -> -- output channel
  (a -> IO b) -> IO () -- functionality
pipestageIO inCC outC f
  = do (inC,input) <- createC
      sendWith rwhnf inCC inC -- send input reply channel
      connectToPort outC -- process I/O stream, using ParPrim directly,
      mapM_ sendIONF input -- triggering one action per input element
      sendData Data [] -- close stream when finished
      where sendIONF x = do b <- f x -- I/O action
                          rnf b 'seq' sendData Stream b
-----

-- 2-stage:
pipe2IO_ :: (NFData a, NFData b, NFData c) =>
  (a -> IO b) -> (b -> IO c) -> [a] -> IO [c]
pipe2IO_ st1 st2 input
  = do (inCC,inC) <- createC
      (in2CC,in2C) <- createC
      (outC,out) <- createC
      spawnProcessAt 0 (pipestageIO in2CC outC st2)
      rnf in2C 'seq' -- wait for channel from 2nd stage
        spawnProcessAt 0 (pipestageIO inCC in2C st1)
      fork (sendNFStream inC input)
      return out

-- using a separate thread to spawn the processes
pipe2IO :: (NFData a, NFData b, NFData c) =>
  (a -> IO b) -> (b -> IO c) -> [a] -> IO [c]
pipe2IO st1 st2 input
  = do (outC,out) <- createC
      fork ( -- separate thread to spawn the pipeline stages:
            do (in2CC,in2C) <- createC
                spawnProcessAt 0 (pipestageIO in2CC outC st2)
                (inCC,inC) <- createC
                -- will block until in2C available from stage 2
                spawnProcessAt 0 (pipestageIO inCC in2C st1)
                sendNFStream inC input)
      return out

-- 3-stage:
pipe3IO_ :: (NFData a, NFData b, NFData c, NFData d) =>
  (a -> IO b) -> (b -> IO c) -> (c -> IO d) ->
  [a] -> IO [d]

```

```

pipe3IO_ st1 st2 st3 input          -- new: st3
  = do (inCC,inC) <- createC
      (in2CC,in2C) <- createC
      (in3CC,in3C) <- createC      -- new cc/c pair
      (outC,out) <- createC
--      rnf outC 'seq'
      spawnProcessAt 0 (pipestageIO in3CC outC st3) -- new
      rnf in3C 'seq' -- wait for channel from 3rd stage -- new
      spawnProcessAt 0 (pipestageIO in2CC in3C st2) -- changed
      rnf in2C 'seq' -- wait for channel from 2nd stage
      spawnProcessAt 0 (pipestageIO inCC in2C st1)
      fork (sendNFStream inC input)
      return out

-- 3-stage, alternative scheme using sep. threads to spawn:
pipe3IO :: (NFData a, NFData b, NFData c, NFData d) =>
  (a -> IO b) -> (b -> IO c) -> (c -> IO d) ->
  [a] -> IO [d]
pipe3IO st1 st2 st3 input
  = do (outC,out) <- createC
      fork ( do (in3CC,in3C) <- createC
              spawnProcessAt 0 (pipestageIO in3CC outC st3)
              (in2CC,in2C) <- createC
              -- and block in in3C, then
              spawnProcessAt 0 (pipestageIO in2CC in3C st2)
              (inCC,inC) <- createC
              -- and block on in2C, then
              spawnProcessAt 0 (pipestageIO inCC in2C st1)
              -- and block on inC, then
              sendNFStream inC input
            )
      return out

{- forked thread will:
   spawn->block on in3C,
   spawn->block on in2C,
   spawn->block on inC,
   send input

this is a foldM, and can include channel creation if we use the right
fold function. unfortunately, the types for different stages do not
match.
-}

pipeIOFold :: (NFData a) =>
  [(a -> IO a)] -> -- all stages as a list (same type)
  [a] -> IO [a]
pipeIOFold stages input
  = do (outC, out) <- createC
      inC <- foldM spawnWithChans outC (reverse stages)
      fork (sendNFStream inC input)
      return out

spawnWithChans :: (NFData a, NFData b) =>
  ChanName' [b] -> -- strict in result channel
  (a -> IO b) -> -- the stage
  IO (ChanName' [a]) -- input channel returned
  -- goes into previous stage

spawnWithChans outC stage
  = do (inCC,inC) <- createC
      spawnProcessAt 0 (pipestageIO inCC outC stage)
      return inC

-----
-- some Template Haskell to generate specialised code for n-staged
-- pipeline with different types inside the PL steps.

mkPipe :: Int -> ExpQ
mkPipe n = do outCN <- newName "outC"
             outN <- newName "out"
             inN <- newName "input"

```

```

    stages <- mapM newName (map (("st"++) . show) [1..n])
    -- generate code:
    (spawnStmts,inCN) <- foldM foldCode ([],outCN) (reverse stages)
    let chanPat    = TupP [VarP outCN,VarP outN]
        chanCreate = BindS chanPat (VarE (mkName "createC"))
        inSend     = apply "sendNFStream" (map VarE [inCN,inN])
        thread     = apply "fork" [DoE (spawnStmts ++ [inSend])]
        retCode    = apply "return" [VarE outN]
    return $
      LamE (map VarP (stages ++ [inN])) -- bindings
          (DoE [chanCreate,thread,retCode]) -- code
-- helper:
apply :: String -> [Exp] -> Stmt
apply fString args = NoBindS (foldl AppE (VarE (mkName fString)) args)

foldCode :: ([Stmt],Name) -> Name -> Q ([Stmt],Name)
foldCode (codeBefore,inChanE) stageFct
  = do (DoE stmts, newCN) <- spawnCode (varE inChanE) (varE stageFct)
      return ((codeBefore ++ stmts), newCN)

spawnCode :: ExpQ -> ExpQ -> Q (Exp, Name)
spawnCode chanE procE
  = do rChan <- newName "rChan"
      rN    <- newName "r"
      let pair = TupP [VarP rChan, VarP rN]
          spawn <- [| spawnProcessAt 0
                      (pipestageIO $(varE rChan) $chanE $procE) |]
      let code = DoE [BindS pair (VarE (mkName "createC")),
                    NoBindS spawn ]
      return (code,rN)
-----

-- Helper functions, suitable interface (choosing no of PEs and
-- agglomerating stages)

optPipe :: NFData x => Int -> [x -> IO x] -> [x] -> IO [x]
optPipe pes stages input
  = let peStages = agglomerateInN pes stages
      in pipeIOFold peStages input

agglomerateInN :: Int -> [x -> IO x] -> [x -> IO x]
agglomerateInN pes fs = map seqPipeIO (splitIntoN pes fs)

seqPipeIO :: [x -> IO x] -> (x -> IO x)
seqPipeIO [] = return
seqPipeIO (f:fs) = \x -> f x >>= \y -> seqPipeIO fs y

splitIntoN :: Int -> [a] -> [[a]]
splitIntoN n xs = takeIter parts xs
  where l = length xs
        parts = zipWith (+)
              ((replicate (l `mod` n) 1) ++ repeat 0)
              (replicate n (l `div` n))

takeIter :: [Int] -> [a] -> [[a]]
takeIter [] [] = []
takeIter [] _ = error "elements left over"
takeIter (t:ts) xs = hs : takeIter ts rest
  where (hs,rest) = splitAt t xs

```

## B.2.3 EdiRing.hs: EDI ring skeletons

```

module EdiRing where
-- Ring skeletons only using EDI (=> no overloading for result!)

import System.IO.Unsafe
import Control.Monad
import Control.Concurrent(yield)

import Edi

ediringnames=["Edi ring (static in)",
              "Edi ring2 (dyn.in)",
              "Edi ringRec (static in)",
              "Edi ringRec2 (caller embedded, static in)"]
edirings :: (NFData a, NFData b, NFData c) =>
  [Int -> (Int -> i -> [a]) -> ([b] -> o) ->
   ((a,[c]) -> (b,[c])) -> i -> o]
edirings = [ring, ring2, ringRec, ringRec2]

ringP :: (NFData b, NFData c) =>
  Int -> Int -> (Int -> i -> [a]) -> ([b] -> o) ->
  ((a,[c]) -> (b,[c])) -> i -> o
ringP n _ = ring n

-- Ring: Interface analog to Eden-Version, but static parent input!
-- Directly programmed (createCs/fork/send* in place of new/parfill)
ring :: (NFData b, NFData c) =>
  Int -> (Int -> i -> [a]) -> ([b] -> o) ->
  ((a,[c]) -> (b,[c])) -> i -> o
ring np split combine ringF input
  = unsafePerformIO $ ediRingIO np split combine ringF input

ediRingIO :: (NFData ro, NFData rr) =>
  -- size, Input/Output processing
  Int -> (Int -> i -> [ri]) -> ([ro] -> o) ->
  ((ri,[rr]) -> (ro,[rr])) -> -- => ring comm forced stream!
  i -> IO o
ediRingIO n dist comb f input
  = do (outCs,outs) <- createCs n -- result channels
       (rChanCs,rChans) <- createCs n -- ch.exchange channels
       -- prepare ring processes (IO actions, parameters supplied)
       let ringIns = dist n input
           ringActions = [ringNode1 rIn f outC rChanC
                         | (rIn, outC, rChanC) <- zip3 ringIns outCs rChanCs ]
       -- spawn children (no explicit placement, rely on automatic RR)
       mapM_ (spawnProcessAt 0) ringActions
       -- send ch.s for ring comm. to all children
       let (ringInCs,succCCs) = unzip rChans
           zipWithM_ (\cc c -> sendNF cc c) (leftrotate succCCs) ringInCs
       -- (lazily!) return combined results
       return (comb outs)

-- ringNode: behaviour of a node, as IO action
ringNode1 :: (NFData b, NFData r) =>
  a -> -- input (static!)
  ((a,[r]) -> (b,[r])) -> -- f
  ChanName' b -> -- result channel (no overloading!)
  (ChanName' (ChanName' [r], ChanName'(ChanName' [r]))) -> -- Back channel
  IO ()
ringNode1 input ringF outC parentCC
  = do
    (ringInC, ringIn) <- createC -- stream channel for Ring
    (succCC, succC) <- createC -- back channel for succ. channel
    sendNF parentCC (ringInC,succCC) -- send channels to caller
    -- computation
    let (out,ringOut) = ringF (input, ringIn)
        -- ring comm (extra thread)
        fork (sendNFStream succC ringOut) -- for successor
    -- output:
    sendNF outC out
    -- which is, explicitly coded:

```

```

-- connectToPort outC
-- (rnf out 'seq' sendData Data out)
-- NEVER SENDS RESULTS AS A STREAM!

-----

-- list rotation
leftrotate, rightrotate :: [a] -> [a]
leftrotate xs = last xs : init xs
rightrotate xs = tail xs ++ [head xs]

-- monadic evaluation control
rnfM :: NFData a => a -> IO ()
--rnfM = return . rnf -- ??? doznwork
rnfM x = case rnf x of { () -> yield }

-----

-- Version which sends input explicitly instead of embedding it:
-- completely analog to Eden version, unless a/b are lists
-- additionally: placement stride in ediRingEIO, set to 1 in outer interface
ring2 :: (NFData a, NFData b, NFData c) =>
  Int -> (Int -> i -> [a]) -> ([b] -> o) ->
  ((a,[c]) -> (b,[c])) -> i -> o
ring2 np split combine ringF input
  = unsafePerformIO $ ediRingEIO np 1 split combine ringF input

ediRingEIO :: (NFData ri, NFData rr, NFData ro) =>
  -- size, stride, Input/Output Generation
  Int -> Int -> (Int -> i -> [ri]) -> ([ro] -> o) ->
  ((ri,[rr]) -> (ro,[rr])) -> -- => ring comm forced stream!
  i -> IO o
ediRingEIO n stride dist comb f input
  = do (outCs, outs) <- createCs n
      (rChanCs, rChans) <- createCs n
      me <- selfPe
      np <- noPe
      let ringActions = [ringNodeE f outC rChanC
                        | (outC, rChanC) <- zip outCs rChanCs ]
          places = map ((+1) . ('mod' np) . (+me) . (*stride)) [0,1..]
      zipWithM_ spawnProcessAt places ringActions

-- difference to skeleton above HERE: receive extra input channels
let (prevCs,succCCs,inputCs) = unzip3 rChans
    ringIns = dist n input
    zipWithM_ (\cc c -> sendNF cc c) (leftrotate succCCs) prevCs
    sequence_ [fork (sendNF iC i) | (iC,i) <- zip inputCs ringIns]
return (comb outs)

-- ringNode:
ringNodeE :: (NFData a, NFData b, NFData r) =>
  ((a,[r]) -> (b,[r])) -> -- f
  ChanName' b -> -- res.channel (no overloading!)
  -- back channel for (predecessor, successor, input)
  (ChanName' (ChanName' [r], ChanName'(ChanName' [r]), ChanName' a)) ->
  IO ()
ringNodeE ringF outC parentCC
  = do
    (ringInC, ringIn) <- createC -- stream channel for Ring
    (succCC, succC) <- createC -- back channel for succ. channel
    (inputC, input) <- createC -- Input
    sendNF parentCC (ringInC,succCC,inputC) -- send channels to caller
    let (out,ringOut) = ringF (input, ringIn)
        fork (sendNFStream succC ringOut) -- to successor (Stream)
        sendNF outC out -- result to caller (NF, no stream)

```

```

-----

-- recursive version with static parent Input
-- Interface analog to Eden Version
type RingSkel i o a b r =
  Int -> (Int -> i -> [a]) -> ([b] -> o) ->
  ((a,[r]) -> (b,[r])) -> i -> o

```

```

ringRec,ringRec2 :: (NFData b, NFData r) => RingSkel i o a b r
ringRec np split combine ringF input
  | length (split np input) /= np
    = error "wrong ring size for distribution function."
  | np == 1 = let (o,r) = ringF (head $ split 1 input, r)
              in combine [o]
  | otherwise
    = unsafePerformIO $ ediRingRecIO np 1 split combine ringF input

ringRec2 np split combine ringF input
  | length (split np input) /= np
    = error "wrong ring size for distribution function."
  | np == 1 = let (o,r) = ringF (head $ split 1 input, r)
              in combine [o]
  | otherwise
    = unsafePerformIO $ ediRingRecIO2 np split combine ringF input

ringRecP :: (NFData b, NFData r) => Int -> RingSkel i o a b r
ringRecP np stride split combine ringF input = ring np split combine ringF input

-- recursively created ring, caller collects results
ediRingRecIO :: (NFData ro, NFData rr) =>
  -- size, stride, input/output processing
  Int -> Int -> (Int -> i -> [ri]) -> ([ro] -> o) ->
  ((ri,[rr]) -> (ro,[rr])) -> -- => ring comm has to be a stream
  i -> IO o
ediRingRecIO n stride dist comb f input
  = do (outCs,outs) <- createCs n
      me <- selfPe
      -- start on successor PE
      spawnProcessAt (me+1) (ringStart (dist n input) outCs)
      return (comb outs)
  where ringStart (i:is) (oC:oCs)
        = do (ringInC,ringIn) <- createC
            (succCC, succC) <- createC
            me <- selfPe
            np <- noPe
            let place = (me+stride) `mod` np -- use stride for placing
                spawnProcessAt place
                    (ringNodeRec stride is ringInC f oCs succCC)
            let (out,ringOut) = f (i, ringIn)
                fork (sendNFStream succC ringOut) -- send to successor
                    sendNF oC out -- result, never sent as stream

-- ringNodeRec: behaviour of a node in the ring (IO action)
ringNodeRec :: (NFData b, NFData r) =>
  Int -> -- placement stride
  [a] -> -- input (static!)
  ChanName' [r] -> -- closing the ring
  ((a,[r]) -> (b,[r])) -> -- f
  [ChanName' b] -> -- result channels (no overloading!)
  (ChanName' (ChanName' [r])) -> -- back channel
  IO ()
ringNodeRec _ [] _ _ _ = error "ringNodeRec: missing input!"
ringNodeRec _ _ _ [] _ = error "ringNodeRec: missing outC!"
ringNodeRec stride (i:inputs) closingC ringF (outC:outCs) predCC
  = do
    (ringInC, ringIn) <- createC -- stream channel for Ring
    sendWith r0 predCC ringInC -- send channel to predecessor
    let (out,ringOut) = ringF (i, ringIn)
        if (null inputs)
        then do
          fork (sendNFStream closingC ringOut)
          sendNF outC out
        else do
          (succCC, succC) <- createC -- create back channel for successor
          me <- selfPe
          np <- noPe
          let place = (me+stride) `mod` np -- use stride for placing
              -- create successor process
              spawnProcessAt place

```



---

```

        (ringNodeRec stride inputs closingC ringF outCs succCC)
    fork (sendNFStream succC ringOut)    -- send to successor
        sendNF outC out

-- recursively created, and caller participates in ring
ediRingRecIO2 :: (NFData ro, NFData rr) =>
    -- size, input/output processing
    Int -> (Int -> i -> [ri]) -> ([ro] -> o) ->
    ((ri,[rr]) -> (ro,[rr])) -> -- => ring comm has to be a stream!
    i -> IO o
ediRingRecIO2 n dist comb f input
    = do (closingC,ringIn) <- createC -- Input channel to close ring
        (succCC,succC)    <- createC -- back channel to successor
        (outCs,outs) <- createCs (n-1)
        let (myIn:inputs) = dist n input
            spawnProcessAt 0 (ringNodeRec 1 inputs closingC f outCs succCC)
            let (myOut,ringOut) = f (myIn,ringIn)
                fork (sendNFStream succC ringOut)
            return (comb (myOut:outs))

```

## B.2.4 PipeRings.hs: definition of a ring using a pipeline skeleton

```

{-# OPTIONS -cpp #-}
-- creating rings by pipeline skeletons (expect deadlocks)
# warning Experimental file, expect deadlocks!

module PipeRings where

import Eden
import Edi

-- pipeline skeletons
import EdenPipes
import EdiPipes

import System.IO.Unsafe -- for EdI parts...

piperings :: (Trans a, Trans b, Trans c) =>
  [ Int -> (Int -> i -> [a]) -> ([b] -> o) ->
    ((a,[c]) -> (b,[c])) -> i -> o
  ]

piperings = map (interface . closePipe) (edenPipelines ++ ediPipelines)
++
  map (interface . embedPipe) (edenPipelines ++ ediPipelines)

piperingsNames = map ("ring constructed using " ++
  (edenPipenames ++ ediPipenames)
  ++
  map ("embedded ring constructed using " ++
  (edenPipenames ++ ediPipenames)

closePipe, embedPipe :: (Trans i,Trans a,Trans o) =>
  ([ [a]->[a] ] -> [a] -> [a]) -> -- pipeline skeleton
  -- resulting ring skeleton
  ((i,[a]) -> (o,[a])) -> -- ringWorker function
  [i] -> [o] -- ring skeleton in/out

closePipe pipeSkel ringF ringIns
  = let rComm = pipeSkel ringNodes rComm -- pipe skeleton
      ringNodes = zipWith (pipeRingNode ringF) ringIns rOutCs
      (rOutCs,rOuts) = createChans (length ringIns) -- channels for results,
  in rnf rOutCs 'seq' -- force all channels
      rComm 'seq' -- activate system
      rOuts

pipeRingNode :: (Trans i,Trans a,Trans o) =>
  ((i,[a]) -> (o,[a])) ->
  i -> ChanName o -> [a]

pipeRingNode ringF rIn rOutC rCommIn
  = let (rOut,rCommOut) = ringF (rIn,rCommIn) -- apply ring function
      in parfill rOutC rOut rCommOut -- concurrently send parent output

embedPipe pipeSkel ringF (firstIn:ringIns)
  = let rCommOut = pipeSkel ringNodes rCommIn -- pipe skeleton, (n-1) instances
      -- local computation = node 1 in ring:
      (myOut,rCommIn) = ringF (firstIn,rCommOut)
      ringNodes= zipWith (pipeRingNode ringF) ringIns rOutCs
      (rOutCs,rOuts) = createChans (length ringIns) -- result channels
  in rnf rOutCs 'seq' -- force all channels
  -- rCommOut 'seq' -- activate system
  (myOut:rOuts)

embedEdiPipe :: (NFData i,NFData a,NFData o) =>
  ([ [a]->[a] ] -> [a] -> [a]) -> -- pipeline skeleton
  -- resulting ring skeleton
  ((i,[a]) -> (o,[a])) -> -- ringWorker function
  [i] -> [o] -- ring skeleton in/out

embedEdiPipe pipeSkel ringF (firstIn:ringIns)
  = let rCommOut = pipeSkel ringNodes rCommIn -- pipe skeleton, (n-1) instances
      -- local computation = node 1 in ring:
      (myOut,rCommIn) = ringF (firstIn,rCommOut)

```

```

    ringNodes= zipWith (ediPipeRingNode ringF) ringIns rOutCs
    (rOutCs,rOuts) = unsafePerformIO (createCs (length ringIns))
    -- result channels
    in rnf rOutCs 'seq'          -- force all channels
--    rCommOut 'seq'            -- activate system
    (myOut:rOuts)
ediPipeRingNode :: (NFData i,NFData a,NFData o) =>
    ((i,[a]) -> (o,[a])) ->
    i -> ChanName' o -> [a] -> [a]
ediPipeRingNode ringF rIn rOutC rCommIn
    = let (rOut,rCommOut) = ringF (rIn,rCommIn) -- apply ring function
    in unsafePerformIO $
        do fork (sendNF rOutC rOut) -- concurrently send parent output
        return rCommOut

createChans :: Trans x => Int -> ([ChanName x], [x])
createChans 0 = ([],[ ])
createChans n = new (\chX valX ->
    let (cs,xs) = createChans (n-1)
    in (chX:cs,valX:xs))

interface :: (Trans a, Trans b, Trans c) =>
    (((a,[c]) -> (b,[c])) -> [a] -> [b] ) -> -- raw ring skel
    -- resulting ring skeleton
    Int -> (Int -> i -> [a]) -> ([b] -> o) ->
    ((a,[c]) -> (b,[c])) -> i -> o
interface rawSkel np splitIn combOut ringF input
    = combOut outs
    where ringIns = splitIn np input
          outs    = rawSkel ringF ringIns

```

# Appendix C

## Erklärung

Ich versichere, dass ich meine Dissertation

Explicit and Implicit  
Parallel Functional Programming:  
Concepts and Implementation

selbstständig ohne unerlaubte Hilfe angefertigt und mich dabei keiner anderen als der von mir ausdrücklich bezeichneten Quellen und Hilfen bedient habe.

Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

Marburg, 18.4.2008

– Jost Berthold –